WILEY

Developing Software for Symbian OS

An Introduction to Creating Smartphone Applications in C++

symbian

Steve Babin

Developing Software for Symbian OS

An Introduction to Creating Smartphone Applications in C++

Steve Babin

With
Richard Harrison

Head of Symbian Press
Phil Northam

Managing Editor
William Carnegie

John Wiley & Sons, Ltd

Developing Software for Symbian OS

TITLES PUBLISHED BY SYMBIAN PRESS

- Wireless Java for Symbian Devices
Jonathan Allin
0471 486841 512pp 2001 Paperback

- Symbian OS Communications Programming
Michael J Jipping
0470 844302 418pp 2002 Paperback

- Programming for the Series 60 Platform and Symbian OS
Digia
0470 849487 550pp 2002 Paperback

- Symbian OS C++ for Mobile Phones, Volume 1
Richard Harrison
0470 856114 826pp 2003 Paperback

- Programming Java 2 Micro Edition on Symbian OS
Martin de Jode
0470 092238 498pp 2004 Paperback

- Symbian OS C++ for Mobile Phones, Volume 2
Richard Harrison
0470 871083 448pp 2004 Paperback

- Symbian OS Explained
Jo Stichbury
0470 021306 448pp 2004 Paperback

- Programming PC Connectivity Applications for Symbian OS
Ian McDowall
0470 090537 480pp 2004 Paperback

- Rapid Mobile Enterprise Development for Symbian OS
Ewan Spence
0470 014857 324pp 2005 Paperback

- Symbian for Software Leaders
David Wood
0470 016833 328pp 2005 Hardback

Developing Software for Symbian OS

An Introduction to Creating Smartphone Applications in C++

Steve Babin

With
Richard Harrison

Head of Symbian Press
Phil Northam

Managing Editor
William Carnegie

John Wiley & Sons, Ltd

Copyright © 2006 John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on 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 Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emailed to permreq@wiley.co.uk, or faxed to (+44) 1243 770620.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

Babin, Steve.
Developing Software for Symbian OS : an introduction to creating smartphone applications in C++ / Steve Babin with Richard Harrison.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-470-01845-3 (pbk. : alk. paper)
ISBN-10: 0-470-01845-3 (pbk. : alk. paper)
1. Mobile communication systems--Computer programs. 2. Operating systems (Computers) 3. C++ (Computer program language) I. Harrison, Richard. II. Title.
TK6570.M6B33 2005
621.3845′6 – dc22

2005021401

British Library Cataloguing in Publication Data

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

ISBN-13: 978-0-470018-45-3
ISBN-10: 0-470018-45-3

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Bell & Bain, Glasgow
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.

Contents

Foreword ix

Author Biography xi

Author Acknowledgements xiii

Symbian Press Acknowledgements xv

1 Smartphones and Symbian OS **1**

1.1 Smartphone Concept 2
1.2 Smartphone Features 2
1.3 Smartphone Messaging 6
1.4 Web Browsing 7
1.5 Local Device Communication Features 8
1.6 The Mobile OS 9
1.7 Symbian – A Little History 10
1.8 Symbian OS Smartphones 13
1.9 Other Smartphone Operating Systems 16

2 Symbian OS Quick Start **19**

2.1 What Do You Need to Get Started? 19
2.2 Firing Up the Development Tools 24
2.3 Simple Example Application 31
2.4 Building and Executing on the Emulator 50
2.5 Building for the Smartphone 51

3 Symbian OS Architecture **55**
3.1 Components in Symbian OS 55
3.2 Multitasking in Symbian OS 56
3.3 Dynamic Link Libraries 57
3.4 Client/Server Model 59
3.5 Memory in Symbian OS 60
3.6 The Kernel 66
3.7 Active Objects and Asynchronous Functions 69
3.8 What Is a Polymorphic DLL? 70
3.9 GUI Architecture 72
3.10 High Performance Graphics 75
3.11 The Communication Architecture 75
3.12 Application Engines, Services and Protocols 79

4 Symbian OS Programming Basics **81**
4.1 Use of C++ in Symbian OS 81
4.2 Nonstandard C++ Characteristics 82
4.3 Basic Data Types 82
4.4 Symbian OS Classes 83
4.5 Exception Error Handling and Cleanup 88
4.6 Libraries 103
4.7 Executable Files 108
4.8 Naming Conventions 110
4.9 Summary 112

5 Symbian OS Build Environment **115**
5.1 SDK Directory Structure 115
5.2 Build System Overview 117
5.3 Build Targets 118
5.4 Basic Build Flow 119
5.5 What Is a UID? 123
5.6 The Emulator 124
5.7 Building DLLs 128
5.8 DLL Interface Freezing 131
5.9 Installing Applications on the Smartphone 136
5.10 Switching Between SDKs 147

6 Strings, Buffers and Data Collections **151**
6.1 Introducing the Text Console 151
6.2 Descriptors for Strings and Binary Data 154
6.3 The Descriptor Classes 158
6.4 Descriptor Methods 172
6.5 Converting Between 8-bit and 16-bit Descriptors 186

6.6 Dynamic Buffers 186
6.7 Arrays 191
6.8 Other Data Collection Classes 200

7 Processes, Threads and Synchronization 203
7.1 Processes 203
7.2 Using Threads 213
7.3 Sharing Memory Between Processes 220
7.4 Memory Chunks 223
7.5 Thread Synchronization 228

8 Asynchronous Functions and Active Objects 235
8.1 Asynchronous Functions 235
8.2 Introducing Active Objects 237
8.3 The Active Scheduler 241
8.4 Active Scheduler Error Handling 246
8.5 Active Object Priorities 247
8.6 Canceling Outstanding Requests 247
8.7 Removing an Active Object 248
8.8 Active Object Example 249
8.9 Active Object Issues 256
8.10 Using Active Objects as Threads 258

9 Client/Server Framework 267
9.1 Client/Server Overview 268
9.2 A Look at the Client/Server Classes 268
9.3 Client/Server Example 270
9.4 Subsessions of the Server 287

10 Symbian OS TCP/IP Network Programming 293
10.1 Introduction to TCP/IP 294
10.2 Network Programming Using Sockets 296
10.3 Symbian OS Socket API 304
10.4 Example: Retrieving Weather Information 315
10.5 Making a Network Connection 322

11 GUI Application Programming 325
11.1 Symbian OS User Interfaces 325
11.2 Anatomy of a GUI Application 332
11.3 Application Classes 333
11.4 Resource Files 340
11.5 Dialogs 348
11.6 Symbian OS Controls 372

11.7 View Architecture 387
11.8 Application Icon and Caption 391

Appendix 1 Specifications of Symbian OS Phones 395

Appendix 2 Security in Symbian OS v9 425

Index 431

Foreword

By the end of March 2005, shipments of Symbian OS phones exceeded an average of two million per month, and cumulative shipments since Symbian's formation reached 32 million phones. Also at that time, there were more than 4500 commercially available, third-party applications for Symbian OS phones. Year on year, phone shipments have been virtually doubling – and that trend appears likely to continue, or even increase, for the foreseeable future.

These figures would suggest that Symbian OS is approaching maturity as the preferred operating system for high- and mid-range mobile phones, and that it offers an ideal platform to developers, on which they can create new and imaginative applications.

Symbian OS is a powerful, open operating system, which means that anyone with the right knowledge, skills and tools can create exciting new applications which will run on any Symbian OS phone. However, precisely because of that power and openness, the task of acquiring the necessary knowledge and skills can be a daunting prospect for a newcomer. Symbian Press aims to ease that task by providing a series of informative texts, covering a wide range of Symbian OS topics, at a variety of levels.

This book represents two milestones for Symbian Press: it is our first book to be written specifically for beginners in Symbian OS development, and it is the first Symbian OS C++ text in the series to have been written by an author who has not spent at least part of his working life as a developer at Symbian Ltd.

These two facts are not entirely unrelated; Steve's background means that he is ideally positioned to understand the problems facing a developer who is approaching Symbian OS for the first time. In consequence, this book provides valuable and practical answers at all stages, from obtaining and setting up a development system to the production of an installable multilingual application.

This book, however, is not just a beginner's guide. In addition to explaining the basic principles, it also describes the underlying mechanisms of a wide range of Symbian OS features, and covers a selection of these topics to a much greater depth than would be expected in an introductory text. As a result, this is a book that will be of continuing value to any Symbian OS application developer.

Richard Harrison

Author Biography

Steve Babin works at IBM developing embedded enterprise software for smartphones based on Symbian OS. He has a BSEE from Louisiana State University and over 19 years' software leadership and development experience on a variety of products – including medical devices, Java accelerators, avionics, Internet appliances, and system-on-chip silicon devices – using numerous embedded operating systems. Steve is married to Sharon and has a daughter named Hillary. They live in Austin, Texas.

Author Acknowledgements

It's surprising how much work a book is to write, and this one would not have been possible without the help of some very talented people. Working with Symbian Press and Wiley has been a great experience for me – their enthusiasm for the book and their timely and thorough responses have been exceptional. Many thanks to William Carnegie, Freddie Gjertsen and Philip Northam at Symbian Press and Sally Tickner at Wiley for their hard work, and thanks to all others who have contributed to the book.

I especially want to thank Richard Harrison at Symbian Press for his invaluable contribution. It's been a pleasure working with someone who is not only very knowledgeable on Symbian OS programming but is an excellent writer as well. His help in pointing out technical problems and making a complex subject more readable has greatly improved the book.

I also want to thank Brian Jepson whose excitement and enthusiasm for smartphones, as well as his fresh perspective on Symbian OS, helped me greatly with the earlier chapters. Also, thanks to Nick Tait for his technical review of some of the earlier chapters.

Last but definitely not least, I want to thank my wife Sharon and daughter Hillary for putting up with me while writing this book on early mornings, late nights and weekends. They have been very supportive, but have been looking forward to its completion so I can spend more time with them. I'll make up for it!

Symbian Acknowledgements

Symbian Press would like to thank Steve Babin for tenaciously toiling in Texas, on this taxing tome. We also extend our warmest thanks to Richard Harrison, a veritable pioneer of the development frontier, who skilfully wrangled some of the more recalcitrant chapters into shape. The lasso of gratitude must also fall on the shoulders of Phil N, Phil S and Freddie G, for their effervescent and, indeed, incoherent banter, depending on which round it was.

1

Smartphones and Symbian OS

Symbian OS is a full-featured mobile operating system that resides in most of today's smartphones. The demand for smartphone software is growing as these devices become more powerful and more widely used.

While Symbian OS-based smartphones are shipped with a variety of useful applications built in, an exciting aspect of these phones is that they are 'open'. This means that users can download, install and uninstall applications written by third-party developers (or by the users themselves). No special carrier service or device manufacturer's agreement is needed to distribute new smartphone applications – they can be downloaded by the user from a PC to the smartphone through a link such as USB or using Bluetooth technology (limited by the smartphone's storage space, of course).

Symbian OS provides a great opportunity for software developers since smartphone users are always looking for good applications for their devices. There is a growing list of Symbian OS software titles available as freeware or as paid downloads on numerous Internet sites (***www.handango.com*** and ***www.epocware.com*** are good examples). Available smartphone applications range from productivity, entertainment, multimedia and communications software to programs that can count fast food calories, improve your golf swing, keep diaries and calculate foreign currency exchange. And – business opportunities aside – sometimes it's just plain fun writing your own code to run on your own smartphone.

The purpose of this book is to help software developers create good software for Symbian OS-based smartphones. But, before launching into programming for Symbian OS, this chapter introduces the smartphone itself and gives an overview of its features and associated technologies. Understanding the smartphone's range of features helps the programmer to exploit them to their full potential.

I'll also discuss the company Symbian Ltd, give an introduction to Symbian OS and discuss how Symbian OS and other operating systems fit into the marketplace.

1.1 Smartphone Concept

A mobile phone that fits in your pocket and lets you communicate from and to anywhere in the world is an amazing invention. Like most inventions, mobile phones are built on a chain of prior technological advancements. Without advancements such as integrated circuits, microprocessors, semiconductor miniaturization, battery technology and, of course, the invention of telephone and radio, the modern mobile phone would not be possible.

Smartphones combine the mobile phone with another stream of technology: the computer, which adds the 'smart' in smartphone. Computers have progressed from centralized mainframes to personal computers with user-downloadable applications and graphical user interfaces. With the introduction of the Internet and email, the PC is a part of everyday life as a productivity, entertainment, and communication device. Laptops were introduced to allow PCs to be portable. Then came the mobile computing device known as the PDA – a true handheld computer.

Since the PDA and the mobile phone are both mobile devices, it's only natural that we would want to combine them into one device. After all, you only have so much pocket or purse space! This is the basic idea of a smartphone – but a smartphone is more than just the sum of two devices.

1.2 Smartphone Features

Like PDAs, smartphones can run applications such as organizers, games, and communications programs (e.g. email, browser). They can, of course, also make telephone calls. The smartphone's goal, however, is not just to limit the number of devices you carry, but also to combine mobile phone and computing technologies in a synergistic way. A simple example is the ability to pull up a person's contact information or even their picture, hit a button and automatically dial the person's phone number. Other examples include taking a picture, adding some text, and sending it instantly to a PC or another smartphone user. There are many more examples of this – and certainly many that have not even been thought of yet.

1.2.1 How Smartphones Communicate

Smartphones, like traditional mobile phones, use radio to communicate with base towers, which in turn act as gateways into landline-based communication infrastructures. While traditional mobile phone systems are based mainly on relaying voice communication between the wireless handset and the wired telephone infrastructure, smartphones provide more features that rely on network data transfer. After all, the basic concept

of the smartphone is to combine a mobile phone with a networked PDA. Improved data transfer is the current challenge for next generation mobile communications; unlike voice transfer which, requires a fixed bandwidth, the rule for data transfer is *the faster the better*.

1.2.2 Generations of Mobile Communication

With faster data speeds come better services. For example, when the bandwidth reaches a certain threshold, applications and services that transfer real-time audio and video become possible. The industry goals in wireless data communications have been categorized into generations – each generation includes a target data bandwidth as well as a set of data services available for it:

- First Generation (1G): Original analog mobile phone technology
- Second Generation (2G): Voice-centric digital systems with increased coverage and capacity and messaging
- Third Generation Transitional (2.5G): Stepping stone to 3G with always-on network connections and bandwidths up to 170 Kbps allowing better Internet browsing, email, and some audio and video; GPRS has been the dominant technology
- Third Generation (3G)/Fourth Generation (4G): Bandwidths up to 2 Mbps and 200 Mbps respectively for high-end services such as video teleconferencing.

The topic of wireless communication protocols is vast and could easily take up another book. But let's briefly cover some of the key communication technologies that apply to smartphones.

1.2.3 GSM

GSM, short for Global System for Mobile Communication, is a digital cell-based communication service that started in Europe, and has quickly spread throughout most of the world. A notable exception is the US, where CDMA is the dominant standard; however, GSM is gaining popularity there. GSM is the most supported protocol in smartphones.

GSM was designed for circuit-switched voice communication. Circuit-switched means that fixed bandwidth is reserved for each direction of a phone call for the entire duration of the voice call, whether you are talking or not.

Although originally designed for voice, GSM now has a variety of higher bandwidth data services (e.g. GPRS and EDGE) available, running on top of the base GSM protocol. This allows for faster data transfer, as we will see shortly.

The following types of GSM exist, each using its own band in the frequency spectrum: GSM 850, GSM 900, GSM 1800 and GSM 1900. The number indicates the frequency band, in MHz, that the protocol uses. Mobile phones supporting GSM 900 and GSM 1800 will ensure coverage in Europe and many other areas outside of the US, while GSM 850 and GSM 1900 are used in the US (mostly GSM 1900).

Fortunately, smartphones support multiple bands to ensure as wide a coverage as possible. It's common to have tri-band phones that support GSM 900, GSM 1800 and GSM 1900 to ensure maximum international coverage – although some still offer separate US models to reduce costs.

A GSM phone uses a Subscriber Identification Module (SIM) to gain access to the GSM network. A SIM contains all the pertinent information regarding a user's account including the services allowed. It is used to identify the user to the GSM network for billing purposes. The user can switch their SIM from one GSM phone to another, provided that the phone is not locked either to a specific carrier or to the carrier that the SIM is associated with.

1.2.4 CDMA

CDMA, which stands for Code Division Multiple Access, is a mobile phone standard that competes with GSM. CDMA currently dominates in the US and Korea, while GSM dominates virtually everywhere else. CDMA supports a high speed data mode called CDMA2000 1xRTT, which tends to hover around 50–70 Kbps, bursting up to 144 Kbps. The forthcoming CDMA2000 1xEV-DO supports rates up to 2.4 Mbps, but initial reports on the Verizon Wireless network in two test markets (San Diego and Washington, DC) made in 2003 show probable speeds of 500–800 Kbps, with peak data rates of 1.2 Mbps.

There are some smartphones based on CDMA, such as the Palm Treo 600 (both GSM and CDMA models are available). At the time of writing, however, there are no CDMA Symbian OS-based smartphones, although several do support W-CDMA (see Section 1.2.9).

1.2.5 CSD

CSD, short for Circuit Switched Data, is the most basic mode of transferring data over a circuit-switched connection like GSM. The connection is established by dialing the number of an ISP, in the same manner that a dial-up connection is started on a land-based telephone line using a PC modem. With CSD you do not need an extra data plan to send data, as you do for GPRS, which costs more (see Section 1.2.6). You can use your existing voice minutes.

There are two major disadvantages to using CSD, however. First, it takes a long time to connect since it involves dialing a number and

waiting for the server to answer the call. Second, it's slow – data transfer speed is only about 9.6 Kbps.

In GSM-based smartphones, this mode is referred to as 'Dial' or simply as GSM data. Earlier smartphones such as the Nokia 9290 rely entirely on this mode of data communication.

1.2.6 GPRS

GPRS, short for General Packet Radio Service, is a wireless technology that allows the smartphone user to quickly connect to the network and obtain good data rates. Connection time is fast since GPRS does not require any dialing (unlike CSD), and the smartphone feels as if it is always connected.

GPRS runs on top of the GSM protocol. While GSM alone is circuit-switched, GPRS is based on packet-switching technology. This means that the radio bandwidth is used only when data is actually transferred, even though you are constantly connected (circuit-switching keeps the full bandwidth reserved throughout a connection).

GPRS, in theory, supports bandwidths up to 170 Kbps. In practice, however, you'll get between 20 and 60 Kbps depending on network conditions – but this is still significantly faster than the GSM dialup data rate! The best way to think of the speed of GPRS is that it matches approximately with a PC connected to the network via a wired telephone modem. However, GPRS can feel better than dialup since it connects almost instantly to the network without the lengthy delay involved in dialing a number and establishing a call.

GPRS is a highly usable communication feature and a good preview of future wireless data communication technologies. Since it is a stepping stone to 3G technology, it is categorized as 2.5G technology. GPRS is available on most newer smartphones.

1.2.7 HSCSD

HSCSD is the high speed version of CSD. HSCSD is another 2.5G standard that supplies a comparable speed to that of GPRS (although on the lower side in many cases), but with a significant difference – the bandwidth is reserved to the smartphone throughout the connection. This is because HSCSD, like CSD and GSM, is a circuit-switched technology. This makes HSCSD better suited for applications that require a constant bit rate, although the practical bandwidth is rather low for good real-time multimedia transfers – which benefit the most from constant bit rates.

HSCSD is not widely used due to the high costs of implementation. The Nokia 6600 and the Motorola A920 are examples of smartphones that support HSCSD.

1.2.8 EDGE

EDGE, short for Enhanced Data Rates for GSM Evolution, is a GSM-based protocol that provides theoretical speeds up to 384 Kbps. It is a 2.5G technology that is sometimes referred to as 3G because of its higher speed. It is not yet as widely used as GPRS, but is gaining support. For example, AT&T has deployed EDGE on its GSM networks in the USA, reaching speeds of around 90 Kbps in practice. Smartphones such as the Nokia 9300 and Nokia 6620 support EDGE.

1.2.9 UMTS

UMTS, short for Universal Mobile Telecommunication Services, is a high speed data transfer protocol which supports bandwidths up 2 Mbps. This protocol is the basis of third generation mobile communications that make many media-rich services a possibility. This is where smartphones will really shine! UMTS is not based on GSM technology – it uses a technology called W-CDMA. However, the UMTS platform is designed to work with GSM systems to ease its deployment.

Although it seems slow in coming, once this communication platform becomes widely implemented, it will revolutionize the way people use smartphone devices.

1.3 Smartphone Messaging

Text messaging, such as email and instant messaging, is widely used on PCs connected to the Internet. It makes sense to use similar modes of communication in mobile devices. Below are the messaging features supported by smartphones.

1.3.1 SMS

SMS stands for Short Messaging Service. SMS allows mobile phone users to send and receive short text messages up to 160 characters. These messages are sent between phones with only a small delay and can occur even while a voice call is in progress. SMS is well suited to many types of communication exchange and is less intrusive than making a voice call. SMS is part of the GSM communication platform and is used by mobile phones all over the world. SMS is not yet widely used in the United States, but is slowly growing in popularity. SMS is a standard feature on today's smartphones.

1.3.2 MMS

MMS, short for Multimedia Messaging Service, is an extension of SMS that provides the ability to send media data such as pictures, audio and

video along with your text message. MMS is a natural complement to smartphones due to their audio and video capabilities. For example, a smartphone user could snap a picture of a landmark, record a quick voice comment on it and send it instantly to another mobile phone user.

MMS messages can even be sent to people who have only SMS capability by sending a text link to a browser URL containing the MMS message. You can also send and receive MMS messages between a smartphone and an email account used from a PC.

1.3.3 Email

Having the ability to keep up with your email while on the road is a standard feature found in smartphones. With high resolution scrollable displays and alphanumeric entry methods, it does not feel much different from email on a PC. Smartphones allow the user to set up multiple POP3 and IMAP email accounts.

1.3.4 Fax

Many smartphones include the ability to send and receive faxes, or can be customized to do so with fax software.

1.4 Web Browsing

Internet browsing is a standard feature for smartphones. There are many different browsers available, and they fall into two main types: WAP and HTML.

1.4.1 WAP

WAP, which stands for Wireless Application Protocol, was specifically designed for Internet browsing on resource-constrained devices. It includes lightweight markup languages designed to minimize the processing power and memory needed by the mobile device to render the web page. WAP also ensures that the page is usable on a small screen. Markup languages include WML and xHTML (mobile profile).

In many cases, proxy servers are used, which will automatically translate traditional HTML web sites to the WAP markup language before transferring to the mobile device. This is known as *transcoding*.

1.4.2 HTML

Although WAP was very important for earlier mobile devices, smartphones today have better memory, processing power and displays.

Because of this, it is feasible to include traditional HTML browsers that directly load web sites in their native format similar to a browser on a PC. Many smartphones have HTML browsers and these usually include WAP capability – sometimes combined in one browser.

1.5 Local Device Communication Features

Smartphones have a variety of communication features in addition to basic access to the cellular network. These features allow a smartphone to directly link with other devices, including PCs, PDAs, wireless headsets and other smartphones, to undertake a wide variety of data transfer functions. Below are the popular device-to-device communication means, along with some of their uses.

1.5.1 USB/Serial Cable Connection

Smartphones can be connected to a PC via either a USB or a serial cable (varies from phone to phone). This high speed link is normally used for downloading new applications to the smartphone as well as synchronizing user data, such as calendar and contact entries. A user can also access the PC's high speed network connection directly from the smartphone for much faster network access than can be achieved through the cellular network. Many products provide a cradle into which the smartphone can be plugged, both for PC connectivity and for charging the phone's battery.

1.5.2 Infrared (IR)

The smartphone provides the capability to communicate through an infrared port to a PC or other device such as a PDA. You can do all the things that can be done with the USB/Serial cable, but without plugging in any wires. IR requires a line-of-sight connection between the devices in the same way that a TV remote control does.

1.5.3 Bluetooth

Bluetooth is a short-range radio technology that enables devices to find and connect to each other. While technologies like GSM replace long lengths of wire, Bluetooth replaces the rat's nest of short wires connecting various pieces of equipment. Unlike infrared, Bluetooth does not require line of sight and will even communicate through walls.

With Bluetooth technology you can connect more conveniently to PCs and PDAs to download applications and synchronize user data than you can with cable or IR. In addition to providing basic PC to

smartphone linkage, Bluetooth technology makes more device-to-device communication scenarios possible. For instance, you can snap a picture on your smartphone and send it to a nearby printer for printing. Another use in a smartphone is in a wireless headset for hands-free operation.

Some smartphones allow themselves to be used as a modem with access to the cellular network. In this case, a device such as a PC connects to the smartphone via Bluetooth technology to provide the PC with Internet connectivity.

As more devices become available, expect many new possibilities for Bluetooth-enabled smartphones.

1.6 The Mobile OS

In the past, portable devices such as mobile phones did not require sophisticated operating systems. These earlier devices used simple, and usually proprietary, system software. In many cases they used no operating system at all and all software remained fixed in the device's Read Only Memory (ROM). Now that mobile devices such as PDAs and smartphones have greater hardware power and implement sophisticated, media-rich (downloadable) applications, it's apparent that a sophisticated operating system is needed.

1.6.1 What Makes a Good Smartphone OS?

Smartphone devices have certain characteristics that are different from traditional desktop computers and that must be addressed by a smartphone operating system:

- Resource-limited hardware Smartphones should be small, have a long battery life and cost as little as possible. To meet these requirements, smartphones, like other mobile devices, have limited memory and processing power as compared to desktop PCs and laptops. The operating system must be frugal in using hardware resources – especially memory. Not only must the OS itself use memory carefully, but the architecture should also provide support to help OS applications limit their use of memory, as well as allowing them to handle low-memory situations gracefully.
- Robustness A user expects a mobile phone to be stable and will not tolerate the device locking up. This is a challenge for any full-featured operating system due to the complexity of the system software itself; however, it is especially challenging for resource-limited devices such as smartphones that also allow third-party applications – which may be of questionable quality – to be downloaded.

Not only must the OS itself be designed to avoid crashing, it must also provide support functions and policies for applications to follow, allowing the device to handle application errors and (as alluded to before) out-of-memory situations, without locking up the phone.

- User interface for limited user hardware The OS should implement a user interface environment that is efficient and intuitive to use, despite the smaller screen and limited user input capabilities of a smartphone. Also, screen sizes and input capabilities vary between different models of smartphones, so the UI architecture should be flexible, so that it can be customized for the varying form factors.
- Library support Smartphone operating systems should contain middleware libraries and frameworks with APIs that implement and abstract the functionality of the features of the smartphone. The purpose is to provide functional consistency and to ease software development. Examples of smartphone middleware include libraries and frameworks for email, SMS, MMS, Bluetooth, cryptography, multimedia, UI features, and GSM/GPRS – the more support for smartphone features the better.
- Application development support Smartphone buyers want to know that there are many good applications available for their device, and that they can expect more and better software for it in the future. In order for this to be a reality, the OS must have good software development tools, support, training and documentation. The more productive the developers, the more powerful, easy to use and bug-free applications will appear for the smartphone.

1.7 Symbian – A Little History

The creation of Symbian OS can be traced back to a talented team of software developers at a company called Psion, an early pioneer in the handheld computer market. After successive generations of software for Psion's handheld devices, the team created an object-oriented operating system called EPOC, which was designed specifically for the unique requirements of mobile computing devices.

Psion realized that there was a need for a mobile OS that could be licensed to other manufacturers for use in their mobile products, and that their EPOC operating system was well suited for this. At the time, the mobile phone industry was looking for a general operating system suitable for mobile phones and was interested in using EPOC. In June 1998, the software team stepped out on their own with the EPOC operating system and Symbian was born. Symbian was formed as a joint venture owned by other major mobile phone manufacturers as well as Psion, with the primary goal of licensing the EPOC operating system and improving it.

Fast forward to today, and we find that Symbian's operating system – now known as Symbian OS – is a major player in the smartphone marketplace, residing in the majority of today's smartphone devices. Symbian is jointly owned by Nokia, Panasonic, Psion, Samsung, Siemens and Sony Ericsson which, together, represent a major portion of the mobile phone industry.

1.7.1 Symbian OS Overview

Symbian OS was designed from the ground up for mobile communications devices. While some competing operating systems (such as Microsoft's Smartphone OS) evolved from operating systems written for larger, more resource-laden systems, Symbian OS approached it from the other direction. Symbian's earlier versions (known as EPOC) would run on devices with as little as 2 MB of memory.

Symbian OS is a multitasking operating system with features that include a file system, a graphical user interface framework, multimedia support, a TCP/IP stack and libraries for all the communication features found on smartphones.

Symbian OS has software development kits available for third-party application development. Also, the hardware layers of the operating system are abstracted, so that phone manufacturers can port the OS to the specific requirements of their phone.

1.7.2 One OS, Various Flavors

It is challenging to create an operating system that provides common core capabilities and a consistent programming environment across all smartphones – yet at the same time allow for manufacturers to differentiate their products. Smartphones come in many different shapes and sizes with varying screen sizes and user input capabilities; the user interface software needs to vary to fit these differences.

Symbian OS has a flexible architecture that allows for different user interfaces to exist on top of the core operating system functionality. Of course, it is not wise to be too flexible for two reasons: having too many different user interfaces inhibits code reuse among different devices and too much work is required by the OEM to create a GUI from scratch for their smartphone.

So, to give the phone makers a starting point, Symbian created a few reference platforms, each packaging the Symbian OS core functionality along with a user interface that matched one of the basic smartphone form factors (screen size and input capability). This was important in the beginning; the idea was for smartphone manufacturers to choose the reference platform that most closely matched their phone's hardware characteristics, and use that as a starting point for their own customized

UI layer. This indeed is what happened, and these reference platforms were the origin of the main flavors of Symbian OS you see today – Series 60, UIQ and Series 80.

Symbian OS no longer supports the original user interface reference platforms and the smartphone programmer has no contact with them at all. Instead, the developer uses the software development kit (SDK) for the platform supported by the phone. Also, there is no generic Symbian OS SDK for the developer – all core functionality is included in the particular platform SDK. A typical platform contains about 80% common Symbian code and 20% platform-specific code.

Here are the major platforms for Symbian OS:

- Nokia Series 60 This user interface is designed for smartphones that have small displays (176×208 pixel) and where user input is performed with the basic phone keys. Nokia based Series 60 on the Symbian reference design known as Pearl, although Nokia did make significant modifications to it. Series 60 is a popular Symbian user interface for lower cost smartphones and resides in the majority of Symbian OS phones shipped. Phones that use the Series 60 user interface include the Nokia 6600, 7650, 3650. Nokia also licenses the Series 60 user interface to other manufacturers – the Sendo X is an example of a non-Nokia phone that uses Series 60.

- Nokia Series 80 Nokia based the Series 80 on a Symbian reference design known as Crystal. Series 80 is designed for phones with a half-VGA screen, a keyboard and hard buttons along the right side of the screen that have dynamic functions as defined by the application. The Nokia 9210/9290 and 9300/9500 communicator devices use the Series 80 user interface.

- UIQ This operating system originated from a Symbian reference design known as Quartz. UIQ is owned, developed, maintained and licensed by UIQ Technology AB – a wholly-owned subsidiary of Symbian Ltd. UIQ is designed for pen-based (i.e. touch screen) smartphones with quarter-VGA display and no keyboard. A virtual screen keyboard and handwriting recognition is provided for user input. The Sony Ericsson P800/P900 and Motorola A920 smartphones are examples of phones that use UIQ.

Symbian OS no longer supports or maintains the original Pearl, Crystal and Quartz reference platforms; however, they do maintain an internal platform known as *Techview*. This UI is used and maintained internally by Symbian to validate development, and is the basis of Symbian's Training SDKs. Unlike the other UIs, the Training SDK does not support building for any target phone hardware.

1.7.3 Applications

One of the exciting things about smartphones is that you can download and install your own software applications – just like you can on a PC and PDA. The number and type of Symbian OS applications are growing rapidly. Current smartphone applications range from productivity and organizer software, to foreign language translators, multimedia players and editors, games, instant messaging clients, third-party web browsers and many specialized applications that are useful for mobile users.

1.8 Symbian OS Smartphones

This section introduces three Symbian OS-based smartphones: the Sony Ericsson P900, Nokia 6600, and Nokia 9500 Communicator. These phones each correspond to a different UI series, as described in the last section, and provide a good sample of the type of smartphones found in the marketplace. All three phones allow you to download Java and C++ software applications and come with basic organizer and game software.

1.8.1 Sony Ericsson P900

The Sony Ericsson P900 (shown in Figure 1.1) is a pen-based smartphone that uses the UIQ user interface. It has a 65K color, 280×320 pixel display with touch screen, virtual keyboard and handwriting recognition, along

Figure 1.1. The Sony Ericsson P900

with many prepackaged organizer and game applications. The device plugs into a cradle that is connected to a PC via USB for downloading applications and synchronizing user data. IR and Bluetooth are also supported. The P900 has an integrated camera that can both take still pictures and record video using MPEG-4. It contains a combination WAP/HTML browser, audio and video playback, email (with attachments), SMS and MMS. The device contains 16 MB of memory for user storage and supports an external memory card to expand this.

For communication the P900 supports GSM 800, 1800 and 1900, GPRS and GSM dialup communication.

1.8.2 Nokia 6600

The Nokia 6600 (shown in Figure 1.2) is a Series 60-based phone with a 176×208 pixel, 65K color screen. Following on from the Series 60 model, this device has no touch screen and all input is via the numeric keys as well as two labeled soft-keys.

Like the P900, the device has a camera capable of taking both still pictures and video. The device has Nokia VPN software as well as digital rights management functions, so you can buy and play music that uses this protection. The device has 6 MB of user memory and it is expandable by a MMC card. In addition, the built-in software includes a WAP browser and a media player, and it supports email, SMS and MMS. Connectivity to other devices is supported via Bluetooth technology and IR, as well as PC connection via USB.

For communication the 6600 supports GSM 800, 1800 and 1900, GPRS and HSCSD.

Figure 1.2 Nokia 6600

Figure 1.3 Nokia 9500

1.8.3 Nokia 9500 Communicator

The Nokia 9500 is the latest smartphone in Nokia's high-end series of phones, known as communicators. Communicators look like traditional mobile phones (although they are a bit heavier), except that the case opens up into an easy to read landscape display and a QWERTY keyboard. Communicators use the Series 80 Symbian OS user interface. They have a 640×200 pixel screen with 4K colors (not a touch screen). The devices include a WAP and HTML browser as well as email and SMS support. User input is via the keyboard (this is the easiest smartphone for entering text) and soft labeled keys along the right side of the display.

The original communicators were Nokia's 9200 series devices. The Nokia 9290 supports GSM 1900 for the USA, the Nokia 9210 supports GSM 900 and 1800.

The 9200 series communicators, while being the easiest to use of the smartphones due to the large keyboard and screen, have two main drawbacks: their size (they are referred to affectionately as 'bricks') and their lack of high-speed data transfer (they only support CSD-style dialup). This however has changed with the recently introduced Nokia 9500 and 9300 communicators.

The Nokia 9500 communicator is smaller and lighter than the 9200 series, and has support for the faster EDGE and GPRS data transfer mechanisms. Also, impressively, it supports WiFi capability as well as Bluetooth technology for local communication. The Nokia 9500 is based on a later version of Symbian OS than the 9200 series phones (v7.0s rather than v6.0), and includes support for multi-homing – the ability to be connected to two connections at the same time (e.g. WiFi and EDGE) – so you may be browsing using EDGE but downloading email at the same time on WiFi, for example. The Nokia 9500 has 80 MB of internal memory as well as supporting a MultiMediaCard (MMC). A camera is also included with this phone.

Even smaller than the Nokia 9500 communicator is the Nokia 9300. This phone is the same as a Nokia 9500, except it has no camera and no WiFi communication. However, this communicator is significantly smaller and is aimed at users who are attracted to the usability of a communicator yet turned off by the size and weight of the previous devices.

1.9 Other Smartphone Operating Systems

The smartphone market is competitive and so, not surprisingly, there are other choices of smartphone operating system besides Symbian OS. At the time of writing, Symbian OS enjoys a wide lead in this market, but competition is expected to become fierce as smartphones become more popular and manufacturers release more phones not based on Symbian OS. There are many factors that will determine who will ultimately win this market (and sadly not all based on who make the best smartphones), but that's not the subject of this book.

This section gives a brief overview of three operating systems that compete with Symbian OS for the smartphone market: Palm OS, Microsoft Smartphone OS and Linux.

1.9.1 Palm OS

Palm OS is a major player in the PDA market and has probably done more for creating the mobile handset market than any other company. The Palm PDA products, which started with the Palm Pilot, are known for being simple to use. Palm OS, like Symbian OS, was designed specifically for lower-resource portable devices.

Since Palm is such a major force in the PDA market, and with wireless communication introduced as early as the Palm VII devices, it's only natural that Palm OS would be a good fit for the smartphone market. One of the biggest advantages is the large number of Palm PDA applications that exist that also can run on their smartphones. There is also a significant base of Palm OS application developers and documentation.

The Handspring Treo 600 is an example of a smartphone based on Palm OS. It supports both GSM and CDMA (via different models). The Treo 600 has all the standard smartphone features, such as SMS, MMS, web browsing and email, as well as the ability to connect to a PC via USB. It has a 160×160 pixel color display, a built-in thumb keyboard and integrated digital camera.

1.9.2 Microsoft Smartphone OS

There is little doubt that Windows is the dominant operating system for the PC, but Microsoft is also gaining a presence in mobile computing devices – including smartphones. This started with the creation

of Windows CE for low-resource handheld devices (or other 'embedded' devices).

Windows CE uses many of the same APIs and architecture as desktop-based Windows and includes a subset of the Windows user interface suitable for handheld devices. They released the Pocket PC as a PDA, which ran the Windows CE-based OS called Pocket PC OS. Although not as widely used as Palm devices, Pocket PCs are quite significant in the PDA market. As of 2003, the Windows CE and Pocket PC operating systems merged into the Windows Mobile family.

Microsoft also aims to be a dominant player in the smartphone market, and has released another variation of Windows Mobile called Windows Mobile Software for Smartphone. As with Palm OS, an advantage of Windows Mobile is the availability of Pocket PC applications that can be run on Microsoft-based smartphones. In addition to this, it supports miniature versions of many of the applications that are dominant in the desktop PC market – Microsoft Word and Excel, for example.

Other advantages are the large Windows developer base, the abundant programming documentation/knowledge base, and the availability of powerful development tools that have been tailored from desktop Windows to work with mobile operating systems.

An example of a smartphone that uses Windows Mobile is the Motorola MPx200, which has some of the functionality of a Pocket PC, along with a mobile phone's voice and messaging capability. This smartphone has a 176×220 pixel 65K color screen and supports GSM and GPRS. Another example is the Orange SPV.

1.9.3 Linux

Smartphones based on the open-source Linux operating system have been appearing on the market. There are many advantages to using an open-source operating system like Linux. No cost and the opportunity to tap into the Linux open source community is appealing. This has made Linux grow, not only for the server and PC market space, but also in the embedded device area including handheld computers. Sharp, for example, has released Linux-based PDAs. Linux is not likely to dominate the smartphone market any time soon, but there are smartphones being released for it and it is likely to be popular in some geographical areas, such as Asia. Motorola is a notable supporter of Linux and has released the A760 smartphone based on this OS.

2

Symbian OS Quick Start

This chapter provides a quick start guide for setting up your Symbian OS development environment, as well as walking through, building and running an example program.

If you already have your environment set up and have built Symbian OS software before, then you may be able to skip this chapter. Or, if you want to delay actual hands-on programming until you get more theory under your belt, you can return to this chapter later.

2.1 What Do You Need to Get Started?

The following are needed for developing Symbian OS smartphone software:

- A PC running Windows XP, 2000 or NT (400+ MHz is recommended).
- The Symbian SDK for your smartphone model.
- A Windows development package (Win32 development tools with an Integrated Development Environment (IDE)) supported by the SDK.
- A Symbian OS smartphone.
- The PC suite used for communication between the PC and the smartphone.

2.1.1 Build Tools Overview

Figure 2.1 shows the basic development pieces. Symbian OS software is developed and built on a host PC. You can build your software to run on the Symbian OS PC-based emulator that comes with the SDK, or you can build for the smartphone itself and load your program to the phone via the PC suite through USB, IR, or Bluetooth.

Once your application is completed, it's deployed to users as an installation file, known as a `sis` file. The user can download this `sis` file from a PC

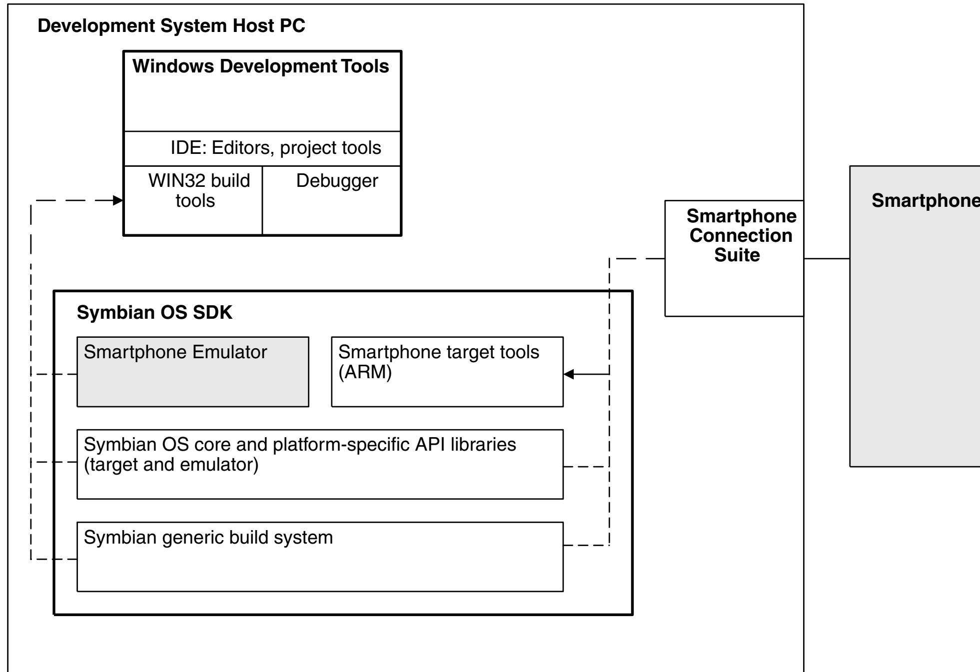

Figure 2.1 Development Tools

to a smartphone using their PC connection suite. Alternatively, they can retrieve it to the smartphone itself by downloading it from a WAP site or a website, or receiving it as an email attachment.

2.1.2 What Is the Symbian OS Emulator?

The emulator is a Windows application that implements a smartphone entirely in software – complete with simulated buttons and display. This allows you to run and debug Symbian OS software on your PC as opposed to running on a real device. Why do this?

- You avoid having to download your code to the smartphone for each code/compile/debug iteration.
- You can take advantage of the debugging support the emulator has, including single stepping and watch points.

The emulator simulates the actual smartphone fairly well, with some differences that I will discuss in more detail in Chapter 5. Each SDK has its own emulator to mimic the smartphone type that it is targeted for.

Figure 2.2 shows a sample emulator screen for the Series 60 platform.

Figure 2.2 Series 60 Emulator

2.1.3 Getting the Symbian OS SDK

Your first priority should be locating the proper SDK for your smartphone. Getting it is straightforward – they can be downloaded freely from the web on the phone manufacturer's website or the Symbian website (***www.symbian.com/developer/sdks.asp***). Normally you need to register and then click through a license agreement before you can download the SDK. Make sure you follow all instructions. You may also need to download versions of Perl and Java runtime software. For example, the Series 60 platform 2.0 SDK requires ActivePerl 518 and Java Runtime Environment 1.3.1 to be installed.

2.1.4 Getting the Windows Development Package

The Symbian OS SDK contains all that's needed for building software for a smartphone device. It also contains the PC-based emulator; however, in order to build and debug software for the emulator, you need a supported Windows development system. The Windows development package contains the Win32 development tool needed to produce emulator executables. The IDEs for these development systems also provide project management features, editors and GUI-based build tools. So with the SDK alone, you will only be able to build and load straight to your smartphone, but will have very limited debug support (normally via log files). In addition, some IDEs (e.g. Metrowerks) provide the ability to debug directly on the phone.

The following Windows development systems are currently supported by Symbian OS SDKs (although not all SDKs support all of these):

- Microsoft Visual C++ 6.0 and .Net
- Metrowerks Code Warrior
- Borland C++ BuilderX or Builder 6.0

I will not advise you which IDE to select, but selecting the IDE comes down to three main questions:

- Does the Symbian SDK for your smartphone model support the particular Windows development system?
- What development/IDE features appeal to you the most?
- What are you willing to pay for the development system?

The first one is most important – you need to make sure the SDK for your phone supports the tool set you buy. As an example, the 9200 Series SDK will not support Borland Builder 6 or Metrowerks Code Warrior tools.

If you already have Microsoft Visual C++ on your system (and you are developing for a smartphone whose SDK supports it), you can just stick with that until you gain enough experience to determine if you want to buy another one. Another option is to download a free trial IDE such as the Borland Mobile IDE (make sure this works on your SDK) until you decide you need something else.

If you have to buy an IDE, it's a good idea to consider what smartphones you may develop for in the future, and make sure the SDKs for those phone models also support your chosen IDE.

To save money, you could use the SDK without any Windows development tool. In this case, however, you would not be able to build, run and debug on the Symbian OS emulator. This can slow down development.

2.1.5 Some Example SDKs

This section describes some example Symbian OS SDKs. They represent the three main Symbian vendor software platforms that exist: Series 60, UIQ and Series 80.

- Series 60 Platform Edition 1 supports Nokia N-Gage, 3660, 3650, 3620 and 3600 as well as Siemens SX1 and Sendo X smartphones, which are based on Symbian OS v6.1. It is available as a download from the Nokia site. The basic SDK version supports Microsoft Visual C++ 6.0 and Borland C++ BuilderX development tools. Separate SDK downloads exist that support Borland Builder 6 and Metrowerks CodeWarrior.

- Series 60 Platform Edition 2 has a basic version that supports the Nokia 6600 smartphone, which is based on Symbian OS v7.0s. As with Edition 1, the standard download of the SDK supports Microsoft Visual C++ 6.0/.Net and Borland C++ BuilderX, while separate downloads are provided that support Borland Builder 6 and Metrowerks CodeWarrior.

  Enhanced versions of this SDK are available, containing additional 'Feature Packs' to support phones based on Symbian OS versions later than v7.0s. At the time of writing, three such versions are available:

  - Feature Pack 1 adds support for Symbian OS v7.0s enhanced, used, for example, on the Nokia 3230, 6670 and 6260, and Panasonic X700 and X800 smartphones.
  - Feature Pack 2 provides support for Symbian OS v8.0, used on the Nokia 6630 and 6680/1/2, and Lenovo P930 smartphones.
  - Feature Pack 3 supports the Nokia N70 and N90 smartphones, which are based on Symbian OS v8.1.

- UIQ comes in two versions: 2.0 and 2.1. Both run on smartphones based on Symbian OS v7.0. The Sony Ericsson P900 is based on UIQ 2.1 while the Sony Ericsson P800, Motorola A920 and BenQ P30 are based on UIQ 2.0. Although separate SDKs exist for the two versions of UIQ, UIQ 2.1 SDK will also support UIQ 2.0 smartphones (as long as you stay away from UIQ 2.1 specific APIs) and thus is the best one to use. Also UIQ 2.1 provides more development tool support than 2.0 (UIQ 2.0 supports Metrowerks tools only).

  UIQ 2.1 SDK supports Borland MobileX and Metrowerks CodeWarrior (via two separate downloads). Although no Microsoft tool support is claimed, Microsoft Visual C++ 6.0 or .Net can be used on the Borland MobileX version of the SDK (available in UIQ 2.1 SDK only) for basic emulator building and debugging. Both UIQ 2.0 and 2.1 SDKs can be downloaded from the Symbian site.

- 9200 Communicator SDK (Series 80) supports Nokia 9200 Communicator series smartphones, which are based on Symbian OS v6.0. Download from the Nokia site. This SDK supports only the Microsoft Visual C++ 6.0 and Borland MobileX development tools.

- Series 80 Platform 2.0 supports the Symbian OS v7.0s-based Nokia 9500 Communicator. It is available as a download from the Nokia site. Versions of the SDK are available that support Borland BuilderX and Microsoft Visual C++ .Net, as well as Metrowerks CodeWarrior.

2.1.6 Is Windows the Only Development System Operating System Supported?

At the time of writing, the only official support for Symbian OS development is on a PC running Microsoft Windows. However, there are efforts

to change this, and GNUPoc is a good example. The site ***www.gnuPoc.sourceforge.net*** provides patch downloads so you can update various Symbian SDKs for use on Linux. The tools required to build for the smartphone device run natively on Linux; however, Windows emulation (via WINE) is required when building for and running the Symbian OS emulator.

Providing native Symbian OS emulator support to other operating systems (without needing Windows emulation) will require an effort by Symbian since the source code for the Symbian OS emulator is not open to the public.

2.2 Firing Up the Development Tools

At this point, you should have your SDK and compatible windows development tool set installed. Now it's time to test your setup and compile some example code.

Here's a tip if you have multiple SDKs installed and the SDK you are using is based on Symbian OS v7.0 or later. At the command prompt, type:

```
devices
```

to list your installed SDKs. Locate the SDK you want to use and ensure it has 'default' displayed next to it. If it does not, then enter:

```
devices –setdefault @<sdk name>
```

where the SDK name is exactly as it appears on the devices line (e.g. `UIQ_21:com.UIQ` is the SDK for UIQ 2.1).

2.2.1 Quick Test Emulator

Type `epoc` from a command prompt. This should bring up the Symbian OS emulator for the supported smartphone type. It displays a smartphone desktop where you can select and run various built-in programs and setup utilities. If it does not start, or locks at some point, then you have a problem with your installation.

An example of such a problem is provided by the earlier Series 60 SDKs, which had a problem when you installed the SDK in a location other than the default. The default installation path was hard-coded in the `epoc.ini` emulator configuration file and caused the emulator not to run if your SDK was located in a different directory. To fix this problem, you would need to manually edit the path contained in `epoc.ini`.

2.2.2 Quick Test Windows Development Package

It's a good idea to do a quick test on your Windows Development System platform to ensure it is installed correctly. For example, if you are using Visual C++, type `cl` from the command line (or `nmake`) and make sure you do not get a 'command not found' error. If you do, then you need to make sure your environment variables are set up correctly for Visual C++ (e.g. running the MS VC++ `vcvars32.bat` program if needed).

2.2.3 Build Some Examples

The SDKs include example projects with source code to help you get familiar with Symbian OS. It's a good idea to build and run some of these to test out, and get familiar with, the SDK.

In the next few pages, I'll run through compiling and executing examples, platform by platform. I'll then provide some steps for building within the Metrowerks, Microsoft and Borland IDEs.

Building a Series 60 Example

The directory structure varies slightly depending on whether you are using the Series 60 v1.2 SDK or the v2.0 SDK (e.g. for Nokia 6600). Go to the Series 60 example directory from a command prompt. This directory is located at `Symbian_Base\Series60Ex`, where *`Symbian_Base`* is your SDK installation directory. I'll assume you have installed the SDK in the default location (`c:\Symbian\6.1\Series60` for v1.2, or `c:\Symbian\7.0s\Series60_v2.0` for v2.0).

For v1.2:

```
C:\>cd \Symbian\6.1\Series60\Series60Ex
```

or for v2.0:

```
C:\>cd \Symbian\7.0s\Series60_v2.0\Series60Ex
```

Type `dir` and you will see a list of folders containing examples. Change directory to `HelloWorld\group` to build the Hello World program (it's called `HelloWorldBasic` in Series 60 v2.0):

```
C:\Symbian\...\Series60Ex>cd helloworld\group
```

Type the following at the command prompt:

```
C:\...\HelloWorld\group>bldmake BLDFILES
C:\...\HelloWorld\group>abld build wins
```

This will build the example and place the output such that it will run in the Windows emulator.

Note, if you are using the Borland or Metrowerks tools then you need to specify a target of `winsb` or `winscw` instead of `wins` when running the `abld` command from the command line. `wins`, `winsb` and `winscw` indicate emulator builds for Microsoft, Borland and Metrowerks development tools respectively.

However, it is worth pointing out that Borland Mobile X can use `wins`, if configured in Microsoft binary mode. Mobile X also supports Metrowerks builds, provided you have a Metrowerks license.

To see the program executed on the emulator, at the command prompt, type:

```
C:\...\HelloWorld\group>epoc
```

You should see the Series 60 emulator come up. Find your HelloWorld icon (on the main desktop or in a folder labeled Other) and select it.

Building a UIQ Example

Go to the UIQ example directory from a command prompt. This directory is located at `Symbian_Base\UIQExamples`, where `Symbian_Base` is your SDK installation directory (e.g. `c:\Symbian\UIQ_21`):

```
C:\>cd \Symbian\UIQ_21\UIQExamples
```

Type `dir` and you will see a list of folders containing examples. Change directory to `HelloWorld` to build the Hello World program:

```
C:\Symbian\UIQ_21\UIQExamples>cd HelloWorld

C:\Symbian\UIQ_21\UIQExamples\HelloWorld>
```

To compile the sample, type the following at the command prompt:

```
C:\Symbian\UIQ_21\UIQExamples\HelloWorld>bldmake BLDFILE
C:\Symbian\UIQ_21\UIQExamples\HelloWorld>abld build wins
```

After the compilation completes, type `epoc` to start the emulator and run the application.

This will build the UIQ example for Microsoft Visual Studio or Borland C++ BuilderX (use `winscw` instead of `wins` for Metrowerks CodeWarrior).

Building a Series 80 Communicator Example – Nokia 9500/9300

Go to the Series 80 example directory from a command prompt. On the Series 80 v2.0 SDK (for the 9500/9300 communicators) this directory is located at `Symbian_Base\Series80Ex`, where `Symbian_Base` is your SDK installation directory (e.g. `c:\Symbian\7.0s\S80_DP2_0_SDK\`):

```
C:\>cd \Symbian\7.0s\S80_DP2_0_SDK\Series80Ex\
```

Type `dir` and you will see a list of folders containing examples.

Change to the `helloworldbasic\group` directory to build the basic hello world program:

```
C:\Symbian\7.0s\S80_DP2_0_SDK\Series80Ex\>cd helloworldbasic\group
```

To compile the sample, type the following bldmake and abld commands at the command prompt:

```
C:\Symbian\...\helloworldbasic\group>bldmake BLDFILE
C:\Symbian\...\helloworldbasic\group>abld build wins
```

After the compilation completes, type `epoc` to start the emulator. Once the emulator is up, select the `helloworldbasic` icon from the desktop to run the application.

If you are using the SDK for the older Nokia 9200 series communicators, Hello World is located at `\Symbian\6.0\NokiaCPP\Epoc32Ex\CrystalUI\HelloWorld`.

In the preceding sections, I mention that the last argument of the `abld` command depends on the Windows-based tool kit you are using. Why is there a different target platform indicator (i.e., `wins`, `winscw`, `winsb`) for each tool set? The reason is that `abld` generates and invokes makefiles that in turn build your program. So `abld` needs to know the target platform to determine what tool set to use. For example, when you specify `wins` in the `abld` command, `abld` creates a Microsoft `nmake` style makefile that contains calls to Microsoft tools (such as `cl` for the compiler). The target platform also specifies what set of libraries to link to since there is a separate set of binary system libraries for each target platform. This will be discussed further in Chapter 5.

Building Using an IDE

The previous sections described how to build the examples from the command line. You can also build from your tool set's IDE if you want,

however, the steps to do this vary depending on your tool set. Here are some basic steps for building using the Metrowerks, Visual C++ and Borland IDEs.

Metrowerks

To build the examples from the Metrowerks Code Warrior IDE:

1. Start Code Warrior IDE.
2. Select `File`, `Import Project From .mmp File`.
3. Select your SDK version and then click on Next.
4. You will be prompted for your `mmp` file and the build platform. Browse to and select the example's `mmp` file (e.g. `helloworld.mmp`) and enter `winscw` as the software platform. Select Next.
5. You should see the project come up with its source files and library folders.
6. Click the green Run icon. The emulator will start.
7. Your program should appear as an item on the emulator's desktop. Select it to run it.

Microsoft Visual C++

To build using the Microsoft Visual C++ Studio 6.0 IDE, you first use the command line tools to create a Microsoft Visual C++ project workspace file. Once you create the project file, you can then load it from the IDE and build, execute and debug with it.

To generate the Visual C++ 6.0 workspace file, enter the following at the command prompt (substitute the correct directory where the example's `mmp` file resides):

```
C:\...\HelloWorld\group>bldmake BLDFILES
C:\...\HelloWorld\group>abld makefile vc6
```

The `vc6` is a special type of target that tells the `abld` command to generate a Microsoft VC++ 6.0 project workspace file as opposed to actual executable output. The workspace file generated by this example is named `helloworld.dsw` and is put in the example's subdirectory under the SDK's `epoc32\build directory.` For example, the Series 60 v1.2 Hello World example's workspace file would be placed in the `<SDK_ROOT>\epoc32\build\symbian\6.1\series60\Series-60ex\HelloWorld\group\HelloWorld\wins` directory.

Next, launch the Visual C++ IDE and perform the following steps:

1. Select `File, Open Workspace` and select the DSW workspace file for your example (e.g. `helloworld.dsw`).
2. Build and run the project by running the execute command (via the build menu, toolbar or by depressing `Ctrl, F5`). You'll be prompted for an executable the first time you run the project. Enter the full pathname for the emulator executable `epoc.exe`, which is located at `epoc32\wins\udeb\epoc.exe` relative to your SDK directory.
3. Once the emulator starts, run and debug your application using features such as break points and single stepping.

Borland C++ BuilderX Mobile

To build using Borland's C++ Builder X Mobile's IDE, perform the following steps:

1. Select `File, New` to bring up the Object Gallery dialog. Click on the Mobile C++ tab.
2. Select `Import Symbian C++ Project` and click `OK`, to start the Import Symbian C++ Project Wizard.
3. Browse to select the example's `bld.inf` file and, if necessary, select the appropriate SDK, platform (e.g. `WINSB`) and Build (e.g. `UDEB`). Click on `Next`.
4. Next type in a suitable project name and click `Finish`.
5. Select `Run, Run Project` to build and run the emulator version (provided you selected `WINSB` above).
6. Select your application's icon in the emulator.

Library Freezing

Sometimes, you may get errors compiling examples that use libraries. If you get an error indicating that a library is missing, it's normally because the library could not be built due to it not being frozen. Library freezing will be explained in detail in Chapter 5, but for now if you get that error, type `abld freeze wins` (or whatever platform you are using), then reissue the `abld build` command. Or it could be that the library has not yet been built at all – some SDKs require you to go to the individual library directories and build them before building the main program.

2.2.4 Resolving Problems

Did everything work? Sometimes you can run into problems. Due to the numerous versions of the SDK and tool sets (not to mention PC

configurations), it's not realistic to provide all the information here to ensure the development tools run correctly, but here are some tips and traps to watch out for when working with the tools:

- Make sure you launch the right command prompt

  You may need to launch the command prompt from a shortcut that was created by your development tools in order to get all the tools you need in your `%PATH%`. For example, Visual Studio .NET creates a shortcut: `Start, All Programs, Microsoft Visual Studio .NET 2003, Visual Studio .NET Tools, Visual Studio. NET 2003 Command Prompt`.

- Install the SDK in the default directory

  Although this situation is getting better, sometimes you can encounter problems when you install your SDK to a location other than the default installation directory. So, unless there is a good reason, stick to this default.

- Watch out for `%EPOCROOT%`

  The `%EPOCROOT%` environment variable controls a number of things, such as where compiled applications are installed. On Symbian SDKs based on Symbian OS versions before 7.0, `%EPOCROOT%` needs to be set up in your command shell environment to point to the directory that contains the `epoc32` subdirectory (such as `\Symbian\6.1\Series60\` for the Series 60 SDK). However, if you use an SDK that is based on Symbian OS 7.0 and higher, you need to make sure the `%EPOCROOT%` variable is not set in your environment. This is because 7.0 SDKs will set the `%EPOCROOT%` internally and if you have it set externally, before the tools are invoked, it will override the correct internal setting (you could end up using the tools of one SDK, but linking to libraries and installing applications in another!). Also, make sure, when using a 7.0 (or higher) SDK, that you do not have tool directories of an earlier SDK still in your `%PATH%`.

- Make sure you correctly manage multiple SDKs.

  Managing multiple SDKs on a single machine is straightforward if all your SDKs involve Symbian OS v7.0 or later. It can, however, be a headache if you have SDKs based on versions earlier than v7.0 (for example, the Series 60 v1.0 or Series 80 v1.0). It is certainly possible, once you know the issues involved, to cleanly switch between the SDKs. Refer to the Switching Between SDKs section in Chapter 5 for information on this.

If you are still having problems, your best bet is to review (and, if needed, post a question on) the different Symbian OS news groups on the Internet, such as ***www.forum.nokia.com*** or the various Symbian support

newsgroups. It's quite likely that someone else will have run into the issue you are having.

2.3 Simple Example Application

This section walks through a simple example of a Symbian OS GUI application, from source code to execution. The example is presented for Series 60, UIQ, and Series 80 smartphones. Much of the example code is common between the platforms but, where there are differences, they are given in separate, platform specific files.

Although you could download this example from the book's website, you may want to actually type in the code in order to get a good feel for the core classes. To build the example, just enter the listings that apply to your chosen target platform, as well as the common code listings. It is instructive, however, to compare the platform-specific portions of the code, to get an idea of variations between platforms.

This chapter will not examine the code in detail – that will be done in later chapters. Some description is provided, but don't worry if you do not understand it all. It will be explained in due time. For now, the main goal is to get a feel for developing a basic application by actually building and running one.

2.3.1 Application Components

Here are the minimum classes required for a Symbian application. You will be creating them for your example application:

- Application View: The root GUI control, this class implements the main window and acts as a container for the other application controls.
- Application UI: This class instantiates the application view and handles the commands sent from the application's GUI controls.
- Application document: This class handles the non-GUI data aspects of the application – the application data. It also instantiates the application's UI class.
- Application: The main application class starts the application by instantiating and starting the document class. It also sets the application's UID (a unique identifier that is required for each application – for more information, see Chapter 4).

2.3.2 Overview of `SimpleEx`

This section presents an example GUI application called `SimpleEx`. When launched, the example displays the text 'Simple Example' in the center of the window. It also has a menu item labeled 'Start' that displays an alert dialog when selected (which displays 'Start selected!'). The menu is slightly different on each platform due to user interface differences. In

the case of Series 80, a softkey is defined to display the dialog, in addition to the menu item, to demonstrate the use of softkeys on that platform.

The example contains the following files organized in the directories as shown:

```
\include
    SimpleEx.h
    SimpleEx.hrh
\src
    SimpleEx.cpp
    SimpleEx_app.cpp
    SimpleEx_doc.cpp
    SimpleEx_ui.cpp
    SimpleEx_view.cpp

\group
    bld.inf
    SimpleEx.rss
    SimpleEx.mmp
    SimpleEx.pkg
```

To create the example you will perform the following steps:

1. Create the application header (`h`) file.
2. Create the resource (`rss` file) and resource command header (`hrh`) files.
3. Create the source code for the application classes declared in the header (`cpp` files).
4. Create project definition (`mmp` and `bld.inf`) files.
5. Build and run the example on the PC-based emulator.
6. Create the target package definition (`pkg`) file.
7. Generate the installation package (`sis` file) and install it on the smartphone.
8. Execute the application on the smartphone.

2.3.3 Header File

This section describes the header files for our example, defined for Series 60, UIQ, and Series 80 platforms. Put the header file for your particular platform in a file named `SimpleEx.h` and place it in the `include` directory.

Example 2.1 shows the Series 60 header file.

Example 2.1. Sample Application header for Series 60

```
#ifndef __SIMP_H
#define __SIMP_H
```

```
/*============================================
   Series 60 SimpleEx Header File
============================================*/

#include <eikenv.h>

#include <eikon.hrh>
#include "SimpleEx.hrh"

#include <akndoc.h>
#include <aknapp.h>
#include <aknappui.h>


// The Application Class

class CSimpleExApplication : public CAknApplication
  {
private:
  CApaDocument* CreateDocumentL();
  TUid AppDllUid() const;

  };

// The Application View Class

class CSimpleExAppView : public CCoeControl
  {
public:
  static CSimpleExAppView* NewL(const TRect& aRect);
  static CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect);
  void ConstructL(const TRect& aRect);
private:
  void Draw(const TRect&) const;
  };

// The UI Class

class CSimpleExAppUi : public CAknAppUi
  {
public:
  void ConstructL();
  ~CSimpleExAppUi();

private:
  void HandleCommandL(TInt aCommand);
  CSimpleExAppView* iAppView;
  };

// The Application Document Class

class CSimpleExDocument : public CAknDocument
  {
public:
  CSimpleExDocument(CEikApplication& aApp): CAknDocument(aApp) { }
private:
  CEik AppUi* CreateAppUiL();
  };
```

```
#define SERIES_60

#endif
```

Example 2.2 shows the UIQ header file.

Example 2.2. Sample Application header for UIQ

```
/*==========================================
    UIQ SimpleEx Header File
==========================================*/

#ifndef __SIMP_H
#define __SIMP_H

#include <eikenv.h>

#include <eikon.hrh>
#include "SimpleEx.hrh"

#include <qikdocument.h>
#include <qikapplication.h>
#include <qikappui.h>


// The Application Class

class CSimpleExApplication : public CQikApplication
  {
private:
  CApaDocument* CreateDocumentL();
  TUid AppDllUid() const;
  };

// The Application View Class

class CSimpleExAppView : public CCoeControl
  {
public:
  static CSimpleExAppView* NewL(const TRect& aRect);
  static CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect);
  void ConstructL(const TRect& aRect);

private:
  void Draw(const TRect&) const;
  };

//  The UI Class

class CSimpleExAppUi : public CQikAppUi
  {
public:
  void ConstructL();
  ~CSimpleExAppUi();
```

```
private:
  void HandleCommandL(TInt aCommand);
  CSimpleExAppView* iAppView;
  };

// The Application Document Class

class CSimpleExDocument : public CQikDocument
  {
public:
  CSimpleExDocument(CEikApplication& aApp) : CQikDocument(aApp) { };

private:
  CEikAppUi* CreateAppUiL();
  };

#endif
```

Example 2.3 shows the Series 80 header file.

Example 2.3. Sample Application header for Series 80

```
/*================================================================
    Series 80 (Communicator Series) SimpleEx Header File
================================================================*/

#ifndef __SIMP_H
#define __SIMP_H

#include <eikenv.h>

#include <eikon.hrh>
#include "SimpleEx.hrh"

#include <coeccntx.h>

#include <eikappui.h>
#include <eikapp.h>
#include <eikdoc.h>
#include <eikmenup.h>

// The Application Class

class CSimpleExApplication : public CEikApplication
  {
private:
  CApaDocument* CreateDocumentL();
  TUid AppDllUid() const;
  };

// The Application View Class

class CSimpleExAppView : public CCoeControl
  {
public:
  static CSimpleExAppView* NewL(const TRect& aRect);
  static CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect);
```

```
  void ConstructL(const TRect& aRect);

private:
  void Draw(const TRect&) const;
  };

// The UI Class

class CSimpleExAppUi : public CEikAppUi
  {
public:
  void ConstructL();
  ~CSimpleExAppUi();

private:
  void HandleCommandL(TInt aCommand);
  CSimpleExAppView* iAppView;
  };

// The Application Document Class

class CSimpleExDocument : public CEikDocument
  {
public:
  CSimpleExDocument(CEikApplication& aApp): CEikDocument(aApp) { }

private:
  CEikAppUi* CreateAppUiL();
  };

#endif
```

Notice from the header files that the application, document, and UI classes are derived from different base classes for each of the platforms (the differences are shown in bold text). For example, the Application UI class is derived from `CAknAppUI`, `CQikAppUI`, and `CEikAppUI` for Series 60, UIQ, and Series 80 respectively.

Figures 2.3, 2.4 and 2.5 show the class hierarchy for `SimpleEx`, for Series 60, UIQ, and Series 80 respectively.

Notice that the application classes for Series 60 and UIQ derive from platform specific classes, which in turn derive from core application classes. Series 80 applications, however, inherit directly from the core `CEIK` classes as shown in Figure 2.5.

Fortunately, the application platform classes for the different platforms are very similar from a development perspective due to their abstracted interface. This will be apparent when you examine the implementation code for `SimpleEx`.

The `application view class` for all three platforms is derived directly from the basic control base class – `CCoeControl` – of the UI Control Framework.

Although, for this example, I have put all the classes in one header file, it is common to separate each class into its own header file.

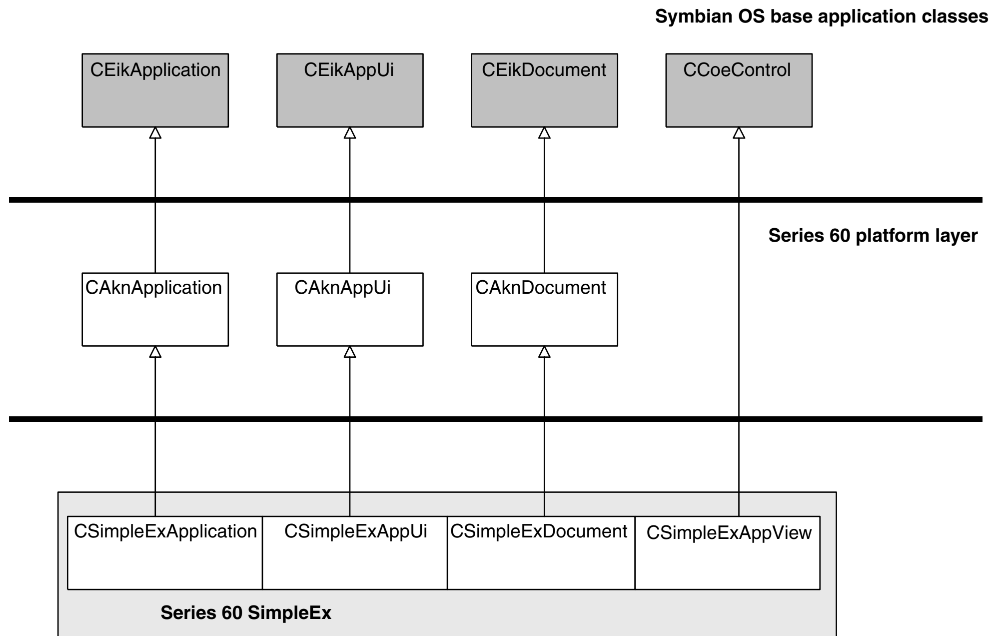

Figure 2.3 `SimpleEx` class hierarchy for Series 60

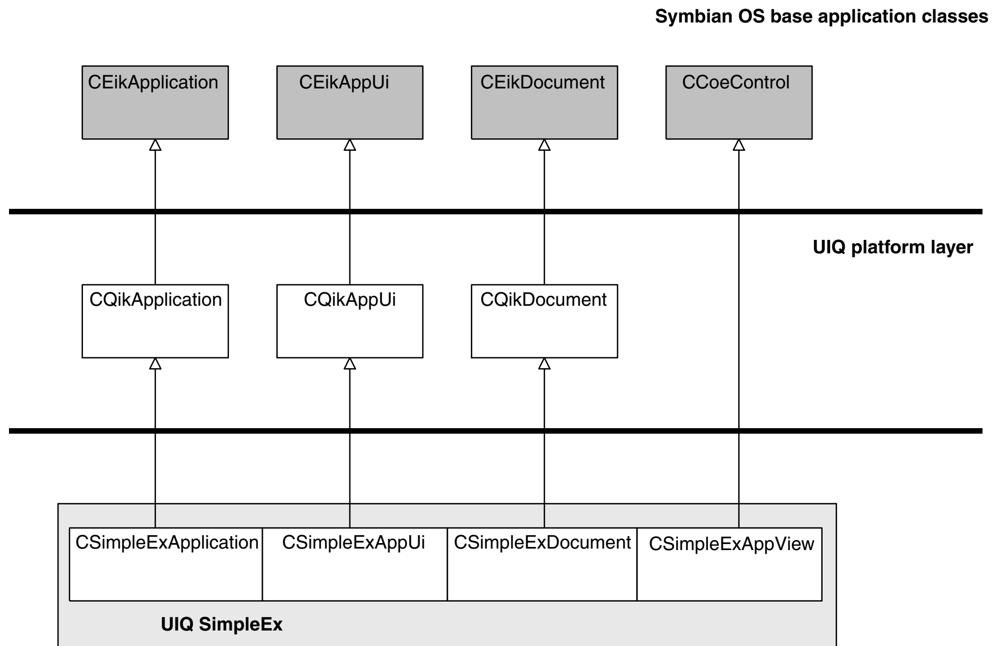

Figure 2.4 `SimpleEx` class hierarchy for UIQ

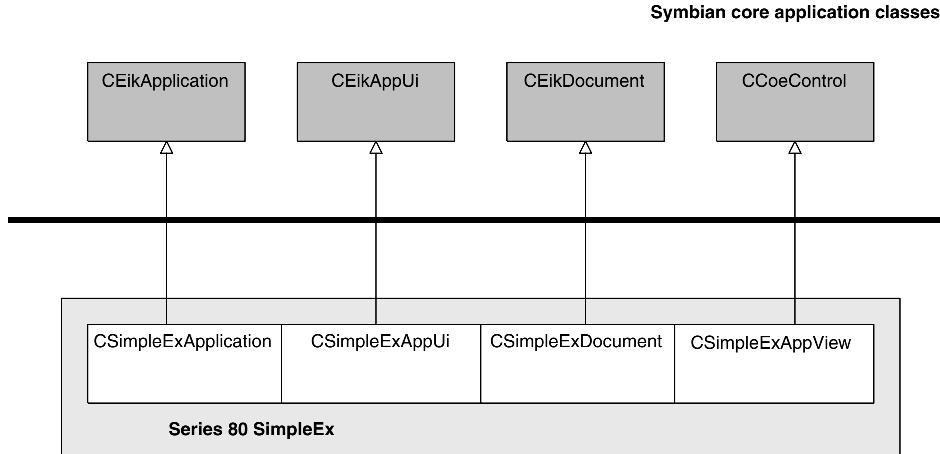

Figure 2.5 `SimpleEx` class hierarchy for Series 80

At this point, you should have, in the include directory, the `SimpleEx.h` that corresponds to your platform.

2.3.4 Resource File

Now, let's create the resource file to define the UI elements – in this case the menu/softkey item used to display the alert dialog.

Examples 2.4, 2.5, and 2.6 show the resource files for Series 60, UIQ, and Series 80 – enter the one corresponding to your platform into a file called `SimpleEx.rss`, and place it in the `group` directory.

Example 2.4 shows the resource file for Series 60.

Example 2.4. Series 60 Resource File

```
/*====================================
  Series 60 SimpleEx Resource File
=====================================*/

NAME SIMP

#include <eikon.rh>
#include <avkon.rh>
#include <avkon.rsg>

#include "SimpleEx.hrh"

RESOURCE RSS_SIGNATURE
  {
  }

RESOURCE TBUF r_default_document_name
  {
  buf="";
  }
```

```
RESOURCE EIK_APP_INFO
  {
  menubar = r_SimpleEx_menubar;
  cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
  }

RESOURCE MENU_BAR r_SimpleEx_menubar
  {
  titles =
    {
    MENU_TITLE
      {
      menu_pane = r_SimpleEx_menu;
      }
    };
  }

RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      }
    };
  }
```

Example 2.5 shows the resource file for UIQ.

Example 2.5. UIQ Resource File

```
/*=====================================
   UIQ SimpleEx Resource File
=====================================*/

NAME SIMP

#include <eikon.rh>

#include "SimpleEx.hrh"

RESOURCE RSS_SIGNATURE
  {
  }

RESOURCE TBUF r_default_document_name
  {
  buf="";
  }

RESOURCE EIK_APP_INFO
  {
  menubar = r_SimpleEx_menubar;
  }
```

```
RESOURCE MENU_BAR r_SimpleEx_menubar
  {
  titles =
    {
    MENU_TITLE
      {
      menu_pane = r_SimpleEx_menu; txt="Simple Menu";
      }
    };
  }

RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      }
    };
  }
```

Example 2.6 shows the resource file for Series 80.

Example 2.6. Series 80 Resource file

```
/*=====================================
   Series 80 SimpleEx Resource File
=====================================*/

NAME SIMP

#include <eikon.rh>

#include "SimpleEx.hrh"

RESOURCE RSS_SIGNATURE
  {
  }

RESOURCE TBUF r_default_document_name
  {
  buf="";
  }

RESOURCE EIK_APP_INFO
  {
  cba=r_simpleEx_cba;
  menubar = r_SimpleEx_menubar;
  }

RESOURCE CBA r_simpleEx_cba
  {
  breadth=80;
  buttons=
```

```
    {
    CBA_BUTTON
      {
      id=ESimpleExCommand;
      txt="Start";
      bmpfile="";
      bmpid=0xffff;
      },

    CBA_BUTTON
      {
      txt=" ";
      bmpfile="";
      bmpid=0xffff;
      },
    CBA_BUTTON
      {
      txt="";
      bmpfile="";
      bmpid=0xffff;
      },
    CBA_BUTTON
      {
      id=EEikCmdExit;
      txt="Close";
      bmpfile="";
      bmpid=0xffff;
      }
    };
  }

RESOURCE MENU_BAR r_SimpleEx_menubar
  {
  titles =
    {
    MENU_TITLE
      {
      menu_pane = r_SimpleEx_menu; txt="Simple Menu";
      }
    };
  }

RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      }
    };
  }
```

A resource file is a text file that defines the user interface elements of an application. As in other operating systems (e.g. Microsoft Windows), the developer can use explicit programming techniques to create GUI

controls; however, the resource file provides a more manageable alternative. In Symbian OS, the resource file must be text; at the time of writing there are no tools to create resource files in a graphical way (as can be done in developer tools such as Visual Studio).

Let's skim through the highlights of the resource file to help understand the `SimpleEx` example. Resource files contain a set of `RESOURCE` structures to define the program's GUI elements. The `EIK_APP_INFO` resource defines general application attributes such as the application's default menu, softkey settings, tool bars, status panes and hotkey definitions. I define two things in the `EIK_APP_INFO` resource for `SimpleEx`: the default menu and default softkey definitions. Note that UIQ has no softkeys, so only the default menu is supplied.

The `menubar` attribute of `EIK_APP_INFO` is assigned a resource of type `MENU_BAR` which specifies the application's default menu. Menu bar resources have one or more menu titles (type `MENU_TITLE`), and each menu title points to a menu pane (type `MENU_PANE`). The menu bar in the example, `r_simplex_menubar`, has a single menu title and this points to menu pane `r_simplex_menu`.

Menu panes define the actual menu items (type `MENU_ITEM`) which the user selects to invoke some operation in the application. `r_simplex_menu` defines a menu item labeled 'Start' that sends the command `ESimpleExCommand` to the GUI command handler code when the user selects it (so the code can display the example's dialog).

For Series 80 and UIQ, I specify the label 'Simple Menu' (`txt` attribute) for the menu title. This is because both Series 80 and UIQ display text for each menu title in the same way as Microsoft Windows – the corresponding menu pane pulls down menus underneath these titles (e.g. like the menu item `File, Open` in Windows applications, in this case 'File' is the title). Series 60 menus, however, do not have these first level menu titles displayed across the top. Selecting the menu softkey displays all the menu items. This is why Series 60 defines no text for its menu title.

Note that Series 80 and UIQ can have multiple menu panes (with titles), but Series 60 should only have one; adding more will show all the items from all the menu panes together, as if you had defined them all in one pane.

The `cba` attribute in the `EIK_APP_INFO` resource defines the application's softkeys (as default, they can be changed dynamically). As mentioned, softkeys apply to Series 60 and Series 80 only – UIQ has no softkeys. Symbian refers to a set of softkeys as a Command Button Array (CBA).

In Series 60, there are two softkeys at the bottom of the screen. In the Series 60 example resource file, I set the attribute `cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT`. This is a predefined system value which specifies that the left softkey brings up the menu and the right one exits the application.

Series 80 has four softkeys along the right side of the screen. In the Series 80 example, I define the softkey structure `r_simpleEx_cba`. This structure defines the four softkeys from top to bottom. I labeled the top key 'Start' and set it to send `ESimpleExCommand` as the Start menu item does. The last softkey is labeled 'Close' and is set to exit the application. The other two softkeys are set to perform no operation.

The file `SimpleEx.hrh`, shown in Example 2.7, is the same for all platforms. This file contains the command values that the controls send (specified in the resource file) for the application code to handle. In this case we have only one, used when Start is selected. The `.hrh` file is an include file that is used by both the resource file and the C++ code which handles the events.

Example 2.7. `SimpleEx.hrh`

```
#ifndef __SIMPLEEX_HRH__
#define __SIMPLEEX_HRH__

/** SimpleEx enumerate command codes */
enum TSimpleExIds
  {
  ESimpleExCommand = 1
  };

#endif // __SIMPLEEX_HRH__
```

2.3.5 Source Files

Examples 2.8 through 2.12 show the application source files. They are written to be common across all platforms. This is a good general practice since it makes for more portable code. It also illustrates the similarities of the application structures between the various Symbian OS platforms. Type the code as shown into their respective files and place them in the `src` directory.

SimpleEx.cpp

This file contains the entry point of the application. E32DLL is required for all DLLs (the application is, in fact, a DLL). It must return a successful status (`KErrNone`) although it does not do anything else. The `NewApplication()` method is called by the Symbian OS application framework to create and return a pointer to the application object that is defined in `SimpleEx_app.cpp`.

Example 2.8. `SimpleEx.cpp`

```
#include "SimpleEx.h"

GLDEF_C TInt E32Dll(TDllReason )
```

```
  {
  return KErrNone;
  }

EXPORT_C CApaApplication* NewApplication()
  {
  return (new CSimpleExApplication);
  }
```

SimpleEx_App.cpp

This file contains the application class implementation. These methods are called by the GUI framework when starting the application. It defines and returns the application's UID and creates and returns the application document object.

Example 2.9. `SimpleEx_App.cpp`

```
#include "SimpleEx.h"

const TUid KUidSimpleExApp = {0x10005B94};

CApaDocument* CSimpleExApplication::CreateDocumentL()
  {
  return new(ELeave) CSimpleExDocument(*this);
  }

TUid CSimpleExApplication::AppDllUid() const
  {
  return KUidSimpleExApp;
  }
```

SimpleEx_Doc.cpp

This file contains the application document object. This example has no actual persistent data, but even so, the example overrides the `Create-AppUiL()` method of the document class since the framework calls this method to create and pass a pointer to the application UI class.

Example 2.10. `SimpleEx_Doc.cpp`

```
#include "SimpleEx.h"

CEikAppUi* CSimpleExDocument::CreateAppUiL()
  {
  return new (ELeave) CSimpleExAppUi;
  }
```

SimpleEx_UI.cpp

Example 2.11 shows `simplex_UI.cpp`, which contains the example application's UI class. The UI class in a GUI application is where the action is, since it is where the application handles user events. All user

events (except alphanumeric keyboard input and low level touch screen events) come through the UI class method `HandleCommandL()`.

When the menu item 'Start' is selected, the GUI framework invokes the `HandleCommandL()` method, passing it the command `ESimpleExCommand` (the command specified in the menu resource in the `SimpleEx` resource file). `HandleCommandL()` responds to this command by popping up an alert window with the message 'Start Selected!'.

I used the `iEikonEnv->AlertWin()` function for the pop-up since this is a core Uikon method available to all platforms. This looks quite good on UIQ, but fairly plain on the other platforms. In practice, you should follow the UI guidelines for the software platform you are using. This may involve using platform-specific classes for controls or message displays instead of core Uikon controls.

As an example, in Series 60 you can use the Series 60 specific UI control class `CAknInformationNote` to pop up a message. This would look better for the Series 60 UI than the core Uikon alternative.

The UI class `ConstructL()` method creates the application view. `ConstructL()` is called by the framework after getting a pointer to the UI object from the document. You will see later how it is common for Symbian C++ objects to be constructed in two steps: instantiating the C++ class then invoking its `ConstructL()` method.

Also note that, in the `HandleCommandL()` method, command `EAknSoftkeyExit` should be in the Series 60 application only. It handles the Series 60 Exit softkey that is put up at application start, as specified by the `R_AVKON_SOFTKEYS_OPTIONS_EXIT` option in the Series 60 resource file.

Example 2.11. `SimpleEx_UI.cpp`

```
#include "SimpleEx.h"

void CSimpleExAppUi::ConstructL()

  {
  BaseConstructL();

  iAppView = CSimpleExAppView::NewL(ClientRect());
  }

CSimpleExAppUi::~CSimpleExAppUi()
  {
  delete iAppView;
  }

void CSimpleExAppUi::HandleCommandL(TInt aCommand)
  {
  switch(aCommand)
    {
    case EEikCmdExit:
#ifdef SERIES_60
```

```
    case EAknSoftkeyExit:
#endif
  Exit();
  break;

    case ESimpleExCommand:
      {
      _LIT(message,"Start Selected!");
      iEikonEnv->AlertWin(message);
      break;
      }
    }
  }
```

SimpleEx_View.cpp

This file contains the application view class. The `ConstructL()` method of the view class is called by the UI framework after the view class is instantiated, and it's this method that creates the main application window and activates it for display.

`Draw()` is a method called by the framework for every control in order to draw it to the screen. The application view is a control, and, for this example, I implement the `Draw()` function to output the text 'Simple Example' in the center of the window. The drawing is performed by opening a graphics context (GC), getting a font, and calling the context's `DrawText()` function. Cleanup is performed on the font upon completion.

Example 2.12. `SimpleEx_View.cpp`

```
#include "eikenv.h"
#include <coemain.h>

#include "SimpleEx.h"

CSimpleExAppView* CSimpleExAppView::NewL(const TRect& aRect)
  {
  CSimpleExAppView* self = CSimpleExAppView::NewLC(aRect);
  CleanupStack::Pop(self);
  return self;
  }

CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect)
  {
  CSimpleExAppView* self = new (ELeave) CSimpleExAppView;
  CleanupStack::PushL(self);
  self->ConstructL(aRect);
  return self;
  }

void CSimpleExAppView::ConstructL(const TRect& aRect)
  {
  CreateWindowL();
```

```
  SetRect(aRect);
  ActivateL();
  }


void CSimpleExAppView::Draw(const TRect& ) const
  {
  CWindowGc& gc = SystemGc();
  const CFont*    font;
  TRect drawRect = Rect();

  gc.Clear();

  font = iEikonEnv->TitleFont();
  gc.UseFont(font);
  TInt   baselineOffset=(drawRect.Height() -
          font->HeightInPixels())/2;
  gc.DrawText(_L("Simple  Example"),drawRect,baselineOffset,
          CGraphicsContext::ECenter, 0);

  gc.DiscardFont();
  }
```

At this point you should have all the source files in the `src` directory.

2.3.6 Project Build Files

Now, let's create the project build files: `SimpleEx.mmp` and `bld.inf`.

Creating the `SimpleEx.mmp` Project Definition File

This section shows the project definition files for the example for all three software platforms. Use the file corresponding to your platform. Name the file `SimpleEx.mmp` and place it in the `group` directory.

Example 2.13 is the Series 60 project file.

Example 2.13. Series 60 project file

```
TARGET      SimpleEx.app
TARGETTYPE  app
UID         0x100039CE 0x10005B94

TARGETPATH  \system\apps\SimpleEx

SOURCEPATH  ..\src
SOURCE      SimpleEx.cpp
SOURCE      SimpleEx_app.cpp
SOURCE      SimpleEx_view.cpp
SOURCE      SimpleEx_ui.cpp
SOURCE      SimpleEx_doc.cpp

SOURCEPATH  ..\group
RESOURCE    SimpleEx.rss
```

```
SYSTEMINCLUDE       \epoc32\include
USERINCLUDE         ..\include

LIBRARY     euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY     avkon.lib
```

Example 2.14 is the UIQ project file.

Example 2.14. UIQ project file

```
TARGET      SimpleEx.app
TARGETTYPE  app
UID 0x100039CE 0x10005B94

TARGETPATH  \system\apps\SimpleEx

SOURCEPATH  ..\src
SOURCE      SimpleEx.cpp
SOURCE      SimpleEx_app.cpp
SOURCE      SimpleEx_view.cpp
SOURCE      SimpleEx_ui.cpp
SOURCE      SimpleEx_doc.cpp

SOURCEPATH  ..\group
RESOURCE    SimpleEx.rss

SYSTEMINCLUDE       \epoc32\include
USERINCLUDE         ..\include

LIBRARY     euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY     qikctl.lib
```

Example 2.15 is the Series 80 project file.

Example 2.15. Series 80 project file

```
TARGET      SimpleEx.app
TARGETTYPE  app
UID 0x100039CE 0x10005B94

TARGETPATH  \system\apps\SimpleEx

SOURCEPATH  ..\src
SOURCE      SimpleEx.cpp
SOURCE      SimpleEx_app.cpp
SOURCE      SimpleEx_view.cpp
SOURCE      SimpleEx_ui.cpp
SOURCE      SimpleEx_doc.cpp

SOURCEPATH  ..\group
RESOURCE    SimpleEx.rss

SYSTEMINCLUDE       \epoc32\include
```

```
USERINCLUDE         ..\include

LIBRARY     euser.lib apparc.lib cone.lib eikcore.lib
```

The project definition file is a Symbian OS specific text file that ends in `.mmp` and defines how to build the application. Symbian OS does not use a makefile because:

- The system supports many types of build tools (including make programs) and so needs a generic solution with platform independence.
- A custom build system can define information specific to Symbian such as the program UIDs.
- IDE tools such as Metrowerks and Borland have the ability to import `mmp` files to create Symbian application projects.

As you will see later, Symbian build tools generate makefiles based on this `mmp` file.

The Symbian build system will be discussed in Chapter 5, but for this example the main thing to note is that it defines what source and resource files should be compiled and what libraries should be used for linking. Include and source paths are also defined.

`TARGET_TYPE` should always be `APP` and the first number under `UID` should always be `0x100039CE` for GUI applications. The second number under `UID` identifies the application specifically and should always match the one returned by `AppDllUid` in the application object (in file `SimpleEx_app.cpp` in this example).

The only difference between the platforms in the `mmp` file is the GUI specific library for that platform. Series 60 uses `avkon.lib` and UIQ uses `qikctl.lib`. In contrast, Series 80 does not use a unique GUI library since it uses the core Uikon classes directly.

Creating the `bld.inf` File

Example 2.16 shows a file called `bld.inf` which points to the `mmp` file. This is required for the build tools to use the example's `mmp` file. Type this into a file and name it `bld.inf` in the Group directory.

Example 2.16. `bld.inf`

```
PRJ_MMPFILES

SimpleEx.mmp
```

2.4 Building and Executing on the Emulator

To build the application with the Visual Studio command-line tools, change directory to the group subdirectory and execute these commands:

```
c:\...\group\>bldmake bldfiles
c:\...\group\>abld build wins
```

Figure 2.6 Series 60 Example

Figure 2.7 UIQ Example

Figure 2.8 Series 80 Example

Use `winsb` or `winscw` instead of `wins` if you are using Borland or Metrowerks development tools.

Run the emulator on the PC (for example, by typing `epoc` at the command prompt) and select the application labeled `SimpleEx` from the desktop (or Other or Extras folder depending on the platform).

Figures 2.6, 2.7, and 2.8 show how the example looks on the different software platforms.

2.5 Building for the Smartphone

To build for a smartphone target, you'll need to specify a different build target; ARM (`armi`) in release (`urel`) mode:

```
c:\...\group\>abld build armi urel
```

This will build an ARM executable suitable for running on your phone.

After you've built the binary, you need to create an installation file (`sis` file) for installing and running the application on the phone. Symbian OS provides a utility called `makesis` to create the `sis` file. You'll need to create a package definition file (`pkg` file); this defines what goes into the `sis` file. The package file specifies various attributes of the installation file and includes a list of files that belong in the installation. This file list includes where each program file is found on the host PC (so `makesis` can locate them to copy them to the `sis` file) and where each of these files should be placed on the phone when the `sis` is installed.

Let's look at the package definition files (`pkg` files) for Series 60, UIQ, and Series 80. Type in the one corresponding to your platform, save it as `SimpleEx.pkg`, and save it in the `group` directory. The text in bold should correspond to your `EPOCROOT`.

Example 2.17 shows the Series 60 package file. The path in bold represents the default location of the Series 60 v1.2 SDK and should be changed for Series 60 v2.0 or if you installed your SDK in a non-default place.

Example 2.17. Series 60 Package File

```
; SimpleEx.pkg  -  Series 60
;

; standard SIS file header
#{"SimpleEx"},(0x10005B94),1,0,0

;Supports Series 60 (all versions)
(0x101F6F88), 0, 0, 0, {"Series60ProductID"}

;
"c:\Symbian\6.1\Series60\epoc32\release\armi\urel\SimpleEx.APP"-
        "!:\system\apps\SimpleEx\SimpleEx.app"
"c:\Symbian\6.1\Series60\Epoc32\Release\armi\urel\SimpleEx.rsc"-
        "!:\system\apps\SimpleEx\SimpleEx.rsc"
```

Example 2.18 shows the UIQ package file.

Example 2.18. UIQ Package File

```
; SimpleEx.pkg -- UIQ
;
; standard SIS file header
#{"SimpleEx"},(0x10005B94),1,0,0

;Supports UIQ version 2.0 and 2.1
(0x101F617B), 0, 0, 0, {"UIQ20ProductID"}

;
"c:\symbian\uiq_21\epoc32\release\armi\urel\SimpleEx.APP"-
        "!:\system\apps\SimpleEx\SimpleEx.app"
"c:\Symbian\uiq_21\epoc32\release\armi\urel\SimpleEx.rsc"-
        "!:\system\apps\SimpleEx\SimpleEx.rsc"
```

Example 2.19 shows the Series 80 package file for the 9200 Series SDK. For Series 80 SDK v2.0 (for Nokia 9500/9300), the portion of the directories shown in bold will be different (e.g. `c:\Symbian\7.0s\Series80_DP2_0_SDK`) depending on where you installed the SDK.

Example 2.19. Series 80 Package file

```
; SimpleEx.pkg -  Series 80
;

; standard SIS file header
#{"SimpleEx"},(0x10005B94),1,0,0

;
"c:\symbian\6.0\nokiacpp\epoc32\release\armi\urel\SimpleEx.APP"-
        "!:\system\apps\SimpleEx\SimpleEx.app"
"c:\symbian\6.0\nokiacpp\epoc32\release\armi\urel\SimpleEx.rsc"-
        "!:\system\apps\SimpleEx\SimpleEx.rsc"
```

The last few lines of the package file specify the individual files to copy to the phone – the first filename specifying where to find the file on the PC and the second where to place it on the phone.

Next, run the following command at the command prompt:

```
c:\...\group\>makesis SimpleEx.pkg
```

Once `SimpleEx.sis` is created, install this to your smartphone through your PC suite (or any method your smartphone has of installing an application `sis` file). The `sis` file is the standard file for installing Symbian applications and installing the application using this file should be well documented in the smartphone's user manual.

Because this application is unsigned, you will get some messages indicating that the application is from an unknown source – ignore these.

Once you load the program on the smartphone you can run it as you did from the emulator. Congratulations on your first Symbian OS program!

3

Symbian OS Architecture

This chapter gives an overview of the architecture of Symbian OS – its main components and its underlying functionality.

Much of the functionality described in this chapter is transparent from a typical Symbian OS application programming view; however, understanding the architectural details of an operating system can be useful in developing software for it – especially for programming on highly reliable, limited-resource devices such as smartphones.

You can skim this chapter on a first read if you want, since an understanding of many of the subjects here is not absolutely essential to developing applications. But if you are like me and prefer to dig into the details in order to gain deeper knowledge, then you should find this chapter useful.

3.1 Components in Symbian OS

There are usually many ways to slice a system up into pieces – the following breakdown of the major parts of Symbian OS is suitable for the detail covered in this chapter:

- Kernel
  The kernel is the central manager and arbiter of Symbian OS. It manages the system memory and schedules programs for execution. It also allocates shared system resources and handles any functionality that requires privileged access to the CPU. The kernel can be extended via Dynamic Link Libraries (DLLs) and device drivers.

- Base libraries
  The base libraries contain APIs that provide functionality such as string manipulation, linked lists, file I/O, database management, error handling and timers. The base libraries also provide access to kernel functions (e.g. thread control and client server communications). This library is used not only by applications, but also by the OS components.

- Application services, engines and protocols
  Application services, engines and protocols provide access for programs to core application data, features and services. An example is an engine to directly manipulate the data of built-in applications that manage contacts, the calendar and to-do lists. Other examples include setting and handling alarms, and access to high-level communication features such as SyncML and HTTP.
- Application framework
  The application framework implements the base functionality of the smartphone's graphical user interface applications. This includes a framework for handling the GUI itself and an architecture framework for handling non-GUI related application functionality.
- Communications architecture
  The communications architecture consists of the APIs and framework that implement data communications. This includes TCP/IP over cellular radio as well as local communication protocols such as Bluetooth, IR, and USB. Also included is the messaging framework for support that includes SMS, MMS and email messaging.
- Middleware feature libraries
  This is a catch-all category for the rest of the APIs and frameworks not covered in the previous items. It includes APIs such as multimedia, animation, and security.

3.2 Multitasking in Symbian OS

Symbian OS is a multitasking operating system – it can run multiple programs at once. Although a smartphone's screen is too small to display more than one application at a time (as you can in Windows, for example), you can switch between running applications as needed. Also, as with operating systems such as Linux and Windows (though not Palm OS), Symbian OS provides true multithreaded behavior in that it allows multiple execution threads to execute in parallel, even in a single application.

Here I'll briefly introduce *threads* and *processes* in Symbian OS – these form the basis of the multitasking capability in Symbian OS. Chapter 7 covers threads and processes and inter-thread communication in more detail.

3.2.1 Threads

Threads are streams of code that run in parallel with each other, based on their priorities. The Symbian OS kernel supports *pre-emptive* multithreading, which means that not only can threads run in parallel, but the

kernel can switch execution from one thread to another without needing any code in the running thread to explicitly relinquish control. Also, a single Symbian OS program can have multiple threads

> While a single Symbian OS application can have multiple threads, you'll see later that it's best to avoid using them in your program directly, and instead use the asynchronous framework. This framework provides an event-driven cooperative multitasking model for your application, which will be introduced later in this chapter (and detailed in Chapter 8).

3.2.2 Processes

A process is a running instance of a program that has its own independent data space as well as one or more threads. The code for a process is contained in a file whose name ends in `.exe`. The kernel creates and starts a separate process for each invocation of an *EXE* file. Multiple processes can run at a time, and the kernel switches to a process whenever one of the threads in that process becomes active.

A process always contains a main thread, and can contain additional threads if needed. All threads within a process share its data space, and, therefore, can directly access its static data.

For protection, Symbian OS does not allow a process to directly access memory in another process. There are ways of providing shared memory access between processes, however, and these will be discussed in Chapter 7.

3.3 Dynamic Link Libraries

A Dynamic Link Library (DLL) is a library that's loaded into memory when needed and its functions are available to all running programs. Only a single copy of each loaded DLL exists in memory at a time. This is more efficient than the traditional static library, where each executable that uses the library's functions links to a separate copy of its code.

DLLs are used extensively in Symbian OS, and there are well over 100 of them on a typical phone. In fact, prior to Symbian OS v9, GUI applications are actually DLLs themselves, although each GUI application runs as a separate process (each GUI application DLL is run from an instance of `apprun.exe`). From Symbian OS v9 onwards, applications are implemented as fully independent executable processes.

3.3.1 Types of DLL

There are two types of DLL, *static interface* DLLs and *polymorphic* DLLs.

Static interface DLLs are the traditional style of libraries, containing a collection of classes and functions that are made available to calling

programs. The base operating system libraries, which provide functions for things such as string manipulations, are examples of static interface DLLs. Static interface DLLs typically end in `.dll`.

Polymorphic DLLs are used as plug-ins (e.g. a device driver is actually a polymorphic DLL), as opposed to simply providing classes and functions as in static interface DLLs. They provide a concrete implementation for some abstracted interface. The concept of this type of DLL is more difficult, and will be discussed in more detail later in this chapter.

3.3.2 Static Data in DLLs

DLLs written using Symbian OS versions earlier than v9.0 cannot contain writable static data. This means you cannot declare global nonconstant variables externally in functions (automatic variables are fine since they are on the stack), or have nonconstant static variables declared within a function or class. This can be limiting, but Symbian made this choice to conserve memory due to the large number of DLLs that can be loaded at a time.

This example will not work in a DLL:

```
int myGlbVar;
void myfunc
  {
  }
```

Nor will the following:

```
void myFunc
  {
  static TInt myVar;
  ...
  }
```

However, constant data is permissible – this will work fine:

```
const int myData=4;
void myFunc()
  {
  int myVar;
  ...
  }
```

When a thread invokes a function within a DLL, that function runs within the context of the calling thread (and the corresponding process) and thus is able to directly access the data space of the calling process.

From Symbian OS v9.0 onwards, this restriction is removed, and DLLs may contain writable static data. While this can greatly simplify the task of importing code from other operating systems, the practice of using writable static data can have a significant impact on memory usage, and is strongly discouraged.

3.4 Client/Server Model

Symbian OS software relies on a client/server architecture to implement much of its functionality. A *server* in Symbian OS has no user interface and acts mainly as an engine to manage some resource or functionality on behalf of some other program, known as a *client*. Servers receive commands from one or more clients and execute these commands, one by one. A server always resides in a separate thread from its clients and in many cases is also contained in its own process.

An example of a server in Symbian OS is the file system server. This server runs as a process (it's an *EXE* file) and receives and executes commands to manage the creation, reading and writing of files on the device's memory drives. The API classes that applications use to manage files (e.g. `RFile`) are actually client-side classes that send commands to the file server (transparently to the API user) which then executes the functionality.

The basic execution flow of a server is:

1. Wait for a command to be received from the client (data may also be sent with this command).
2. When the command is received, execute it.
3. Return the status (and any data) to the client.
4. Go to step 1.

Not only are many applications written using this model, but much of the OS itself is implemented using it. In most cases the details of the communication between the client and the server are hidden in user API calls.

Figure 3.1 shows multiple clients communicating with servers. As mentioned, servers are always in separate threads from their clients (although multiple servers can exist in a single thread). Data is transferred between the client and the server using inter-thread communication functionality within Symbian OS.

Symbian OS provides a client/server framework that handles the details of the communications between the client and server. Chapter 9 describes this framework in detail, and shows how to use it to write your own server and the client-side class that interfaces with it.

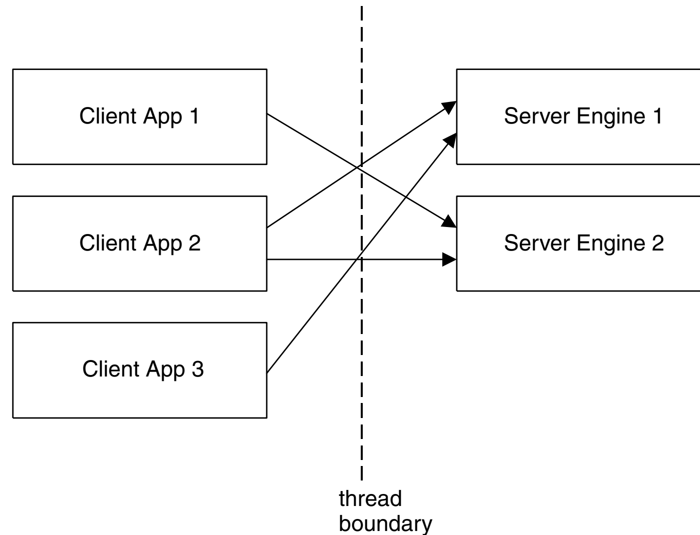

Figure 3.1 Client/Server Interaction

3.5 Memory in Symbian OS

Let's look at the different types of memory that exist on a smartphone based on Symbian OS:

- Random Access Memory (RAM)

  RAM is the volatile execution and data memory used by running programs. Applications vary in how much RAM they use, and this also depends on what the application is doing at the time. For example, a browser application loading a web page needs to allocate more RAM for the web page data as it's loaded. Also, the more RAM space you have, the more programs you can run on your smartphone at once. Typically, mobile phones have between 7 and 30 MB of RAM available for applications to use.

- Read Only Memory (ROM)

  The ROM is where the Symbian OS software itself resides. It includes all the startup code to boot the device, as well as all device drivers and other hardware-specific code. This area cannot be written to by a user, although some of it can be seen by the file system as drive *z*:. Viewing the *z*: drive will show all the built-in applications of the OS, as well as the system DLLs, device drivers and system configuration files. For added efficiency, code in ROM is executed in place – i.e. it is not loaded into RAM before executing.

  Typically a phone has between 16 and 32 MB of ROM.

- Internal Flash Disk

  The internal flash acts like a disk drive and allows for reading and writing files to it via the Symbian OS file system. The file system is fully featured and supports a hierarchical directory structure, with very similar features to those you would find on high-end operating systems. The internal flash drive is represented as the *c*: drive to the file system. This memory contains user-loaded applications, as well as data such as documents, pictures, video, bookmarks, calendar entries, etc.

  The size of the internal flash disk varies with the phone, but it can be quite generous. For example, the Nokia 9500 has 80 MB of internal flash space available to the user. On many phones, however, available internal user space is significantly less. The Nokia 6600, for example, has 6 MB of flash space available to the user.

- Removable memory cards

  Memory cards act as removable disk drives and allow you to expand the storage provided internally. You can also read from and write to a memory card just as to the internal disk – including operations such as saving user data, and even installing applications. This card is treated as another disk volume by the file system and is represented by a drive letter such as *d*: or *e*: (this varies between phones).

  The memory card formats (MMC and SD are examples) and available sizes vary by phone. Memory card sizes can vary from 16 MB (or even less) to 1 GB.

In this section, I'll describe how Symbian OS organizes its memory map and how processes use it. This information is not strictly needed to develop typical Symbian OS applications since this functionality occurs behind the scenes. However, it can help when dealing with some difficult issues (or, perhaps, for tweaking performance) or even for providing a deeper understanding of some of the system APIs.

3.5.1 How Memory Is Addressed

At the time of writing, Symbian OS smartphones exclusively use ARM microprocessors. ARMs use 32-bit memory addresses and thus are capable of addressing 4 GB of memory – much more memory than a smartphone will ever need (well, maybe not ever).

There are two types of memory addresses: *virtual* and *physical*. Virtual addresses are the addresses that software deals with. When you set or examine a C or C++ pointer, or even look at an address at the assembly level, you are dealing with a virtual address. Physical addresses are the unchanging hardware addresses of the memory hardware (they

reflect how the hardware address lines are hooked to the memory components).

In older CPUs, there was no concept of virtual and physical addresses since the software simply always used the memory's fixed physical address. However, most modern processors, including the ARM processor, have a *Memory Management Unit* (MMU) that allows the system software to flexibly map and remap virtual addresses to represent different physical addresses. With the MMU, the operating system can organize and manage its own memory map by setting up a memory translation table in the MMU. Memory blocks can be moved almost instantaneously without any copying, only remapping, of addresses (e.g. when switching process data in and out).

In addition to providing address translation, the MMU also provides the capability of protecting memory regions from being accessed by software not running at a specified privilege level or higher.

3.5.2 Chunks in Symbian OS

Symbian OS uses *chunks* to represent contiguous regions of virtual memory. The size of a chunk is variable. The kernel uses the MMU to map physical memory to the virtual address range of the chunk, and to remap it quickly to different areas of virtual memory as needed (mainly for context switches, as you'll soon see).

While chunks reserve a range of virtual memory addresses, the entire range need not have actual physical memory behind it. The kernel can add more physical memory behind the chunk as needed. Remember: virtual addresses are plentiful (4 GB!), real physical memory is much more scarce.

Symbian OS provides a public API so that you can use chunks directly (*RChunk* class, described in Chapter 7). It's not very common for the typical application to use them (although they can come in handy if you need system-wide global data). Symbian OS itself, however, makes extensive use of chunks to manage your programs and its data behind the scenes, as described in the next section.

3.5.3 A Process in Memory

When a process is created, Symbian OS creates the following chunks for it (at a minimum):

Stack and Heap Chunk

This chunk is where the stack and heap resides for the main thread of the process (it's possible that additional threads in the process can have their own stack and heap, and thus separate chunks).

Static data chunk

Where all the static variables are kept for the process.

Code Chunk

The code chunk contains a copy of the code. There is only one copy of a code chunk in memory, shared by all running instances of that process executable. Note that if the executable is on the phone's Read Only Memory (ROM), then the code is run in place, without copying it to a code chunk.

3.5.4 Virtual Memory Map in Symbian OS

The basic map of virtual memory usage in Symbian OS is shown in Figure 3.2.

In Figure 3.2, the two memory areas of interest are the *home area* and the *run area*. The home area is where the data chunks for a process are kept when the process has been launched but is not the currently active process. The home area is a protected region of memory – only kernel-level code can read and write it. When a process is scheduled to execute, its data chunks are moved (remapped via the MMU) from the home area of virtual memory to the area known as the run area, and process execution starts.

You can think of the run area as a sort of stage for processes, and the home area as the backstage area. When a process is ready to perform, it goes on stage – i.e. its data chunks are moved to the run area, and the process executes.

Why aren't the process data chunks simply left in the home area when the process executes? The reason is that the process code always expects its data to reside in the same place. This is because the linker places data

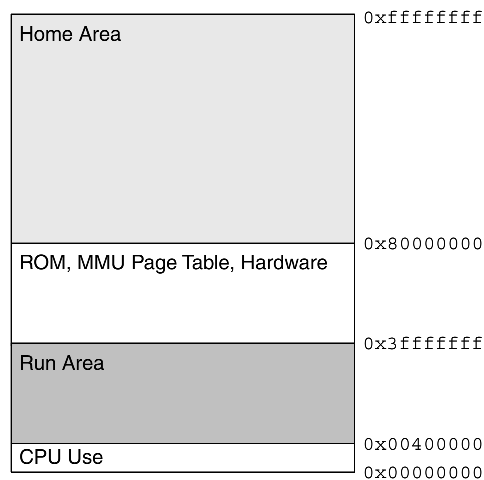

Figure 3.2 Virtual Memory Map

address references (when code references a static variable, for example) in the code image at build time. Thus, the process code expects its data in the single place specified by the linker (i.e. the run area) – no matter what instance of the program is running.

Note, however, that code chunks are never moved to the run area. This is because, unlike data chunks, you do not have separate copies of code for each process instance and the code can be run from its location in the home area.

Re-mapping the data of a running process to a common virtual address area is not unique to Symbian OS. Many other multi-tasking operating systems do this as well – although the memory map and switching details are different.

3.5.5 Switching Processes – Detailed Example

Figure 3.3 illustrates process switching, as described in the last section, in more detail.

Program A and Program B represent executables contained in `exe` files. The '_xx' represents a process instance of that executable.

As mentioned, every code image assumes its data is in the run area, and the kernel handles moving the data into the run area when the code is run.

Below is a sample scenario:

1. Program A is invoked for the first time and the kernel loads the code from flash disk to RAM and creates a process (programA_01). The kernel then allocates data chunks for that process in the home area.
2. Another instance of Program A is invoked and the kernel creates a new process (programA_02). It associates it with the same code area from Step 1 and creates new data chunks for that process in the home area.
3. The kernel schedules programA_01 for execution.
4. The MMU page table is changed to remap all physical memory pages associated with programA_01's data from its home area location to the common run area.
5. The code image associated with Program A executes.
6. The kernel switches context from programA_01 to programA_02.
7. The MMU page table is changed to remap the data chunks of programA_01 from the run area to its original home area location.
8. The data chunks of programA_02 are then mapped to the run area and control is passed back to the Program A code region, but at the appropriate instruction, with respect to the thread context.

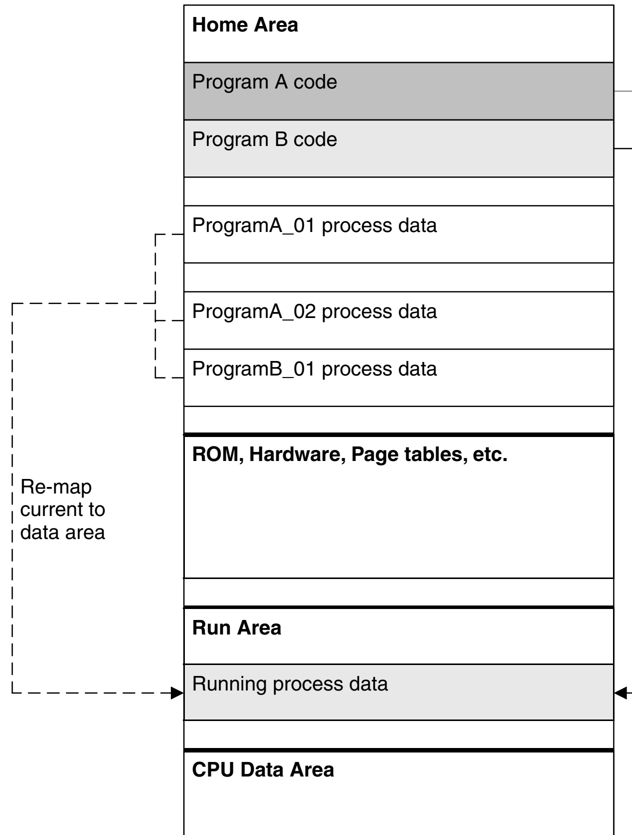

Figure 3.3 Structure of the virtual machine

3.5.6 Protecting Processes from Each Other

Another feature of memory handling in Symbian OS is protection. Only the run area can be accessed by user mode programs; the rest of memory can only be accessed in the CPU's privileged mode (and therefore only by the kernel, device drivers, and other selected OEM components). This means that a user process does not have direct access to the data of other processes. Also, a user process cannot access hardware devices or CPU data structures such as the MMU page table.

The run area is sometimes referred to as a *sandbox* since it provides an isolated world for the process to run in.

3.5.7 Performance in Switching Processes

Although using the MMU to relocate data in the virtual memory map is much faster than copying the actual data, there is still a performance penalty. If the cache is virtually tagged (i.e. the cache is indexed by virtual

addresses instead of physical addresses – the most likely case on Symbian OS devices), then a cache flush must be performed when the MMU page tables are changed, thus causing a significant performance hit. That is why switching between threads in the same process is considered fast, since no memory areas need to be remapped. However, process switches are more expensive due to the remap.

3.5.8 Fixed Processes

Some OS-level processes are switched to so often that the performance impact of remapping their process data areas (via the MMU) between the home and run areas is not acceptable. Examples are the kernel server itself and the file system process. To get around this, Symbian OS has the concept of a *fixed process*, where the process data stays in the home area, even when executing.

Fixed processes are faster to switch to since the MMU tables are not modified. However, the cost of doing this is that there can be only one instance of the process running at a time and that, since the code image points to the data directly, the data location must be reserved and fixed in the system.

Only Symbian OS components (or OEM-specific customizations) can be fixed processes – this capability is not available to the application developer.

3.6 The Kernel

The Symbian OS kernel consists of a set of executables and data files which runs in the CPU's privileged mode and provides basic system management and control. The kernel handles the creation and scheduling of threads and processes. It also manages communication between threads and processes with objects such as mutexes and semaphores, as well as with functions for inter-process data transfers. In addition, the kernel manages all the system memory, and acts as a gateway that provides access to device hardware.

Figure 3.4 shows the kernel architecture. The kernel, hardware abstraction layer (HAL), device drivers, and kernel extensions run in privileged mode and therefore have access to all memory and hardware resources.

3.6.1 Abstracting the Hardware

Kernel functionality depends on the underlying hardware on a device. For example, timers are needed for task scheduling and timer services. Another example is the control of the MMU and flushing the cache. The methods of controlling these features vary with the hardware and CPU.

The kernel is divided in such a way that the bulk of the kernel code is abstracted from the hardware (i.e. it is written so that the detailed

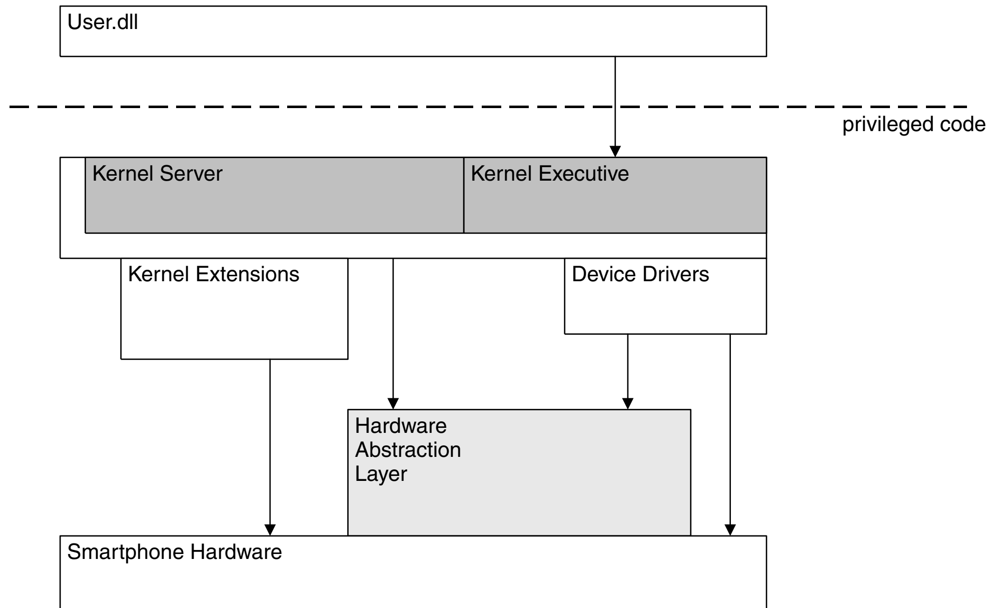

Figure 3.4 Kernel

specifics of the hardware do not matter). This is done through the *hardware abstraction layer* (HAL).

The HAL provides a generic hardware-independent API to the kernel for accessing the required hardware features. The HAL is customized to the OEM hardware and is used to make hardware implementation details transparent to the kernel.

While the HAL is used for adapting the kernel to the underlying hardware, device drivers provide a more flexible abstraction of hardware devices. User-mode software – including applications – can load and use device drivers. Device drivers primarily control specific hardware peripherals such as communications ports, radio modems, or external storage devices. The kernel implements the functions for user programs to load, and communicate with, device drivers.

Kernel extensions are hardware-specific modules written by the OEM. They are more tightly integrated with the kernel than device drivers. Kernel extensions are implemented as DLLs and are detected and initialized at boot time. They are primarily used for user input hardware, such as buttons and keyboards.

3.6.2 User Library

The user library is a DLL that provides user-mode access to kernel functionality. Since the kernel runs in privileged mode, the functions in

`user.dll` will switch the processor from user mode to kernel mode, then invoke the appropriate kernel function. Upon return, the mode is switched back to user mode.

The user library actually contains more than just kernel functions – it also contains basic functions to do things such as string and array manipulation.

3.6.3 Kernel Executive and Server

The kernel consists of two main components: an *executive* and a *server*.

The executive consists of a set of software interrupt handlers, which, when invoked, will switch the CPU from user mode to privileged mode (if it is not in privileged mode already). When a kernel function is called from an application (via `user.dll`), a software interrupt (the SWI instruction on ARM) occurs that invokes the appropriate function handler in the kernel executive. The kernel server is the kernel's own independent process that has its own data space.

There are two types of kernel function which differ by the component of the kernel that the function is executed in. Kernel executive functions are executed completely from within the software interrupt handler of the kernel executive component, without invoking the kernel server. They execute quickly since the kernel function runs in the same context as the calling application, without a context switch. There are two types of executive kernel calls: fast and slow. Fast calls disable interrupts, and thus context switches, while the kernel call is being executed. By contrast, slow calls allow interrupts to occur.

Kernel server functions start in the kernel executive, but the kernel executive software interrupt handler does not actually execute the kernel function. Instead, the interrupt handler sends one or more commands to the kernel server for it to execute. Kernel server function calls are more expensive than kernel executive calls, since the OS must switch from the application process to the kernel server process to execute. The advantage, however, is that these functions have full access to the data space of the kernel process which stores various global kernel data including the list of all the system's running processes and the list of all chunks and semaphores. Kernel executive functions, on the other hand, have no way to access these structures.

The kernel server is a *fixed process*, so it minimizes some of the overhead normally associated with server calls.

Since applications use `user.dll` to call kernel services, it is transparent to the programmer how the actual kernel code is invoked. However, it is useful to know how the call is handled when considering application performance.

3.7 Active Objects and Asynchronous Functions

Although Symbian OS has support for multiple threads within a process, this capability is not much used. In fact it is discouraged. One reason is performance – lots of threads can bog down a system due to context switches. Other reasons result from the way Symbian OS is designed. For example, when a system object is created it can usually be accessed only by the thread that created it. The correct way to handle threaded behavior – without actually having a separate thread – is through *active objects*.

To understand active objects, consider how threads are normally used in an application. Usually, code is put in a separate thread when it needs to wait for an event and then process the event when it occurs – perhaps even in a continuous fashion. It is put in its own thread so that the whole program does not block waiting for that event, and other productive things can occur during the wait.

Active objects simulate multiple threads in a single process, but in fact are executed in a single thread. The thread consists of what is known as an *active scheduler*, which contains two main elements: an event dispatcher and a list of attached active objects. The active scheduler loop waits for an event (at a semaphore) and then invokes the event handler of the active object that is expecting the event. The active scheduler then waits for the next event.

The events received by the active scheduler are generated from *asynchronous functions* which, unlike traditional functions, return immediately after being called and run in parallel with the calling thread, sending the event when complete. You invoke an asynchronous function from a method of an active object (a C++ class derived from CActive); when the active scheduler receives the asynchronous function's completion event, it invokes the active object's command handler. Your single program thread can have multiple asynchronous functions in progress at one time. So although, obviously, the asynchronous functions themselves execute in their own thread, the combination of asynchronous functions and active objects allows you to process multiple operations in parallel without actually implementing threads in your program. But how do you handle the situation when a function you call blocks from within an active object? Won't it still block the entire program?

Yes, if a function blocks within an active object, the entire program (and its other active objects) stops, which is clearly not desirable. Asynchronous functions execute without blocking the program and you should use them. Most of the Symbian OS APIs that involve any sort of waiting are implemented as asynchronous functions.

It's very important to have a solid understanding of active objects and asynchronous functions when programming in Symbian OS, since they are used extensively. I provide more information on them in Chapter 8 and show you how to develop and use them.

3.8 What Is a Polymorphic DLL?

Before discussing the GUI and communications architecture, now is a good time to introduce *polymorphic DLLs* since they are used heavily in these architectures (as well as in many other areas of the operating system). Polymorphic DLLs, in a similar fashion to static interface DLLs, are loaded when needed and linked to at runtime. Unlike a static DLL though, a polymorphic DLL acts as a plug-in that implements an abstract interface. In other words it implements a concrete C++ class that's inherited from a known abstract base class. The first (and usually only) exported function of the polymorphic DLL just instantiates its concrete class and returns a pointer to it.

With polymorphic DLLs, you can implement custom plug-ins that present consistent interfaces to applications. Polymorphic DLLs are used heavily by Symbian OS for this purpose. Polymorphic DLLs normally do not end in `.dll`, but in an extension that is more descriptive of the type of plug-in they are implementing (e.g. `.prn`, `.prt` and `.app`).

Polymorphism is built into C++ – and is one of its best features. To implement polymorphism in C++, you have a base class with one or more methods declared as *virtual* (and in many cases it's pure virtual, meaning there is no code for it in the base class at all). This base class then acts as an abstract interface to classes derived from the base class.

The methods themselves are implemented in concrete classes derived from the base class. However, to implement polymorphic behavior, you access the objects through a base class pointer, relying on C++ to automatically call the overridden method in whatever concrete class is assigned to that base pointer. In this way you have common code that behaves the same, no matter which concrete class is assigned to that base pointer.

Consider the following example of an abstract printer base class named `GenericPrinter`:

```
class GenericPrinter
  {
  ...
  virtual TInt PrintDocument(TDesC& aDocName)=0;
  }
```

Now consider two different class implementations of concrete classes derived from `GenericPrinter`.

```
class PrinterX : public GenericPrinter
  {
  ...
  TInt PrintDocument(TDesC& aDocName);
  };
  TInt PrinterX::PrintDocument(TDesC& aDocName)
  {
  // specific code to format and print for printerX
```

```
...
};
TInt PrinterX::PrintDocument(TDesC& aDocName)
{

// PrinterX specific way of printing a document
}
class PrinterY : public GenericPrinter
{
...
TInt PrintDocument(TDesC& aDocName);
};
TInt PrinterY::PrintDocument(TDesC& aDocName)
{
// specific code to format and print for printerY
...
};

TInt PrinterY::PrintDocument(TDesC& aDocName)
{

// PrinterY specific way of printing a document
}
```

Then, with `printer` declared as a pointer to `GenericPrinter`, you can call:

```
printer->PrintDocument(MyDocument);
```

This line calls `PrintDocument()` of whatever concrete class is assigned to base class pointer `printer`. For example, if you assigned the base pointer via `printer=new PrinterX;`, then `PrinterX::PrintDocument()` would be called. The only requirement is that the methods called using the base class pointer must be declared as *virtual*.

Polymorphic DLLs use this exact concept, except now the concrete classes are contained in their own separate DLL which can be thought of as a plug-in to a base class interface. In our example, classes `PrinterX` and `PrinterY` would each reside in separate polymorphic DLLs.

As previously mentioned, a polymorphic DLL also always contains a function (it must be the first exported function) that instantiates the DLL's class and returns a pointer to it. This is how you assign your base class pointer so you can use the plug-in.

Also, the polymorphic DLL is given a type and this type identifies the base class (and thus, the interface) that the DLL's contained concrete class is derived from.

To give you an idea of how this works, here is some code to load a polymorphic DLL and call a method:

```
RLibrary library;
library.Load(_L("printerX.apr")); // load our plugin
TLibraryFunction entry=library.Lookup(1)
```

```
GenericPrinter *printer = (GenericPrinter *) entry();

// ...
// common abstracted code:
printer->PrintDocument(MyDoc);
// ...
```

Don't worry if you do not understand the details in this code (and it's not really complete: it does not verify the DLL type or have error checking). The intent is not to teach you how to write a polymorphic DLL, just to show the general concept.

This code snippet uses the class, `RLibrary`, to load the polymorphic DLL containing the `PrinterX` concrete class (I just used extension `.apr`, standing for ''**a pr**inter''). The code then calls the first exported function in the DLL (`Lookup(1)` returns a pointer to this first function) – this will instantiate the `PrinterX` class and return a pointer to it. This pointer is then assigned to the printer base class pointer for controlling the printer through the abstracted base class interface.

3.9 GUI Architecture

Symbian OS has a powerful and flexible GUI architecture. This section presents an overview of its features.

3.9.1 Customizing the UI

In Chapter 1, I discussed the importance of product differentiation in marketing smartphone devices, and how Symbian addresses this by providing a flexible user interface architecture. I also introduced the major UI customizations including those found in UIQ, Series 60 and Series 80 software platforms.

To understand the architecture rationale further, consider the two extremes an OEM can choose in selecting software for their smartphone. At one extreme, the OEM can create its own OS for maximum differentiation. However, this results in extensive up-front development costs and having little to no third party application support. At the other extreme, the OEM can choose an OS with a complete, built-in GUI. The advantages of this include the low development cost in implementing the phone, and having more applications that run on it. The disadvantage is that there is little product differentiation since it will be very similar to other phones using that OS.

The GUI architecture in Symbian OS is a good balance between these extremes, and is well suited for the smartphone market. This is accomplished by having a powerful, common GUI core that is customized for a specific smartphone product (or product series) via vendor GUI

layers. Although some customization work is required by the OEM, they get a full-featured OS (they do not have to implement their own) and have flexibility in differentiating their product.

While it is true that Symbian applications written for one UI customization will not directly run on another, it is fairly easy to port between UIs – for example UIQ to Series 60 – since the main complexity is in the common UI framework code.

3.9.2 Introducing the GUI Framework

The main components of the GUI architecture and their relation to each other are shown in Figure 3.5.

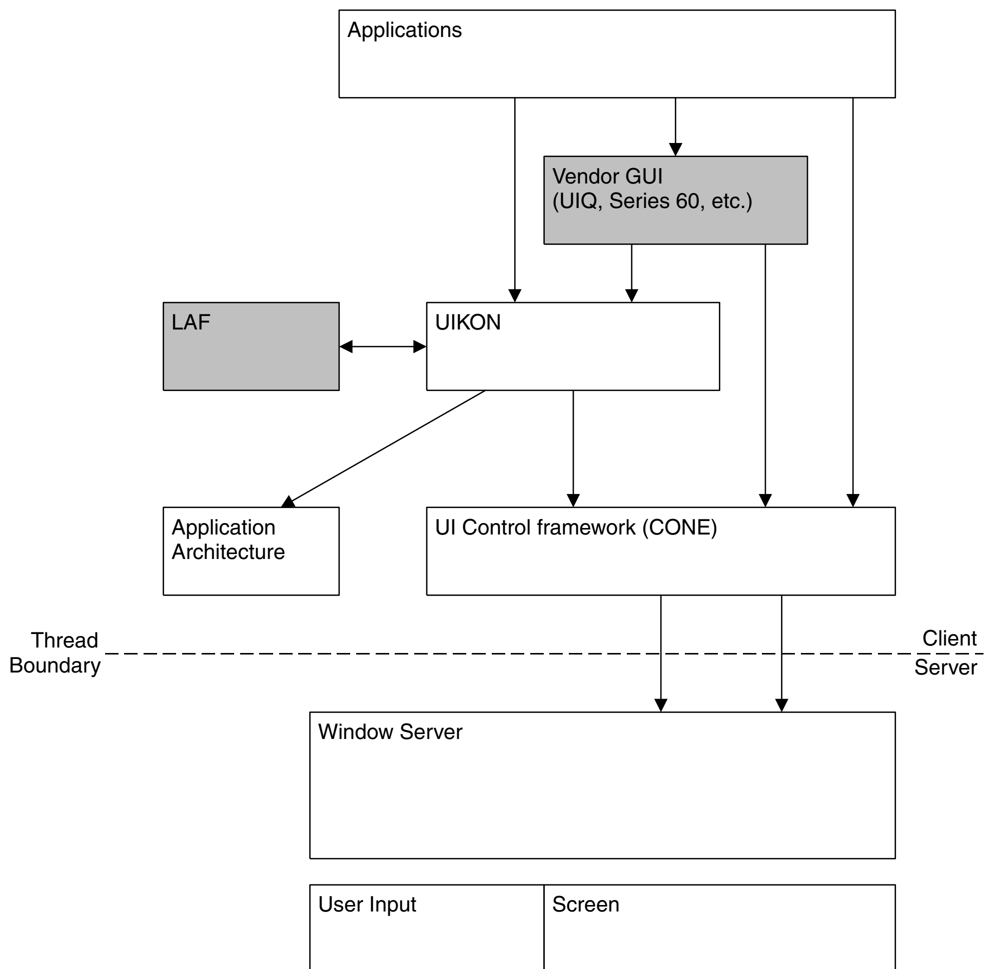

Figure 3.5 GUI architecture

Near the bottom of the diagram, we see the *window server* which provides centralized access to the screen and user input devices across all applications. As the name implies, the window server is a server process – applications (using libraries) act as clients to this server. The window server handles the details of drawing window and control objects to the screen, as well as keeping track of which windows belong to which applications. The window server will also ensure that events such as key presses, pointer events and redraw events are routed to the correct application for handling.

The window server does not enforce any particular UI policy since its commands are low level – the GUI look is handled by the upper GUI layers.

Above the window server is the *UI control framework*, which is sometimes referred to as CONE (control environment). This is a library of C++ abstract classes which communicate directly with the window server via the client/server IPC channel. The UI control framework provides higher-level functionality than the window server and is more suitable for application use. The library contains no concrete controls – upper GUI layers use these base classes to derive their own specific controls. The derived classes need not worry about the details of the client/server communication with the window server since the base classes of the UI control framework handle this.

UIKON is the core Symbian OS application framework library. While the UI control framework contains mainly abstract classes, UIKON provides a set of concrete controls and event handler classes. These classes are derived from UI control framework base classes. UIKON also implements classes derived from the *application architecture* library, which will handle the basic application framework itself and non-display-related application behavior such as managing application documents and handling the command line.

LAF (Look and Feel) is a library that allows the appearance (e.g. size and color) of UIKON controls to be changed by a vendor without actually modifying any UIKON code. The purpose of LAF is to allow minor look and feel modifications to occur without needing to derive new controls.

While having UIKON plus LAF allows customization to a certain extent, a vendor GUI layer also exists for maximum UI flexibility. This vendor layer consists of C++ classes, which derive from UIKON classes as well as directly from the UI control framework. The vendor can therefore supply its own custom controls or extend the functionality of existing UIKON controls. It can also customize application architecture-oriented behavior.

Applications use classes in the vendor GUI layer as well as from UIKON directly to implement the user interface. As mentioned in Chapter 1, a vendor's software platform will have its own SDK with guidelines; applications should follow these guidelines when determining what classes to

call. In addition to using vendor and UIKON classes, applications can create their own custom controls by deriving directly from the UI Control Framework. Also, there is nothing to prevent user programs from directly calling the window server when more screen control is desired.

An *application* is a polymorphic DLL (with a `.app` file suffix) and thus cannot have static data – however, each application runs as an independent process. How is this possible? This is accomplished by invoking a process named `apprun.exe` and having it call the application DLL. So a separate process instance of `apprun.exe` exists for every open application and each application process has its own client session with the window server (through the other GUI layers).

3.10 High Performance Graphics

While the GUI framework presented in the last section is fast enough for standard GUI applications, some applications use graphics more heavily and require quicker screen response. Obviously, games fall into this category, and support for good games is always important for any computing product, including the smartphones.

Symbian OS provides two mechanisms for providing high performance graphics: animation plug-ins and APIs for direct screen access.

Animation plug-ins are polymorphic DLLs which are developed by the application programmer, and plugged directly into the window server. Once the animation is started, the animation plug-in executes in the context of the window server – drawing frames to the desired screen areas at the specified animation rate. This avoids having messages sent from the application to the window server for drawing each animation frame. The client-side animation API provides applications with the capability to install animation plug-ins as well as send commands to them.

In addition to animation plug-ins, Symbian OS provides APIs for drawing on the screen directly. In this manner, low-level drawing primitives (such as drawing a rectangle) can be performed by bypassing the window server and, again, avoiding client/server IPC messages (although some initial ones are required to coordinate with the window server).

3.11 The Communication Architecture

Communications is key to smartphones, and Symbian OS contains an extensive and flexible communication architecture to support it. This section looks at the Symbian OS communications architecture. Chapter 10 will discuss communications in more detail as you learn to develop communication applications. This section (and Chapter 10) assumes some basic knowledge of network communications.

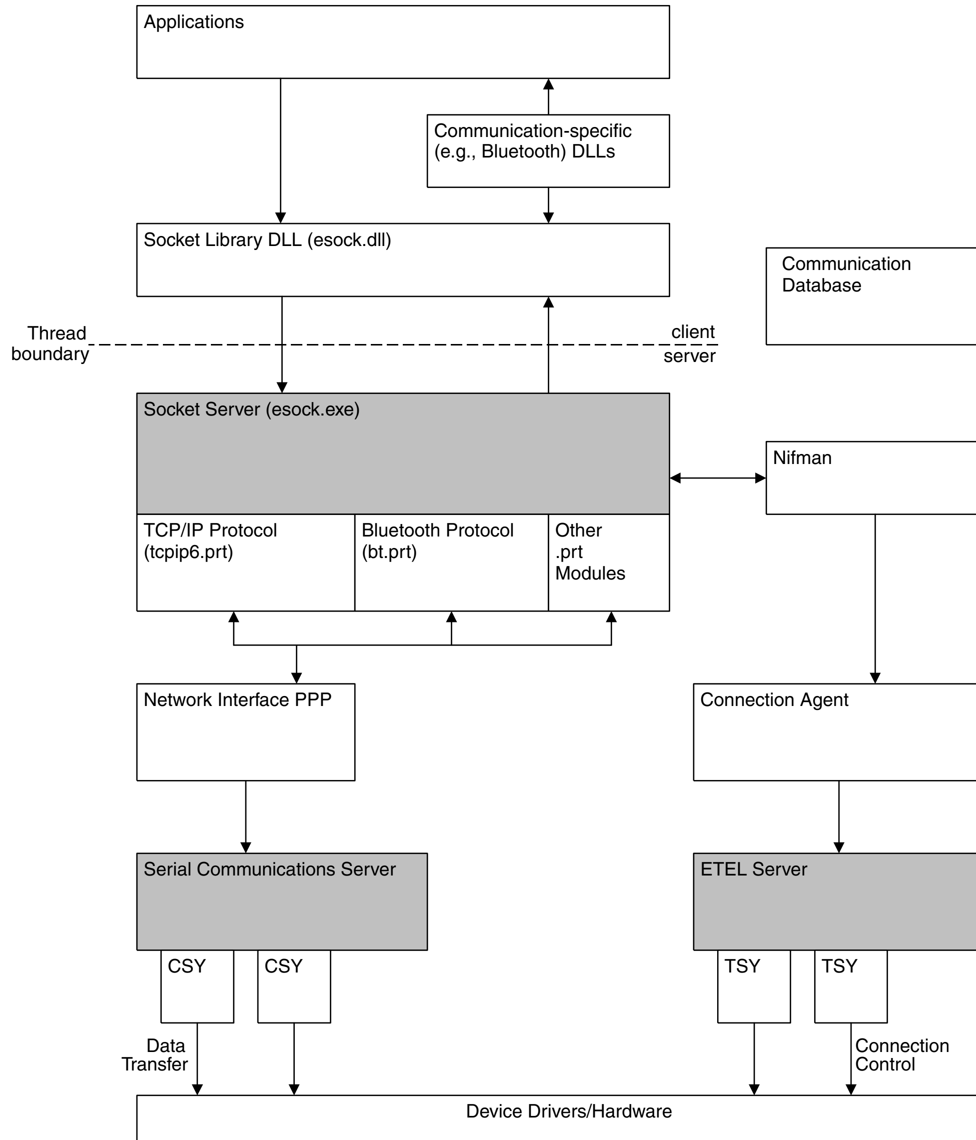

Figure 3.6 Communication architecture

Figure 3.6 shows the main components of the Symbian OS communication structure.

The communication architecture is a good example of how software is constructed in Symbian OS. The architecture consists of application-level DLLs, multiple servers and subsystem DLLs, which link to the servers and server plug-in modules (which are polymorphic DLLs) to support

different protocols and devices. The overall goal is to provide maximum power and flexibility for communications support, while at the same time providing a common interface, not only to the application but throughout the various lower system levels.

The following list describes the components of the Symbian communication architecture:

- Applications and DLLs

  Applications use networking API classes in DLLs to access communications features. As with other application-level DLLs, the communication DLLs hide the details of the underlying architecture. Symbian OS provides a socket-based API that operates in a similar way to the BSD socket API. I will discuss sockets in much more detail (and use them in programs) in Chapter 10.

- Communication-Specific Functionality

  In addition to the socket API, there are also APIs specific to certain types of communication, such as Bluetooth. A Bluetooth program would call functions from the Bluetooth DLL for device discovery, for example, then use the socket API for the bulk of the data communications.

- Socket Server

  The *socket server* is a process that implements and manages communication sockets. Applications act as clients to this server through the application-level communication DLLs. As with the GUI DLLs, these functions hide the actual client/server communications from the socket server.

- Protocol Modules

  The socket server uses *protocol modules* for handling the network data protocols. These are polymorphic DLLs (`prt` files) that implement different communication protocols, while providing a consistent interface to the socket server. Examples of protocol modules are TCP/IP, Bluetooth and IR. New protocol modules can be created and used.

  Protocol modules are independent of the data-link layer – bringing up the connection and exchanging data with the device is done through an abstracted interface. This interface is accomplished with two other plug-in modules that attach to the socket server: a network interface (which is usually the PPP module) and a connection agent.

- Network Interface Manager (Nifman)

  The socket server with protocol modules use *Nifman* (network interface manager) to establish the connection and set up the data path to the data link level. In order to start a particular physical connection, Nifman will load a *connection agent*.

- Connection Agent

  A *connection agent* is a polymorphic DLL that is responsible for starting and stopping the communication connection. Not only is it responsible for establishing the connection itself (e.g. dialing a number for GSM or starting GPRS), but it provides information to set up the data communication path between the physical device and the network protocol module. The connection agent will normally use ETEL (described further down) to start the connection.

- Communication Database

  A connection agent will consult the *communication database* to determine how to establish the network connection. This database contains all the settings applicable to communication connections. Depending on database settings, the connection agent can choose to start a preferred connection or it may prompt the user to select a connection. Once the connection is chosen, the agent will extract all the applicable connection parameters from the database to start the connection.

  A connection to a network on a Symbian OS smartphone is known as an Internet Access Point (IAP). An example is a GSM CSD connection using a specific ISP phone number and login information – all stored in the communication database entries for that IAP.

- ETEL Server

  ETEL is a low-level server used to establish a connection with a communication device. It provides an abstracted telephony API to its clients, with functions for tasks such as establishing the connection, terminating the connection, and retrieving line status and device capabilities. Modules, called *TSY* modules, are installed to contain the implementation for the target device. An ETEL client will load the appropriate TSY, then use the ETEL-abstracted API to control the device. Symbian OS has many built-in TSYs for devices such as GPRS and GSM (files end in `.tsy`).

  In addition to loading a Connection Agent, Nifman will load a network interface module (DLLs suffixed with `.nif`). This is usually `PPP.nif` which implements the PPP data link protocol. This module uses the abstracted API of the Communications Server to transfer data to the device.

- Serial Communications Server

  The *Serial Communications Server* provides an abstraction for serial communication across multiple devices. Reading and writing serial data and managing data flow control are example functions of this abstracted API. The details of the low-level protocols for handling a specific device are implemented in DLL CSY modules (suffixed by `.csy`). Example CSY modules include IR, GPRS and UART.

- CSY Modules

  CSY modules communicate with the hardware through device drivers. The device drivers handle the actual control of the communications hardware.

Symbian OS v7.0 and previous versions could have only one active IAP connection at a time. Symbian OS v7.0s introduced a multi-homing capability: the ability to have multiple IAP connections – each with its own IP address – active at once. This is useful, for example, if you want multiple functions active that use different GPRS contexts (such as MMS and web browsing). Another example is having interfaces such as WLAN and GPRS up at the same time.

This feature opens up many possibilities for devices that support multiple ways of accessing the Internet and will become more important for future smartphones.

3.12 Application Engines, Services and Protocols

This section briefly covers application engines, services and protocols. The SDK documentation can be referenced for more detailed information.

Symbian OS provides *application engines* to access and manipulate data from core Symbian applications, such as agenda and contacts. This is useful in creating companion applications that work in conjunction with these core applications. API classes are provided to read and write calendar entries, to-do lists and contact entries.

Application services provide high-level utility functions for applications to use and consist of several client-side APIs and servers. For example, there is a World Server that provides central access to information about different cities (area code, time zone, country, map position, etc.). Other examples are the alarm and log servers, which handle setting and initiating alarms and logging various types of system information, respectively.

There are three framework APIs which currently make up a library group known as *application protocols*:

- ECOM is a software framework used for implementing plug-ins – it has custom functionality for different entities of a particular type, while maintaining a consistent, abstract interface to them. Isn't that what polymorphic DLLs do? Yes, but ECOM is a more extensive framework for this. While polymorphic DLLs allow for abstracted interfaces, the method of finding and loading the available DLLs falls on each application. ECOM provides for a generic framework for handling this higher level plug-in functionality.

- The HTTP library provides an API for handling the Web-based HTTP protocol.
- SyncML is a standard for synchronizing user data between devices, and a set of APIs is provided for applications that wish to use it.

4

Symbian OS Programming Basics

This chapter focuses on the fundamentals of Symbian OS programming. So far, I've described smartphones in general, presented some steps to get started with the SDK, walked through some example code, and described the general architecture of the operating system. This chapter, however, marks the real beginning of your Symbian OS programming training as we get down to the basics.

You will not find any references to Series 60, Series 80 or UIQ in this chapter. The information presented here is generic for all Symbian reference platforms.

I begin the chapter with an overview of the use of C++ in Symbian OS, followed by a look at the basic data types and the key types of classes you'll use and create. Then, I show how to program using the error-handling mechanism in Symbian OS, using leaves and traps, and how to use the cleanup stack. Next, I cover libraries in Symbian OS – both statically linked and DLLs.

Finally, I outline the key naming conventions used when developing Symbian OS code and provide a summary of key points to remember when writing Symbian OS software.

4.1 Use of C++ in Symbian OS

C++ is the primary language for software development on Symbian OS since it provides the most efficient and natural interface to the system-level frameworks and APIs which themselves are written in C++. In fact, Symbian OS itself is written almost entirely in C++. When developing Symbian software, you'll be using many of the standard C++ language features, including *inheritance, encapsulation, virtual functions*, function *overloading*, and *templates*.

These language features are not only used for implementing your application logic, but also in using the system APIs. For example, some APIs are abstract classes that your application classes can inherit from and extend their functionality as needed. Other APIs are classes that

are instantiated and used directly. Still others are simple function calls implemented as static class methods that can be called directly in the same manner as C-based APIs – no class instantiation is required (the static API class *User* is a good example of this).

4.2 Nonstandard C++ Characteristics

Although Symbian OS uses many of the object-oriented features of C++, some of its functionality is implemented in nonstandard ways. This can require an adjustment, even for experienced C++ programmers. For example, Symbian implements its own exception-based mechanism for handling errors such as low memory conditions in place of the C++ throw/catch exception feature. Also, Symbian OS does not use the Standard Template Library (STL) and instead has Symbian OS-specific implementation classes for functions such as string manipulation and complex collection types. Symbian decided on this course for a variety of reasons including making the implementation more efficient for resource-constrained devices.

To begin the discussion of Symbian OS basics, let's start with the basic data types.

4.3 Basic Data Types

To provide machine and compiler independence, Symbian OS provides a set of data types that should be used in place of the standard C++ types, such as `int`, `long` and `char`:

- **`TInt, TUint`**: An integer whose size is the natural machine word length (at least 32 bits). These are mapped to `int` and `unsigned int`.
- **`TInt8, TInt16, TInt32`**: Signed integers of 8, 16 and 32 bits respectively.
- **`TUint8, TUint16, TUint32`**: Unsigned integers of 8, 16 and 32 bits respectively.
- **`TInt64`**: A 64-bit integer, implemented in two unsigned 32-bit integers. The class implements operator methods (such as, `+`, `*`, and `=`) and thus can be used like a normal data type. Some nonoperator methods such as `Low()` and `High()` (to get the lower and upper 32 bits of the data) are also provided.
- **`TText8, TText16, TText`**: Simple character data. TText8 and TText16 are mapped to `unsigned char` and `unsigned short` respectively. However, `TText` is the best one to use, since it will

be defined as either 8- or 16-bit, depending on whether or not your application is configured for a Unicode build.

- **`TChar`**: A class (as opposed to the simple *typedefs* used for the `TText` types) that represents a character. It contains various character detection and manipulation methods, such as converting between upper and lower case, and checking whether it's a control character. `TText` should be used if possible, since it has less overhead cost. `TChar` forms the basic building block for the string functionality in Symbian OS.
- **`TBool`**: A Boolean type, whose value is either `ETrue` or `EFalse`. This type is mapped to `int`.
- **`TReal32, TReal64, TReal`**: Floating point numbers. `TReal64` and `TReal` both represent double precision 64-bit real numbers and are mapped to the `double` data type. `TReal32` represents a 32-bit floating point value. This smaller precision can be limiting; however, it's useful in cases where performance is more important than precision. The smaller data size results in faster floating point operations.
- **`TAny`**: Mapped to the standard *void* data type in C and C++. Symbian uses `TAny` because it is more descriptive than `void` when representing a 'pointer to anything'. Functions that return no value still use `void`, since in that case `void` is accurately descriptive.

See Example 4.1 for some sample declarations for the basic data types.

Example 4.1. Basic data types

```
TInt foo(TInt aParm1, TText aParm2, TAny *aPtr)
                         // returns an int
                         // takes an int, a character, and a void pointer

  {

  TInt var1;
  TChar dummyC;
  dummyC = 'A';
  for (TInt i=0;i<10;i++)
    {  /* some stuff */ }

  dummyC.lowerCase();  // converts the 'A' to 'a'

  }
```

4.4 Symbian OS Classes

There are four main categories of C++ class in Symbian OS. To improve code readability, Symbian OS has a convention of prefixing class names with a letter to identify the class type. This convention should be followed

when creating your own classes that fall into these categories:

- T – Data type classes
- C – Heap allocated classes derived from `CBase`
- R – Resource classes
- M – Interface classes

4.4.1 Data Type Classes

Data type classes encapsulate a value of a specific type. These classes have optional methods for manipulating, comparing, and otherwise controlling the object's contained value. The `TChar` class described in the previous section is a good example of this: each instantiation of `TChar` holds a character value. The `TChar` class methods can be used to perform operations on that value.

Data type classes start with `T`, but this convention is not limited to classes. As we have seen in the previous section, a `T` is prefixed to any declaration that represents a data type. This includes *typedefs* and *enums*.

4.4.2 Heap Classes

Heap classes inherit from Symbian's `CBase` class, which is why the `C` prefix is used. As the name suggests, they are instantiated on the heap (i.e., with `new`) as opposed to on the stack as automatic variables or as class members. Heap classes are referenced by pointers.

Deriving a class from `CBase` ensures that:

- The destructor of the derived class is called when the object is deleted through a base class pointer. (`CBase` declares a virtual destructor.)
- All data members in the class are initialized to zero when instantiated. This prevents problems such as uninitialized pointers.

Heap classes should not be allocated on the stack. Since the zero initialization is done by `CBase` as a result of the `new` operator, and since `new` is not performed in the case of stack instantiation – then the member variables will contain undefined data when instantiated (as is normal with non-`CBase` derived classes). This can cause a problem if the class (or class user) is written to assume that the data is initialized to zero – an assumption that is valid for correctly instantiated Symbian OS heap classes.

4.4.3 Resource Classes

Resource classes are used to control objects that are implemented and owned somewhere else. For example, client classes in a client/server

structure are implemented as resource classes since the actual resource is controlled by the server. Also, the Symbian OS API provides numerous resource classes that allow user programs to control objects that are owned and implemented by the kernel (threads, processes, mutexes and memory chunks are examples of this).

Resource classes begin with `R`, which stands for resource (you can also think of the `R` as standing for remote). These classes are normally allocated on the stack or as class member variables – although they can also be created on the heap.

Since an `R` class instance is a handle to a resource, deleting it does not delete the actual resource itself. This is different from the behavior of `T` classes where deleting a `T` class instance also deletes the data the `T` class instance represents (since the data is just a member of the class).

The Symbian `RFile` API class is a good example of a resource class. Files are opened by instantiating an `RFile` object and calling its `Open()` method. The object then acts as a handle to read and write the file (using the `Read()` and `Write()` methods of `RFile`). Deleting the `RFile` object does not delete the file associated with it.

Another example is `RThread`, as illustrated in Example 4.2. (The class `RThread` will be covered in more detail in Chapter 7.)

Example 4.2. Resource Class Example

```
void func1()
  {
  RThread thread;

  // opens reference to thread with id ThreadXId
  thread.Open(ThreadXId);

  // Raise priority one notch above the default priority
  thread.SetPriority(EPriorityMore)

  thread.Close();
  }
```

The code in this example will raise the priority of the thread whose thread ID is `ThreadXId`. Although `RThread`'s `Close()` method is called – and the `RThread` object itself is destroyed when the function exits – the actual thread is not deleted, since `RThread` is simply a handle to it. Note that, for simplicity, no error checking is done in this example.

`RSocket`, `RProcess` and `RSemaphore` are other examples of resource classes.

Resource classes follow certain patterns. They usually use an `Open()` method (and sometimes `Connect()`) to create a handle to the resource, and a `Close()` to close the handle to the resource. Creating and closing a resource handle results in a reference count being incremented and decremented respectively, and the Symbian OS kernel will not allow the

actual resource to be deleted if there are any open handles to it. For some resources, the resource is deleted automatically by the system when the last handle to it is closed.

4.4.4 Interface Classes

Interface classes, which are prefixed with `M` for `Mixin`, are abstract classes whose purpose is to define an interface (sometimes known as a *protocol*) for other classes to use, as opposed to implementing functionality themselves. Interface classes have no member variables and in most cases contain only pure virtual functions. Typically, you derive a class from one or more interface classes using multiple inheritance, and then override and implement the interface's functions as appropriate for your class, for example:

```
class MInterface1
  {
  virtual void DoThis()=0;
  };
class MInterface2
  {
  virtual void Callback()=0;
  };

class AClass: public ABaseClass, MInterface1, MInterface2
  {
  virtual void DoThis();  // override method from MInterface1
  virtual void Callback();  // override method from MInterface2
  }

void AClass::DoThis()
  {
  //implementation here
  }

void AClass::Callback()
  {
  // implementation here
  }
```

This example shows two interface classes, `MInterface1` and `MInterface2`, each consisting of a single abstract function. Class `AClass` uses multiple inheritance to inherit from a normal base class, `ABaseClass`, and from the two interface classes. `AClass` then implements the actual functionality behind the interfaces by overriding the interface functions.

Having separate classes for interfaces is more manageable than simply adding all the interface methods directly to your class (or in one of your base classes). Also, inheriting from interface classes allows you to use an interface class pointer when operating on an object and not care

what the actual derived class type of the object is (this is a typical C++ polymorphism). For example:

```
void CallMeBack(MInterface2 *aObj)
  {
  aObj->Callback();
  }
```

An object of any class type that inherits from `MInterface2` can be passed to `CallMeBack()` and the appropriate `Callback()` implementation of the passed object is called.

Deriving from interface classes is the only situation in Symbian OS where multiple inheritance is used. You will run into problems if you attempt to use other forms of multiple inheritance, since the standard base classes were not designed to support them.

Figure 4.1 provides an example interface class relationship.

In Figure 4.1 `Class2` implements two interface classes, `MProtocol1` and `MProtocol2`. `Class1` implements `MProtocol1` only. Both classes, as is typical, also inherit from a normal, non-interface class in addition to the interfaces they implement (shown as `Class1Base` and `Class2Base` in the figure).

`Function1(MProtocol1 aProt1)` will accept an argument of type `Class1` or `Class2` since both of these classes are derived from `MProtocol1`. The argument is cast down to an `MProtocol1` type and

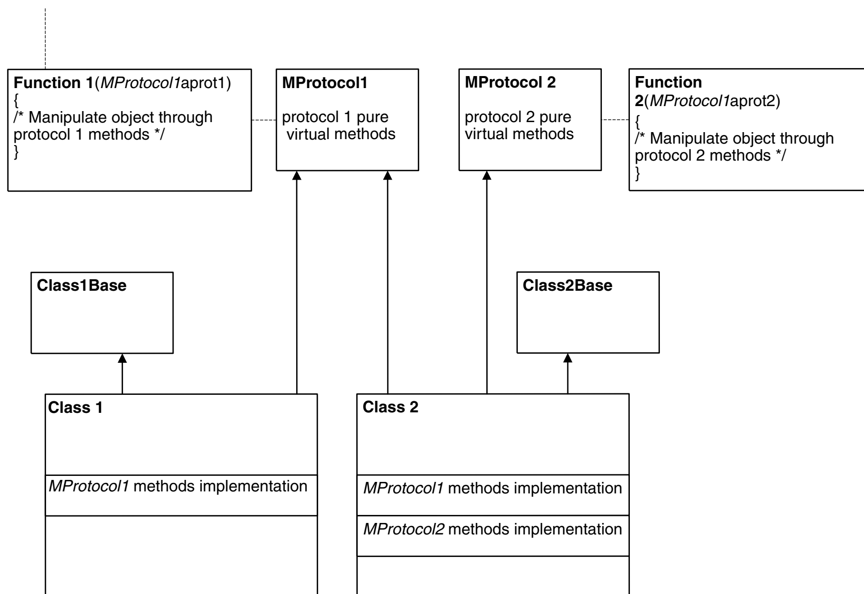

Figure 4.1 Interface Classes Example

`Function1()` will then manipulate the object as needed through the `MProtocol1` interface methods.

`Function2(Mprotocol2 aProt2)` will accept objects of type `Class2` since `Class2` inherits from `MProtocol2`. Passing `Class1` to this function will generate a compiler error since `Class1` does not inherit from class `MProtocol2` (in other words, `Class1` does not support the `MProtocol2` protocol).

From an object-oriented point of view, there are many benefits of using interface classes for the purpose of managing objects using specific protocols. While I did not go into the theory in much detail here, hopefully this gives you a better idea of what interface classes are and how they are used.

If you are a Java programmer, you may recognize the `M` class concept as being a C++ implementation of the Java `interface` keyword.

4.5 Exception Error Handling and Cleanup

Good error handling and recovery are essential for limited resource devices such as smartphones. For example, if an application runs out of memory, the user should not lose any data and the smartphone should not crash.

Symbian OS provides an extensive error handling and recovery mechanism that is used heavily in Symbian OS software. You'll need to understand this functionality since it will comprise a significant portion of your Symbian OS software design and development effort. This section describes how to use this functionality.

4.5.1 Error Handling via Return Codes

Traditionally, functions are written to return status codes that indicate either success or some particular failure. Symbian OS uses this method for many of its APIs – a function returns `KErrNone` on success, and a particular error code (e.g. `KErrNotFound`, `KErrNoMemory`) on failure, as defined in `e32std.h`. A simple `if` after the function call can test for and handle an error.

But providing return statuses alone is not enough for a robust user experience. Why? Two reasons: firstly, not all return codes are tested by the programmer when invoking functions or creating objects. Secondly, the calling function may not know how to handle particular errors, which can result in inconsistent behavior for 'core' error conditions, such as running out of memory.

4.5.2 The Leave/Trap Mechanism

To solve the problems just mentioned, Symbian OS provides an exception-based error handling and recovery mechanism based on *leaves* and *traps*. When an error occurs, the software invokes a leave, which causes the function to exit immediately. Control returns to the calling function, which, if no trap exists, will also exit at that point. This process continues up the calling chain until a trap is encountered, at which time the error is handled. Figure 4.2 illustrates this exiting process up an example nested calling chain to where a trap is defined.

As you can see, the leave is a more proactive way of indicating an error. Unlike simple return codes, a leave cannot be ignored.

When a function is interrupted and exited due to a leave (as `Func2()`, `Func3()` and `Func4()` are in Figure 4.2), it will act as if a return occurred at that point: all automatic variables will go out of scope and thus will

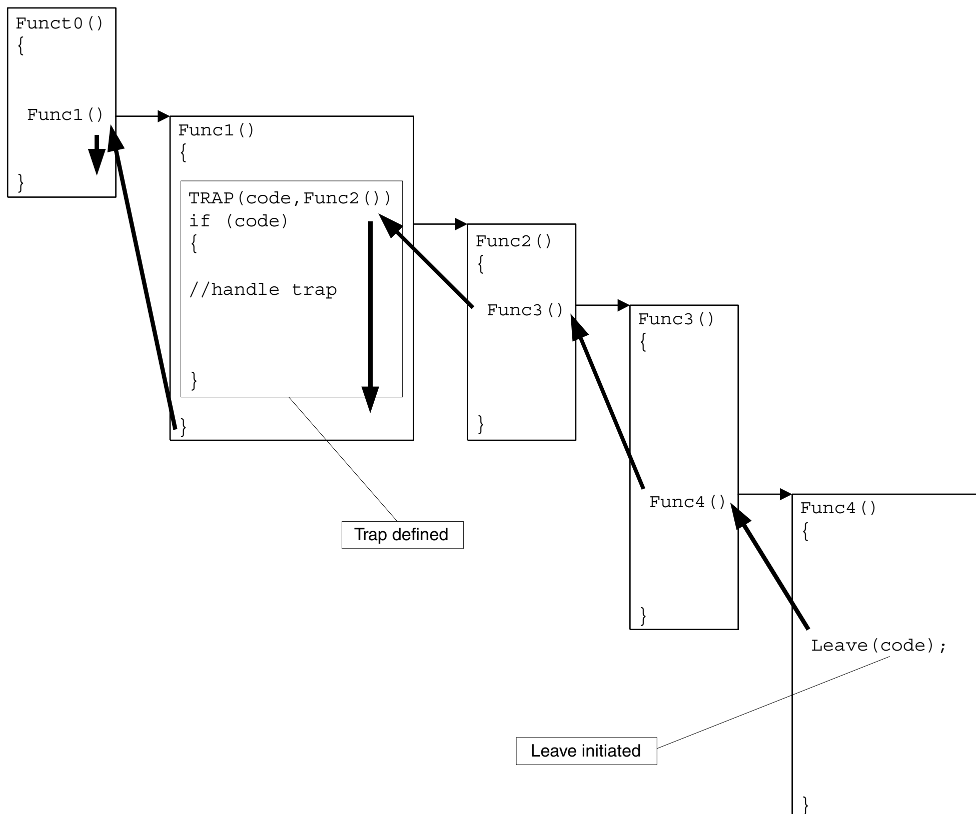

Figure 4.2 Leave/Trap

be deallocated. However, any cleanup that requires explicit code (e.g. delete statements) will be skipped (if they occur after the leave). This is an issue that must be accounted for and the correct way to handle this is discussed in Section 4.5.6.

Since this leave/trap method is similar to C++'s built-in throw/catch exception handling, you may wonder why Symbian did not use that instead of inventing its own method. The reason is that this mechanism was not a part of C++ at the time Symbian OS was written. Also, the leave/trap method in Symbian OS is more lightweight and efficient than the C++ throw/catch method.

Example 4.3 shows example code of leave/trap usage. The macro `TRAP` is used to invoke the function and trap any leave code that occurs. The function `User::Leave()` is used to execute a leave in the case of an error.

Example 4.3. Leave/Trap Example

```
void fooBarL ()

  {
  TInt rc;

  rc = SomeFunction();
  if (rc)
    {
    User::Leave(KAnError);   // leave invoked
    }
    // The code here is not executed if Leave above occurred.
  }

void MyFunctionL()

  {

  ....
  fooBarL();
  // The code from here on will not be executed
  // if Leave was called in FooBarL()

  ...

  }

void StartHere()

  {
  TInt LeaveError;

// invoke MyFunctionL(), with a trap to catch leaves

  TRAP(LeaveError,MyFunctionL());
  if (LeaveError)
```

```
    {
          // MyFunctionL() Leave occurred, handle here

    }

    // code is always executed here - leave or not
  }
```

In Example 4.3, execution starts at StartHere(). StartHere() invokes MyFunctionL() through the TRAP macro and MyFunctionL() invokes fooBarL() (without a TRAP). If rc is set in fooBarL() (due to a SomeFunction() failure), then the system static API function User::Leave() is called. This will cause fooBarL() to stop executing at that point and return to MyFunctionL(). Since MyFunctionL() did not define a trap handler when calling fooBarL() (i.e. the TRAP macro was not used) then MyFunctionL() will exit, immediately after the call to FooBarL(), propagating the exception to its calling function, StartHere().

Since we invoked MyFunctionL() using the TRAP macro in StartHere(), then StartHere() will not automatically exit. Instead, the leave code (KAnError) is written to the first argument of the TRAP macro (LeaveError) and execution continues normally. If a leave did not occur, LeaveError is set to KErrNone (0).

Execution always continues normally after the TRAP macro, whether or not a leave event occurred. A simple if after the TRAP handles LeaveError.

4.5.3 The TRAP and TRAPD Macros

Let's look at the TRAP macro in more detail. TRAP takes two arguments. The first is a TInt variable in which the leave code is placed. The second is the function you want to invoke and trap the leave codes from.

Here is an example TRAP call:

```
TRAP(LeaveCode,functionL())
```

This statement invokes functionL() and if functionL() returns due to a leave (in it or on down the calling chain), then LeaveCode is set to the value passed to the User::Leave() function call, and execution is continued normally. If a leave did not occur (the normal case), Leavecode is set to KErrNone (0) and execution also proceeds. Note that the first argument can be any variable name and that variable must have been declared previously (as a TInt) or the compiler will complain.

A variation on TRAP is TRAPD (the D is for declare). TRAPD is the same as TRAP except the TRAPD macro will declare the first argument (the leave code variable) so you do not have to. For example:

```
TRAPD(LeaveError,MyFunctionL());
```

is equivalent to:

```
TInt LeaveError;
TRAP(LeaveError,MyFunctionL());
```

In Example 4.3, if TRAPD was used in place of TRAP (in the StartHere() function), then the TInt LeaveCode; line would not be needed. In fact, the compiler would generate a multiple declaration error if you left that line in since using the TRAPD in this case results in the compiler seeing two TInt LeaveCode declarations. The same error would occur if you have multiple TRAPD calls using the same leave code variable as the first argument. To avoid multiple declaration errors in that case, you could use TRAPD first and TRAP thereafter.

You can also use TRAP(LeaveCode,ret = functionL()) to assign a return value. Keep in mind that the return value will only be valid in the case of LeaveCode being set to KerrNone (indicating that no leave occurred).

What if you do not use TRAP or TRAPD in your code and a leave occurs? In this case, the operating system code will handle it depending on the error. In many cases, the thread is killed.

4.5.4 Leave Functions

Symbian OS has a set of static API functions grouped in a class called User. This is where the leave functions reside.

Here are the different variations of the leave function:

```
User::Leave(code);         // simple leave, passing leave code
User::LeaveNoMemory();     // equivalent to User::Leave(KErrNoMemory)

User::LeaveIfError(error);    // if error is negative, do a leave using
                              // error as the reason. Just return
                              // 'error' if not negative.

User::LeaveIfNull(TAny *ptr); // if ptr is NULL, do a leave with
                              // reason as KErrNoMemory
```

In most cases, your experience will be in trapping (and handling leave cleanup issues) from system APIs that have the potential to leave. If you look at the SDK API reference, you will see the possible return codes for

the function and possible leave codes (if any). Functions that may leave have a suffix of L (or LC) as discussed next.

4.5.5 What Do the 'L' Suffixes Mean?

The Symbian OS convention is to add an L to the name of all functions that may leave. Why is this needed? The first reason is that it gives you a clue that you may want to trap some of the leave codes that could occur. The next reason (which is the most important) is that you need to know that the function may actually exit at that point and not execute the lines further down. You need to look at your code and think about this carefully. Make sure the code will cleanup properly if a leave occurs.

Example 4.4 shows some code with a cleanup problem.

Example 4.4. An accident waiting to happen

```
MyFunction()
  {
  TInt *buff;
  buff = new CThisObject;

  ...
  funcL();
  ...
  delete buff;
  }
```

If a leave occurs in `funcL()`, the `delete buff` line would never be called and you would be left with allocated memory on the heap. Since `buff` is an automatic variable, it will go out of scope upon exit and you are left with orphaned memory with no reference to it (a textbook example of a memory leak).

To avoid this situation, you could structure your code such that deletes are never needed after functions that may leave, but this is an awkward, if not impossible, solution. Another option is to always use class member variables instead of automatic variables when allocating heap memory, and perform the deletes in the class destructor. This could work for classes allocated on stack, since they will go out of scope on leaves, but it can be limiting. But what else is left? Well, you can `TRAP` the function and have the `TRAP` handler call the delete, and then just reissue the `User::Leave()` so that the real `TRAP` can handle further up. Clever, but still awkward to do for every call of an L function in your program. So what is the solution? Thankfully, Symbian provides a method of handling this situation – the *cleanup stack*.

4.5.6 Cleanup Stack

Automatic pointer variables can be pushed onto a cleanup stack during a function's execution. If a leave occurs in the function (either directly

via the `User::Leave()` function call, or from an `L` function that leaves), each pointer that was pushed on the cleanup stack is popped and freed before the function is exited. This will prevent the problem described in the last section.

Items that were pushed onto the cleanup stack must be manually popped off when there is no more danger of a leave occurring before the deletion. As with all stacks, items are popped from the cleanup stack on a last-in–first-out basis and the stack must be kept balanced in order to perform as expected.

Symbian OS provides a static API class called `CleanupStack` for accessing the cleanup stack. The basic functions in this class are: `CleanupStack::PushL()` and `CleanupStack::Pop()`. These functions push items to and pop items from the cleanup stack respectively.

Example 4.5 shows the cleanup stack in use.

Example 4.5. Using the cleanup stack

```
Func1L()
  {
  CMyObject *myObj = new CMyObject;
  CleanupStack::PushL(myObj);
  TInt *buff = new TInt[1000];
  CleanupStack::PushL(buff)
  DoSomethingL();
  CleanupStack::Pop(2); // Pop last two items off cleanup stack

  delete myObj;
  delete buff;

  }
```

Both `myObj` and `buff` are pushed onto the cleanup stack with `CleanupStack::PushL()`. If function `DoSomethingL()` leaves, execution stops at that point; however, before control is returned to the calling function, each pointer on the cleanup stack (`myObj` and `buff` in this case) is freed.

If `DoSomethingL()` does not leave (normal case), the items are popped off the cleanup stack manually via `CleanupStack::Pop(2)`. The argument '2' means to remove the last two items pushed.

A variation of the Pop function is `CleanupStack::PopAndDestroy()`. This function removes the item from the cleanup stack and deallocates it. The item is cleaned up in the same way it would be if a leave had occurred. Since popping and deallocating are often done at one time, this function is convenient. In Example 4.5, `CleanupStack::PopAndDestroy(2)` could replace `CleanupStack::Pop(2)` and the two `delete` statements.

Before the cleanup stack can be used, it must be created for the thread that uses it. You will not have to worry about this for GUI applications and servers since the cleanup stack is created automatically in these cases. However, in other types of programs (or in user-created threads within a GUI program, for example), you will have to do this yourself by adding `CTrapCleanup *trap = CTrapCleanup::New()` to your code and calling `delete trap` when finished.

4.5.7 Object Types and the Cleanup Stack

The following are the `CleanupStack::PushL()` methods:

- `PushL(CBase *)`
- `PushL(TAny *)`

If a `CBase`-derived object is pushed on the stack, upon cleanup (performed as a result of a leave or a `CleanupStack::PopAndDestroy()` call) a `delete` will be performed on that object, causing the object's destructor to be called. Since the `CBase` destructor is virtual, the destructor of the derived class is called. This is the ideal cleanup case.

If a non-`CBase` object is pushed on the cleanup stack (causing the `PushL(TAny *)` version of the function to be called), then the corresponding `PopAndDestroy()` function does not call `delete` on the pushed pointer, but instead calls `User::Free()`. This simply frees the memory allocated to the object, without calling the destructor.

The reason `PopAndDestroy()` does not call `delete` in this case is because `PushL(Tany *)` cannot be sure that the passed class has a virtual constructor (which it knows `CBase` has). Since the pointer could be to a base class, it will not know if the correct destructor of the concrete class would be called. So, if you push an object on the cleanup stack that is not of type `CBase`, only partial cleanup may take place when `PopAndDestroy()` is called, or a leave occurs. (To resolve this issue see Section 4.5.9.)

Therefore, a good rule is only to use `CleanupStack::PushL()` on objects derived from `CBase`. But you can also safely use it for objects (such as simple memory allocations) which have no destructor code.

Note that `PushL()` can itself leave due to an error. However, it will only leave after the item is pushed on the cleanup stack, so you can be sure that the item will be cleaned up even when `PushL()` fails.

4.5.8 More Complex Cleanup

In some cases, deleting memory that is referenced by automatic pointers is not the only type of cleanup that is needed if a leave occurs. You may have application-specific cleanup (e.g. tidying up a state machine in a

file), or may need to call specific methods in automatic objects before they go out of scope (e.g. `Close()`).

To handle the just mentioned requirement, Symbian OS provides another `CleanupStack::PushL()` overloaded function:

```
PushL(TCleanupItem userCleanup)
```

Using this form of `PushL()`, you push a reference to your own cleanup handling function on to the cleanup stack. Upon cleanup (via a leave or `PopAndDestroy()` call), your function is invoked when this item is retrieved from the cleanup stack. `TCleanupItem` is a wrapper class for a simple function call that returns void and takes one `TAny*` argument. A code example should clarify this – see Example 4.6.

Example 4.6. Using a user cleanup function

```
void myCleanupFunc(TAny *arg)
  {

 // Will execute on leave or PopAndDestroy. Do special cleanup here.
  }

void foo()
  {

  CleanupStack::PushL(TCleanupItem(myCleanupFunc,&data));

  // ...

  Func_1L();

  // ...

  CleanupStack:: PopAndDestroy();

  }
```

If `Func_1L()` leaves, `myCleanupFunc()` will execute with the argument set to `data`.

4.5.9 Other Cleanup Functions

There are three more cleanup stack functions that are useful: `CleanupClosePushL()`, `CleanupReleasePushL()`, and `CleanupDeletePushL()`. These are static API functions that do not belong to any class. These functions use a combination of C++ templates and the `TCleanupItem` form of `CleanupStack::PushL()`, just described, to implement their functionality.

CleanupClosePushL<class T>(T& obj)

This function will push `obj` on the cleanup stack. When cleanup occurs (via leave or `PopAndDestroy()`), `obj.Close()` is called. This is perfect for resource classes ('R') that are allocated on the stack and require the `Close()` method to be called to cleanup.

Example 4.7 shows this function in action.

Example 4.7. CleanupClosePushL() Example

```
void FooL()
  {

  RFile f;

  ...

  f.Open(...);

  ...
  CleanupClosePushL(f);
  func1L();  // may leave, if so f.Close() called

  ...
  CleanupStack::PopAndDestroy();  // f.Close() called
  }
```

You do not need to add the template declaration after `CleanupClose-PushL()` (or any of the three functions of this section) since the compiler can unambiguously determine the class type for the template from the function argument.

`CleanupReleasePushL` <class T> `(T&obj)`

`CleanupReleasePushL` acts the same as `CleanupClosePushL` except that method `Release()` is called on cleanup. Calling `Release()` is required to cleanup some interfaces.

`CleanupDeletePushL` <class T> `(T *obj)`

Pushing an object on the cleanup stack using this function will cause a `delete` to be called on `obj` upon cleanup. How is this different from `CleanupStack::PushL()` ? Since `CleanupDeletePushL()` uses templates, the class type of the object is passed in addition to the object itself. This enables the actual destructor of the passed object to be called upon cleanup regardless of the object's type. Contrast this with `CleanupStack::PushL(CBase*)` where the passed class must be derived from `CBase`.

Thus `CleanupDeletePushL()` should be used for all non-`CBase` classes that have destructors defined. If a class is derived from `CBase`, use

`CleanupStack::PushL(CBase*)`. As you will remember, `CleanupStack::PushL(TAny*)` can be used for simple classes without destructors.

The object passed to `CleanupDeletePushL()` need not have a virtual destructor since the object type is not cast down to a base pointer, as it is in `CleanupStack::PushL(CBase*)`. Whatever derived object you pass to `CleanupDeletePushL()`, that same derived class' destructor is called when cleaning up.

Example 4.8 shows this function in action.

Example 4.8. Using `CleanupDeletePushL()`

```
class myClass
  {
  public:

  myClass();
  ~myClass(){ // will be called on cleanup in this example }
  };

func1L()

  {

  myClass *obj = new myClass();

  // do stuff with obj

  CleanupDeletePushL(obj);

  FooL();  // if leave occurs, delete obj will be called

  CleanupStack::PopAndDestroy();  // delete obj called

  }
```

4.5.10 LC Functions

Functions that end in `LC` provide an added convenience – upon successful completion the return value is pushed on the cleanup stack for you, as shown in Example 4.9.

Example 4.9. Using an LC Function

```
void Func1L()
  {
  TInt *BuffPtr;

  BuffPtr = User::AllocLC(1000); // system static API which allocates
                                 // memory
```

```
...
FooL();

CleanupStack::PopAndDestroy();
}
```

In `Func1()`, if `User::AllocLC()` allocates memory successfully, it returns the buffer pointer to `BuffPtr` and pushes that pointer on the cleanup stack. This saves you a statement, but don't forget to pop the pointers off the cleanup stack after calling `LC` functions!

4.5.11 Leaves when Creating Objects

When an object is constructed using the `new` operator, a memory allocation occurs. Although a return value of `NULL` will indicate that the memory allocation failed, many times you will want it to generate a leave instead. How can you do this? Just insert an `(ELeave)` between the `new` and the class name as in the example below:

```
CMyObject *obj = new (ELeave) CMyObject;
```

This may seem cryptic at first, but it's valid C++ syntax for invoking an overloaded `new` operator function.

For a traditional `new` statement (e.g. `CMyObject obj = new CMyObject`), the compiler invokes the built-in `new` function prototyped as `new(TInt)` – the `TInt` argument being the size of the object. However, if you add `(ELeave)` after the `new` keyword, the compiler invokes the function prototyped as `new(TInt, TLeave)` instead, where `TInt` is the object's size and `TLeave` is the data type for the argument `ELeave`. `ELeave` is just a dummy variable whose purpose is to cause this overloaded `new` function to be invoked. Symbian OS implements this overloaded `new` function (overriding C++'s built-in `new` function) to leave on memory allocation failures.

Remember to take care in cleanup when constructing objects using `ELeave` since there is the possibility that a leave can occur during construction. For example, can you spot the error in Example 4.10?

Example 4.10. Spot the error

```
Func1L()
  {

  TInt *buff1 = new (ELeave) myBuff[1000];
  CMyClass *obj = new (ELeave) CMyClass;

  CleanupStack::PushL(buff1);
```

```
  CleanupStack::PushL(obj);

  Call1L();
  Call2L();

  CleanupStack::PopAndDestroy(2);

  }
```

The problem is that if a leave occurs when constructing `CMyClass`, then `buff1` will not be destroyed. You must push `buff1` before constructing `CMyClass`. Example 4.11 shows the corrected code.

Example 4.11. Corrected Code

```
Func1L()
  {

  TInt *buff1 = new (ELeave) myBuff[1000];
  CleanupStack::PushL(buff1);
  CMyClass *obj = new (ELeave) CMyClass;

  CleanupStack::PushL(obj);
  Call1L();
  Call2L();
  CleanupStack::PopAndDestroy(2);

  }
```

The original code has an additional problem in that if the `PushL()` of `buff1` leaves, then the same issue of `buff1` not being destroyed occurs.

4.5.12 Leaves in Constructors

We have seen how adding `ELeave` will cause a `new` to leave if a memory allocation occurs, but what if a leave occurs in the class constructor itself? This is a problem in Symbian OS, and thus is not allowed. Why? Because the constructor is called immediately (and behind the scenes) after the memory allocation in the `new` operator function, with no chance for the programmer to push the pointer to the allocated memory to the cleanup stack. So if a leave occurs during the constructor, you have an orphaned pointer to the memory allocated for the class.

In other words, a constructor should never leave, so don't call leave in them, or call any functions that may leave (i.e. with `L` suffix) unless you trap them. But isn't that unrealistic? Surely you may want to do memory allocations – or otherwise call functions that may leave – when an object is constructed? This is why Symbian implements what is known as a *two-phase constructor*.

4.5.13 Two-Phase Constructors

The two-phase constructor concept is simple: a method is supplied in your class named `ConstructL()` which completes the object construction. A leave can occur in this method since it is just a normal function. See Example 4.12.

Example 4.12. Implementing a two-phase constructor

```
void  fooL()
  {
  CmyObj *obj = new (ELeave)CmyObj;

  CleanupStack::PushL(obj);
  Obj->ConstructL();

  CleanupStack::PopAndDestroy();

  ...

  }
```

Of course, if you can write your whole constructor without the possibility of a leave occurring, then the two-phase method is not needed.

It is important to know whether an object has a `ConstructL()` before using it. Not calling `ConstructL()` on an object that relies on two-phase construction will result in fatal consequences.

Symbian OS classes are often implemented with a static `NewL()` that will create the object correctly by performing both a `new` and the `ConstructL()` call, as in Example 4.13.

Example 4.13. Implementing `NewL()`

```
CMyObj* MyObj::NewL()
  {
  CMyObj* self = new (ELeave) CMyObj;
  CleanupStack::PushL(self);
  self->ConstructL();
  CleanupStack::Pop();
  return self;
  }
```

Or if you want to provide a `NewLC()` function, you could implement both, as in Example 4.14.

Example 4.14. Implementing `NewLC()`

```
CMyObj* MyObj::NewL()
  {
  CmyObj* self = NewLC();
```

```
  CleanupStack::Pop();
  return self;
  }

CMyObj* MyObj::NewLC()  // the returned object pointer will be on cleanup
                        // stack on exit, to save user from pushing.
  {
  CMyObj* self = new (ELeave) CMyObj;
  CleanupStack::PushL(self);
  self->ConstructL();
  return self;
  }
```

4.5.14 Panics

A panic occurs on any error that is not recoverable, at which time the thread exits immediately and the system displays a popup with information regarding the error (the SDK documentation contains a list of the system panics). In general, a panic occurs as a result of a programming error of some kind. An example is if you use an API improperly. For example, if you try to write to a file via the API class `RFile`, without first calling `Open()`, a panic results.

A panic consists of a category name as well as a reason number – the SDK documentation contains a list of these, with a description of what they mean.

You can invoke a panic in your code in response to an error you detect by calling:

```
User::Panic(const TDes& aCategory, TInt aReason);
```

`TDes` will be covered in Chapter 6, but for now you only need to know that it represents a Symbian string. `User::Panic()` will cause the thread to exit and an information box to appear indicating the thread name, as well as the category name and reason code passed to the panic function. Released code should never generate a panic condition (although, unfortunately, some does).

On Series 60, when a panic occurs a box that simply says 'Program Closed' is displayed. To cause the full panic information to appear you need to create a dummy file (which can be empty) in `\system\bootdata\errd` on the target system's `C:` drive. This works for both the emulator and the smartphone.

Example 4.15 shows an example of calling panic.

Example 4.15. Calling Panic

```
_LIT(KFooProgram,"Foo program");  // Defines a string and assigns to
                                  // KFooProgram
```

```
void foo(TInt aX, TAny *aBuff)
  {

  if (aBuff == NULL)
    {
    User::Panic(KFooProgram,3);
    }
  }
```

KFooProgram is a string constant indicating the category (do not worry about this string syntax for now) and 3 is the reason code.

4.5.15 Assert Macros

Assert macros, __ASSERT_ALWAYS and __ASSERT_DEBUG, are usually used in a situation such as in Example 4.15. The macro implements a simple if statement, so you could replace:

```
if (aBuff == NULL)
  {
  User::Panic(KfooProgram,3);
  }
```

with one of the following lines:

```
__ASSERT_ALWAYS(aBuff!=NULL,User::Panic(KfooProgram,3));
```

or

```
__ASSERT_DEBUG(aBuff!=NULL,User::Panic(KfooProgram,3));
```

__ASSERT_DEBUG will only throw the panic in debug builds.

4.6 Libraries

The two main types of library in Symbian OS are static libraries and dynamic link libraries (DLLs). Static libraries are linked to a program at build time – the library functions are extracted and included as part of the calling program's executable. DLLs, on the other hand, are loaded and linked to at runtime. In other words, the complete DLL contents are loaded into a shared memory region and programs call the functions in that region directly as needed. DLLs are efficient since only a single copy of each library function exists in memory, and these can be shared by multiple programs.

Both static libraries and DLLs can contain C++ classes. The library classes can be base classes, from which user programs derive their own classes. Many Symbian OS API classes fall in this category. Libraries can also contain derived, concrete classes which the user manipulates via base class pointers (without knowledge of the details of the derived class). This capability is used by polymorphic DLLs to implement plug-ins.

Of course, the library can also contain classes and functions that can be instantiated and used directly (like `RFile`, `RSocket` or the static `User` classes, for example).

4.6.1 Creating a Static Library

To create a static library, just enter your code in one or more source and header files and create a project definition file like the one in Example 4.16.

Example 4.16. Defining a Static Library MMP

```
TARGET Mystatic.lib
TARGETTYPE LIB
UID 0x1000008D 0x01000023

USERINCLUDE .
SYSTEMINCLUDE \epoc32\include

SOURCEPATH .

SOURCE MySource1.cpp MySource2.cpp
```

The line `TARGETTYPE LIB` indicates that the build is for a static library.

> As we saw in Chapter 2, the `mmp` file is used for building your program and is used with the `abld build wins` or `abld build armi` commands. It can also be imported to a supported Windows development IDE. Chapter 5 will discuss building in more detail.

Add the following to the `mmp` file for the programs that use this library:

```
STATICLIBRARY MyStatic.lib
```

When you build your program the functions required from `MyStatic.lib` are pulled out and included as part of your program's executable. No separate runtime module is required for it to run.

4.6.2 Creating a DLL

Building DLLs is a bit more complicated than building static libraries. Chapter 5 discusses the various build issues encountered with DLLs

(including freezing exports and the def file inner-workings). This section outlines a few programming points regarding DLLs as well as presenting the basic DLL code structure.

These rules must be followed when writing a DLL:

- In the h files, add IMPORT_C before the declaration of each function (class method or external function) that you want to be available to DLL users.
- In the cpp files, add EXPORT_C before the implementation of each function you want available to DLL users.
- Include the entry point function E32DLL(). It can be a stub, but is needed for all DLLs.

While in static libraries, the functions are linked to as if you had included the source files directly in your build – DLLs are handled a little differently. Functions within the DLL can access other DLL functions in a normal fashion (using standard C++ scoping rules), but in order for a function to be available for outside use, it must be exported via the IMPORT_C and EXPORT_C macros above.

IMPORT_C and EXPORT_C are mapped to compiler-specific keywords for indicating access to DLLs. Some development tools may be more lenient (such as the Microsoft compiler, which lets you get away with just adding the IMPORT_Cs in the header) but, for portability, you should use both IMPORT_C and EXPORT_C as specified.

See Example 4.17 for an example of a skeleton DLL.

Example 4.17. Implementing a DLL skeleton

```
//MyDLL.h

class DLLClass

  {

  IMPORT_C DLLClass();
  IMPORT_C void Func1();
  IMPORT_C void Func2();
  void Func3();
  virtual void Func4();

  };

//MyDll.cpp

EXPORT_C void Func1()
  {
  // do stuff
  }

EXPORT_C void Func2()
```

```
  {
  // do stuff
  }

void Func3()
  {
  // do stuff
  }

void Func4()
  {
  // do stuff
  }

TInt E32Dll(TDllReason /*aReason*/)
  {
  return(KErrNone);
  }
```

In Example 4.17, the constructor, `Func1()`, and `Func2()` are exported and available for use by other programs when the DLL is loaded (indicated by `IMPORT_C`/`EXPORT_C` declarations). `Func3()` is not exported and thus is not available – a program will get an error if it tries to call it. `Func4()` is available for outside use. How is that, since `EXPORT_C` and `IMPORT_C` are not used? The reason is that it is virtual. All virtual functions are exported (although it does not hurt to add `IMPORT_C/EXPORT_C`).

Example 4.18 shows the DLL `mmp` file. `TARGETTYPE dll` is what determines whether a DLL is built.

Example 4.18. mmp file for DLL skeleton

```
//MMP

TARGET        mydll.dll
TARGETTYPE    dll
UID           0x10004262 0x10004264

SOURCEPATH    ..\src
SOURCE        mydll.cpp

USERINCLUDE   .
USERINCLUDE   ..\include
SYSTEMINCLUDE \Epoc32\include

EXPORTUNFROZEN
```

4.6.3 Import Libraries

When you build a DLL, you actually build two files: a `dll` and a `lib`. The `lib` is a static library known as an import library. Programs that use a DLL will statically link to its import library. Import libraries contain function wrappers for each DLL function that, when run, will locate and execute the real function's code in the appropriate runtime-loaded DLL.

The import library also handles the details of loading the appropriate DLL, if it is not already loaded.

For your program to use a DLL, you link at build time to the DLL import library by adding it to your LIBRARY line in your mmp. Then, as long as the DLL exists on your target, your program can invoke the exported DLL classes and functions as needed.

The system APIs are implemented as DLLs. The SDK will indicate what include file to use as well as what import library to include in the LIBRARY line of your project's mmp file.

4.6.4 RLibrary API

You do not have to link to the DLL import library to use a DLL (although in most cases it's more convenient). You can use the RLibrary API to manually load the DLL into memory and invoke its DLL functions. The functions are invoked by ordinal number, and you have to know what functions correspond to what ordinal. DLL ordinal numbers and RLibrary are described in detail in Chapter 5.

4.6.5 Polymorphic DLLs

A polymorphic DLL is just a normal DLL, but with only one exported function – a static function that creates an instance of an object and returns it. See Example 4.19.

Example 4.19. Polymorphic DLL

```
//H file

IMPORT_C CmyPolyDll *NewMyPolyL();

class CmyDerivedPolyPlugin : public CmyPoly
  {
  // ...
  };

// CPP file

EXPORT_C CmyPolyDll *NewMyPolyL()
  {
  return new (Eleave) CMyDerivedPolyPlugin;
  }
// Implement rest of class member functions here
```

As covered in Chapter 3, polymorphic DLLs implement virtual functions. You can create multiple DLLs that all implement this NewMyPoly() function, but returning different CMyPolyDll-derived class implementations. The program then chooses which of these DLLs to load, and loads it with the RLibrary load command. Then, the program executes

common code that uses the DLL class through the common base pointer (i.e. CMyPolyDll). This common code does not care which CMyPoly plug-in DLL you have loaded.

4.6.6 Static Data in DLLs

As I mentioned in Chapter 3, writable static data is not allowed in a DLL. This is a Symbian design choice for efficiency. Tacking on a data memory region for DLLs is costly considering the large amount of DLLs available. Whatever the reasons, you will have to live with this, and it can be a challenge sometimes. This is especially true when porting existing code. Note that you can have global variables – but they must be read-only and of type const.

Be aware that the emulator will allow you to put static data in your DLLs, but the target system will not. This can surprise you if you have been doing the bulk of your development on the emulator and then do your initial build using the smartphone target ARM tools and get errors that look something like this one:

```
PETRAN - PE file preprocessor V01.00 (Build 175)
Copyright (c) 1996-2000 Symbian Ltd.
ERROR: Dll 'XXXApp[appUid].APP' has uninitialized data.
NMAKE :fatal error U1077: 'petran' : return code '0xfffffffe'
Stop.
```

This indicates that you need to hunt down those writable static variables and either put const in front of them (if they are used as read only) or devise another method for keeping global data. This subject will be discussed further in Chapter 12.

4.7 Executable Files

The exe file is the basic executable image file for Symbian OS and all programs reside in one. For example, as we have already seen, in Symbian OS releases before v9, a GUI application is actually a DLL launched transparently from a process instance of apprun.exe.

You will need to implement your own exe files for non-GUI related processes such as when you implement a server.

Example 4.20 shows a simple EXE.

Example 4.20. Implementing a simple exe

```
TInt ExtGlobal=0;   // ok to use here in EXE

GLDEF_C TInt E32Main()
```

```
{
for (;;)
  {
  User::After(10000000);  // wait 10 seconds
  User::InfoPrint(_L("Ping Message"));
  ExtGlobal++;  // not used, just to illustrate that you can use
                // writable globals in exe files
  }
}
```

Example 4.21 shows the corresponding mmp file for building the source of this EXE.

Example 4.21. Build file for the exe

```
// exe mmp file
TARGET        myexe.exe
TARGETTYPE    exe
SOURCEPATH    ..\src
SOURCE        myexe.cpp

USERINCLUDE   .
USERINCLUDE   ..\include
SYSTEMINCLUDE \Epoc32\include
```

The executable in Example 4.21 loops forever and displays a message to the screen every 10 seconds. InfoPrint() function displays a message for a short period of time before it disappears (until invoked again after the next User::After()).

As you can see, the basic structure of an exe is simple. The exe only requires the entry point E32Main() and your code takes it from there.

TARGETTYPE exe indicates that the output is an exe file.

Note that you can use writable static data in exe files since a process has its own data area.

An application can start an exe file by calling EikDll::StartExe(_L("c:\programs\myexe.exe")) – assuming that the executable resides in that directory. As currently written, the StartExe() in the code in Example 4.21 will only work on the smartphone and not on the emulator. This is because the emulator does not let you run separate processes within it, and the exe must be compiled as a Windows DLL. The emulator does a fairly good job of simulating exe files with DLLs, but some ifdefs for the emulator are required. Chapter 5 discusses this in more detail.

4.8 Naming Conventions

Symbian OS has a set of naming conventions that should be used when developing Symbian OS software. The operating system itself uses these for its APIs and data.

Naming conventions make code easier to understand and aid correct usage of classes and variables. For example, code such as the following (which appears to be pushing member data to the cleanup stack, since the 'i' prefix indicates a member variable) should be regarded with suspicion:

```
CleanupStack::PushL(iMyData)
```

This is because you should never push a class member variable on the cleanup stack.

Another suspicious line would be:

```
CSomeClass sc;
```

A class based on `CBase` (indicated by the prefix `C`) should never be statically instantiated or instantiated on the stack – `new` (or a static `NewL()`/`NewLC()`) should be used instead.

Yet another example is that if you see a class that begins with an `R` (a resource class) and no `Close()` function called on it – that should encourage you to look at it more closely.

Class Names

We looked at the conventions for class names at the beginning of the chapter. To recap: `T` is prefixed to structures and class names that represent data types, `C` is prefixed to the names of heap classes derived from `CBase`, `R` is prefixed to resource class names and `M` is prefixed to interface class names.

Variable Names

Class member variables should begin with 'i'. Function arguments should begin with 'a'. For example:

```
class TMyClass
  {
  ...
  TInt iMyValue;
  void MyAddFunc(TInt aArg1, TInt aArg2);
  };

void TMyClass::MyAddFunc(TInt aArg1, TInt aArg2)
```

```
{
iMyValue = aArg1+aArg2;

}
```

Global variables (although their use is discouraged) should begin with an uppercase character.

Constants

Prefix constants with `K`. For example:

```
const int KMyConstant;
```

or

```
_LIT(KMyConstantString,"string");)
```

Enumerations

Enumeration types begin with `T` (since they are types). The actual `enum` members should begin with `E`. For example:

```
enum TColors
  {
  ERed,
  EGreen,
  EBlue,
  EPurple,
...
  };
```

Macros

Macros should be all uppercase. For example:

```
#define MY_HARDCODED_VALUE 25
```

Function Names

Function names should be descriptive and, in most cases, are verbs. Function names have suffixes to indicate if they could leave on an error, and/or if they push anything on the cleanup stack. We've looked at these suffixes already when discussing error handling, but here is a recap:

- `L` – Functions in which a leave may occur end in `L` (e.g. `myFuncL().`)
- `LC` – Functions that may leave, having previously pushed their results to the cleanup stack, end in `LC`.

A function suffix not previously discussed is D. A class method ending in D means that the function takes responsibility for the object the method is called from – i.e. it will delete the object when it is finished with it. Thus, the calling program should not delete it (your software will crash if you do!). An example function of this type is the dialog function CEikDialog::ExecuteLD(), which will launch the dialog and destroy the dialog object itself once the dialog is dismissed by the user (note the L before the D, which indicates it may also leave).

4.9 Summary

This section recaps some key points to remember when developing Symbian OS software. Some of these have already been discussed and are included here again for convenience.

- When calling functions that may leave, consider what happens if the program exits at that point and use the cleanup API functions as needed.
- Any function that has a possibility of leaving should end in L (e.g. fooL()).
- Use (ELeave) for instantiating objects (e.g. CClass = new (ELeave) CClass). However, remember that the code could leave at that point.
- Always declare a heap class (indicated by the C prefix) as a pointer only, and create it via new (or NewL()/NewLC()); never declare or instantiate it directly as an automatic variable.
- If you call a function that ends in LC, you need to pop the pointer returned by that function from the cleanup stack (at a suitable place) or your program will crash when the calling function exits.
- Only use CleanupStack::PushL() for CBase objects, and for simple buffers and objects that have no destructors.
- When writing a DLL, place IMPORT_C in the h file before the declaration of each function that you want accessible to the DLL user (e.g. IMPORT_C void method1(TInt aArg1)) and place EXPORT_C in the cpp file before each function implementation (e.g. EXPORT_C void MyClass::method1(TInt aArg1) ...).
- Do not use global writable data in your DLLs (which includes GUI applications). Any external, global variables must be of type const. The emulator build will let you get away with this (this can trick you), but it will complain when you build for the phone.
- You can use writable global data in exe files.

- Follow the naming standards for member variables, arguments, enums, constants and macros.
- When creating an object that will be instantiated on the heap, derive it from `CBase` (or a class already derived from it) and prefix a `C` to your class name (e.g. `CMyClass`).
- When using a Symbian OS API, include the header file and import library specified in the SDK documentation.

5

Symbian OS Build Environment

This chapter examines the Symbian OS Software Development Kit (SDK), the overall build process and tools, and how to create the various build configuration files required to successfully build and install your program. This chapter also covers other key topics, such as using the emulator, and building and freezing DLLs.

I'll mainly use the command line in this chapter – even if you're using an IDE, it's helpful to have a basic understanding of what goes on in the background.

5.1 SDK Directory Structure

The SDK is placed, by default, in a directory called `symbian`. Nokia SDKs are placed in subdirectories of `\symbian`, using subdirectories that specify the Symbian OS version number and the product name. For example, the Series 60 v1.2 SDK is installed at `/symbian/6.1/series60_v1.2`, while the Series 60 v2.0 SDK is installed at `/symbian/7.0s/series60_2.1`.

Let's look at some of the key directories in the SDK.

5.1.1 The `epoc32` Directory

This directory is common to all Symbian OS SDKs (although the earlier Nokia SDKs separate this into two `epoc32` directories in the SDK – `Shared` and `NokiaCPP`). Let's look at the `epoc32` subdirectories.

epoc32/include

This directory contains the system include files needed for your software. The file `e32std.h` is a good one to skim through. It contains common system API class declarations (although most of the API declarations are distributed between numerous include files) as well as error codes that may be encountered.

You'll also notice a `stdlib` directory, which contains C headers for the standard C library implemented by Symbian OS.

epoc32/build

This directory is where the build tools place their intermediate files. As builds occur, you will see directories being created in `/epoc32/build` that mirror your project's location where you executed the build. For example, a build performed at `c:\myProject\group` will create an `epoc32/build/myProject/group` directory. There is also a subdirectory for each component in the build – and under that are subdirectories for every platform you have built, each containing the object files generated for that platform.

If you are curious, you can explore this directory and examine the makefiles generated for each platform – but you'll find that normally you do not need to worry about this directory.

epoc32/tools

This directory contains the Windows-based tools used in the SDK. You'll see a mixture of batch files, Perl scripts, Windows and DOS executables and Java executables.

epoc32/gcc

This directory contains the ARM cross-compiler toolchain, used for building software to run on smartphone devices. When you are building for the emulator target, the Windows development tools are used.

epoc32/release

The `/epoc32/release` directory contains the executables for all supported target platforms. This is where the final executables (e.g. `exe`, `dll`, `app` files, etc.) are placed when you build your software. Directory `/epoc32/release` has subdirectories for each platform supported by the SDK (e.g. `WINS`, `ARMI`, `WINSCW`). These platforms – known as build targets – in turn contain `UDEB` and `UREL` directories. These directories contain the actual executables – `UDEB` contains versions of the executables built with debug symbols, while `UREL` has no debug symbols and is suitable for release.

On emulator build targets (e.g. `WINS`, `WINSCW`), under both `UDEB` and `UREL`, there is a directory called `Z` that contains (together with the contents of `/epoc32/data/z`) the contents of the simulated ROM (`Z`) drive for the emulator.

For smartphone build targets (e.g. `ARMI`, `ARM4`, `THUMB`), the release directory is used mainly for storing executables before they are packaged

to an install file for installation to the smartphone. However, the emulator build targets contain the emulator executable itself, all the Symbian OS system components built for the emulator, and the executables produced by your program builds.

epoc32/data/z

The files in this directory are combined with the build target's `Z` directory to make up the simulated `Z` drive of the emulated smartphone.

epoc32/wins

This is the default location for other emulator memory drives.

5.1.2 Example Directories

Each SDK has a set of standard Symbian OS examples that are common between all platforms. These examples cover a variety of different areas of Symbian OS.

In addition to the generic Symbian examples, the SDKs contain examples specific to the platform. The directories for these vary, and you'll need to locate them. For example, Series 60 SDK's examples are in a directory named `Series60Ex`; UIQ SDKs have a directory called `UIQExamples`.

If you are compiling GUI application examples, use the ones in your platform-specific examples directory instead of the generic Symbian examples. This is because the common examples do not use the vendor-specific API classes and sometimes do not work properly.

5.1.3 Documentation Directories

The SDKs contain documentation that provides a reference for the system APIs and the build tools, as well as other general information, examples, and tutorials for building. Each SDK has the documentation organized differently. For example, the UIQ SDK documentation is entirely in HTML, while the Nokia ones tend to use Windows help and PDF files.

5.2 Build System Overview

The Symbian OS build system is platform-independent; therefore, makefiles are not used directly. After all, different development systems have different make, compiler, and linker tools and these are invoked differently. For example, the compiler is invoked as `cl` for Microsoft compiler and `gcc` for the smartphone. Also there are different makefile formats (i.e. `nmake` and `make`). You would not want to keep track of separate makefiles when compiling your software application for both the emulator and smartphone device.

If you don't write makefiles, how do you define your build? Symbian OS has its own build file format that you must use to specify how your program is built. It contains information similar to that in a makefile, but it is platform-independent – it contains no specific platform or development tool commands. The build command takes the target platform as an argument, generates the necessary makefiles for that platform, and executes them.

I describe how to create the build files, and discuss the build command, shortly, but first let's look at the platforms Symbian OS supports and the concept of a build target.

5.3 Build Targets

Build targets represent the various binary formats (and thus the target platform) which could be used for a build. The ones supported by Symbian OS are listed below.

Smartphone devices:

- **ARM4** – 32-bit ARM instruction set
- **THUMB** – 16-bit ARM instruction set
- **ARMI** – ARM interchange format

Emulators:

- **WINS** – Microsoft
- **WINSCW** – Code Warrior
- **WINSB** – Borland

When you specify one of these build targets in your build command, the build generates and executes a makefile that invokes the development tools needed to produce the appropriate binary output. The executables are then placed in the `/epoc/release` directory under the appropriate build target's name as described previously.

The smartphone build targets use the GNU tools to produce code for the ARM processor – all current Symbian OS smartphones are based on ARM. But why are there multiple ARM build targets, and which one do you use? `ARM4`, `THUMB` and `ARMI` are known as Application Binary Interfaces (ABI) and represent different ARM binary outputs. The ARM processor has two instruction sets: a 32-bit set (`ARM4`) and a 16-bit set (`THUMB`). The first is fast, but uses more memory, the latter is compact, but slower. `ARMI` is the 32-bit instruction set with extra logic to allow it to call `THUMB` code in addition to other 32-bit code. `ARMI` is known as ARM interchange format.

So which one should you use? The most commonly used device build target is ARMI – use this one when in doubt. It is the safest for third-party developers since it will interface with code compiled as ARM4, THUMB and other ARMI code, and will work on any available Symbian OS smartphone. If memory size is a significant concern you can use THUMB. Since THUMB uses 16-bit instructions, the executables are somewhat smaller than ARMI, but ARMI is faster. Many popular smartphones support THUMB.

Do not use ARM4 unless you are writing system-level code such as device drivers and board support software. ARM4 is used by phone manufacturers and is not usually supported on the smartphone for user-level programs.

WINS, WINSCW, and WINSB are emulator targets for Microsoft, Code Warrior, and Borland Windows development tools respectively. The emulator targets generate x86-based Windows binaries; however, you need to use the build target that corresponds to the Windows toolset you have on your PC. This ensures that your Windows development tools are invoked when building. In addition to invoking the correct tools, each emulator build target has its own emulator executable (actually two: a UDEB version and a UREL version). It's required that the emulator, system code, and user programs are compiled with the same Windows compiler – this is needed so that they can link together correctly.

While all SDKs will support the three smartphone device build targets, the emulator build target support varies with the SDK.

5.4 Basic Build Flow

To build a Symbian OS program, you need two build files:

- Component description file (always named bld.inf)
- The Project definition file (suffixed by .mmp)

The component description file is a text file that, in its simplest form, lists the project definition files to be included in an overall build. In most cases, bld.inf will only list a single component. An example bld.inf is shown below:

```
PRJ_MMPFILES
simpleEx.mmp
```

The project definition file (known as an mmp file) specifies the information needed to build a specific program. This includes a list of the program's source files, the paths to the program's include files, and the libraries your program needs to link to. This is the key definition file for the build and its format will be discussed in more detail shortly.

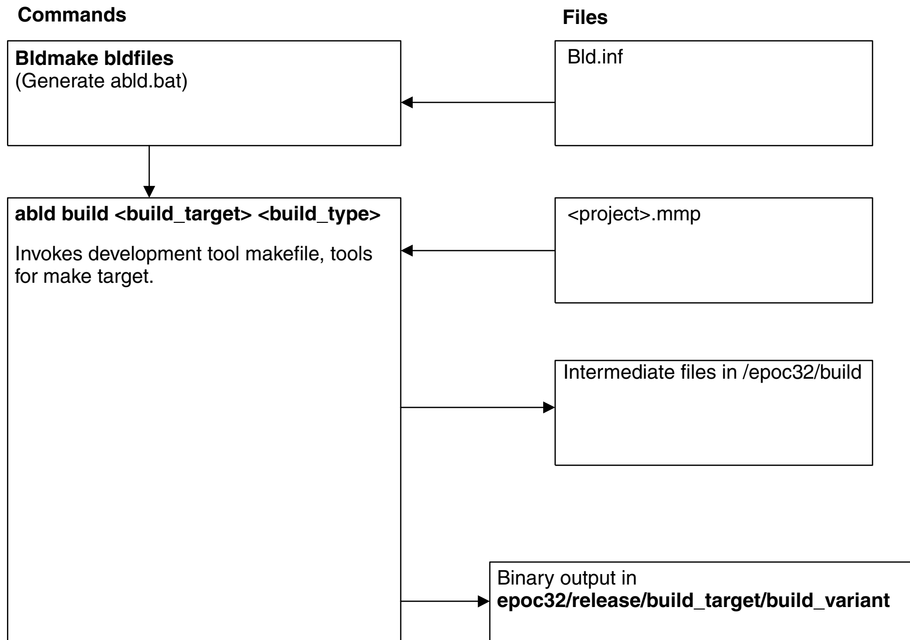

Figure 5.1 Build Flow

As shown in Chapter 2, once these two files are created, you use the commands `bldmake` and `abld` to perform a build based on the `bld.inf` and `mmp` files. For example, you would type:

```
bldmake bldfiles
abld  build wins udeb
```

to perform a wins debug emulator build. `bldmake` actually generates the `abld.bat` file (it's a batch file that ends up calling a Perl program). You only need to invoke it when you build your project for the first time, or if you modify `bld.inf` or move your project directory to another location. From then on you can build by just using the `abld` command.

Figure 5.1 shows the basic flow of the build.

5.4.1 A Closer Look at `abld.bat`

`abld` can be invoked with other arguments besides `build`. The more detailed syntax of `abld` is:

```
abld  [test] command [options] build_target  build_type
```

As you can see, `build` is just one of the commands you can use with `abld`. See the SDK documentation for a complete list. For the most

part there are only two commands besides `build` that you are likely to need: `clean` and `freeze`. The `abld clean` command performs the equivalent of a `make clean`, where all binary files are removed so the software can be completely rebuilt (e.g. **`abld clean armi urel`**).

The `freeze` command is used to freeze the interfaces to DLLs and will be discussed in detail later in this chapter.

Normally, the `options` are left blank, but sometimes it's useful to add `-v`. This generates verbose output so you can see the development system tool commands as they are invoked.

5.4.2 The MMP File

An `mmp` file (sometimes referred to as the project definition file) is a text file used to define a build in a platform-independent way. Each statement in the file begins with a keyword. Statements can span multiple lines by using a forward slash at the end of the line to be continued.

Example 5.1 shows the project definition file for a Series 60 example program.

Example 5.1. Series 60 mmp file

```
TARGET            SimpleEx.app
TARGETTYPE        app
UID               0x100039CE 0x10005B94

TARGETPATH        \system\apps\simpleEx

SOURCEPATH        ..\src
SOURCE            simpleEx.cpp
SOURCE            simpleEx_app.cpp
SOURCE            simpleEx_view.cpp
SOURCE            simpleEx_ui.cpp
SOURCE            simpleEx_doc.cpp

SOURCEPATH        ..\group
RESOURCE          simpleEx.rss

SYSTEMINCLUDE     \epoc32\include
USERINCLUDE       ..\include

LIBRARY           euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY           avkon.lib
```

This section describes some of the main `mmp` file statements. See the SDK documentation for the complete list and description of `mmp` statements.

- `TARGET program_name` specifies your program's executable file name, for example:

```
TARGET myprocess.exe
TARGET myfuncs.dll
```

- TARGETPATH target_path specifies the location where the final executables for emulator builds are to be placed, relative to \epoc32\release\build_target\build_type\z (the emulated Z drive of the smartphone), for example:

```
TARGETPATH \system\programs
```

will put your final executable, for a WINS debug build, in a \system\programs subdirectory of <sdk_dir_root>\epoc32\release\wins\udeb\z.

If the build is for the smartphone instead of the emulator (or the TARGETPATH is not specified), then the executables are placed in \epoc32\release\build_target\build_type (e.g. \epoc32\release\armi\urel).

- TARGETTYPE type specifies the type of executable that is to be created. Here are some of the key types that can be specified in this statement:
  - app – GUI application
  - dll – Dynamic Link Library (DLL)
  - exe – Process executable
  - lib – Static library
  - epocexe – Process executable that can run on both the emulator and the target device (explained later in this chapter).
- UID uid2 uid3 – specifies the second and third UIDs for your component. Refer to the next section for information on UIDs in Symbian OS, for example:

```
UID 0x100039CE 0x10005B94
```

  indicates a GUI application (0x100039CE) with a unique UID of 0x10005B94.
- SOURCEPATH path specifies the directories to search through to find the source files listed in the SOURCE statements.

  path is either relative to the mmp file location or can be a fully qualified path.

  Only one source path is in effect at a time, and it is active until changed by the next SOURCEPATH statement.
- SOURCE source_file_1 source_file_2 ... specifies the source files that make up your project. Multiple statements can be used and more than one file can be included in each statement.

SOURCEPATH and SOURCE are used together to specify your project's source files, as in the following example:

```
SOURCEPATH ../myclass
SOURCE classx.cpp classy.cpp
SOURCE classz.cpp
SOURCEPATH ../myfuncs
SOURCE func1.cpp func2.cpp
```

These statements specify that the build includes classx.cpp, classy.cpp and classz.cpp from the ../myclass directory, and func1.cpp and func2.cpp from the ../myfuncs directory.

- RESOURCE resource_file_1 resource_file_2 ... specifies the application resource files to be compiled. A SOURCEPATH statement can be used with this statement to specify where the resource files are located. The build will compile these resources once for every language that appears in the LANG statement. This will be discussed further in Chapter 11, when we discuss language translations.
- SYSTEMINCLUDE include_path_1 include_path 2 ... contains a list of paths that will be searched for system include files (e.g. #include <estlib.h>).
- USERINCLUDE include_path_1 include_path 2 ... contains a list of paths that will be searched for nonsystem include files (e.g. #include "myinc.h").
- MACRO macro-1 macro-2 ... defines each macro in the list to have the value '1' (as in the compiler -D option). For example, if a project has the following line in its mmp:

```
MACRO TEST_FLAG
```

and a source file in that project implements:

```
#ifdef TEST_FLAG
```

the #ifdef evaluates to true.

5.5 What Is a UID?

Symbian OS uses unique identifiers (UID) extensively for identifying components. Each component is identified by three 32-bit UID integers – UID1, UID2 and UID3.

UID1 is the most general identifier. Examples of UID1s are KExecutableImageUid (0x1000007a), to specify an EXE, and KDynamicLibraryUid (0x10000079), to specify a DLL. You need not worry about specifying UID1 in your mmp file, the build command can determine this UID from your mmp file's TARGETTYPE statement.

UID2 specifies further what type of component it is. For example, in Symbian OS releases before v9, a GUI application is a DLL, so its UID1 is KDynamicLibraryUid (0x10000079) and its UID2 is KUidApp (0x100039CE) to indicate that the DLL is a GUI application. UID2 is used extensively for polymorphic DLLs (where UID1 is KDynamicLibraryUid and UID2 indicates the specific polymorphic 'plug-in' type). An API can use this UID as a sanity check, to make sure it is loading the correct type of DLL.

UID3 is the most specific identifier for the component. It must be unique – no two executables in the system can have the same UID3, or undefined behavior can result.

How do you obtain a unique UID3 for your program? You can reserve a block of unique UIDs (they are assigned in groups of ten) from Symbian by sending an email to *UID@symbiandevnet.com*. In the email, include your name or your program's name, your email address, and how many UIDs you need (be reasonable).

Alternatively, during development, you can use UIDs in the range of 0x01000000 to 0x0fffffff and be assured that no released program will conflict with them (although you should make sure you do not have multiple programs yourself with the same UID).

UID2 and UID3 are specified in the UID statement of your mmp file. Note that exe files do not need a UID2 or UID3 so the UID statement is usually set to 0 (i.e. UID 0) if targettype is set to EXE.

5.6 The Emulator

The Symbian OS emulator is a Windows application that simulates the smartphone on your host PC. You'll find it a very helpful aid while developing your Symbian OS software. With the emulator, the change, build, run cycle occurs more quickly since you can run your program without loading it onto the device. More importantly, since the emulator is a Windows application, you can perform advanced debugging (e.g. single stepping, break points, variable examination) of your Symbian OS applications using your Windows development IDE.

Although all SDK emulators are based on a common core, each SDK has its own emulator variation that looks and acts like the SDK's target smartphone. This includes supporting the device screen size, input devices and graphical user interface. Using your application on the emulator is very similar to using it on the target phone – not only functionally, but aesthetically as well.

Compiling and running on the emulator is straightforward. First, build your software for one of the supported emulator build targets (e.g. `abld build wins udeb`), then launch the emulator via the `epoc` command and run it. The emulator emulates the entire smartphone environment – you select and run your program as you would on the actual device.

Although the emulator is fairly similar to the target device there are some differences that will be discussed in Section 5.6.3. First, let's look at how the emulator is configured.

5.6.1 Running the Emulator

There is a different emulator executable for each emulator target platform and target type. Just typing `epoc` will run the version corresponding to the SDK's principal target platform, with a `UDEB` build type. Alternatively, type

```
epoc -urel
```

to run the non-debug version of the emulator. You can also specify the build target. For example, entering

```
epoc -wins -urel
```

will run the `WINS UREL` version of the emulator (with its associated executables).

5.6.2 Emulator Configuration

The emulator is configured through a file called `epoc.ini`. This file is located in the `%EPOCROOT%/epoc32/data/` directory of your SDK. You'll normally not need to touch it, but it can be used to customize emulator behavior.

Virtual Drives

The emulator simulates the ROM and flash drives on the smartphone by mapping the `Z` and `C` drives as directories on the PC.

The combined files in SDK directories `epoc32\release\emulator_build, target\build_type\z` (e.g. `epoc32\wins\udeb\z`) and `epoc32\data\z` make up the simulated `Z` drive. Files are combined on a directory basis. For example, the simulated smartphone directory `z:\system\lib` directory for a Microsoft WINS UDEB build target will contain the combined files of the `epoc32\release\wins\udeb\z\system\lib` and `epoc32\data\z\system\lib` directories.

By default, the simulated C drive of the smartphone is mapped to the SDK's epoc32\wins\c directory.

Customizing Virtual Drives

The virtual drives can be customized via the EPOC_DRIVE_?***location*** statements in epoc.ini. For example, you can add a D drive to point to a specific PC directory. For example, you could add the following to epoc.ini:

```
EPOC_DRIVE_D c:\myMMCCard
```

This results in the simulated phone's D drive being mapped to c:\myMMCCard on the PC.

You can also change the C and Z drives to map to where you want, but note that for the Z drive the specified PC directory must be named z.

Memory Capacity

The default maximum heap size for your software running in the emulator is determined by the following statement in the epoc.ini file:

```
MegabytesOfFreeMemory  size_in_MB
```

If this statement is not there the emulator uses 64 MB. Check your SDK's epoc.ini to see what size is being used. You can change it as needed to simulate the limited memory conditions of the device. Some SDKs set it to realistic settings already. Series 60 v2.0, for example, sets this at 16 MB although you can set it lower to stress test low-memory handling functionality.

The emulator always claims 1 MB to account for general system usage, so to simulate an 8 MB device, use MegabytesOfFreeMemory 7.

Other Emulator Configurations

There are a variety of other settings in epoc.ini that you can use to customize emulator behavior. For example, you can define the text in the emulator title bar via the WindowTitle statement. You can also define virtual buttons and hot keys for the emulator, mapping them to key code events via the VirtualKey and KeyMap statements. The individual SDKs use these settings to simulate specific phones, so normally you would not modify them – however you may want to customize them in developing specific tests or demos.

Here is an excerpt from Series 60 `epoc.ini`:

```
# Series 60 in emulator title bar.
WindowTitle Series 60
# button at defined rect pixel area simulates 0 on keyboard.
VirtualKey 0       rect 126,568 64,28
# Following causes left Alt-1 to send EstdKeyDevice0
#  keycode (keycodes in e32keys.h).
KeyMap LeftAlt 1 EStdKeyDevice0
```

See the SDK documentation for more details of these configurations.

5.6.3 Emulator versus Device Functionality

The emulator behaves very similarly to a real device. The entire Symbian OS code is compiled for both the target device and emulator build targets using the same source code – with some required deviations (e.g. if you have any assembly language functions, you must obviously provide both x86 and ARM versions). So, not only can the emulator be used for GUI applications, but you can use it to develop system-level code.

Will everything that works on the smartphone, also work on the emulator? Not everything – no emulator is that good. But for the most part it is equivalent. Here are the main differences between the emulator and the device:

- Hardware

  The most obvious difference is that the underlying hardware of the emulator is different from that of the device. The PC processor instruction sets are different – the PC uses *x*86 and the device uses ARM – but this is easily hidden via the C/C++ language. More importantly, however, the peripheral hardware is different, so you cannot use the same device driver and hardware abstraction layer code on both. On the emulator, hardware accesses are mapped to appropriate Windows API calls.

- Pixels and fonts

  Although in most cases, the display of a GUI application will be very similar on the emulator and on the device, there are likely to be slight differences in pixel sizing between the two. For instance, it is possible for text to be truncated on the emulator and not on the real device, or vice versa. This can be an issue, if you rely on the emulator alone to perform language translation testing, for example.

- Static variables in DLLs

  Static variables are allowed in the emulator, but not in the real device. Be careful of this if you are doing most of your development on the emulator – you'll want to avoid having massive global variable search and destroy missions late in the project.

- Single process versus multiple processes

  The emulator runs as a single process, while the device supports the multiprocessing capabilities of Symbian OS. In Symbian OS v8 and above, this difference is hidden and the APIs that start and control processes are emulated. However, in Symbian OS versions before that, you need to have special logic (enclosed by `#ifdef __WINS__`) that uses threads to emulate the processes. Thankfully, there is not much code needed to implement this. Chapter 9 shows an example of transforming your process to run on a pre-v8.0 emulator.

5.7 Building DLLs

In this section, I show how to build DLLs and the issues involved. Some aspects of building a DLL can be confusing at first, but once you understand how it works, and the issues have been addressed, you'll find it straightforward to use.

5.7.1 `mmp` File for DLL

To build a DLL, set `targettype` to `dll` in your DLL's `mmp` file. Also set the first number in the `UID` statement to indicate the type of DLL to build. The static interface DLL is the most popular and basic DLL, and is what we'll cover in this section. For this DLL type, set the UID to 0x1000008d. Example 5.2 shows a sample DLL `mmp` file (from Chapter 4, Example 4.19):

Example 5.2. DLL mmp File

```
//MMP

TARGET     mydll.dll
TARGETTYPE dll
UID        0x1000008d 0x10004264

SOURCEPATH ..\src
SOURCE     mydll.cpp

USERINCLUDE       .
USERINCLUDE       ..\include
SYSTEMINCLUDE     \Epoc32\include

EXPORTUNFROZEN
```

Once you create the mmp, run the bldmake bldfiles and abld build ***build_target build_type*** commands as you would with other projects. At some point, you will also need to use the abld freeze command to freeze the interface to your DLL for release. I discuss this in more detail in Section 5.8. During development however, you should add EXPORTUNFROZEN to your mmp file (as indicated in Example 5.2) to disable interface freezing. The abld freeze command is not required when this option is set.

Building a DLL produces two outputs: the DLL itself and the import library (this is a LIB file). Both outputs are placed in the build platform's release directory (e.g. epoc32\release\armi\urel). The emulator can load and use the DLL directly from that directory. On the target phone, DLLs should be placed in the /system/lib directory.

Programs that use the DLL need to statically link to the import library. Adding the DLL's import library name to the LIBRARY statement in the program's mmp file will allow this. The program will then access the DLL functions through this import library.

Figure 5.2 shows the relationship between the import library and the DLL.

The DLL's import functions (simple wrappers whose only job is to invoke the actual code contained in the DLL) reside in the DLL user's executable since the import library is statically linked. To illustrate this,

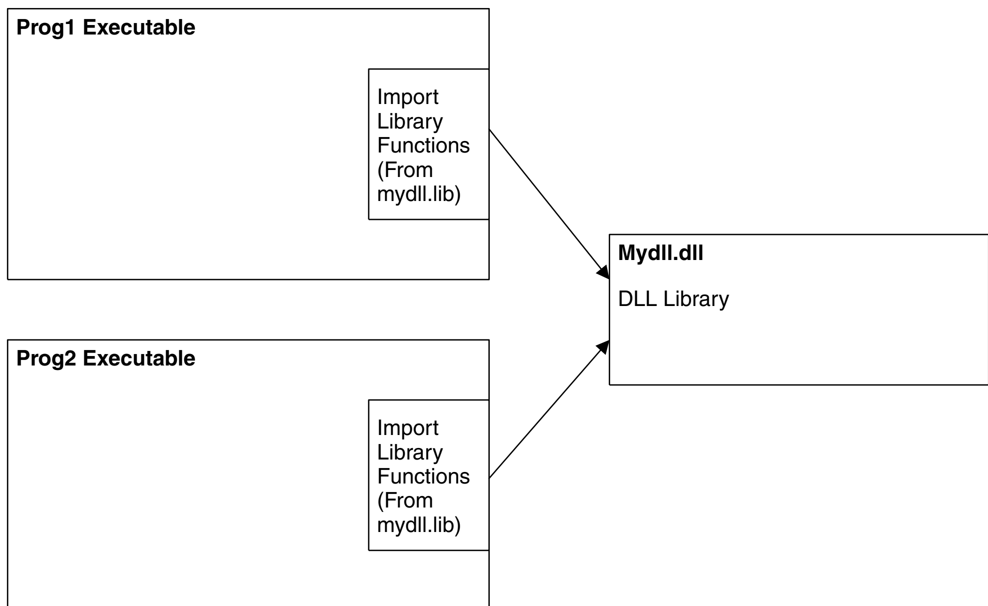

Figure 5.2 Import and DLL Libraries

executing a library function called `myDllFunc1()` first invokes it in the import library, which then locates and invokes it in the DLL. The import library will also load the DLL if necessary.

Note that, on the smartphone device, when a DLL is loaded, it is searched for in the `/system/lib` directory starting at the highest drive letter before `Z`, then moving down until it reaches the lowest drive letter, finally checking the system ROM `Z` drive. Therefore, you can copy a `dll` in the `/system/lib` drive on `C` and have it override the one on ROM (I would not suggest this). Or you can put it on a memory card (e.g. `D drive`) and it will override the ones on `C` and `Z`.

5.7.2 Referencing Functions by Ordinal

To understand the discussions that follow, you need to understand how functions within an import library invoke their DLL function counterparts.

As you saw in Chapter 4, `EXPORT_C` and `IMPORT_C` are used to indicate which functions in a DLL are exported (i.e. available for outside use). When the DLL is built each exported function is assigned a unique integer value known as an *ordinal*. DLL functions are invoked at runtime using these ordinal values. To illustrate this, suppose that `myDllFunc1()` is assigned an ordinal of 5 when the DLL is built. The import library's `myDllFunc1()` will invoke the corresponding `myDllFunc1()` function in the DLL by looking up function number 5 in that DLL (by using `RLibrary` class's `Lookup()` method described in the next section) and executing it.

The import library and the DLL ordinal numbers must line up. Imagine the trouble you will have if you update your DLL, and rebuild it such that `myDllFunc1()` has a different ordinal value. If applications linked with the older import library are run – and the `myDllFunc1()` import function is called – the wrong function in the DLL will be invoked (i.e. whatever function is now at ordinal 5). This situation is exactly what the interface-freezing feature of Symbian OS is meant to prevent, as you will see.

5.7.3 `RLibrary` API Class

Symbian OS provides an API class called `RLibrary` to load and invoke DLL functions at runtime. The import library uses this API to access the DLL and so it's instructive to have a quick look at this class, even though the import library shields you from needing to use it.

The key methods of `RLibrary` are `Load()` and `Lookup()`. The `Load()` method is used to load a specific DLL and associate it with the class. The `Lookup()` method will look up the DLL function with the ordinal value passed to it.

The following is a simple example of using `RLibrary`. It loads a DLL called `MyDll.dll` and calls the DLL function whose ordinal value is 1:

```
RLibrary lib;

lib.Load(_L("MyDll.dll"));
TLibraryFunction MyFirstFunc=library.Lookup(1);

MyFirstFunc();

lib.Close();
```

In addition to its use by system code to access import libraries, an application can make explicit use of the `RLibrary` class. Explicit use is necessary when using DLLs that act as plug-ins, such as polymorphic DLLs. See Chapter 4 for more information.

5.8 DLL Interface Freezing

DLL freezing is a mechanism to ensure that newly released DLLs will remain backward compatible with previously released versions of the DLL's import libraries. It works by ensuring that the function ordinals currently available in a released import library correspond with the ordinals used by future versions of the DLL, even when new functions are added.

Why Is DLL freezing important?

Imagine that you are developing a DLL that will be released for widespread use. On initial shipment, you release the DLL itself and a corresponding import library. Now imagine that several companies use your DLL, by linking their application with your import library. Now imagine that you want to update your DLL and rerelease it. If you modify the DLL such that the function ordinals change, then that DLL is no longer compatible with the applications that use your DLL, since the import library they link to uses the previous function ordinals.

Of course, you could release a new import library so that all the applications can be relinked, but this is not very realistic. The end-user will not want to load and install new revisions of all the applications that use the DLL in order for them to continue working (not to mention that if they forget to upgrade one, it will be likely to crash, or to do other harm).

To solve this issue, you freeze the DLL interface before your initial release. From then on, as you make changes to your DLL, the build tools will keep the DLL ordinal numbers assigned to the same function names, as specified in the frozen interface. If you add new functions they are assigned new ordinal numbers, with values above the existing ones in the frozen interface.

The old import library will, of course, not provide access to the new functions, but an application can still find the older ones that it uses, in the same place as before.

If you were to delete a function in your DLL the order would be lost – but the good news is that this is considered a violation of the frozen interface and will generate a build error until you refreeze the interface.

Disabling Interface Freezing

Interface freezing should be disabled in early development. Add `EXPORTUNFROZEN` to your DLL's `mmp` file to do this. Building in this mode is straightforward – you run `bldmake bldfiles` and `abld build` ***`build_target build_type`*** and it generates an updated DLL and import library. However, since the interface is not frozen, all applications that use your DLL must be relinked to the updated import library because the previous import library's backward compatibility is not guaranteed. This is because the ordinals assigned to exported functions are free to change with each build, so only using the import library and DLL produced from the same build is safe.

However, having the DLL's ordinals change during early development is not an issue since you have control over the applications that use the DLL. Many times you will build both the DLL and the applications that use it together at this stage, and the new import library will be picked up automatically.

Enabling Interface Freezing

In the later stages of DLL development, you'll want to enable interface freezing in your builds and freeze the DLL's exports each time you release. This will ensure that updates to your DLL will remain backward compatible with previously released import libraries.

How do you do this? Remove the `EXPORTUNFROZEN` statement in your `mmp` file. With this statement removed, the build will require that the DLL interface (i.e. exported function ordinals) be frozen. Then you perform an interface freeze with the `abld freeze` command each time you release a new DLL and import library.

What Does the `abld freeze` Command Do?

It creates a `def` file that records the current exported interface of the DLL. The `def` file defines the frozen interface by listing each exported function name along with its ordinal number.

How Is the DEF File Used?

With freezing enabled, each DLL project is associated with a `def` file (see the note at the end of this section for how it is associated) that defines

where the `abld freeze` writes to. The build then uses this `def` file in the following situations:

- Linking the DLL

  When the DLL is built, the project's frozen `def` file is consulted. The linker will ensure that all functions specified in the `def` file will remain at the same specified ordinal position ensuring backward compatibility with the import library produced after the last freeze. New functions in the DLL will receive new, higher-numbered ordinals (which will be added to the `def` on the next freeze).

- Generating the import library

  With interface freezing enabled, the import library is directly generated using the DLL interface defined in the project's `def` file. While interface freezing is disabled, the import library is always generated using the interface from the just-generated DLL.

  This is why the first build of a DLL will fail to generate an import library if an `abld freeze` command was not done – no `def` file yet exists.

- Associating a `def` file with your project

  The name of the def file associated with your project defaults to `<your_projectname>U.DEF` and is located in your project's `BARM` (for ARM build targets) or `BWINS` (for emulator build targets) directory. U stands for Unicode build in your BARM directory. You can specify a new name and location for this file by using the `DEFNAME` statement in the DLL's `mmp` file. Also, `nostrictdef` can be added to the `mmp` file to cause the `U` not to be added to the def file name.

There is an additional `def` file that is generated on each DLL build, and this should not be confused with the `def` file discussed above. This file is an intermediate file, located in the project's `epoc32/build` directory, and always reflects the current interface of the DLL. When interface freezing is disabled (`EXPORTUNFROZEN` in your `mmp`) the import library is generated from this intermediate `def` file.

First Build of a DLL

A typical first set of commands to build a DLL without the `EXPORTUNFROZEN` statement is as follows:

```
cd  <your dll build directory>
bldmake bldfiles
abld build wins
abld freeze wins
abld build wins
```

Doesn't it seem strange to run `abld build wins` twice? Actually, what is happening is that the first `abld build` command will successfully build the DLL, but will not build an import library since the interface is not frozen. Executing the `abld freeze` command will examine the interface of the DLL just built and freeze it (by recording the interface in a `def` file, as we will see). The next `abld build` command will successfully generate the import library corresponding to the DLL.

> You could substitute `abld library wins` in place of the last `abld build wins` command. This command builds the import library from the `def` file just created by the `abld freeze wins` command. `abld build` does this, but it also does a complete DLL build.

Now the DLL can be updated as needed and rebuilt. This DLL will be backward compatible with the last frozen interface and thus will work with those older frozen import libraries.

Sample `DEF` File

Example 5.3 shows the source code for a sample DLL.

Example 5.3. Source code for the sample DLL

```
#include <e32base.h>

class CMyClass : public CBase
  {
public:
  IMPORT_C CMyClass(void);
  IMPORT_C void FuncA();
  IMPORT_C void FuncB();
  IMPORT_C void FuncC();
  IMPORT_C void FuncD();
  IMPORT_C void FuncE();
  };

#include "Mydll.h"

EXPORT_C TInt MyTest()
  {
  return 0;
  }

GLDEF_C TInt E32Dll(TDllReason /*aReason*/)
  {
  return(KErrNone);
  }

EXPORT_C CMyClass* NewL()
  {
  return new (ELeave) CMyClass;
  }
```

```
EXPORT_C void CMyClass::FuncA()
  {
  /* FuncA code */
  }
EXPORT_C void CMyClass::FuncB()
  {
  /* FuncB code */
  }
EXPORT_C void CMyClass::FuncC()
  {
  /* FuncC code */
  }
EXPORT_C void CMyClass::FuncD()
  {
  /* FuncD code */
  }
EXPORT_C void CMyClass::FuncE()
  {
  /* FuncE code */
  }
EXPORT_C CMyClass::CMyClass()
  {
  }
```

Example 5.4 shows a sample `def` file generated (by `abld freeze`) for the sample DLL. The `MyClass` constructor is assigned ordinal 1, and functions `FuncA()` through `FuncE()` are assigned ordinals 2 through 6. The function `MyTest()` was assigned ordinal 7 and the `NewL()` function of MyClass was assigned ordinal 8.

Example 5.4. `def` File

```
EXPORTS
    ??0CMyClass@@QAE@XZ @ 1 NONAME ; public: __thiscall
            CMyClass::CMyClass(void)
    ?FuncA@CMyClass@@QAEXXZ @ 2 NONAME ; public: void __thiscall
            CMyClass::FuncA(void)
    ?FuncB@CMyClass@@QAEXXZ @ 3 NONAME ; public: void __thiscall
            CMyClass::FuncB(void)
    ?FuncC@CMyClass@@QAEXXZ @ 4 NONAME ; public: void __thiscall
            CMyClass::FuncC(void)
    ?FuncD@CMyClass@@QAEXXZ @ 5 NONAME ; public: void __thiscall
            CMyClass::FuncD(void)
    ?FuncE@CMyClass@@QAEXXZ @ 6 NONAME ; public: void __thiscall
            CMyClass::FuncE(void)
    ?MyTest@@YAHXZ @ 7 NONAME ; int __cdecl MyTest(void)
    ?NewL@@YAPAVCMyClass@@XZ @ 8 NONAME ; class CMyClass * __cdecl
            NewL(void)
```

Inserting a New Function

In the example just discussed, suppose you insert a new class method, say `FuncC_1()`,after `FuncC()`. If freezing were disabled, the ordinals could change in the next build; however, since the DLL is frozen, you can be assured that the ordinals will remain the same and `FuncC_1()` will

receive the next higher ordinal (9). Note that the frozen import libraries will not have access to this new function yet (you need to refreeze and release a new import library to use it), but at least the other functions will still work correctly.

Interface Violation

Let's consider what would happen if you froze the example DLL, and then you removed one of the methods (`FuncA()`, for example). The next time you built the DLL, you'd see an error such as the following:

```
MAKEDEF ERROR: 1 Frozen Export(s) missing from object files:
  \SYM_PR\~1\STAGE\SERIES60\SIMPLEEX\BWINS\MYDLL.DEF(3) :
?FuncA@CMyClass@@QAEXXZ @2
NMAKE : fatal error U1077: 'perl' : return code '0xff'
Stop.
```

The reason is that backward compatibility would be broken since existing applications may depend on the function you deleted.

Unfreezing a DLL

In some cases you will have frozen the DLL (without releasing it) and then want to rearrange things, by either renaming methods or deleting them. You will not be able to build, however, since it will violate the interfaces and you will get errors such as the one just discussed. To reset, unfreeze your interface by deleting your project's `def` file. Then run `abld build`, `abld freeze`, `abld build`. This will create a new `def` with the new interface.

5.9 Installing Applications on the Smartphone

An application is installed on a smartphone via an installation file that has a *sis* suffix. This installation file, which is referred to as a `sis` file, contains all the executables and data files for the application. In addition it contains installation information, such as where to put each executable/data file on the target device's flash memory.

You can install a `sis` file in several ways:

- Using PC suite on the PC.
- On UIQ-powered smartphones, you can simply click on the `sis` file in File Explorer and it will install itself on the smartphone.
- Download `sis` files from the web or via email onto the smartphone itself and install them.
- Beam the `sis` file to the phone using infrared or Bluetooth technology.

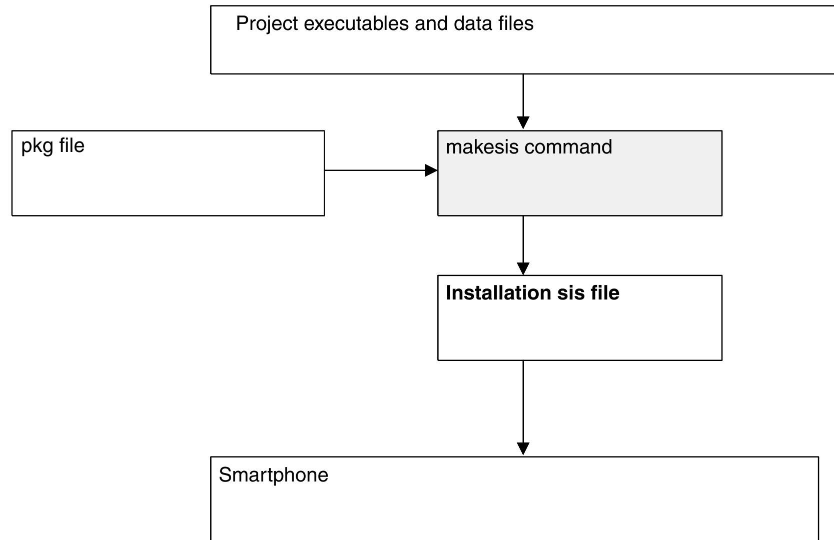

Figure 5.3 makesis Flow

The smartphone keeps track of all installed programs, and allows the user to uninstall them. The information from the `sis` file is used to determine which files to delete.

To create a `sis` file for your program, Symbian provides a tool called `makesis`. `makesis` uses a package definition file (a text file) as input, which specifies what files are included in the package (and their location on the PC) and where these files go on the smartphone. The package definition file has a suffix of `pkg`. Running `makesis` with the argument set to the name of the `pkg` file will generate the `sis` file – ready for installation.

Figure 5.3 shows the operation of `makesis`.

5.9.1 Where Do I Put My Files?

Let's look at the key directories on the smartphone device. The directory structures differ somewhat between phones, but the ones listed here are consistent across all devices.

- `/system/apps`

  This is where applications reside. Each application has its own subdirectory. The application's executable and resource files are placed in this directory.

- `/system/data`

  This is where configuration files are normally put.

- `/system/libs`

  All DLLs go in this directory.

5.9.2 Format of the PKG File

The real work in creating a sis file is in creating the pkg file for your project. First I will discuss the basic statements in the pkg file using a simple illustrative example. Then I will talk about some of the more advanced features.

A minimum pkg file has two lines describing the application and the target smartphone device. These lines are followed by one or more lines that specify what files on the development PC go in the package file, and where those files should be placed on the target device when the sis file is installed. Example 5.5 shows the pkg file from Chapter 2.

Example 5.5. Series 60 Example Package File

```
; SimpleEx.pkg - Series 60
;

; standard SIS file header
#{"SimpleEx"},(0x10005B94),1,0,0

;Supports Series 60 (all versions)
(0x101F6F88), 0, 0, 0, {"Series60ProductID"}

;
"c:\Symbian\6.1\Series60\epoc32\release\armi\urel\simpleEx.APP"-
        "!:\system\apps\simpleEx\simpleEx.app"
"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps
        \simpleEx\SimpleEx.rsc"-"!:\system\apps\simpleEx\SimpleEx.rsc"
```

Let’s look at the format of these lines in more detail.

Package File Comments

Lines that begin in a semi-colon and blank lines are ignored by makesis.

Package Header

The package header contains information about the program that you are installing. The format is as follows:

```
#{" program name"},{ProgramUID},Major_Version_#,Minor_Version_#,
        Build_#[,package options] [,Type=Package Type]
```

In Example 5.5, the package header is:

```
#{"SimpleEx"},(0x10005B94),1,0,0
```

This indicates that the program's name is `SimpleEx`, the UID is 0x10005B94, and the version is 1.0.0. The program's name and version are displayed while installing (or uninstalling) the software. The name and version are also used to identify the program for reference by other package file commands (see Section 5.9.3).

All `sis` files must have a UID. In the example I used the one assigned to the application contained in the `sis` file, but, even if you are installing components that do not have a UID, one is needed here.

Package Options

There are two package options that can be specified in the package header: `SHUTDOWNAPPS (SH)` and `NOCOMPRESS (NC)`. The options can be spelled out in full, or the two-letter abbreviation can be used. The `SHUTDOWNAPPS` option will cause all applications to be closed on the smartphone before the installation starts. `NOCOMPRESS` will cause the files to be put in the `sis` file in an uncompressed format. Generally this should not be used.

The operation of these options may vary between phones, and they are not normally used. Our example does not define any package options.

Package Type

Package type should be set to indicate what kind of component is being installed. It is used for presenting this information to the user and applying certain characteristics or restrictions for installing or uninstalling.

The default package type is `SISAPP`, which indicates a standard stand-alone application.

The complete list of package types is:

- `SISAPP` indicates that this is an application (the default).
- `SISOPTION` indicates an optional component.
- `SISSYSTEM` indicates a shared component, such as a DLL, which may be used by multiple applications (it will not be removed until the last user is uninstalled).
- `SISCONFIG` configures an existing application. It does not appear in the list to uninstall.
- `SISPATCH` patches an existing component. The user may remove it.
- `SISUPGRADE` upgrades an existing component. Like `SISCONFIG`, this is not available for uninstall.

Product Target

The line after the package header indicates the target platform for the installation. Consider the product target line from Example 5.5:

```
(0x101F6F88), 0, 0, 0, {"Series60ProductID"}
```

The first number is the platform's UID, the last string is the name of the platform. This line indicates that the target is Series 60 version 0.9 (which will work on all Series 60 platforms).

Here are some common platform UIDs, along with product names:

Series 60 v0.9	0x101F6F88	Series60ProductID
Series 60 v1.0	0x101F795F	Series60ProductID
Series 60 v1.1	0x101F8201	Series60ProductID
Series 60 v1.2	0x101F8202	Series60ProductID
Series 60 v2.0	0x101F7960	Series60ProductID
UIQ v2.0	0x101F617B	UIQ20ProductID
UIQ v2.1	0x101F61CE	UIQ21ProductID

Select the lowest version of a target platform that your software works on. As an example, if you write a Series 60 application that works with any Series 60 platform, then you should use the v0.9 platform UID (0x101F6F88) so that the `sis` file will be able to be installed on smartphones with all versions of Series 60. Make sure it will work on the lower versions however. If, for example, you use Series 60 v1.2 APIs that work on v1.2 and v2.0, then use v1.2 (0x101F8202). If you use a lower one, problems will result if installed on pre-v1.2 phones.

On 9200 Series communicator, Series 80 platforms, no target platform line is needed.

Specifying Files to Install

To specify the files that will be installed, enter lines of the following format:

```
"source"-"destination"
```

where *source* specifies the file to include in the `sis` file, and *destination* indicates the name and location of the file when it is installed. For example, in Example 5.5, the line:

```
"c:\Symbian\6.1\Series60\epoc32\release\armi\urel\simpleEx.APP"-
        "!:\system\apps\simpleEx\simpleEx.app"
```

copies, into the sis file, a file named simpleEx.app from the c:\symbian\6.1\series60\epoc32\release\armi\urel\ directory on the PC. Then, on installation, this file is copied to the smartphone at \system\apps\simpleEx. In this example, the installed file has the same name, but you can, if necessary, change the destination name.

But what about the target drive – what is drive '!:'? When installing a sis file, the user is prompted to select which drive to install the software on. The '!:' just means to use the drive that is specified in response to this prompt. So, if the user selects C, then SimpleEx.app is placed in c:\system\apps\simpleEx. You can hardcode the drive too. For example, if you specify 'c:' instead of '!:' as the target path, then the file is always copied to C, irrespective of which drive the user selects.

Relative paths can also be specified for the source file in these statements. Such paths are relative to the location from which you run makesis. You can also use the makesis -d option to specify the directory you want relative file paths to be based on. For example, if you run the command:

```
makesis -d c:\Symbian\6.1\Series60
```

a file specified as '\epoc32\release\armi\urel\simpleEx.APP' will be interpreted as being relative to the Series 60 SDK path specified in the -d option. This can be useful if the software is built on different systems and you do not want to hardcode SDK paths in the pkg file.

5.9.3 Advanced PKG File Options

For most programs, the pkg file is simple, and similar to the minimum example described above. But there are some more powerful features of the installation tool that can be taken advantage of. I will not go through all of them here – see the SDK documentation for that – but I will go over a few useful and interesting ones.

Text Notices

You can specify a text file to be displayed to the user during the installation process. The file itself is not copied to the target. This is useful for displaying basic readme information or license agreements. To do this, add the line below to the pkg file:

```
"license.txt" - " ", FILETEXT, TEXTCONTINUE
```

FILETEXT indicates to display the file during install. TEXTCONTINUE will provide a continue button that will dismiss the text file and continue the installation.

Instead of `TEXTCONTINUE`, you can specify one of the following:

- `TEXTSKIP` displays a Yes/No option. If Yes is selected, installation continues. If No is selected, the next statement is skipped, but installation continues normally afterwards.
- `TEXTEXIT` displays a Yes/No option. If Yes is selected, installation continues. If No is selected, the installation stops and any files that have already been installed are removed.
- `TEXTABORT` displays a Yes/No option, but when No is selected, the installation just stops, without removing any installed files.

Removing Runtime-Generated Files

When you uninstall a program via its `sis` file, the uninstaller will remove all the files that were copied to the phone by the installer as specified in the file specification lines of the `pkg` file. But what if a file is generated at runtime? Since the file is not copied to the target during the installation, it is not listed for removal when the program is uninstalled. You can specify that you want such files to be removed by using the `FILENULL` specification as follows:

```
""-"C\system\data\my_runtime_generated_file",FILENULL
```

This indicates that no file is to be installed at the target location, but that the specified file is to be removed during the uninstall process.

Note that such a file is removed only during a true uninstallation, and not on update installations, where an old version of the program is first removed, then replaced by a later version. The assumption is that you will want to keep your existing runtime-generated files when installing a new version of your program.

Embedding `sis` Files

You can include another `sis` file within your `sis` file with the following line:

```
@"sis file name",{UID}
```

For example: `@"prog1.sis",{0x12341234}` installs *prog1.sis*, with UID 0x12341234, at the point where this line is encountered.

Note that, on uninstallation, this embedded `sis` file will not be uninstalled until the system determines that no other currently installed components use it (i.e. there is no other installed component that also includes that `sis` file in its `pkg` file).

Running Executables on Install or Uninstall

You can specify that an executable be run during an installation by adding `FILERUN (FR)` and `RUNINSTALL (RI)` keywords at the end of the executable's file specification line. For example:

```
"\Symbian\6.1\Series60\Epoc32\release\armi\urel\myprogram.exe"-
        "!:system\programs\myprogram.exe",FR,RI
```

will install myprogram.exe and execute it during the installation.

The `RUNINSTALL` keyword can be replaced by either of the following alternatives:

- `RUNREMOVE (RR)` causes execution to occur only during uninstallation.
- `RUNBOTH (RB)` causes the executable to be run on both installation and uninstallation.

Any of these three options may be further qualified by use of the `RUNWAITEND (RW)` keyword, which causes the installation to wait for the executable to complete before continuing. If not specified, then installation continues immediately after the executable is launched.

Requisite Lines

You can use a requisite line to specify that a particular component must already be installed in order for the current installation to continue. It has the following format:

```
{UID}, Major_Version_#,Minor_Version_#,Build_#,{"Product Name"}
```

This means that the component with the specified UID and Product Name, with a version number not earlier than the one specified, must exist for the installation to continue.

For example:

```
{0x10000123},1,0,0,{"MyDll"}
```

indicates that a component named `MyDll`, with UID 0x10000123 and a version number of at least 1.0.0 must exist already before installation can proceed.

The requisite line should look familiar – it is how the target platform line is implemented. The example target platform line:

```
(0x101F6F88), 0, 0, 0, {"Series60ProductID"}
```

is a requisite statement that the 'component' named Series60ProductID, with a UID of 0x101F6F88, and version number 0.0.0 or higher, must exist in order for the installation to continue.

5.9.4 Language Support

Multiple translated versions of an application can exist within a single `sis` file. When a user installs a `sis file`, they are prompted to select which language they would like installed.

To specify the language variants that you want to be included, add a language line at the top of your `pkg` file. The language line begins with '&' and contains a list of comma-separated language codes from the following list.

AM – US English

AS – Austrian German

AU – Australian English

BF – Belgian French

BL – Belgian Flemish

CS – Czech

DA – Danish

DU – Dutch

EN – UK English

FI – Finnish

FR – French

GE – German

HK – Hong Kong Chinese

HU – Hungarian

IC – Icelandic

IF – International French

IT – Italian

JA – Japanese

NO – Norwegian

NZ – New Zealand

PL – Polish

PO – Portuguese

RU – Russian

SF – Swiss French

SG – Swiss German

SK – Slovak

SL – Slovenian

SP – Spanish

SW – Swedish

TC – Taiwan Chinese

TH – Thai

TU – Turkish

ZH – Prc Chinese

An example language line is:

```
&EN, FR, FI
```

which specifies that the `sis` file contains English, French and Finnish language variants.

If a language line is not included, `&EN` is assumed.

How Does `makesis` Use the Language Information?

So far, we have used only language-independent statements in the `pkg` file. Example 5.5 will install exactly the same regardless of an added language line, or a language selection by the user.

In order to use the language information, you must use language-dependent versions of the applicable `pkg` statements.

Language-Dependent Files

The first rule in internationalizing an application is to keep the language-dependent parts of your application separate (i.e. in different files) from the language-independent parts. For example, Symbian OS uses resource files to contain text strings, and a separate resource file would exist for each language. When the user selects a particular language to install, you want to install the appropriate resource file for that language.

As an example, the following pkg line specifies the installation of a resource file based on the language:

```
&EN, FR, FI
...
{"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps
        \simpleEx\SimpleEx.en",
"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps\simpleEx\SimpleEx.fr",
"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps\simpleEx\SimpleEx.fi",
}
-"!:\system\apps\simpleEx\SimpleEx.rsc"
```

In this example, makesis includes all three resource files in the sis file. However, the language chosen during the installation determines which file is actually copied to smartphone file \system\apps\simpleEx\SimpleEx.rsc. The order in which the source files are listed must agree with the order of the languages in the language statement – so that UK English chooses SimpleEx.en, French chooses SimpleEx.fr and Finnish chooses SimpleEx.fi.

Note that, if you use this language-dependent version of a file specification line, you must include a source file for each of the languages listed in the language line.

As another example, you could also have language-dependent versions of a text notice, such as:

```
{"license.en.txt",”license.fr.txt”,”license.fi.txt”} - " ", FILETEXT,
        TEXTCONTINUE
```

Other Language-Dependent Statements

When specifying multiple languages you will need to ensure that your product header provides a component name for each language, and that your target platform lines (as well as other requisite lines) provide product id strings for each language. Although it is common for the component name to be in English for each language variant, there still needs to be a string entered for each language, otherwise an error will occur when makesis runs.

Example 5.6 shows the pkg file of Example 5.5, after being modified to support multiple languages.

Example 5.6. pkg file supporting multiple languages

```
; SimpleEx.pkg – Series 60
;
&EN,FR,FI
; standard SIS file header
```

```
#{"SimpleEx",”SimpleEx”,”SimpleEx”},(0x10005B94),1,0,0

;Supports Series 60 (all versions)
(0x101F6F88), 0, 0, 0,
{"Series60ProductID",”Series60ProductID”,”Series60ProductID”}
"c:\Symbian\6.1\Series60\epoc32\release\armi\urel\simpleEx.APP"-
"!:\system\apps\simpleEx\simpleEx.app"

{"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps\
        simpleEx\SimpleEx.en",

"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps\
        simpleEx\SimpleEx.fr",

"c:\Symbian\6.1\Series60\...epoc32\data\z\system\apps\
        simpleEx\SimpleEx.fi",
} -"!:\system\apps\simpleEx\SimpleEx.rsc"
```

5.10 Switching Between SDKs

At some point, you may want to develop software for multiple smartphone models, and thus need to run multiple Symbian OS SDKs on the same PC. Here are some methods to switch between these SDKs.

Two main things must be set up in your environment in order to run an SDK. First, the environment must point to the proper SDK's tool directories so that the correct build commands can be invoked. Second, an indicator must be set so that the build tools themselves know the base directory of the SDK for finding include files, using the active SDK's `build` and `release` directories, etc. With SDKs that predate Symbian OS version 7.0, this is done by:

- setting your `PATH` environment variable to point to the SDK tool directories
- setting an environment variable called `EPOCROOT` to the location of the active SDK.

SDKs based on Symbian OS v7.0 and later, however, provide a command called `devices` that makes it easier to switch between SDKs. This works especially well if you use Symbian OS v7 or later SDKs. As you install these SDKs, the installer registers the SDK as an SDK 'device' and gives it a name. You can run `devices` on the command line to see the names of all your installed SDKs.

The following is a sample of the output of *devices:*

```
Series60_v20:com.nokia.series60 -  default
UIQ_70:com.symbian.UIQ
```

This shows that you have both the Series 60 v2.0 and UIQ SDKs installed on your machine. The 'default' indicates that the Series60 v2.0 is currently active.

To switch to an SDK, execute `devices – setdefault @<SDK name>`. From then on, Symbian OS builds will be done using that SDK. As an alternative to setting the SDK as default, you can specify the `@<SDK_name>` after each build command – but that is very awkward and using `setdefault` is more straightforward. In the preceding example, executing

```
devices –setdefault @UIQ_70:com.symbian.UIQ
```

switches your environment so that UIQ is the active project.

5.10.1 What if Some SDKs Predate Symbian OS v7.0?

If at least one of your installed SDKs is based on Symbian OS v7.0 or higher, then the `devices` command can be used to switch between all your SDKs. However, installing a pre-v7.0 SDK will not automatically add the SDK name for the `devices` command to use. You have to add the SDK manually by using the following option:

```
devices –add <location of epoc32\release directory> <location of
        epoc32\tools directory> @<SDK_NAME>
```

For example, the following statement adds a device that represents a Symbian OS v6.0 Series 80 (Nokia Communicator) SDK.

```
devices –add c:\symbian\6.0\NokiaCPP c:\symbian\6.0\Shared
        @Series80_9200:com.nokia.series80
```

5.10.2 How Does the `devices` Command Work?

When an SDK is added – either by installing a Symbian OS version 7.0 or greater SDK, or by executing the `devices –add` command – an entry is added to a file called `devices.xml`, located in `\Program Files\Common Files\Symbian`. This file contains a list of all installed SDKs, including their names and where they are installed. The directory `\Program Files\Common Files\Symbian\Tools` contains a small stub for each Symbian OS tool. When a tool stub is called, it refers to the `devices.xml` file in order to set `EPOCROOT` to point to the active SDK's location and get the location of the active SDK's tools directory and invoke the actual tool.

Therefore, when using `devices`, your `PATH` no longer points directly at your SDK tool directories, but at `\Program Files\Common Files\Symbian\Tools` – for all SDKs. The tools stubs will ensure that the correct SDK tools are called. Make sure that this directory is first in your path – other directories in your path that point directly to your SDK (perhaps left over from an installation of a pre-v7.0 SDK) will prevent the `devices` command from switching SDKs.

For backward compatibility, setting `EPOCROOT` manually before invoking the build tools will override the settings in `devices.xml`. A common problem encountered after installing a Symbian OS v7 SDK is that your environment may still have `EPOCROOT` set, from a previous pre-v7.0 SDK installation. Since this `EPOCROOT` setting overrides the settings in `devices.xml`, your new SDK will not work properly when you switch to it with the `devices` command (the `devices` command is, in effect, ignored). To solve this, make sure that your command line environment does not initialize `EPOCROOT`.

5.10.3 What If All SDKs Predate Symbian OS v7.0?

In this case, it's best to create a batch file to switch between them. The batch file should update the PC's path to point to the SDK's tool directories (`<`**`SDK_PATH`**`>\epoc32\tools;<`**`SDK_PATH`**`>\epoc32\gcc\bin`) and set `EPOCROOT` to point to the SDK's location.

For example, to set up a Series 80 v6.0 SDK:

```
set EPOCROOT=\symbian\6.0\NokiaCpp\
set PATH=c:\symbian\6.0\Shared\epoc32\tools;
    c:\symbian\6.0\Shared\epoc32\tools\gcc\bin;%PATH%
```

6

Strings, Buffers and Data Collections

This chapter covers the basic string and data buffer APIs, as well as other common data organization classes. These classes are part of what is known in Symbian OS as the base APIs, and they reside in `user.dll`.

This chapter covers the following types of data classes:

- Descriptors for handling strings and binary data
- Dynamic buffers for buffers that grow at runtime
- Array classes
- Other data organization classes like linked lists and circular queues.

The chapter includes numerous examples, and the complete source of the examples can be downloaded from the book's website. The examples output their results via a `printf()` style function, to what is known as a *text console*.

Before diving into string and buffer management, let's take a look at how a text console program works. This provides an easy way to compile the examples in this chapter and do experiments of your own without writing a full GUI program.

6.1 Introducing the Text Console

Symbian OS provides a text console API class called `CConsoleBase` that allows you to output formatted text to the screen, without the overhead of using the GUI framework. The class also accepts keyboard input. While the text console is not very useful for product software, it's excellent for learning and experimenting with non-GUI related Symbian OS functionality.

Below is a very minimal console `cpp` file that outputs 'Hello' to the text console so you can get the general idea:

```
#include <e32base.h>
#include <e32cons.h>
```

```
CConsoleBase* console;

TInt E32Main()
  {
  _LIT(KName,"Tests");
  _LIT(KAnyKey,"[Press any key]");

  console=Console::NewL(KName,TSize(KConsFullScreen,KConsFullScreen));

  console->Printf(_L("Hello\n"));

  console->Printf(KAnyKey);
  console->Getch();

  delete console;
  return(0);
  }
```

Symbian OS uses `_LIT` and `_L` to define string literals. We will discuss them in Section 6.2.3.

`CConsoleBase::Printf()` works in much the same way that a standard C `printf()` function works – it accepts a format string, and a variable number of arguments to output using the specified formatted string. The above example shows the simplest possible form, with no format elements or arguments beyond the text string itself. The format string has the same syntax as the C `printf()` format string, but has some extra, Symbian OS-specific format identifiers. For example, `%S` is used to print the contents of a string descriptor – you will use that one frequently.

The method `CConsoleBase::Getch()` is used to wait for and retrieve a key from the keyboard, although the return value is discarded.

Example 6.1 shows an expanded version of a console program that provides a general framework for running experiments, including the examples in this chapter.

Example 6.1. Expanded Console Framework (tests.cpp)

```
#include <e32base.h>
#include <e32cons.h>
CConsoleBase* console;

void RunExampleL()
  {
  console->Printf(_L("Example Code\n"));

  // Add example code here

  }

void RunConsoleL()
  {
  _LIT(KName,"Tests");
  _LIT(KAnyKey,"[Press any key]\n");
```

```
  console=Console::NewL(KName,TSize(KConsFullScreen,KConsFullScreen));
  CleanupStack::PushL(console);
  RunExampleL();
  console->Printf(KAnyKey);
  console->Getch();
  CleanupStack::PopAndDestroy(console);
  }

TInt E32Main()
  {
  __UHEAP_MARK;
  CTrapCleanup* cleanupStack = CTrapCleanup::New();
  TRAPD(error,RunConsoleL());
  __ASSERT_ALWAYS(!error,User::Panic(_L("Example"),error));
  delete cleanupStack;
  __UHEAP_MARKEND;
  return(0);
  }
```

Example 6.1 creates a cleanup stack for your test code to use if needed. It also traps leaves that occur in your test code.

`__UHEAP_MARK` and `__UHEAP_MARKEND` are useful macros that detect memory leaks on the heap. When `__UHEAP_MARK` is called, the heap level is internally recorded. Then when `__UHEAP_MARKEND` is called, if the current heap level does not match (i.e. there are allocations on the heap that were not there when `__UHEAP_MARK` was called), then an `ALLOC` panic is generated. We used them in the above code so that you can see if any of the test code you entered did not properly free up allocated memory. Note that these macros are only used in debug builds and are ignored otherwise. They are useful in catching memory leaks.

The `mmp` file for the console project is shown in Example 6.2.

Example 6.2. Console `mmp` file

```
TARGET        tests.exe
TARGETTYPE    exe
UID           0

SOURCEPATH    .
SOURCE        tests.cpp

USERINCLUDE   .
SYSTEMINCLUDE \Epoc32\include

LIBRARY       euser.lib
```

To compile this console program, use the following commands (you only need to issue the `bldmake` command once to set up the makefiles):

```
bldmake BLDFILES
abld build wins udeb
```

Figure 6.1 Console output

Substitute your particular build target for `wins` (e.g. `winscw` if you are using Code Warrior), if required.

This creates a windows executable called `tests.exe` in the `%EPOCROOT%\epoc32\release\wins\udeb` directory. If you run this executable, it brings up the emulator and immediately runs your console app. Figure 6.1 shows the output for `tests.exe` from Example 6.2. To simplify running, you can create a batch file in the same directory as the source file, which executes the `exe` in the release directory (so you do not have to change to that directory).

Now, let's get on with our discussion of strings and buffer management, starting with the most commonly used Symbian OS data classes: descriptors.

6.2 Descriptors for Strings and Binary Data

Descriptors are classes that represent data buffers and allow you to safely access them. Symbian OS uses descriptors to store and manipulate strings (as opposed to `NULL`-terminated C strings), as well as to manage binary data. Descriptor classes, although containing many features, are optimized for minimal overhead, since they are designed to run on memory-constrained devices.

You'll need to thoroughly understand descriptors in order to develop Symbian OS code, since they are so widely used. In fact, you'll need to use them just to call many of the Symbian OS API functions, since descriptors are often passed to them as arguments. Descriptors are powerful, but since their use is so unique when compared to other operating systems, they can be a source of confusion to programmers starting out in Symbian OS.

A descriptor class encapsulates a data buffer as well as the buffer's size – the size being used to prevent buffer overruns. There are multiple descriptor classes you will need to be familiar with – these classes differ in how the data buffer is stored and referenced, as well as the width of the buffer's data and whether the buffer is modifiable or not. Also, descriptor classes contain numerous methods that allow you to read and write the buffers as well as transform the data, using an interface consistent across the different descriptor types.

6.2.1 Strings Versus Binary Data

Both strings and binary data buffers are treated as data buffers of a specific length. Of course, if your descriptor contains binary data, then the string manipulation methods of the descriptor (e.g. `LowerCase()`) are not applicable. Another difference is that strings are usually stored in 16-bit descriptors while binary data is stored in 8-bit descriptors. This is because Symbian OS uses Unicode and thus deals with 16-bit characters. For binary data, however, 8-bit descriptors are normally used, since the binary data is treated as simply a buffer of bytes.

6.2.2 Preventing Memory Overruns

A memory overrun occurs when your software writes past the end of an allocated buffer. The worst thing about a memory overrun is that it will often go unnoticed at first and then manifest itself later as an intermittent crash – often in functions far removed from where the overrun occurred. As a result, a memory overrun can be extremely hard to debug and they always seem to occur close to – or after – product release.

A big advantage of a descriptor is that it can prevent data from being written outside of the allocated buffer. When an access is attempted beyond the buffer limit, the descriptor generates an exception when the actual overrun occurs, making it significantly easier to find and fix the problem. However, nothing will prevent a memory-overrun attempt, so you need to avoid such attempts and test vigorously to avoid having these exceptions occur in your product.

6.2.3 Simple Descriptor Example

Before describing the descriptor classes in detail, let's look at a simple string example – comparing its implementation both in C and in Symbian OS using descriptors.

The example implements a function called makeName() which concatenates the string passed as its argument to the string literal 'Name: ', and prints the results.

First, let's look at the example, written using C strings:

```
char *namePrefix="Name:";
void makeName(char *name)
  {
  char str[80];
  strcpy(str,namePrefix);
  strcat(str,name);
  printf("str=  %s ",str);
  }

void MainFunc()
  {
  makeName("Sharon");
  }
```

In C, strings are represented as a set of characters terminated by a NULL. The literal namePrefix is declared as a char * and assigned the string 'Name: ' – a literal stored in the code image. makeName() accepts its string argument as a char *. It declares a temporary string buffer as a char array (str) and then uses strcpy() and strcat() to copy the name prefix and append the name passed to the function into the temporary string. When the code invokes makeName(), it passes its string argument as a quoted string.

Now let's look at the same example rewritten to use descriptors:

```
_LIT(KNamePrefix,"Name:");

void makeName(const TDesC& aName)
  {
  TBuf<80> str;
  str.Copy(KNamePrefix);
  str.Append(aName);
  console->Printf(_L("str = %S\n"),&str);
  }

void MainFunc()
  {
  makeName(_L("Sharon"));
  }
```

The first thing to note is how string literals are declared. In Symbian OS, string literals are declared as descriptors using either the _LIT macro or the _L macro.

The prefix string literal is declared as:

```
_LIT(KNamePrefix,"Name:");
```

The _LIT macro is called to take the string "Name:" and stores both the string (no NULL) and the string's size in the descriptor literal KNamePrefix.

Also notice that the example invokes makeName() as follows:

```
makeName(_L("Sharon"));
```

The _L macro is like _LIT except that this one does not assign an intermediate constant as _LIT does. I discuss other differences between these macros in Section 6.3.2.

makeName() accepts its string argument as a descriptor instead of a char *:

```
void makeName(TDesC &aName)
```

There are several different types of descriptor classes (see Section 6.3), and TDesC is the base class of all descriptors – thus declaring the argument in this way ensures that the function will accept any type of descriptor.

In the C example, the temporary string in makeName() was declared as an array of 80 characters as follows:

```
char str[80];
```

In Symbian OS, str is declared as a descriptor instead:

```
TBuf<80> str;
```

TBuf is a 16-bit modifiable descriptor class with a maximum size (specified as the template parameter) of 80 characters. Like an array, TBuf stores the string buffer on the stack.

The example then builds the final string into the temporary descriptor by copying the name prefix into the temporary descriptor and appending the passed name as follows:

```
str.Copy(KNamePrefix);
str.Append(aName);
```

The Copy() and Append() descriptor methods are the counterparts to C's strcpy() and strcat() functions. Copy() here copies the specified

descriptor data `KNamePrefix` to `str`'s descriptor buffer, replacing anything that's there. `Append()` appends the descriptor string data in `aName` to `str`'s buffer.

If the name passed into `makeName()` was large enough such that `str` exceeded 80 characters, the C version of the code would overrun its buffer. However, the descriptor version will immediately panic if the string exceeds 80 characters, since the `Copy()` and `Append()` methods know that the size allocated to the descriptor is 80 characters.

6.3 The Descriptor Classes

In this section, we look closely at all of the descriptor classes.

There are 10 descriptor classes available for the programmer to use. These are divided into three types: *buffer, pointer* and *heap*. Buffer descriptors contain their data buffers in the descriptor classes themselves; their class names begin with `TBuf`. Pointer descriptors contain a pointer to a data buffer located outside the descriptor; their names begin with `TPtr`. Heap descriptors are used for managing descriptors on the heap. Heap descriptor names begin with `HBuf`.

A descriptor can be modifiable or non-modifiable. A `C` (for constant) is appended to the class names mentioned above to indicate that the descriptor is non-modifiable.

Also, a descriptor buffer can contain 8-bit or 16-bit data. Adding 8 or 16 at the end of the class name indicates this. So, for example, `TBufC16` is a 16-bit non-modifiable buffer descriptor.

Listed here are all the descriptor classes that can be instantiated. These classes are directly instantiated without inheritance and they are defined in `e32des8.h` and `e32des16.h` in the `%EPOCROOT%\epoc\include` directory.

- `TBuf8<n>`: modifiable, 8-bit buffer descriptor, n is the buffer size
- `TBuf16<n>`: modifiable 16-bit buffer descriptor, n is the buffer size
- `TBufC8<n>`: non-modifiable 8-bit buffer descriptor, n is the buffer size
- `TBufC16<n>`: non-modifiable 16-bit buffer descriptor, n is the buffer size
- `TPtr8`: modifiable 8-bit pointer descriptor
- `TPtr16`: modifiable 16-bit pointer descriptor
- `TPtrC8`: non-modifiable 8-bit pointer descriptor
- `TPtrC16`: non-modifiable, 16-bit pointer descriptor

- `HBufC8`: non-modifiable, 8-bit heap descriptor
- `HBufC16`: non-modifiable, 16-bit heap descriptor

16-bit Default for Unicode

Most times you will see strings represented with descriptor classes with no data width appended to the class name (e.g. just `TBuf`). If you leave the data width off the class name, it defaults to a 16-bit descriptor. Actually the default depends on whether the build is using a 16-bit Unicode character set or not (remember descriptors are used mainly for strings). But since all current Symbian OS platforms use Unicode, then the default is always 16-bit. Examine the include file `e32std.h` if you are interested in how this default mapping to the descriptor classes is accomplished.

Since 8- and 16-bit descriptors behave identically in almost all respects, I will use the default 16-bit descriptors (by leaving the number off) for simplicity for most of this chapter. Unless stated otherwise, you can assume that the code for 16-bit descriptors applies also to 8-bit descriptors.

Descriptor Class Hierarchy

Figure 6.2 shows a class diagram of the descriptor classes. As indicated above, the diagram shows 16-bit classes only – there is a separate, but

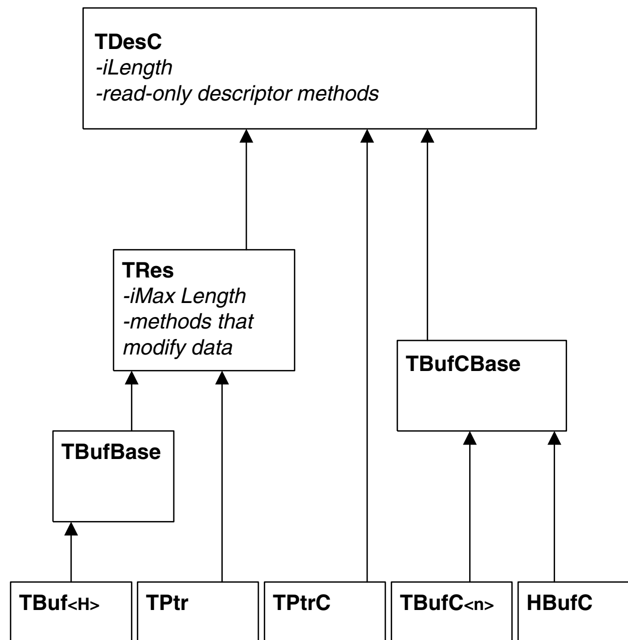

Figure 6.2 Descriptor Class Diagram

equivalent hierarchy for 8-bit descriptors that start with TDesC8 (just tack an 8 on every class name, and you'll have it).

6.3.1 Descriptor Base Classes

TDes and TDesC are the base classes for descriptors, and they contain methods for operating on the descriptor's buffer. As you can see, TDes inherits from TDesC. All modifiable classes inherit from TDes, while the non-modifiable ones inherit directly from TDesC. Why is this? The answer is simple: TDesC provides all the methods that involve only reading descriptor data. Since all descriptors allow reading, then it serves as a base for all descriptors. TDes extends TDesC by adding the methods that involve writing descriptor data, which is why only the modifiable descriptors inherit from it.

Remember that when you see a reference to a TDesC object, it does not mean it represents only non-modifiable descriptors – modifiable ones can also be referenced through pointers or references of this type, but no writing will be done on them. However, TDes pointers and references can only be used with modifiable descriptors.

The TBufCBase and TBufBase classes shown in the class diagram are for implementation only and have no public methods or members and thus will not be discussed.

Earlier we discussed that, unlike traditional C style arrays, descriptors contain the size of their data buffer so that it can be safely accessed. The descriptor size is stored as a member variable in TDesC. TDesC provides two methods to access this size: Size() and Length(). Size() returns the buffer size in bytes. Length() returns the buffer size in either 8- or 16-bit units depending on whether it is an 8- or 16-bit descriptor. For example, if a 10-character Unicode string is stored in a 16-bit descriptor, Length() will return 10 and Size() will return 20.

Class TDes (which inherits from TDesC) adds an additional length value that specifies the maximum limit of the descriptor buffer. This is used for modifiable descriptors to ensure that write operations do not occur past the end of the allocated buffer. Therefore modifiable descriptors have two lengths associated with them – the size of the data currently in the buffer (from TDesC) and the maximum size of the data (from TDes).

TDes and TDesC cannot be directly instantiated; however, you will see and use these types frequently in function prototypes. Using base class references like this allows you to use descriptors without needing to know what kind of descriptor it is. However, as previously mentioned, while TDesC can represent all descriptors – TDes can only represent modifiable descriptors (e.g. TBuf). Also, TDes and TDesC can only represent 16-bit descriptors and TDes8 and TDesC8 can only represent 8-bit descriptors.

In the following example:

```
_LIT(KSuffix,".suffix");
void AddSuffix(TDes& string)

  {
  string.Append( KSuffix);
  }
```

the function `AddSuffix()` will add the string `".suffix"` to the end of any modifiable descriptor type passed.

6.3.2 String Literals

We've already seen string literals defined in some of the examples using `_LIT` and `_L`, but before moving onto describing the different descriptor types in detail, let's look more closely at how string literals are handled in Symbian OS.

String literals are used to store and reference strings in the code image itself. In C, you simply specify a quoted string (or one with an `L` prefix for 16-bit strings) and the compiler stores it – along with a terminating `NULL` – in the code image, and then substitutes a pointer to that location. This is simple and efficient for C since a pointer to a `NULL`-terminated string is how C uses strings. An example of a C string declaration is:

```
const char *str="Hello";
```

`_LIT` and `_L` both take a quoted string as an argument and produce a literal that appears, for all practical purposes, like a descriptor. Both macros are used often, but `_LIT` is preferred because it is implemented in a very efficient way, such that no runtime class construction occurs. `_L`, on the other hand just instantiates a `TPtrC` descriptor at runtime, which is not as efficient.

Here is an example of using `_LIT`:

```
_LIT(KMyString,"My String");
```

This defines the literal `KMyString`, used to reference the string `"My String"`. You can use `KMyString` as if it were a non-modifiable descriptor – you can pass it to functions that accept `TDesC` arguments (but not `TDes`!), you can assign it to `TDesC` pointers, and you can even call descriptor methods directly if you use the `()` operator (e.g. `MyString().Length()`).

The `_LIT` macro creates a special descriptor class called `TLitC` for string literals.

Let's look at how the _LIT macro is implemented.

```
_LIT(KMyString,"My String")
```

expands to:

```
const static TLitC<10> KMyString={9, L"My String" }
```

The buffer data and its calculated size are initialized to data members of a TLitC class.

The class TLitC does not inherit from TDesC – but it appears in memory like TBufC (see Section 6.3.3). This makes it possible for the compiler to statically initialize the data since it is all in one class. This, along with some operators that cast its type to TDesC, provides an effective trick to allow you to store a TDesC type descriptor in the code image without a constructor being called at runtime.

Like _LIT, the _L macro also defines a literal that can be treated as a TDesC (actually it is one in this case).

_L("Hello") expands to TPtrC((const TText *) L"Hello")

When an _L is encountered, a temporary TPtrC object is constructed and allocated. This is why _L is not as efficient as _LIT, (remember that _LIT involves no runtime initialization). _L, however, is sometimes more convenient since you do not need a separate line to define the literal (e.g. User::PrintInfo(L("Hello"))).

_L is officially deprecated, and is recommended for use only in cases (such as in test code) where source clarity is more important than runtime efficiency. That being said, support is likely to continue for the foreseeable future.

6.3.3 Buffer Descriptors

TBuf and TBufC are buffer descriptors, i.e. they contain their data buffers within their classes. The buffer's size is specified by an integer passed as a template argument during the class declaration.

For example, TBuf<10> Buf; creates a 16-bit descriptor object that contains a buffer big enough for ten 16-bit values (20 bytes). For 8-bit descriptors, the value specifies the number of 8-bit values allocated, so TBuf8<10> would allocate 10 bytes rather than 20.

TBuf and TBufC are commonly used for small buffers and are often declared on the stack as automatic variables. You can think of them as arrays – in fact, these classes implement their data buffers as member arrays, whose size is determined from the template argument.

TBuf is modifiable – it inherits from both TDesC and TDes and thus has both the read-only (TDesC) methods and the read/write (TDes)

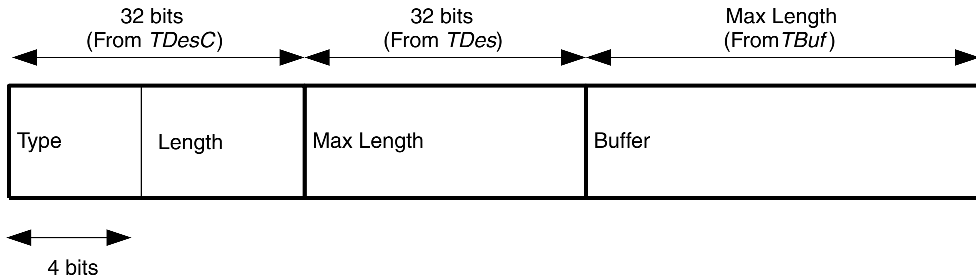

Figure 6.3 `TBuf` Memory Layout

descriptor methods available to it. `TBufC`, however, inherits only from `TDesC`, and therefore has only the read-only `TDesC` methods available to it.

Figure 6.3 shows how a `TBuf` descriptor appears in memory.

The fields labeled `type` and `length` make up a 32-bit value declared as part of `TDesC`. `type` is a 4-bit value that specifies the type of descriptor that this memory region represents. The value of `type` is 3 for `TBuf` descriptors.

Length is a 28-bit value that indicates the length of the data currently in the data buffer. This is the value returned by the `Length()` (in units of data width) and `Size()` (in bytes) methods. `Max Length` is a 32-bit value that comes from class `TDes` and contains the actual size of the allocated buffer. It is used to prevent the buffer from being accessed beyond the buffer's boundary. `buffer` is the actual allocated data buffer array and it is declared in the `TBuf` class itself.

You may wonder why `type` is stored with the descriptor in `TDesC`. If the descriptor methods in `TDes` and `TDesC` are declared virtual and the derived classes override the functions as needed, then this type of information should not be needed. That would be correct – except that virtual functions are not used in descriptors. Descriptors were written to be space efficient, and virtual functions are more of an overhead than just storing the 4-bit `type` value. The descriptor methods in `TDes` and `TDesC` use a `switch` statement on the `type` value to perform the operation correctly for the specified descriptor.

Let's step through some `TBuf` operations and show how memory is handled.

When you declare the `TBuf` as:

```
TBuf<10> buf;
```

the descriptor in memory appears as shown in Figure 6.4.

The `type` value is set to indicate a `TBuf` (3), the length is zero, since no data is yet in the buffer and the maximum length is equal to the

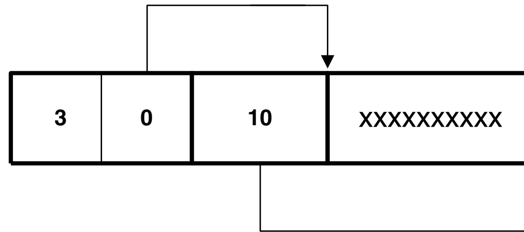

Figure 6.4 Initial state of `TBuf<10>`

allocated buffer size of 10. The buffer data is shown as a row of Xs which indicate uninitialized memory.

To copy some data to it, you can pass a value to the `TBuf` constructor when declared as in the following lines:

```
_LIT(KString,"Test");
TBuf<10> buf(KString);
```

or use the `Copy()` method as follows:

```
_LIT(KString,"Test");
TBuf<10> buf;
buf.Copy(KString);
```

Both will result in the descriptor appearing in memory as shown in Figure 6.5.

Now let's append some data by adding the following:

```
_LIT(KString1,"!!!");
buf.Append(KString1);
```

`Append()` will append the data to the descriptor buffer starting at the current length. The length is updated appropriately. The descriptor will now look as in Figure 6.6.

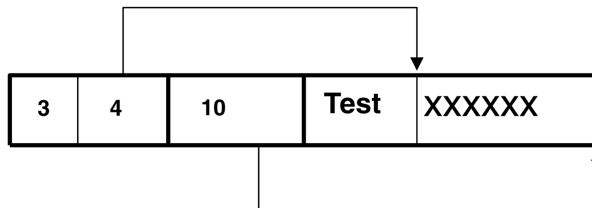

Figure 6.5 Copying data to `TBuf`

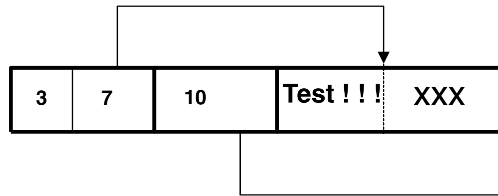

Figure 6.6 Appending to TBuf

If you then add the following:

```
_LIT(KString2,"1234");
buf.Append(KString2);
```

what happens? An exception occurs, since this would write past the end of the allocated buffer.

A TBufC descriptor is declared in the same way as a TBuf descriptor. A TBufC descriptor, however, is not modifiable with the following exception – data can be completely replaced in the buffer by using the = operator.

Figure 6.7 shows how TBufC is stored in memory.

Note that TBufC has only one length value (from TDesC) stored in memory instead of two as in TBuf. A type value of 0 indicates TBufC.

If you declare a TBufC as follows:

```
_LIT(KString1,"Sam");
TBufC<10> cBuf(KString1);
```

the memory layout will be as shown in Figure 6.8.

You cannot add to the buffer with a non-modifiable descriptor; however – as mentioned – you can replace it.

For example, you can add the following to the previous code to reassign the buffer data from KString1 ("Sam") to KString2 ("Merry"):

```
_LIT(KString2,"Merry");
cBuf=KString2;
```

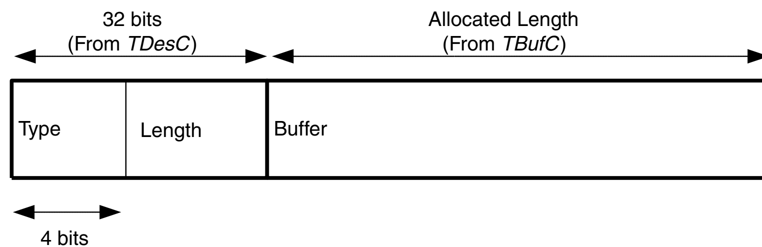

Figure 6.7 TBufC Memory Layout

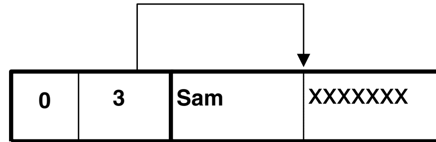

Figure 6.8 TBufC containing "Sam"

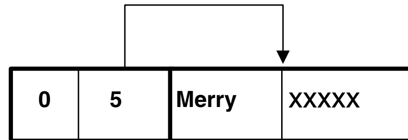

Figure 6.9 TBufC containing "Merry"

The descriptor memory will then appear as in Figure 6.9.

What if you try to replace a TBufC string with one that is too big for the buffer? Since no maximum size is stored with the descriptor, will it allow you to overwrite the buffer? The answer is no, it will throw an exception, as it would on a TBuf. The = operator uses the maximum value that is supplied at compile time via the template argument – no storage required – to see if the buffer would be overwritten.

You may then wonder why modifiable descriptors need to store the maximum size value. It's because, most times, modifiable descriptors are operated on from base class pointers (TDes) and thus will not know the template size passed over and must rely on a member variable to know the allocated buffer size, to protect against overruns.

6.3.4 Pointer Descriptors

Pointer descriptors behave like buffer descriptors except that they contain a pointer to an external data buffer instead of the data buffer itself. TPtr and TPtrC are pointer descriptors.

TPtr is a modifiable pointer descriptor and is stored in memory as shown in Figure 6.10.

You can see that the TPtr descriptor memory looks similar to TBuf except that the buffer resides outside the descriptor. The type field is set to 2 for TPtr.

Figure 6.11 shows how TPtrC is stored in memory.

Like its buffer descriptor counterpart TBufC, TPtrC does not store the buffer's maximum length (since it does not inherit from TDes). Also, like TBufC, the buffer data cannot be modified via this descriptor except by direct replacement of the data.

How do you initially set your TPtr or TPtrC buffer pointer to point to a memory region? The buffer pointer can be set when the pointer

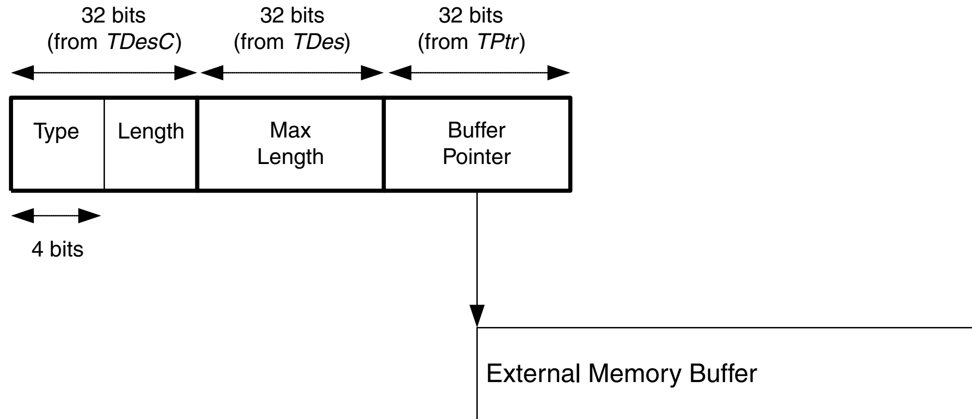

Figure 6.10 `TPtr` Memory Layout

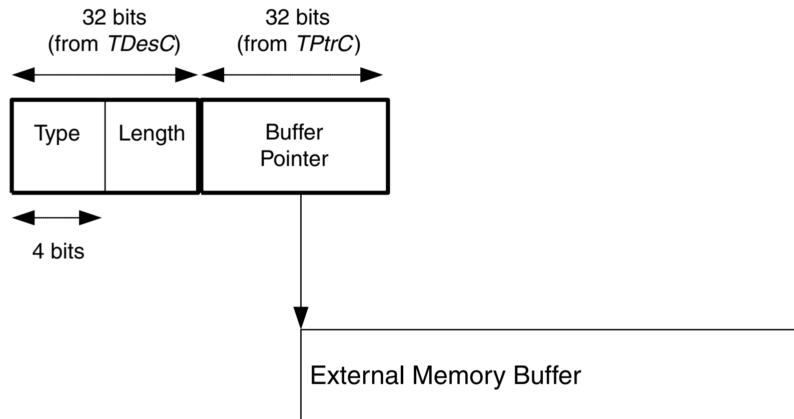

Figure 6.11 `TPtrC` Memory Layout

descriptor is constructed. For example:

```
_LIT(KSting1,"some data");
TBufC<10> someDes(KString1);
TPtrC myDes(someDes);
```

creates a non-modifiable pointer descriptor to the descriptor `someDes`.

Another example:

```
TInt bufArray[100];
TPtr myDes(bufArray,sizeof(bufArray));
// constructor arguments: buffer pointer and buffer size.
```

This will create a modifiable pointer descriptor called myDes that points to the buffer's allocated memory. For a TPtr such as in this example, the size of myDes will be zero (indicating it's empty so far), and the maximum size is the size of the array passed as the second argument.

Data can then be copied and appended to the buffer using TPtr as it would be with a TBuf descriptor.

```
_LIT(KString1,"Test");
_LIT(KString2,"!!!");
myDes.Copy(KString1);
myDes.Append(KString2);
```

This will result in the memory appearing as in Figure 6.12.

In some cases, especially if you are interfacing with ported C code, you may have a buffer that already contains data, and you want to assign it to a descriptor. In that case, you will want the pointer descriptor to be initialized with the length of the data in the buffer, in addition to the maximum length.

This can be done with the following TPtr constructor:

```
TPtr myDes(TInt *buff, TInt Length, TInt max_length);
```

Where Length is the length of the data already in the buffer and max_length is the allocated buffer size.

Or you can construct the TPtr/TPtrC, and then afterwards set the buffer with the Set(buff,length,max_length) (only in the case of TPtr) or Set(buff,max_length) where length would default to zero.

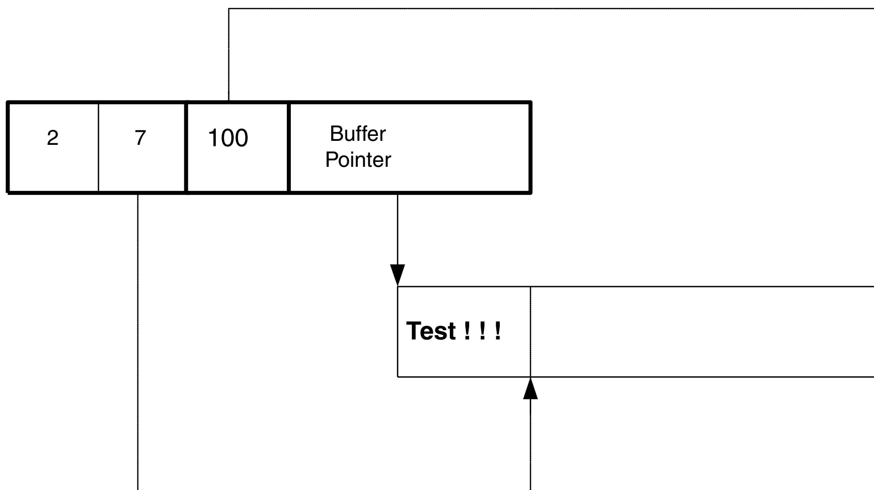

Figure 6.12 TPtr After Append

There are other ways of pointing TPtr and TPtrC to memory regions. You can reference the SDK documentation for the various overloaded constructors and Set() methods for this.

6.3.5 Heap Descriptors

HBufC is a descriptor that is allocated on the heap and referenced as a pointer. Only a non-modifiable form of the heap descriptor is supplied, but there is a way to modify it, as you will see.

In memory, an HBufC looks just like a TBufC (see Figure 6.13).

The type field for HBufC is set to 0 as for TBufC, since it appears like TBufC in memory.

HBufC provides a static New() method for instantiating a HBufC. The following line shows how to create one:

```
HBufC* myDes=HBufC::New(100);
```

This line will allocate a 16-bit descriptor on the heap with a buffer length of 100 (200 bytes).

NewL() and NewLC() methods are also available but leave on error. NewLC() pushes the created pointer on the cleanup stack.

You may wonder why you cannot just use a pointer to a TBufC instead of having another class – like the following:

```
TBufC<100>* myDes = new TBufC<100>;
```

This also works, but HBufC should be used. The reason is that HBufC provides some extra methods for dealing with the heap. For example, HBufC has a method called ReAlloc(). ReAlloc(TInt new_length) creates a new descriptor on the heap, of size new_length, copies the descriptor data to it and deletes the old one.

Modifying a Heap Descriptor's Data

Although there is no modifiable HBuf version of a heap descriptor, you can modify the data in an HBufC buffer by using the Des() method. Des() returns a TPtr whose buffer pointer is initialized to point to the

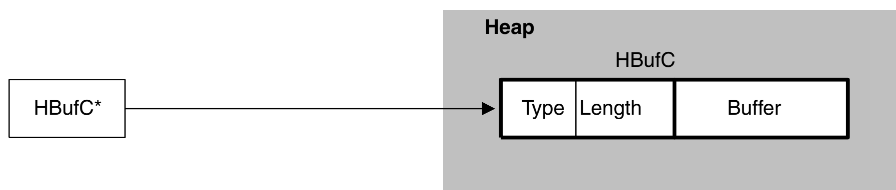

Figure 6.13 HBufC Memory Layout

heap-based buffer in the HBufC descriptor. Since the returned TPtr descriptor is a modifiable descriptor, you can use it to write the heap descriptor's data buffer. An interesting feature is that as you change the HBufC data via this TPtr, the buffer size in the TPtr descriptor and the HBufC whose data it is pointing to, are both updated.

See Figure 6.14 for how the TPtr returned by Des() is related to HBufC. The max length of TPtr is set to the length of the allocated HBufC buffer passed in the New() function. As data is changed, both lengths in HBufC and TPtr are updated together.

Let's look at the HBufC example shown in Example 6.3.

Figure 6.14 HBufC's Des() method

Example 6.3. HBufC Example

```
void heapDesExampleL()
  {
  _LIT(KString1,"Test");
  _LIT(KString2,"My Heap String");

  HBufC* myHeapDes = HBufC::NewL(KString1().Length());

  CleanupStack::PushL(myHeapDes);
  *myHeapDes = KString1;

  _LIT(KFormat1,"myHeapDes = \"%S\"length = %d\n");
  console->Printf(KFormat1,myHeapDes,myHeapDes->Length());

  myHeapDes->ReAllocL(KString2().Length()); // if you do not do this,
                                            // next line would panic
```

```
/* Replace entire string in the HBufC */

  *myHeapDes = KString2;

  console->Printf(KFormat1,myHeapDes,myHeapDes->Length());

/* Get a modifiable pointer to the HBufC's data buffer */

  TPtr myPtr = myHeapDes->Des();

  _LIT(KString3,"Hello");
  _LIT(KString4,"!!!");

/* Modify the HBufC buffer area through the TPtr, using Copy()
 * and Append()
*/

  myPtr.Copy(KString3);
  myPtr.Append(KString4);

  _LIT(KFormat2,"myHeapDes = \"%S\"length = %d  myPtr = \" %S\ "length =
          %d\n");
  console->Printf(KFormat2,myHeapDes,myHeapDes->Length(),
          &myPtr,myPtr.Length());
  CleanupStack::PopAndDestroy(myHeapDes);
  }
```

The output for Example 6.3 is as follows:

```
myHeapDes = "Test" length = 4
myHeapDes = "My Heap String" length = 14
myHeapDes = "Hello!!!" length = 8 myPtr = "Hello!!!" length = 8
```

The example code first allocates an `HBufC` with a buffer big enough to fit the string `"Test"`. It is then assigned that string. Then the buffer is reallocated using the `ReAlloc()` method and set to a bigger string. The example then shows how to modify the buffer using a modifiable `TPtr` returned from the `HBufC::Des()` method. Notice in the last output lines that the lengths of both the `TPtr` and the `HBufC` descriptors are updated and that they do indeed both point to the same, changed data.

Creating a Heap Descriptor from Another Descriptor

`TDesC` provides a method called `Alloc()` which will create a heap descriptor and initialize it with the data of the descriptor on which `Alloc()` was called. For example:

```
TBuf<80> myStr(_L("Some string data"));
HBufC* myHeapDes;
myHeapDes=myStr.Alloc();
```

The above code creates a heap descriptor initialized with the contents of myStr ("Some string data"), and assigns it to myHeapDes. You can also call Alloc() on a literal. For example, the following code creates a heap descriptor and initializes it with the contents of KMyString:

```
_LIT(KMyString,"My string");
HBufC* myHeapDes;
myHeapDes = KMyString().Alloc();
```

AllocL() and AllocLC() versions of the Alloc() function also exist which will leave on allocation failures and, in the case of AllocLC(), push the allocated heap descriptor on the cleanup stack. Alloc() will return NULL if the memory allocation fails.

6.4 Descriptor Methods

This section describes the key methods of descriptors and gives examples of how to use them.

I have divided the methods into two parts – those methods that involve reading descriptor data only (the TDesC methods) and those that involve setting and otherwise modifying the descriptor's buffer (the TDes methods). You will see that the overwhelming majority of functions are for operating on strings.

See the SDK API reference for the complete list of descriptor methods as well as the detailed function prototypes and return status descriptions for each descriptor method.

6.4.1 Non-Modifying Methods

This section describes the key descriptor methods that involve no writing to the descriptor data buffer. These methods are implemented in TDesC and thus can be used by all descriptors.

Comparing Descriptor Data

To compare the contents of one descriptor to the contents of another, use one of the Compare() methods.

For example:

```
des1.Compare(des2);
```

compares the data in descriptor des1 with the data in descriptor des2 and returns 0 if the data is the same, or a negative or positive number

if the des2 data is less than or greater than des1 (in alphabetic order) respectively. Compare() behaves like strcmp() does in C.

In addition to Compare(), you can also use CompareF() and CompareC(). These methods are the same as Compare() except that they compare the data in a normalized form for more tolerant comparisons. CompareF() compares the data normalized via *folding*. Folding is a simple locale-independent normalization method where case and accents are ignored. CompareC() performs the compare with a more powerful, locale-dependent normalization known as *collation*. While folding only does simple one-to-one mappings (e.g. lower to upper case) for comparisons, collation uses a dictionary-like ordering where it can make more complex decisions about string differences that can be safely ignored, and these rules are dependent on the locale.

Example 6.4 shows an example of using the compare functions.

Example 6.4. Compare Example

```
void CompareExample()
  {
  _LIT(KString1,"My String");
  _LIT(KString2,"MY STRING");
  _LIT(KString3,"Another string");

  TBuf<20> str1(KString1);
  TBuf<20> str2(KString2);

  TInt res;
  /* Compare shows a match since str1 is initialized to KString1 */

  res = str1.Compare(KString1);
  _LIT(KFormatCompare1,"Compare() string 1 and string 1 = %d\n");
  console->Printf(KFormatCompare1, res);

  /* Compare shows a no match since str1 and str2 contents do
  * not exactly match
  */

  res = str1.Compare(str2);
  _LIT(KFormatCompare2,"Compare() string 1 and string 2 = %d\n");
  console->Printf(KFormatCompare2, res);

  /* Compare shows a match since a folded compare is case insensitive */

  res = str1.CompareF(str2);
  _LIT(KFormatCompare3,"CompareF() string 1 and string 2 = %d\n”);
  console->Printf(KFormatCompare3, res);

  /* Compare shows a mismatch since string 1 and 3 are different */

  res = str1.Compare(KString3);
  _LIT(KFormatCompare4,"Compare() string 1 and string 3 = %d\n");

  console->Printf(KFormatCompare4, res);
  }
```

The output of Example 6.4 is:

```
Compare() string 1 and string 1 = 0
Compare() string 1 and string 2 = 32
CompareF() string 1 and string 2 = 0
Compare() string 1 and string 3 = 12
```

Finding Sub-Strings Within a Descriptor

To locate a sub-string within a descriptor, you can use `Find()`. `Find()` looks for the first occurrence of a sub-string within a descriptor and returns its start position, if it is found.

For example:

```
des1.Find(KSomeSubString);
```

returns `KErrNotFound` (−1) if the sub-string `KSomeSubString` is not found in `des1`, or the starting position of the sub-string in `des1` if the sub-string is found.

`FindF()` and `FindC()` are the same as `Find()` except they use the tolerant fold and collation comparisons (respectively) to search for the string.

Example 6.5 shows an example of the find functions.

Example 6.5. Find Example

```
void FindExample()
  {
  _LIT(KString1,"This is a test string");
  _LIT(KString2,"test");
  _LIT(KString3,"car");
  _LIT(KString4,"TEST");

  TBuf<40> buf(KString1);
  TInt res;

  /* Find returns position of "test" in KString1 */

  res = buf.Find(KString2);
  _LIT(KFormat1,"Find of string 2 in string 1 res = %d\n");
  console->Printf(KFormat1,res);

  /* "car" does not occur in KString1, so KErrNotFound reported */

  res = buf.Find(KString3);
  _LIT(KFormat2,"Find of string 3 in string 1 res = %d\n");
  console->Printf(KFormat2,res);

  /* Since FindF does a fold compare, "TEST" is found and position
  * is returned
  */
```

```
  res = buf.FindF(KString4);

  _LIT(KFormat3,"Find of string 4 in string 1 res = %d\n");
  console->Printf(KFormat3,res);
  }
```

The output of Example 6.5 is:

```
Find of string 2 in string 1 res = 10
Find of string 3 in string 1 res = -1
Find of string 4 in string 1 res = 10
```

For more powerful searching of sub-strings within descriptors, you can use `Match()` instead of `Find()`. `Match()` behaves like `Find()` except you can supply wildcard characters when searching for a string match. '*' represents a sequence of any characters; '?' represents an occurrence of any single character.

Example 6.6 shows an example.

Example 6.6. Match Example

```
void MatchExample()
  {
  _LIT(KString1,"This is test string A");
  _LIT(KString2,"This is test string ?");
  _LIT(KString3,"*is test string ?");
  _LIT(KString4,"*");
  _LIT(KString5,"*B");

  TBuf<40> buf(KString1);
  TInt res;

  /* A match since ? indicates any single character */

  res = buf.Match(KString2);
  _LIT(KFormat1,"Match: string 2 and string 1 res = %d\n");
  console->Printf(KFormat1,res);

  /* A match using '*' and '?' in string */
  res = buf.Match(KString3);
  _LIT(KFormat2,"Match: string 3 and string 1 res = %d\n");
  console->Printf(KFormat2,res);

  /* '*' matches any string */

  res = buf.Match(KString4);
  _LIT(KFormat3,"Match: string 4 and string 1 res = %d\n");
  console->Printf(KFormat3,res);

  /* no match since KString1 does not end in 'B' */
  res = buf.Match(KString5);
  _LIT(KFormat4,"Match: string 5 and string 1 res = %d\n");
  console->Printf(KFormat4,res);
  }
```

The output of Example 6.6 is as follows:

```
Match: string 2 and string 1 res = 0
Match: string 3 and string 1 res = 5
Match: string 4 amd string 1 res = 0
Match: string 5 and string 1 res = -1
```

Extracting Sub-Strings from Descriptors

To extract specific portions of a descriptor string, use the methods: `Left()`, `Right()` or `Mid()`.

These methods return a `TPtrC` descriptor that points to a specified sub-string within the descriptor the methods are invoked on.

`Left()` defines a sub-string starting at the beginning of the descriptor and of a specified length. `Right()` defines a sub-string that starts a specified length before the end of the descriptor. `Mid()` specifies a sub-string that starts from a specified position and is of a specified length.

Example 6.7 shows an example of using all three sub-string extraction functions.

Example 6.7. Sub-string Example

```
void SubstringExample()
  {
  _LIT(KString1,"This is my string");

  TBufC<40> buff(KString1);

  TPtrC SubStr= buff.Left(4);

  /* Get left 4 characters of string */

  _LIT(KFormat1,"Left(4): SubStr = \"%S\"\n");
  console->Printf(KFormat1,&SubStr);

  /* Get right 3 characters of string */

  SubStr.Set(buff.Right(3));

  _LIT(KFormat2,"Right(3): SubStr = \"%S\"\n");
  console->Printf(KFormat2,&SubStr);

  /* get 6 characters in middle, starting at position 8 */

  SubStr.Set(buff.Mid(8,6));

  _LIT(KFormat3,"Mid(8,6): SubStr = \"%S\ "\n");
  console->Printf(KFormat3,&SubStr);
  }
```

The output of Example 6.7 is as follows:

```
Left(4): SubStr = “This”
Right(3): SubStr = “ing”
Mid(8,6): SubStr = “my str”
```

6.4.2 Methods that Write Descriptor Data

This section describes some key `TDes` class methods which are available to all modifiable descriptors.

Copying Data to a Descriptor

We have already looked at using `Copy()` to write data into a descriptor. To recap, `Copy()` will copy data into the descriptor's buffer, replacing any data that exists, and update the descriptor size to match the size of the data copied. The data to be copied can be specified as an 8- or 16-bit descriptor, a `NULL`-terminated string or a buffer specified with a pointer and size.

In addition to `Copy()`, you can use `CopyC()`, `CopyF()`, `CopyCP()`, `CopyLC()` or `CopyUC()`. These variations are equivalent to `Copy()` except that each will perform a specific transformation on the data before the copy. `CopyC()` and `CopyF()` will collate and fold the data respectively, before the copy. These can be used to normalize strings for tolerant sorts and compares.

`CopyCP()`, `CopyLC()`, `CopyUC()` will perform case conversions – capitalization, lower case and upper case, respectively – before copying (these are performed depending on locale).

Example 6.8 shows an example of the copy functions.

Example 6.8. Copy Example

```
void CopyExample()
  {
  TUint8 binData[6] = {0xB0,0xB1,0xB2,0xB3,0xB4,0xB5};

  /* Copy standard C array into binary descriptor */

  TBuf8<sizeof(binData)> binDes;

  binDes.Copy(binData,sizeof(binData));

  _LIT(KFormat1,"binDes[0]=%x  binDes[1]=%x\n");
  console->Printf(KFormat1,binDes[0],binDes[1]);

  /* Copy binary descriptor to another 8 bit binary descriptor */

  TBuf8<20> buf8;
```

```
buf8.Copy(binDes);

_LIT(KFormat2,"buf8[0]=%x  buf8[1]=%x\n");
console->Printf(KFormat2,buf8[0],buf8[1]);

/* Copy literal into descriptor */

_LIT(KString1,"My string");

TBuf<20>  buf16;
buf16.Copy(KString1);

_LIT(KFormat3,"buf16 = %S\n");
console->Printf(KFormat3,&buf16);

/* Copy C style 8-bit string into descriptor (first 8-bit then 16-bit)*/

TUint8 *C_str=(TUint8 *) "Hello there.";  /* NULL-terminated
                                                8-bit string */
buf8.Copy(C_str);
buf16.Copy(buf8);
console->Printf(KFormat3,&buf16);  /* Printf just prints 16-bit
                                        descriptor strings */

/* Copy, converting to upper case */

TBuf<20> newBuf;
newBuf.CopyUC(buf16);
_LIT(KFormat4,"CopyUC(): newBuf = %S\n");
console->Printf(KFormat4,&newBuf);

/* Copy, converting to lower case */
newBuf.CopyLC(buf16);
_LIT(KFormat5,"CopyLC(): newBuf = %S\n");
console->Printf(KFormat5,&newBuf);

/* Copy, capitalize */

newBuf.CopyCP(buf16);
_LIT(KFormat6,"CopyCP(): newBuf = %S\n");
console->Printf(KFormat6,&newBuf);
}
```

The output from Example 6.8 is as follows:

```
binDes[0]=b0 binDes[1]=b1
buf8[0]=b0 buf8[1]=b1
buf16 = My string
buf16 = Hello there.
CopyIC(): newBuf = HELLO THERE.
CopyUC(): newBuf = hello there.
CopyCP(): newBuf = Hello there.
```

In addition to copying data to a descriptor you can fill the descriptor with repeating data using `Fill()`. `Fill()` will fill the data buffer with the specified character (`TChar`), for the specified number of characters,

starting from the beginning. If length is not specified, the data is filled up to the current length. `FillZ()` works the same way, except that the fill character is always 0.

Example 6.9 shows the fill function.

Example 6.9. Fill Example

```
void FillExample()
  {
  TBuf<40> buf;
  buf.Fill('*',10);
  _LIT(KFormat1,"buf = \"%S\"\n");
  console->Printf(KFormat1,&buf);
  buf.Fill('-');
  console->Printf(KFormat1,&buf);
  }
```

The output for Example 6.9 is as follows:

```
buf = “**********”
buf = “----------“
```

Appending Data to a Descriptor

We have also discussed `Append()` previously – it behaves like `Copy()` except that it concatenates the specified data to the descriptor instead of replacing it. The data specified to `Append()` can be an 8- or 16-bit descriptor, a `NULL`-terminated string, a buffer pointer with length, or a `TChar`.

Here are a few other functions that append data to a descriptor.

`AppendFill()` appends a specified number of repeats of a specified character (a `TChar`) to the descriptor. `AppendJustify()` will justify a specified string (left, center or right) and append it to the descriptor.

`AppendNum()` will convert an integer to a string and append it to the descriptor. You can also specify a radix which can be binary, octal, hexadecimal or decimal. `AppendNumFixedWidth()` will result in a fixed-width number string being added, with leading zeros if needed. The methods with UC at the end will result in upper case letters being appended for hexadecimal numbers.

Example 6.10 shows the append functions in action.

Example 6.10. Append Example

```
void AppendExample()
  {
  _LIT(KMyString1,"String:");
  _LIT(KMyString2,"num vals are");
```

```
_LIT(KMyString3,"Justify");

TInt num=0x0b4a;

TBuf<40> str(KMyString1);
/* Simple Append of a literal */
str.Append(KMyString2);

/* Append num in various forms */

str.AppendNum(num);
str.Append(' ');
str.AppendNum(num,EHex);
str.Append(' ');
str.AppendNumUC(num,EHex);
str.Append(' ');
str.AppendNumFixedWidthUC(num,EHex,5);

_LIT(KFormat1,"str = %S\n");
console->Printf(KFormat1,&str);

TBuf<40> just;

/* Justify a string */

just.AppendJustify(KMyString3,12,ERight,' ');
/* can replace " " with any fill character */

_LIT(KFormat2,"%S\n");
console->Printf(KFormat2,&just);

/* Add repeated character */

just.AppendFill('!',5);
console->Printf(KFormat2,&just);
}
```

The output for Example 6.10 is as follows:

```
str = String: num vals are 2890 b4a B4A 00B4A
    Justify
    Justify!!!!!
```

Formatting Descriptor Data

It's handy to be able to format a string in the same way as in C's `sprintf()` and `printf()` functions. The descriptor method `Format()` does this.

The format string supplied to `Format()` is very similar to the format string in C, supporting `%d`, `%s`, `%f`, etc. There are also some Symbian OS specific formats, however. For example, the format indicator `%S` takes a descriptor and outputs the descriptor's string contents.

Format(), like Copy() replaces any existing descriptor data. Alternatively, you can use AppendFormat() to append the formatted string to the descriptor.

Example 6.11 shows the format functions in action.

Example 6.11. Format Example

```
void FormatExample()
  {
  _LIT(KString1,"My string");

  TInt value=10,value1=20;
  _LIT(KMyDesFormat,"Descriptor = %S, value = %d");

  TBuf<100> buf;
  buf.Format(KMyDesFormat,&KString1,value);

  _LIT(KMyDesFormat1,"--also value1 = %d");
  buf.AppendFormat(KMyDesFormat1,value1);

  _LIT(KFormat1,"%S\n");
  console->Printf(KFormat1,&buf);
  }
```

The output of Example 6.11 is as follows:

```
Descriptor = My string, value = 10--also value1 = 20
```

Changing the Case of a Descriptor String

Use Capitalize() to capitalize a descriptor string (performed as defined by the phone's locale). Use LowerCase() and UpperCase() to convert all characters in the descriptor to lower and upper case respectively.

Example 6.12 shows an example of the case switching functions.

Example 6.12. Case Conversions

```
void CaseExample()
  {
  _LIT(KString1,"hillary");
  TBuf<7> name(KString1);

  name.UpperCase();
  _LIT(KFormat1,"name=%S\n");
  console->Printf(KFormat1,&name);

  name.LowerCase();
  console->Printf(KFormat1,&name);

  name.Capitalize();
  console->Printf(KFormat1,&name);
  }
```

The output of Example 6.12 is as follows:

```
name=HILLARY
name=hillary
name=Hillary
```

Deleting Data from a Descriptor

Use `Delete()` to remove a selected portion of a descriptor buffer.
For example:

```
des1.Delete(2,3);
```

deletes the data in `des1` starting at position two, for a length of three. So if `des1` contained 'abcdedf' before this line, it would contain 'abdf' after, and the length of `des1` would be changed from seven to four.

Descriptors also have methods for removing unwanted spaces in strings. `TrimLeft()` and `TrimRight()` will delete leading and trailing spaces respectively. `TrimAll()` will delete leading and trailing spaces, as well as trimming any consecutive spaces in the data to one space.

Example 6.13 shows the deletion functions.

Example 6.13. Deletion functions

```
void DeletionExample()
  {
  _LIT(KString1,"Heeeello");
  _LIT(KString2,"This    is   a test  ");

  TBuf<20> buf1(KString1);
  TBuf<40> buf2(KString2);

  _LIT(KFormat1,"buf1=\"%S\"buf1 length = %d\n");
  console->Printf(KFormat1,&buf1,buf1.Length());

  /* delete characters starting at position 1, length
  * 3 -  will reduce descriptor size appropriately
  */

  buf1.Delete(1,3);
  console->Printf(KFormat1,&buf1,buf1.Length());

  _LIT(KFormat2,"buf2=\"%S\"buf2 length = %d\n");
  console->Printf(KFormat2,&buf2,buf2.Length());

  /* TrimLeft deletes leading spaces */
  buf2.TrimLeft();
  _LIT(KFormat3,"TrimLeft(): buf2=\"%S\"buf2 length = %d\n");
  console->Printf(KFormat3,&buf2,buf2.Length());
```

```
/* reset string to KString2 */
buf2.Copy(KString2);

/* Trim right deletes trailing spaces */
buf2.TrimRight();
_LIT(KFormat4,"TrimRight(): buf2=\"%S\"buf2 length = %d\n");
console->Printf(KFormat4,&buf2,buf2.Length());

/* reset string to KString2 */
buf2.Copy(KString2);

/* Trim deletes leading and trailing spaces */
buf2.Trim();
_LIT(KFormat5,"Trim(): buf2=\"%S\"buf2 length = %d\n");
console->Printf(KFormat5,&buf2,buf2.Length());

/* reset string to KString2 */
buf2.Copy(KString2);

/* Trimall deletes leading and trailing spaces, and extra
* spaces in middle
*/
buf2.TrimAll();
_LIT(KFormat6,"TrimAll(): buf2=\"%S\"buf2 length = %d\n");
console->Printf(KFormat6,&buf2,buf2.Length());
}
```

The output of Example 6.13 is as follows:

```
buf1="Heeeello" buf1 length = 8
buf1="Hello" buf1 length = 5
buf2="This               is   a  test    " buf2 length = 24
TrimLeft(): buf2= "This    is   a test" buf2 length = 21
TrimRight(): buf2="This    is   a test" buf2 length = 22
Trim(): buf2="This    is   a test" buf2 length = 19
TrimAll(): buf2="This is a test" buf2 length = 14
```

Converting to `NULL`-terminated Strings

Sometimes its useful to convert a descriptor string to a C-style `NULL`-terminated string – especially in cases where you are working with ported C code in the Symbian OS environment. Descriptors provide a few methods to help with this.

`ZeroTerminate()` adds a `NULL` to the end of the descriptor data. The descriptor length is not updated. This is used for translating the string to a C-style `NULL`-terminated string, as you will see in the example.

`PtrZ()` is the same as `ZeroTerminate()` except that it returns a pointer to the `NULL`-terminated descriptor data. It is equivalent to calling `ZeroTerminate()` and then `Ptr()`.

Example 6.14 shows an example.

Example 6.14. Zero Termination

```
void ZeroTerminationExample()
  {
  _LIT(KMyString,"My string");
  TBuf<20> buf(KMyString);
  const TText *str;
  str = buf.PtrZ();

  /* str now points to a 16 bit NULL terminated string */
  _LIT(KFormat1,"str=%s\n");
  console->Printf(KFormat1,str);

  const unsigned char *nStr;
  TBuf8<20> buf8;
  buf8.Copy(buf);  /* copy 16 bit string into 8-bit
                      descriptor (converts to 8-bit chars)*/
  nStr = buf8.PtrZ();

  /* nStr points to a standard C style narrow string (can't
  * print out directly with console->Printf)
  */
  }
```

The output of Example 6.14 is:

```
str=My string
```

Setting the Descriptor Size

You can manually change the length of a modifiable descriptor using `SetLength()` and `SetMax()`. Note that both of these only modify the length stored in `TDes`, and not the maximum length stored in `TDesC` (the name `SetMax()` is misleading).

`SetLength()` changes the current length of the descriptor data. If you lower the length, for example, you effectively chop off data from the end of the buffer. `Zero()` is equivalent to `SetLength(0)`. The current length also determines where data will be appended.

`SetMax()` will set the current buffer length value (the one stored in `TDesC`) to equal the maximum buffer size (also stored in `TDes`). Why do this? Typically, you use `SetMax()` when assigning an external buffer (e.g. `char *` buffer) to a pointer descriptor.

For example, assume you have a `char *` buffer called `externalBuf`, and its size is `buffSize`. You can assign this buffer to a descriptor as follows:

```
TPtr myDes(externalBuf,buffSize);
```

This will cause `myDes` to point to `externalBuf` correctly and the maximum buffer size stored in `TDesC` is also correct. However, the

actual size of the descriptor as specified in `TDes` (the size returned by `Size()` and `Length()` and where append operations would start) is 0 after this line. So if you pass this descriptor to a function, or otherwise operate on it, it will treat this descriptor like an empty one.

To solve this, call:

```
myDes.SetMax()
```

after the declaration. This will set the descriptor size to the maximum size (to `buffSize` in the example).

6.4.3 Using a Descriptor as an Array

You can use the descriptor's `[]` operator to access your descriptor data in the same way you would do with a C array.

Example 6.15 shows an example using a binary buffer (of course, strings can also be used).

Example 6.15. `[]` Operator

```
void ArrayIndexExample()
  {
  _LIT(KString1,"This is my string");

  TChar c;

  TBuf<20> str(KString1);

  /* character access using [ ] */

  _LIT(KFormat1,"str[0]=%c  str[3]=%c\n");
  console->Printf(KFormat1,str[0],str[3]);

  /* Binary buffer access using [] */

  TUint8 binData[6] = {0xB0,0xB1,0xB2,0xB3,0xB4,0xB5};

  TBuf8<sizeof(binData)> binDes;
  binDes.Copy(binData,sizeof(binData));

  _LIT(KFormat2,"binDes[0]=%x  binDes[1]=%x  binDes[5]=%x\n");
  console->Printf(KFormat2,binDes[0],binDes[1],binDes[5]);
  }
```

The output of Example 6.15 is:

```
str[0] = T str[3] = s
binDes[0] = b0 binDes[1] = b1 binDes[5] = b5
```

As you can see from the example, the [] operator can be used to read descriptor data as if it were a standard C-style array. You may wonder why a binary buffer would be put into a descriptor like this, since it was an array already. The reason is that having it as a descriptor provides safe access and will throw an exception immediately if you go over the end of the array. Try it and see.

You can also write descriptor data (if it's a modifiable descriptor) using the [] operator.

6.5 Converting Between 8-bit and 16-bit Descriptors

You may have 8-bit strings that need to be in the form of a 16-bit descriptor. You can convert the 8-bit descriptor to a 16-bit descriptor by calling the Copy(TDesC8&) method of the 16-bit descriptor – this will expand each 8-bit data value to 16 bits, setting the high-order bytes to zero.

This is straightforward and may be suitable for simply getting a narrow string in the correct format – however, if the 8-bit string is coded in UTF-8, this simple 8-bit to 16-bit conversion will not work correctly. This is because the multi-byte sequences of the string will just be copied as is, resulting in a corrupted 16-bit string.

The API class CnvUtfConverter is used to translate between UTF-8 and Unicode. The class contains two methods to do this:

```
CnvUtfConverter::ConvertToUnicodeFromUtf8(src8,dest16)
```

where src8 is an 8-bit descriptor containing the UTF-8 string and dest16 is the 16-bit descriptor where the converted Unicode string will be put.

```
CnvUtfConverter::ConvertFromUnicodeToUtf8(dest8,src16)
```

where src16 is the 16-bit descriptor that contains the Unicode string and dest8 is the 8-bit descriptor where the UTF-8 string is placed.

Make sure you include utf.h in your source file and charconv.lib in the LIBRARY statement of the mmp file when using these functions.

6.6 Dynamic Buffers

CBufBase, CBufSeg and CBufFlat are API classes used for managing expandable buffers known as dynamic buffers. While a descriptor has a fixed-size buffer allocated to it, a dynamic buffer can be resized as needed at runtime – and in some cases this resizing is automatic as you add more data.

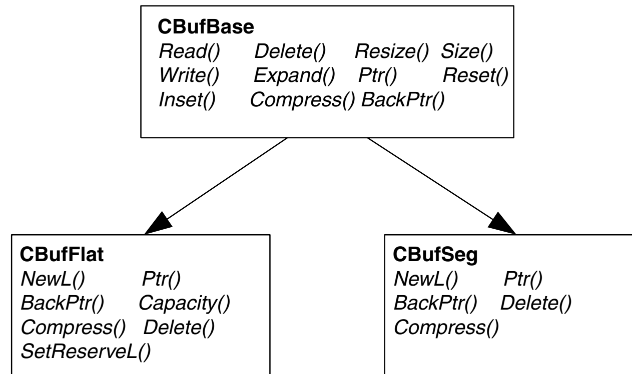

Figure 6.15 Dynamic Buffer Class Diagram

Figure 6.15 shows the class hierarchy for dynamic buffers along with their methods.

6.6.1 When Should I Use Dynamic Buffers?

Use dynamic buffers when you do not know what the maximum size of the buffer will be. Use descriptors if you do know the size, or can specify a maximum size big enough that you can be sure it will not be exceeded at runtime (although this can waste memory, or cause an exception if you guess wrongly).

Dynamic buffers are not used much directly, but are used by other APIs, including collection classes such as the array API classes (see Section 6.7).

6.6.2 Flat and Segmented Buffers

`CBufFlat` and `CBufSeg` represent the two types of dynamic buffers: flat and segmented. From an API perspective, these buffers act virtually the same – they implement the abstracted dynamic buffer interface provided by `CBufBase`. It's instructive to understand them though, so that you can choose the most efficient dynamic buffer type for your situation.

Flat buffers are allocated as a single memory region on the heap. As more space in the buffer is needed, the single cell is reallocated to a bigger size.

To create a flat buffer, call `CBufFlat::NewL(granularity)`, where *granularity* is the number of bytes by which the buffer size is increased (or decreased) at one time. For example, if your granularity is 512, then 512 bytes are initially allocated on the heap before the first data is written. An allocation of 512 more bytes occurs when you write past the 512th byte. The buffer is increased again by another 512 bytes when you write past the 1024th byte and so on.

The advantage of flat buffers is that the memory is always contiguous, and thus is more straightforward and efficient to access. However, buffer expansions are expensive, since they involve a reallocation – which requires the data to be copied to a new heap cell.

Therefore, use flat buffers for buffers that need to be expanded at runtime, but where expansions are rare.

Unlike a flat buffer, a segmented dynamic buffer allocates a new memory region on the heap when more buffer space is needed, as opposed to performing an allocation of a single heap region. Thus, segmented buffers are more efficient when expanding. In addition, segmented buffers are more efficient when inserting and deleting data since the shuffling of data can be minimized.

To create a segmented buffer, call `CBufSeg::NewL(granularity)`, where granularity is the number of bytes in a segment.

While a segmented buffer can be expanded much more efficiently than a flat buffer, the disadvantage is that the region is not contiguous and thus may be more difficult to access, depending on how you use the buffer.

Note that, although the buffer is not contiguous, this fact is hidden when accessing the buffer through the dynamic buffer class methods. `CBufFlat` and `CBufSeg` map the position value to the memory address transparently. Only when you manipulate the memory buffer directly (i.e. by using a pointer returned by the `Ptr()` method) will you need to be concerned about the buffer being segmented.

6.6.3 Dynamic Buffer Methods

Let's briefly look at the key methods that are available to both segmented and flat buffers through the abstract interface of `CBufBase`. Consult the SDK documentation for more details of how these functions are called.

Reading and Writing a Dynamic Buffer

The methods `Read()` and `Write()` allow you to read from and write to a dynamic buffer starting at a specified position. The data is read into or written from an 8-bit descriptor, or a raw memory region specified by a pointer and the data size.

Example 6.16 shows this in action.

Example 6.16. Dynamic Buffer

```
void DynamicBufferExampleL()
  {
  TUint8 dataAry1[100];
  TUint8 dataAry2[100];

  TUint8 outAry[120];
```

```
// initialize dataAry1 and dataAry2 with some stuff
    TUint8 j=100;
    for (TUint8 i=0; i<100; i++)
      {
      dataAry1[i]=i;
      dataAry2[i]= j--;
      }

    TPtrC8 desAry1(dataAry1,100); // create descriptor for dataAry1

    CBufFlat* dynBuf = CBufFlat::NewL(20);
    CleanupStack::PushL(dynBuf);

    dynBuf->ResizeL(100);   // allocate memory to buffer, none to start

    dynBuf->Write(0,desAry1);     // write desAry1 to
                                  // dynBuf starting at position 0.
    dynBuf->Write(3,dataAry2,50); // write first 50 bytes of dataAry2 to
                                  // dynBuf at position 3
    dynBuf->Read(0,outAry,50);    // reads 50 bytes starting at
                                  // position 0, putting data into buffer
                                  // to outAry

    _LIT(KFormat1,"dynbuf pos=%d : %d %d %d %d %d\n");
    console->Printf(KFormat1,0,

    outAry[0],outAry[1],outAry[2],outAry[3],outAry[4]);

    dynBuf->ResizeL(120);    // add some room to the buffer

    dynBuf->Write(90,dataAry1,30);

    dynBuf->Read(100,outAry,20);
    console->Printf(KFormat1,100,
    outAry[0],outAry[1],outAry[2],outAry[3],outAry[4]);

    CleanupStack::PopAndDestroy();
  }
Output:

dynbuf pos=0       : 0 1 2 100 99
dynbuf pos=100 : 10 11 12 13 14
```

Example 6.16 should be self-explanatory. One thing to note is that since `Write()` does not expand the buffer automatically, the method `ResizeL()` is used to allocate memory to the buffer.

Inserting and Deleting Data

You can insert data into, and delete data from, a dynamic buffer by using the `InsertL()` and `Delete()` methods for the dynamic buffer class. `InsertL()` acts the same as a `Write()` except that the data currently in the buffer at the insertion point is shifted up in position. Also, unlike with `Write()`, `InsertL()` will expand the buffer, if necessary, to make room for the new data.

So in Example 6.17, if the line:

```
dynBuf->Write(90,dataAry1,30);
```

was replaced by

```
dynBuf->InsertL(90,dataAry1,30),
```

then the `ResizeL()` would not be needed since `InsertL()` would see that more room is needed in the buffer and resize accordingly. Also, if you insert data anywhere in the buffer, then, since all data is shifted up, a memory allocation may occur too.

Use `Delete()` to delete a specified number of bytes at the specified dynamic buffer position. Upon deletion, the data is shifted down appropriately (as is also done in the `Delete()` descriptor method).

Manually Changing the Size of a Dynamic Buffer

Even though `InsertL()` will expand the buffer size automatically as needed, you may still want to manually change the size of your buffer. There are two reasons for this:

1. So that you will know immediately if you have enough memory for the particular sequence of data writes you are performing,
2. So that the allocation is done at once, for efficiency, instead of many allocations taking place as you insert new data.

`ResizeL()` and `ExpandL()` will manually reserve space in the dynamic buffer. These functions are similar in that they force a reallocation of the buffer.

`ResizeL()` we've seen in the example – it simply causes the dynamic buffer to be reallocated (or initially allocated) to a new size, with the buffer space increasing or decreasing as appropriate.

`ExpandL()` inserts an uninitialized data region at a given buffer position, thus forcing the buffer size to increase by that amount. In other words, it sets aside some space in the buffer for future use.

For example:

```
dynbuf->ExpandL(5,200);
```

would insert an uninitialized data region at position 5 to 200, pushing the rest of the data up (and doing a memory allocation if needed). This carves out a space in the buffer so that you can now use `Write()` to write to that region, knowing that the memory is preallocated.

Getting a Pointer to an Area in a Dynamic Buffer

`Ptr()` returns a `TPtr8` which points to the address of the specified position in the dynamic buffer. `Ptr(4)`, for example, returns a `TPtr8` referencing the region starting at the byte at position 4 in the buffer up to either the buffer end or – in the case of a segment buffer – the end of the current segment.

`BackPtr()` returns a `TPtr8` that points to the memory region, starting at the beginning of the contiguous memory region containing the byte, and ending at the position specified when calling `BackPtr()`. For a flat buffer, the beginning will always be the first byte of the buffer. For a segmented buffer, it will be the first byte of the segment containing the byte.

6.7 Arrays

Symbian provides a large assortment of API classes for implementing data arrays. See the SDK documentation for the complete list of these classes as well as their method descriptions. I will cover the key array classes here and provide some examples.

6.7.1 Fixed Arrays

`TFixedArray` is a thin API class that uses templates to define an array of data items of user-defined type. The class overrides the `[]` operator and thus acts like a traditional array, but with an important difference – range checking is performed on indexes to prevent out of bound accesses.

A fixed array is declared as follows:

```
TFixedArray<type,size> myArray
```

This line will allocate a fixed array called `myArray` where type represents the data type of the data in the array and size indicates the maximum number of `MyObjs` the array is allocated for.

This declaration is effectively the same as `type myArray[size]`.

Example 6.17 shows an example of using fixed arrays.

Example 6.17. Fixed Array Example

```
/* Initialize a simple fixed array of integers */

  TFixedArray<Tint,10> Array;

  for (j=0;j<10;j++)
    Array[j] = j;

  Array[10] = 10;   // Generates exception, outside of array boundary
```

Fixed arrays are a lightweight and efficient way of implementing the array while providing access range checking. Like traditional arrays, the number of items in the array is preallocated at compile time.

6.7.2 Descriptor Arrays

Descriptor array classes implement arrays of `TDesC`-based buffer descriptors. The purpose of these classes, in most cases, is to implement an array of strings. For example, a list box keeps its list of selection item strings in a descriptor array.

Descriptor arrays use dynamic buffers to store their data. Therefore, the array size does not need to be preallocated, as it is for a fixed array. The array is expanded as needed when new data items are added to it.

Descriptor arrays can be flat or segmented, contain 8- or 16-bit descriptors and contain either copies of the descriptors (in `HBufC`s) or pointers to descriptors (in `TPtr`s).

Here are the instantiable descriptor array classes:

- `CDesC16ArrayFlat`: An array of 16-bit descriptors stored in a flat dynamic buffer
- `CDesC16ArraySeg`: Same as `CDesC16ArrayFlat`, but data is stored in a segmented dynamic buffer
- `CDesC8ArrayFlat`: An array of 8-bit descriptors stored in a flat dynamic buffer
- `CDesC8ArraySeg`: Same as `CDesC8ArrayFlat`, but data is stored in a segmented dynamic buffer
- `CDesCArrayFlat`: Same as `CDesC16ArrayFlat` for the standard Unicode build
- `CDesCArraySeg`: Same as `CDesC16ArraySeg` for the standard Unicode build
- `CPtrC16Array`: Array of `TPtrC` objects that point to the descriptor data elements
- `CPtrC8Array`: Same as `CPtrC16Array`, but stores 8-bit descriptors
- `CPtrCArray`: Equivalent to `CPtrC16Array` for the standard Unicode build

Descriptor array classes that end in `Flat` and `Seg` indicate the flat and segmented type of dynamic buffer used to store the array's data. Refer to the previous section on dynamic buffers for more information. Use a flat array if the array is not expanded very often, otherwise, use a segmented array.

Note that the classes that begin in `CPtrC` use flat buffers only – no segmented versions of these classes are supplied.

Descriptor array classes that begin with `CDesC` are implemented as an array of `HBufC` pointers. When a descriptor is added to a `CDesC` array, the array class will allocate a new `HBufC`, copy the data from the descriptor to this `HBufC`, and, finally, write the `HBufC`'s pointer to the appropriate position in the array.

Consider Example 6.18.

Example 6.18. `CDesC` Array Class

```
_LIT(KString1,"My String");
_LIT(KString2,"Test One Two");

TBufC<20> MyDes(KString1);
TBuf<20> MyDes1(KString2);

CDesCArrayFlat myArray = new (ELeave) CDesCArrayFlat(5);
CleanupStack::PushL(myArray);  // in case the appends leave

myArray->AppendL(MyDes);
myArray->AppendL(MyDes1);

/* ...  */

CleanupStack::PopAndDestroy();
```

Figure 6.16 shows how `CDesCArray` is stored, when `MyDes` and `MyDes1` are appended.

To use the descriptor arrays, you need to include file `badesca.h` and link to library `bafl.lib`.

Since copies of the descriptors are made and referenced in the array, it does not matter if the user deletes the objects after they are added to

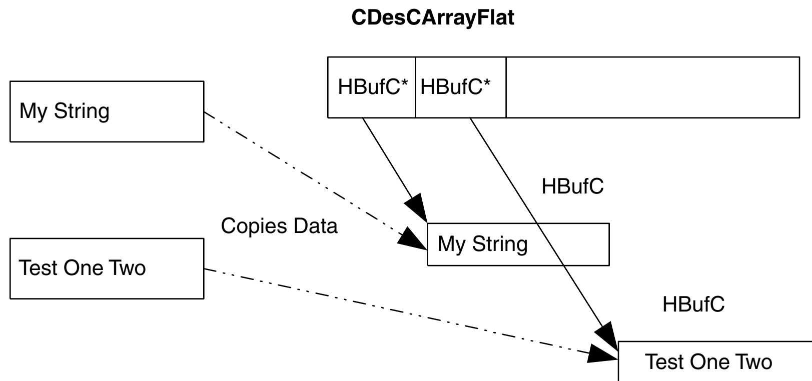

Figure 6.16 `CDesCArrayFlat`

the array. Of course, the disadvantage of using this type of array is that you have overheads in both performance (doing the copy) and memory (duplicating descriptor data in memory).

An array class that begins with CPtrC contains descriptor pointers (TPtrC s) as its elements.

If the descriptor array were of type CPtrCArray instead of CDesCArrayFlat in Example 6.18, it would be stored as shown in Figure 6.17.

Unlike the CDesC array classes, CPtrC classes do not have to copy or store the data in the descriptors that are added to the array, since they simply point to the descriptor data. However, you need to make sure that you do not add any descriptors to the array that may go out of scope or be otherwise deleted since the array would contain a TPtrC pointing to an undefined area.

6.7.3 Dynamic Arrays

Symbian provides a set of classes for implementing dynamic arrays. Like descriptor arrays, dynamic arrays are based on dynamic buffers, and are thus expandable. However, unlike descriptor arrays, templates are used so that the array can contain items of any data type, as defined by the user.

A wide assortment of dynamic array classes exists. Here is a sample of them:

- CArrayFixFlat<class T>: Holds fixed length objects of type T and uses a flat dynamic buffer
- CArrayFixSeg<class T>: Holds fixed length objects of type T and uses a segmented dynamic buffer
- CArrayPtrFlat<class T>: Holds pointers to type T objects using a flat dynamic buffer
- CArrayPtrSeg<class T>: Holds pointers to type T objects using a segmented dynamic buffer

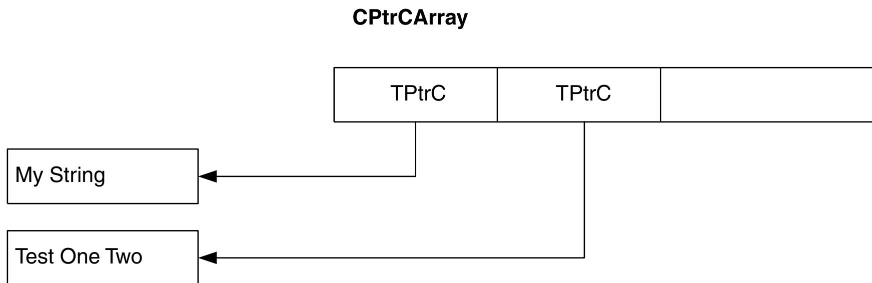

Figure 6.17 CPtrCArray

- `RArray<class T>`: A full-featured, efficient array class for fixed length data of type T, using a flat buffer
- `RPointerArray<class T>`: Same as `RArray`, but uses pointers to type `T` objects

Although the classes that begin with `CArray` provide more data representations, for the most part `RArray` and `RPointerArray` have practically everything you need and are the most efficient, so I will cover these in more detail.

RArray

An example declaration of an `RArray` is as follows:

```
RArray<MyObj> myArray;
```

This will define a dynamic array `myArray` that contains objects of type `MyObjs`. Note that this is similar to `TArrayFixed` except an array size is not specified with the template. This is because the size is not preallocated as it is for `TArrayFixed`.

The `R` at the beginning of `RArray` indicates that this class can be declared as an automatic variable; doing so will result in class data being stored on the stack. Unlike other `R` classes, this one also has heap storage associated with it – in fact, the dynamic buffer that holds the data itself resides on the heap. Therefore, you always need to call the `Close()` method of `RArray` to free the heap data storage when finished with the array.

In addition to these default declarations, the following constructors are also provided:

```
RArray(TInt granularity)
```

This specifies how the array is to be expanded in the background. `RArray<MyObj> myArray(10)` for example will cause the memory for `myArray` to be expanded in units of ten items. The default is eight.

```
RArray(Tint granularity, TInt key_offset)
```

In addition to granularity, this version of the constructor specifies the class offset of an integer that acts as a *key* for the list. A key is a data member of the array element class that is used as a reference value when searching, inserting, and sorting the objects in the array. For example, if you sort the array, the objects in the array are ordered based on comparing the array element's data members that are defined as the key.

The `_FOFF(class,membername)` macro is used to specify this key value location in the array data members.

This example creates an array of `TAccount` elements using `TAccount::account_number` as the array's key:

```
class TAccount

  {
  public:
    TAccount();

    TInt account_number;

  };

RArray<TAccount> myAccounts(10,_FOFF(TAccount,account_number));
```

Inserting and Appending Array Data

Methods: `Insert()`, `InsertInSignedKeyOrder()`, `InsertInUnsignedKeyOrder()`, `InsertInOrder()`, `InsertInOrderAllowRepeats()`

`Insert()` and `Append()` methods are provided to add data to the array. `Insert()` inserts the data at the specified position and shuffles everything up. `Append()` adds data to the object at the end of the array. If the number of data items exceeds the memory allocated by the array, the array memory is automatically expanded by the number of data items specified as the granularity.

You can also insert items in the list in key order using `InsertInSignedKeyOrder(const &T aItem)` and `InsertInUnsignedKeyOrder(const &T aItem)` functions. These functions insert `aItem` ordered by the value in the data object's key value whose location was given in the `RArray` constructor (assuming the correct overridden constructor was used). `InsertInSignedKeyOrder()` treats the key as a signed value; `InsertInUnsignedKeyOrder()` treats the key as an unsigned value.

To provide maximum flexibility, `RArray` also allows you to create your own callback function that will compare two array members (supplied as arguments to your callback) and determine their relation. The function returns a negative value if the first is less than the second, positive if it is more and 0 if equal. This callback is wrapped in a class called `TLinearOrder`. The function `InsertInOrder(const &T aEntry, TLinearOrder<T>aOrder)` is then used to insert the item in the order determined by the callback function of `TLinearOrder`.

`TLinearOrder` can also be supplied to the `RArray` sort methods so that the array can be sorted in a flexible way.

Finding Data in an Array

Methods: `Find()`, `FindInOrder()`, `FindInSignedKeyOrder()`, `FindInUnsignedKeyOrder()`

Use the find methods to locate an element within the array. These functions will return the index of the array element that matches specified data, or `KErrNotFound`, if the data was not found in the array.

`Find(const &T aObject)` does a linear search for an array element whose integer key matches the key value of `aObject` (`T` is the type specified in the template argument supplied when creating the array). You can also implement a callback function to do the element compares yourself (returns 1 if a match, 0 otherwise). Then wrap this callback in the class `TIdentityRelation`, and use `Find(const &T aObject, TIdentityRelation)` to find the object using your callback function for comparing elements.

The find functions that end in `Order()` assume that your array is in order so that a more efficient binary search can be done to find a match.

`FindInOrder()` does a binary search assuming that your array is already ordered by an ordering callback, supplied as a `TLinearOrder` type. `FindInSignedKeyOrder(const &T aObject)` and `FindInUnsignedKeyOrder(const &T aObject)` will do a binary search of the object assuming the list is currently ordered by key value (treating key value as signed and unsigned respectively).

Sorting an Array

Methods: `Sort()`, `SortSigned()`, `SortUnsigned()`

`Sort(TLinearOrder<T> anOrder)` sorts the array using the `TLinearOrder` callback.

`SortSigned()` and `SortUnsigned()` orders all the objects in the array by the integer key value assigned in the `RArray` constructor – treating the key as signed and unsigned respectively.

Miscellaneous Array Functions

To remove an element, call `Remove(TInt aIndex)`. The element at that position is removed and the list is shuffled down.

Call `Reset()` or `Close()` before destroying the array – these will free all the memory allocated on the heap. `Reset()` frees all heap memory and prepares the array for reuse.

An Example using `RArray`

Example 6.19 shows an example using `RArray`.

Example 6.19. `RArray` Example

```
class TTenant  // T class representing a tenant
  {
public:

  TTenant(const TDesC& aName,TInt aNum) { iName.Copy(aName),
          iApartmentNum=aNum; }

  TBuf<40> iName;
  TInt iApartmentNum;
  };

void TestArrayL()
  {
/* Define array of tenants, use the apartment number as the array's key */

  RArray<TTenant> renters(10,_FOFF(TTenant,iApartmentNum));

/* Add some tenants to the array, random apartment order */

  TTenant renter1(_L("Sue"),520);
  TTenant renter2(_L("Bob"),132);
  TTenant renter3(_L("Sally"),1004);

  User::LeaveIfError(renters.Append(renter1));// if append fails, leave
  User::LeaveIfError(renters.Append(renter2));
  User::LeaveIfError(renters.Append(renter3));

/* Sort array to be in order of apartment numbers (the key) */

  renters.SortUnsigned();

/* Insert new tenant, in apartment number order */

  TTenant newRenter(_L("Pippin"),755);

  User::LeaveIfError(renters.InsertInUnsignedKeyOrder(newRenter));

/* Print list of tenants, will be in order of apartment now */

  _LIT(KFormat1,"Name=%S, Unit=%d\n");

  for (TInt i=0;i<renters.Count();i++)
    {

    console->Printf(KFormat1,
    &renters[i].iName,renters[i].iApartmentNum);

    }
/* See who is at apartment 520 */

  TTenant findT(KNullDesC,520);
  TInt index;

  index = renters.FindInUnsignedKeyOrder(findT);

  if (index != KErrNotFound)
```

```
    {
    _LIT(KFormat2,"Who’s at apartment 520? %S!");
    console->Printf(KFormat2,&renters[index].iName);
    }

  renters.Close();  /* must be done to free the RArray heap memory */
  }
```

The example in Example 6.19 outputs as follows:

```
Name=Bob, Unit=132
Name=Sue, Unit=520
Name=Pippin, Unit=755
Name=Sally, Unit=1004
Who’s at apartment 520? Sue!
```

Example 6.19 illustrates how to set and use a key for the array, including insert data in key order, and finding data in the array using the key.

RPointerArray

`RPointerArray` is like `RArray` except that pointers to the objects are held in the array instead of copies of the objects. When using these, ensure that you do not delete the objects after they are added to the array (and before deleting the array) since this could result in an exception accessing allocated data.

`RPointerArray` is declared as follows:

```
RPointerArray<MyObj> myArray;
```

This is like `RArray`, except that in this case an array of pointers to type `MyObj` is stored. The granularity of the array can also be specified, but note that keys are not supported for this type of array.

Items can be added to the array via `Append(const T* anEntry)` and `Insert(const T* anEntry, TInt pos)` where `T` is the type passed in the template when declaring the pointer array. You can also insert objects in order of pointer addresses with `InsertInAddressOrder(const T* anEntry)` and `InsertInAddressOrderAllowingRepeats(const T* anEntry)`.

These functions return `KErrNone` if successful, otherwise they return a system error code.

Although integer keys are not supported as they are in `RArray`, you can specify your own function wrapped in `TLinearOrder` to perform sorting and to perform binary searches. See the SDK documentation for more detail on this.

6.8 Other Data Collection Classes

Symbian OS provides a wide assortment of data collection structures. They are too numerous to cover them all, but here is a useful subset.

6.8.1 Linked Lists

Class `TDblQue<class T>` can be used to create a doubly-linked list of objects of type `T`. The class that the array contains must have a member variable of type `TDblQueLink` (contains the forward and backward linked list pointers). Then when you construct the `TDblQue`, you specify the offset of the `TDblQueLink` member variable in the constructor.

Here is an example declaration

```
class CMyObj : public CBase
  {
  ...
  TDblQueLink iLink;
  ...
  };

...

/* construct list, supply offset of link member
 * variable in class
 */

  TbdlQue<CMyObj> linkList(_FOFF(CMyObj,iLink));
```

Objects are added to the linked list with `AddFirst()` and `AddLast()` to add to the beginning and end of the list respectively. `First()` and `Last()` will return pointers to the first and last elements of the list. To insert and delete items from a point in the middle of the list, use the current object's `TdblQueLink::Enque()` and `TdblQueLink::Deque()` methods respectively – these will insert and delete at that point in the list.

You can create an iterator to the list as follows:

```
TDblQueIter<MyObj> iter(linkList);
```

Use the iterator's `++` and `--` operators to traverse the list and return pointers to list items.

A singly-linked list is available via the `TSglQue<class T>`, `TSglQueLink` and `TSglQueIter<class T>` classes.

6.8.2 Circular Buffers

Use `CCirBuf<class T>` to create a circular buffer of objects of type `T`. Use the `TInt Add(const T* aPtr)` to copy the data from the class

pointed to by `aPtr` to the buffer. Items are removed using the `TInt Remove(T* aPtr)` which will copy the data at the current retrieve position in the buffer to the area pointed to by `aPtr`. The data is extracted in a first in, first out fashion

Before adding anything to the circular buffer, method `SetLengthL()` must be called to set the maximum length of the buffer. If the buffer fills up due to the data not being removed fast enough (via `Remove()`), then the next `Add()` method will return 0 indicating that the data cannot be added.

7

Processes, Threads and Synchronization

In Chapter 3, I gave an overview of the multitasking capability of Symbian OS, and introduced how processes and threads are used. This chapter continues that discussion by showing you specifically how to create and manage your own processes and threads using the Symbian OS API. I'll also describe how to synchronize, and otherwise communicate between threads, using inter-thread data transfers and shared memory regions as well as using semaphores, mutexes and critical sections.

Understanding the material in this chapter is not absolutely necessary for basic Symbian OS programming, since processes and threads are handled by the system for the most part. However, at some point you will find that you need to create your own processes or threads. For example, you may want to create a server that runs as a separate process, launched by your program when needed. Or you may want to create your own threads if you are porting code from an environment that relies heavily on them (however, in general implementing threads in your program is discouraged (see Section 7.2)).

Also, understanding the details of how processes and threads function and communicate will provide you with a deeper understanding of how the various frameworks, such as active object and client/server frameworks, operate (these frameworks are covered in Chapters 8 and 9).

7.1 Processes

A Symbian OS process is an executable that has its own data space, stack and heap. A process is contained in a file that ends in `exe`. Multiple processes can be active in memory at one time, including multiple instances of the same `exe` file.

By default, a process contains a single execution thread – its main thread – but additional threads can run in the process as well. A

process is switched to whenever one of the threads in that process becomes active.

Threads that run in the same process have access to the data space of that process and this makes exchanging data between these threads straightforward. However, exchanging data between threads in different processes is more involved. This is because a process cannot directly access memory that belongs to another process without causing a fatal exception.

7.1.1 An Example Process

Example 7.1 shows an example process that displays an information message every two seconds once it's started.

Example 7.1. Simple process

```
#include <e32base.h>

TInt E32Main()
  {
  LIT(KMsgTxt,"Process");
  for (Tint i=0;i<100;i++)
    {
    User::InfoPrint(KMsgTxt);
    User::After(2000000);
    }
  return(0);
  }
```

All processes contain the function `E32Main()`, which is where execution begins. When `E32Main()` exits, the process terminates.

Example 7.2 shows the `mmp` file which builds the process in an executable called `MyProc.exe`.

Example 7.2. Build file for MyProc.exe

```
// exe mmp file
TARGET     MyProc.exe
TARGETTYPE exe
SOURCEPATH ..\src
SOURCE     MyProc.cpp

USERINCLUDE       .
USERINCLUDE       ..\include
SYSTEMINCLUDE     \Epoc32\include

LIBRARY           euser.lib
```

7.1.2 Launching a Process

The following code loads and runs an instance of `MyProc.exe`:

```
void LaunchProcessL()
  {
  _LIT(KMyExeFile,"c:\\system\\programs\\MyProc.exe");

  RProcess proc;

/* This will launch the MyProc.exe, passing the specified command line
        data to it */

  User::LeaveIfError(proc.Create(KMyExeFile,KNullDesC));
  proc.Resume(); // start the process running

/* ... */

  proc.Close();
  }
```

`RProcess` is the core API class for representing and controlling a process. `RProcess` acts as a handle to a process (it's an `R` class) and allows you not only to launch new processes, but also to open a handle to an already running process. This means you can perform operations on that process, such as changing its priority, terminating it and retrieving information (e.g. memory usage) from it.

The code instantiates an `RProcess` object and then invokes its `Create()` method to load the `exe` file specified as the first argument for the new process. After calling `Create()`, the process is created, but suspended. To start the process you call the `Resume()` method, as in the example.

Note that `Create()` can fail (for example, if the `exe` is not found) and thus the error should be handled. In this case, the process leaves if `Create()` returns an error (using `User::LeaveIfError()`).

When you are finished with the process handle, you call `Close()` – this closes the process handle only, it does not stop the actual process itself.

Let's look more closely at the `Create()` function. There are a few overloaded versions of the `Create()` function (refer to the SDK API document for details), and the one used in the example is prototyped as follows:

```
TInt RProcess::Create(const TDesC& aExecutableFile,
const TDesC& aCommand,TOwnerType aType=EOwnerProcess)
```

aExecutableFile contains the path to the process executable file. aCommand is a descriptor containing a command line argument that specifies data to be passed to the process when launching. aType specifies handle ownership and has a default value of EOwnerProcess to indicate that this RProcess handle can be used by any thread in the creating process. If aType is set to EOwnerThread, then only the creating thread can access the process via this handle.

In some code, you may see a process launched with EikDll::StartExeL(), as the following example shows:

```
#include<EikDll.h>
/* ... */
LIT(KMyProcName,"c:\\system\\programs\\MyProc.exe");
EikDll::StartExeL(KMyProcName);
```

However, although this is used sometimes, it's not officially supported by Symbian OS (marked as 'for internal use') and, in fact, is deprecated from Symbian OS v8.0. You should use RProcess instead.

7.1.3 Setting and Retrieving Process Arguments

As mentioned in the previous section, you can pass a command line argument to your process via the second argument of RProcess Create(). For example:

```
void LaunchProcessWithArgL()
  {
  _LIT(KMyExeFile,"c:\\system\\programs\\MyProc.exe");
  _LIT(KMyExeFileCmd,"-x 20 -y 30");

  RProcess proc;

 /* This will launch the MyProc.exe, passing the specified command line
         data to it */

  User::LeaveIfError(proc.Create(KMyExeFile,KMyExeFileCmd));
  proc.Resume(); // start the process running
  }
```

This will pass the argument string "-x 20 -y 30" to the process created, which can then retrieve this argument by calling RProcess CommandLine() on an RProcess handle opened to itself, as the following shows:

```
TBuf<200> cmdLine;

RProcess me;
me.CommandLine(cmdLine);
```

By default, the `RProcess` class constructor opens a handle to the currently running process. I could have skipped declaring `me` altogether and replaced the last two lines with the line: `RProcess().CommandLine()`. This calls both the constructor and the `CommandLine()` method (this is a common practice).

When `CommandLine()` is called, it sets the passed descriptor, `cmdLine`, to contain the command line string ("`-x 20 -y 30`" if it was launched from our previous example).

7.1.4 Communicating with Other Processes

You can open an `RProcess` handle to some other, already running, process by calling the `Open()` method for `RProcess`. Once opened, the `RProcess` object acts as a handle to that process and you can then use other `RProcess` methods to operate on the referenced process. You can open the process by either its numeric process ID or its ASCII name.

The following code opens a handle to a process by its process ID:

```
RProcess myProcess;
myProcess.Open(AProcId);
if (rc != KErrNone)
  {
  /*  open failed, handle error */
  }

/* ... */
myProcess.Close();  // close handle when finished
```

`AProcId` is the ID (type `TProcessId`) of the process you want to open. Process IDs are represented by the `TProcessId` class, which is a simple wrapper class for an integer.

To get the numeric ID of a process, use the `RProcess Id()` method. For example, the line:

```
RProcess().Id();
```

will get the ID of the currently executing process.

Again, make sure you close the `RProcess` handle when you are finished, by using `Close()` – and as mentioned before, closing the handle does not terminate the actual process it represents.

`RProcess Open()` returns `KErrNone` if successful, otherwise it returns a system error. For example, if it cannot find the process ID supplied in the `Open()` argument, then it returns `KErrNotFound`.

7.1.5 Process Names

Each process has an ASCII name associated with it. The default name for a process is the name of the exe file that contains it, minus the directory path and the exe extension. You can rename a process (from the process itself, or from another process) using the RProcess Rename() method if desired.

You can open a handle to a process by its name using Open(TDesC& aName) in RProcess. Note, however, that the system automatically appends a UID and instance number to the name of the process, and you must supply this full name to Open(TDesC& aName) for it to succeed. This can make opening a process by name using Open(TDesC& aName) awkward.

A better way to open a process by name is to use the Open(TFindProcess& aFind) version of Open(). This call allows you to use a wildcard match of the process name, as in the following example:

```
RProcess proc;
_LIT(KMatchName,"MyProc*");
TFindProcess procName(KMatchName);
TInt rc = proc.Open(procName);
if (rc != KErrNone)
  {
  /*  open failed, handle error */
  }
```

This code will open the first process it finds that starts with 'MyProc', which our myproc.exe example would match. The actual process name will look something like:

```
"MyProc[00000000]0001"
```

which is the name followed by a program UID (set by the RProcess::SetType() method) and an instance number – the kernel appends the latter two items to the base process name automatically.

TFindProcess is a class inherited from TFindHandleBase, which is a generic class for searching through kernel objects of a specific type and returning matches based on supplied match strings. TFindProcess specifically looks for running processes whose names match the supplied string.

7.1.6 Querying the Phone's Running Processes

You can also use TFindProcess directly to traverse the list of processes in the system. It has a method called Next() that you can use to

step through the filtered list (filtered by the match pattern) one at a time, and get each process's full name.

The following example shows how you can use `TFindProcess` to query a list of running processes on the phone:

```
TFindProcess fp;
TFullName procName;
while(fp.Next(procName)==KErrNone)
  {
  console->Printf(_L("process: %S\n"),&procName);
  }
```

The default match string of `TFindProcess` is '*', so this example simply traverses the list of all processes in the system.

Let's extend the example a bit and print some information about each process using the `RProcess GetMemoryInfo()` method. This is shown in Example 7.3.

Example 7.3. Listing Running Processes

```
void ListProcessesL()
  {
  TFindProcess fp;
  TFullName procName;
  TProcessMemoryInfo memInfo;
  RProcess process;
  while(fp.Next(procName)==KErrNone)
    {
    User::LeaveIfError(process.Open(procName));
    process.GetMemoryInfo(memInfo);
    console->Printf(_L("process: %S\n"),&procName);
    console->Printf(_L("  code base=%x\n"),memInfo.iCodeBase);
    console->Printf(_L("  code size =%x\n"),memInfo.iCodeSize);
    console->Printf(_L("  const data
                  size=%x\n"),memInfo.iConstDataSize);
    console->Printf(_L("  initialized data size=%x\n"),
                  memInfo.iInitialisedDataSize);
    console->Printf(_L("  uninitialized data size=%x\n"),
                  memInfo.iUninitialisedDataSize);
    process.Close();
    }
  }
```

A sample portion of output from Example 7.3 follows:

```
Process: EKern[100000b9]
    code base = 5000d000
    code size = 172480
    const data size = 2584
    uninitialized data size = 7380
```

```
    initialized data size = 0
Process: EFile[100000bb]
    code base = 5016e8e4
    code size = 92896
    const data size = 296
    uninitialized data size = 3752
    initialized data size = 16
Process: Emon[00000000]0001
    code base = 50bac814
    code size = 10268
    const data size = 4
    uninitialized data size = 0
    initialized data size = 0
Process: EInfoServer[00000000]0001
    code base = 5020f954
    code size = 9692
    const data size = 0
    uninitialized data size = 0
    initialized data size = 0
Process: EwSrv[10003b20]
    code base = 502fc6a4
    code size = 158948
    const data size = 4
    uninitialized data size = 17764
    initialized data size = 260
```

The `RProcess` `GetMemoryInfo()` method is only supported by Symbian OS v7.0 and later. But, if you are using a pre-7.0 version, you can still use `TFindProcess` to list all the process names.

7.1.7 Process Priority

Each process is assigned a priority value. The kernel uses process priorities to decide which process to switch to when more than one process is ready for execution at a time.

Use `SetPriority()` in `RProcess` to set the priority for a process. The priority can be one of the following (from the `TProcessPriority` enum):

- `EPriorityLow (150)`
- `EPriorityBackground (250)`
- `EPriorityForeground (350)`
- `EPriorityHigh (450)`

As an example, the following line changes the priority of the current process to the highest priority:

```
RProcess().SetPriority(EPriorityHigh);
```

You can also set the priority of a process from another process. The following example creates a process and then sets its priority:

```
RProcess proc;
User::LeaveIfError(proc.Create(KMyProc));  // create new process
                                           // (or you could open it if
                                           // you want to change already
                                           // running process's priority).
proc.SetPriority(EPriorityLow);  // change priority
proc.Resume();  // start process running
```

Alternatively, you can set the process priority at build time, and not use `RProcess` at all. To do this, add the `epocprocesspriority` keyword to your `exe` program's `mmp` file. Set the keyword to `low`, `background`, `foreground` or `high` (e.g. `epocprocesspriority=foreground`).

To read the priority of a process, use the `RProcess Priority()` method – this will return the current priority of a process.

7.1.8 Terminating a Process

Typically, a process ends after running its course (i.e. returning from its `E32Main()` function). However, you can terminate a process before then by calling `RProcess Kill()`. `Kill(TInt aReason)` takes one integer argument – a code to indicate the reason the process was terminated. A process can kill itself, or any other process it has a handle to.

`RProcess Panic(const TDesC& aCategory,TInt aReason)` also terminates a process. `Panic()` indicates that some unrecoverable error was detected. `aCategory` is a string indicating the type of the panic; `aReason` gives a more specific error number.

7.1.9 Checking the Status of a Process

To check if a process is still running and, if not, how and why it was terminated, use the `RProcess ExitType()` and `ExitReason()` methods.

`ExitType()` returns `EExitKill` if the process has ended – either normally, via a return from `E32Main()`, or forcibly, via an `RProcess::Kill()` call. `ExitType()` returns `EExitPanic` to indicate that the process has been terminated by a call to the `RProcess Panic()` method. If `ExitType()` returns `EExitPending`, this means that the process is still running.

If `ExitType()` indicates that the process is no longer running (by returning `EExitKill` or `EExitPanic`), then you can call `ExitReason()` to get more information about the termination. `ExitReason()`

returns the value returned by the E32Main() function of the process for a normal exit, the reason code passed to the Kill() method where Kill() was called, or, if the process has ended due to a Panic(), then ExitReason() will return the panic code.

Why is a normal process exit reported in the same way as a forced process Kill()? The reason is that whenever a process returns from E32Main(), the system automatically calls Kill() on that process, passing E32Main()'s return value as the parameter to the Kill() function.

7.1.10 Signaling when a Process Ends

The RProcess::Logon() method can be used to wait for a process to complete as shown in Example 7.4.

Example 7.4. RProcess Logon

```
void StartProcessWaitEndL()
  {
  RProcess proc;

  User::LeaveIfError(proc.Create(MyExeFile));

  TRequestStatus istat;
  proc.Logon(istat);
  proc.Resume();

// Thread is executing. Can add code here to run in parallel...

  User::WaitForRequest(istat); // blocks here while process is running

// Process is ended, you can use proc.ExitType()
// proc.ExitReason() and proc.ExitCategory()
// to get information on how the process ended.
  }
```

RProcess::Logon() is an asynchronous function as indicated by its TRequestStatus argument. Asynchronous functions are covered in detail in Chapter 8 along with active objects, but for now it is sufficient to say that they always return immediately, but will send an event at a later time, when the function actually completes. The TRequestStatus variable will contain the status of the completion.

To realize the full power of asynchronous functions, you use an active object to set up a call-back to be run when the function completes. But you can also simply wait for completion of the function via User::WaitForRequest(), as we do in Example 7.4. Don't worry if you do not understand how asynchronous functions work

at this point. My purpose here was just to introduce the `Logon()` method.

Although the example code actually created the process in which `Logon()` was called, you can also use `Logon()` on processes which are already running, opened via `RProcess::Open()`. In fact, multiple programs can have process handles open to the same process, and all these programs could use `Logon()` – and so be notified when that single process ends.

7.1.11 Protecting a Process

You may want to prevent other processes from changing the priority of, or terminating, a particular process. Use the `RProcess SetProtected()` method for this – either on a `RProcess` handle opened on the process, or by calling `RProcess().SetProtected()` to protect the currently running process.

7.1.12 Other Process Facts

The following are other facts concerning processes:

- Switching between threads in different processes (and thus requiring a process switch) is expensive compared to switching between threads within the same process. The reason is that a process switch requires that the data areas of the two processes be remapped by the Memory Management Unit (MMU). Switching between threads in the same process involves no such memory mapping changes.
- Unlike in DLLs, you can have writable static data in a process executable (`exe` file).
- Although GUI applications are DLLs, they launch as separate processes. A process called `apprun.exe` is called behind the scenes that, in turn, launches the application framework, and your application.
- The emulator does not support multiple processes since it executes completely as a single process. Although Symbian OS version 8.0 will make the emulator behave more like the real device in this regard, in pre-v8 versions, you will have to simulate your process by using threads. Normally, this is done by creating a `DLL` (in place of the `exe`) and starting the `DLL`'s main function as a thread.

7.2 Using Threads

Threads form the basis for multitasking and allow for multiple sequences of code to execute at once. You can create multiple threads in your

program for parallel execution. However, in many cases the better way to go is to use asynchronous functions and active objects, so consider your use of threads carefully.

While Symbian OS relies on threads to implement its multitasking capabilities, you'll find that using multiple threads in your own program can sometimes be a problem. One reason is that some Symbian OS objects can only be used in the thread in which they are created. A common example is that only the main thread in a GUI program can draw to the screen – other threads can require a complex handshaking scheme to coordinate with the main GUI thread for screen updates.

So while operating systems such as Linux and Windows rely heavily on creating separate threads for applications, in Symbian OS it's best to avoid using real threads and instead use active objects (see Chapter 8). This is because active objects can simulate multithreaded behavior, while actually running in a single thread – thus avoiding threading problems such as the ones I mentioned.

However, you may find that creating your own threads is the best solution in some situations. Also, having an understanding of the way they work helps with understanding Symbian OS and its various frameworks better.

Symbian OS provides the `RThread` API class for creating and managing threads. Like `RProcess`, `RThread` is a handle class and the actual thread object is owned by the kernel. Also like `RProcess`, `RThread` is instantiated directly, and usually on the stack.

Example 7.5 shows an example of creating and starting a thread using `RThread`.

Example 7.5. Starting a thread

```
TInt threadFunc(TAny *)
  {
  for (TInt i=0;i<10;i++)
    {
    User::InfoPrint(_L("Thread"));
    User::After(4000000);
    }
  return(0);
  }
void StartThreadL()
  {
    RThread thd;
    User::LeaveIfError(thd.Create(_L("MyThread"),
           threadFunc,KDefaultStackSize,NULL,NULL));
    thd.Resume();
  }
```

In Example 7.5, function `StartThreadL()` will create a thread at function `threadFunc()`. A thread is created in the suspended

state, so, to start the thread, you need to call the `Resume()` method. Once `thd.Resume()` is executed, a separate thread starts at `threadFunc()` while the creating thread continues and returns from `StartThreadL()`. The created thread will display an information message (a message which stays up for a couple of seconds then disappears) ten times, at intervals of four seconds, and the function will then exit, thus ending the thread. So after `StartThreadL()` returns (after the `thd.Resume()`), you have two threads of execution within the process.

Note that the thread runs in the same process as `StartThreadL()`, so it has access to any public variables within the process – but, as mentioned, some Symbian objects created in one thread cannot be used in another without an exception being generated.

7.2.1 Creating a Thread

The `Rthread::Create()` method is used to create a new thread. Threads are not contained in separate executable files as processes are – they execute code in their parent process executable – however, each thread executes as an independent execution stream. A thread is associated with a particular function in the process, and that function's name is specified as an argument to the `Create()` method. The execution stream starts at that function call and ends when the function returns.

Let's look at the `RThread::Create()` method in more detail. There are a few overridden variations of this function, but they vary by only minor differences.

```
TInt Create(const TDesC& aName, TThreadFunction aFunction, TInt
        aStackSize,
RHeap* aHeap, TAny* aPtr, TOwnerType aType=EOwnerProcess)
```

- `aName` defines the name of the thread. This name can be used when opening up a handle (via another `RThread`) to this thread from another thread. Also, this name will appear in the exception pop-up boxes if a system exception occurs within the thread.
- `aFunction` specifies the function where thread execution starts. `TThreadFunction` is defined as:

  ```
  typedef TInt (*TThreadFunction)(TAny *aPtr);
  ```

- Upon return from this function, the thread automatically ends. `RThread::ExitReason()` can then be used to obtain the function's return value.

- aStackSize defines the size of the stack used by the process in bytes. The constant KDefaultStackSize can be used to indicate a default stack size.
- aHeap passes a heap via an RHeap object pointer. If the value is NULL, the heap of the creating thread is used. Note that there are other forms of Create() that allow a separate heap to be created automatically and function arguments are supplied for the minimum and maximum sizes of this heap.
- aPtr specifies the argument passed to the thread function defined in aFunction. NULL can be used if no argument is used.
- aType is EOwnerProcess by default. This indicates that this RThread handle can be used by any thread within the current process. aType can also be set to EOwnerThread to indicate that this RThread instance can only be used by the thread it was created in.

Like RProcess::Create(), RThread::Create() returns KErrNone if successful and a system error code otherwise.

7.2.2 Opening an Existing Thread

As in the case of a process, a handle to an existing thread can be opened by either name or ID using the RThread::Open() method.

Example 7.6 will open the thread created from Example 7.5 and (just to create interest), if it is still running, will suspend it. At this point the thread will be suspended until a Resume() is performed.

Example 7.6. Opening, Suspending and Resuming a Thread

```
RThread thd;
TInt rc=thd.Open(_L("MyThread"));
if (rc != KErrNone)
  {
  /* handle open error */
  }
if (thd.ExitType() == EExitPending)
     thd.Suspend();

...
thd.Resume();  // continue thread execution
```

Since the system does not append any numbers to the end of thread names (unlike with processes) you do not normally need to use the partial name matching version of the RThread::Open() method.

However, a `TFindThread` class is supplied that works in the same manner as the `TFindProcess` discussed earlier, in that it can use a pattern to match the thread name. A `TFindThread` object can be passed to `RThread::Open()` instead of the thread's full name, if desired.

You can also open a thread by its integer ID, which is represented by the type-safe class `TThreadId`. Like `RProcess`, `RThread` has an `Id()` method that returns the thread's ID. Also, as in the case of processes, you need a way of supplying, at runtime, this ID to the process and thread that need to open your thread. This is because thread IDs vary on each program run, unlike thread names which are constant.

7.2.3 Thread Priorities

A thread's priority can be set relative to the priority of its owning process or to an absolute priority, independent of the priority of its owning process. Thread priorities are set by the `SetPriority()` method.

Symbian OS defines the following process-relative priorities:

- `EPriorityNull` (−30),
- `EPriorityMuchLess` (−20)
- `EPriorityLess` (−10)
- `EPriorityNormal` (0)
- `EPriorityMore` (+10)
- `EPriorityMuchMore` (+20)
- `EPriorityRealTime` (+30)

The default thread priority is `EPriorityNormal`, which means that the thread's priority is the same as that of the owning process. The other values indicate a thread's priority in relation to the priority of its owning process. The numbers in parentheses indicate the values that are added to the priority of the process to form the thread's absolute priority. As the priority of the process is changed, the relative priorities of all its threads are automatically adjusted.

Figure 7.1 shows the relative thread priorities when, for example, the process priority is `EPriorityForeground`.

If you do not want your thread's priority to be set relative to the process priority, you can use an absolute priority instead. Absolute priorities stay fixed, regardless of the process's priority. Figure 7.2 shows the absolute

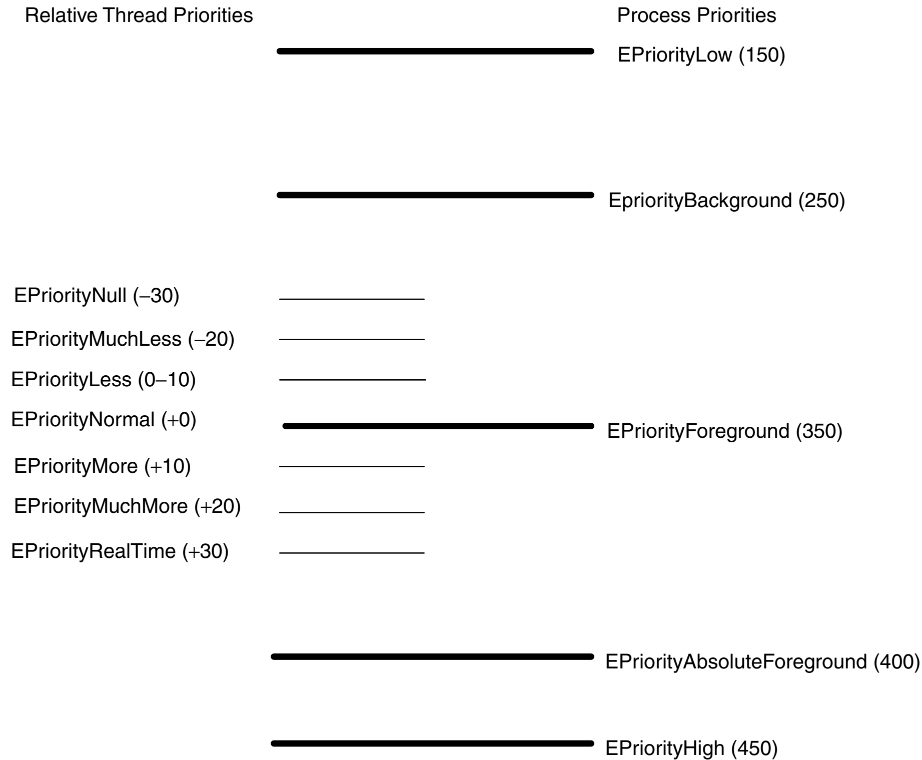

Figure 7.1 Relative Thread and Process Priorities

thread values that can be used, and how they relate to process priority values.

You use the same `SetPriority()` method to set both relative and absolute priorities. The function automatically determines if the priority is relative or absolute by the argument's `enum` value.

7.2.4 Terminating a Thread

You can use `RThread::Kill(TInt aReason)` to terminate a thread, either remotely or from within the thread itself. Also, as in `RProcess`, `RThread` provides the following methods for determining why a process has ended: `ExitType()` and `ExitReason()`.

`ExitType()` returns one of the following values:

- `EExitKill` means that the thread function returned or that the `Kill()` method was explicitly called.

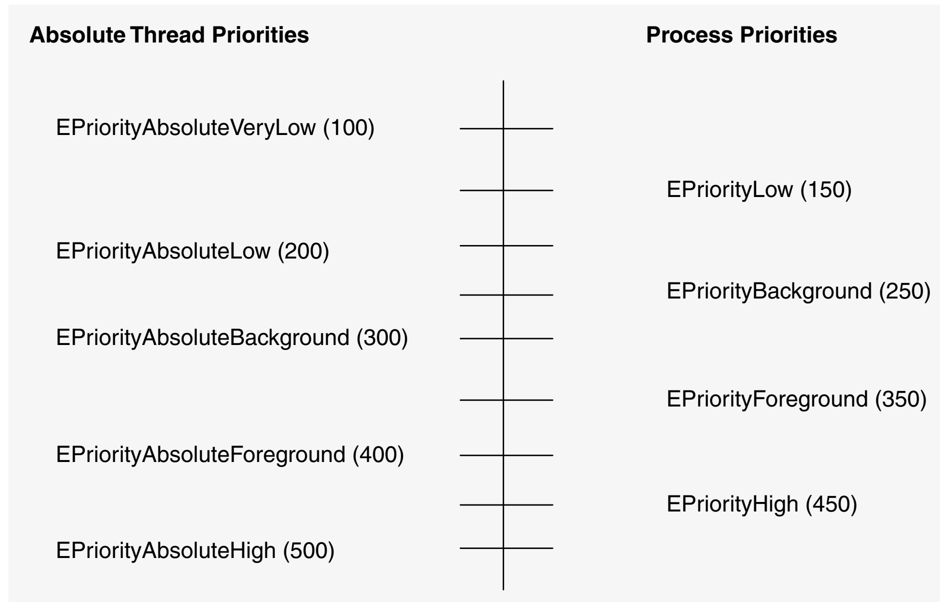

Figure 7.2 Absolute Thread and Process Priorities

- `EExitPanic` means the thread ended due to a panic.
- `EExitPending` means that the thread is still running.

`ExitReason()` returns one of the following values:

- the return code if the thread function returns normally.
- the termination reason code if the thread ends via a `Kill()` call.
- the panic reason code if the thread exits due to a panic.
- zero if the thread is still running.

7.2.5 Waiting for a Thread to End

Similarly to `RProcess`, `RThread` has a `Logon()` method (`RThread::Logon(TRequestStatus aStat)`) that can be used to wait for a thread to end. You can wait for the signal via the `User::WaitForRequest(iStat)` call or use an active object such that the active object's `RunL()` method is called when the thread completes.

7.3 Sharing Memory Between Processes

Processes cannot directly access each other's memory space. For example, if you obtain a pointer to a memory buffer that resides in some other process, and then try to directly read from or write to that buffer using the pointer, you will raise an exception. In fact, the memory pointed to by that pointer no longer even points to the intended data since the memory space of a process is moved to a different area when it is not active.

Let's look at an example, where we assume that process A has a data buffer defined as a descriptor:

```
//process A
TBuf8<300> processAData;
```

Suppose you want to write to this buffer from process B which has obtained a pointer to it, called `processADataPtr`. (A common way to pass a pointer like this is via a client/server message, as you will see in Chapter 9.)

You might consider trying the following in process B:

```
(*processADataPtr).Copy(someData);
```

This would work if `processADataPtr` pointed to a descriptor in the same process, but it does not work on a pointer from another process, since the memory space of that process is swapped out and now resides somewhere else.

The correct way to do this is with the `RThread` `WriteL()` method, as the following example shows:

```
//process B
RThread thd;
User::LeaveIfError(thd.Open(processAThreadId));
thd.WriteL(processADataPtr,_L("data"));
thd.Close();
```

This code opens an `RThread` handle to a thread in process A, then calls `WriteL()` to write the data `"data"` into the process A descriptor.

Note that process B needs the thread ID (or name) of process A's main thread (or, in fact, of any thread in process A) in addition to a pointer to the descriptor buffer to write to.

If process B wants to read the process A buffer, it uses `ReadL()` in the following way:

```
TBuf<200> myBuffer;
thd.ReadL(processADataPtr,myBuffer);
```

7.3.1 ReadL() and WriteL()

ReadL() and WriteL() are RThread functions rather than RProcess functions, which would seem to make more sense. This can be confusing, since it's entirely possible to directly access memory between threads in the same process without using ReadL() and WriteL(), since they all share the memory space of their parent process. Threads in the same process can exchange pointers or even directly access the global static data of their process. It's only across different processes that you must use the ReadL() and WriteL() functions.

However, it does not hurt to use ReadL() and WriteL() between threads in the same process – in that case it will just transfer the data by direct use of the pointer. So it's always safe to use these functions when transferring data between threads.

The prototype of ReadL() is:

```
void ReadL(const TAny* aSrcPtr,TDes &aDes,TInt aOffset)
```

- aSrcPtr specifies the buffer to read from, in the memory space of the other thread. This should be a pointer to a descriptor (TDesC) in the other thread's data space. Although aSrcPtr is of type TAny*, the method performs a sanity check to ensure that it is pointing to an actual descriptor, and a fatal exception will occur if does not.
- aDes is the destination descriptor in the current thread to which the data from the other thread is copied.
- aOffset is the byte offset from the start of the source descriptor where the copy should begin.

The syntax of WriteL() is:

```
void WriteL(const TAny* aDestPtr,const TDesC& aSrc,TInt aOffset)
```

- aDestPtr specifies the destination data region for the transfer. This should be a pointer to a descriptor (TDes) in the other thread's data space. As in the case of the source buffer in ReadL(), a sanity check is made to ensure that the pointer is pointing to an actual descriptor.
- aSrc specifies a descriptor in the current thread where the data is copied from. The size of this source descriptor determines the transfer size.
- aOffset specifies the offset in the destination descriptor for the copy to start. The resulting length of the destination descriptor will be the length of the source descriptor plus this offset.

Both ReadL() and WriteL() have versions that transfer data using 8-bit descriptors, in addition to 16-bit descriptor versions.

Note that these functions *always* require the buffers to be specified as descriptors, so if you have a buffer that is not a descriptor (e.g. a static char array), then you will need to wrap it in a descriptor, such as a TPtr, before using these functions.

7.3.2 Inter-Thread Memory Access – Background Information

Some of you may be interested in what's going on in the background when transferring data between processes. This section can be skipped unless you are interested in more of the architecture behind inter-process memory handling.

As discussed in Chapter 3, the data area of all process instances resides in an area of memory known as the Home area. When a process becomes active (only one is active at a time) then its data memory is mapped by the processor MMU from the Home area, to an area of memory known as the Run area – which is a sandbox for the use of the current process (refer to Figure 7.3).

The left side shows how memory looks when process A is running. Process A's data has been mapped to the Run area, and process B, since it is not active, remains in the Home area.

Now imagine that process A creates a pointer to myArray and sends this pointer to process B. This pointer, since it was assigned while process A was active, points to the myArray's location in the Run area. When process B becomes active, however (see right side of figure), the pointer will no longer point to myArray since:

- myArray[] is no longer at the run address the pointer was set to – it's now in process A's Home area.
- Process A's Home area is protected and an exception would result if it were accessed.

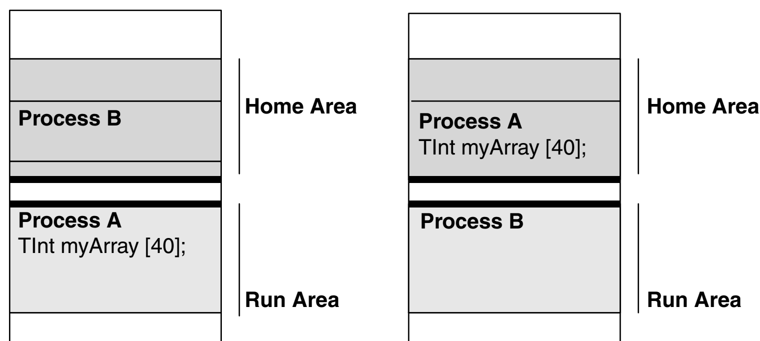

Figure 7.3 Process Pointers

This is where the inter-thread transfer methods ReadL() and WriteL() in the RThread class come in. These functions will convert the myArray pointer from its process A Run area address to the process A Home area – where the array resides now that process A is not running. Also, since the Home area is protected memory, the functions will get the needed CPU privilege required to access this Home area so they can read it or write the data to it.

Note that if you call ReadL() or WriteL() to transfer data between threads in the same process, no conversion between Run area and Home area is required. This is because the same shared parent process is active in the Run area for both threads and thus the data remains in the same Run area spot. In that case, ReadL() and WriteL() simply transfer the data using the passed buffer pointers with no address conversion.

7.4 Memory Chunks

In addition to the inter-thread access functions (ReadL() and WriteL()), Symbian OS also provides support for shared memory regions that can be directly accessed across multiple processes. These shared memory regions are known as *global memory chunks*. You can create your own memory chunks or access existing memory chunks by using the RChunk API class.

Let's look at a simple example of creating and using a global memory chunk.

```
//Process A
RChunk chk;
_LIT(KChunkName,''My Global Chunk'');
TInt rc=chk.CreateGlobal(KChunkName,0x1000,0x1000);
if (rc != KErrNone)
  {
  /* error occurred creating chunk, handle here */
  }

TInt *ptr=(TInt *)chk.Base();
//write some data into chunk using *ptr
```

This code creates a global memory chunk named "My Global Chunk", and initializes it with some data.

Any other program in the system can read and write this memory chunk if it knows the chunk's name, as shown in the following:

```
//Process B
RChunk chk;
_LIT(KChunkName,"My Global Chunk");
TInt rc=chk.OpenGlobal(KChunkName,0);
if (rc != KErrNone)
```

```
  {
  /* error occurred, handle here (e.g. KErrNotFound is
  returned if it cannot open the chunk. */
  }
TInt *ptr=(TInt *)chk.Base();
//read from or write some data into chunk using *ptr
```

Global chunks are created via the `RChunk CreateGlobal()` method. The first argument is the name of the chunk. The next two arguments specify the physical RAM assigned to the chunk (known as committed memory) and the amount of virtual memory to reserve for the chunk.

To understand this, let's briefly review the concepts of virtual memory and physical memory.

All addresses used by software are virtual memory addresses. There are 4 GB of virtual memory in the system. Virtual memory is only usable by software when it is mapped to physical memory – i.e. actual RAM that resides on the smartphone. Virtual memory is mapped to physical memory by the CPU's Memory Management Unit (MMU), in units of the memory page size (usually 4 KB). When virtual memory has physical memory mapped to it, it is considered as committed. Virtual memory addresses are very plentiful, while physical memory is a scarce resource. Refer to Chapter 3 for more details of memory usage in Symbian OS.

Figure 7.4 shows the chunk memory layout.

The committed size (the second argument of `CreateGlobal()`) specifies the size of the memory in the chunk that you can actually read and

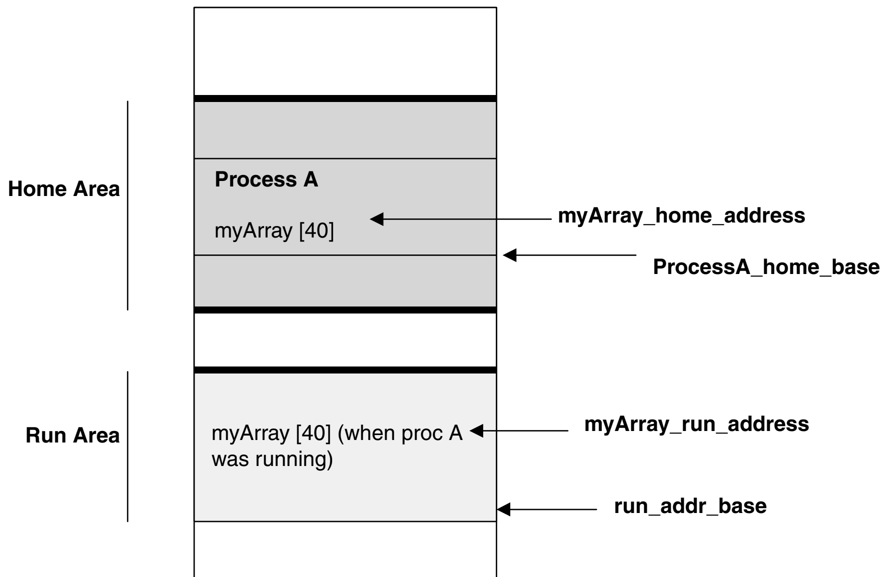

Figure 7.4 Layout of the Memory Chunk

write. You can reserve a larger block of virtual memory (via the third argument) when creating the chunk so that you can expand the chunk's committed memory, while keeping it contiguous.

To expand the chunk's committed memory size, use the `RChunk::Adjust(TInt newSize)` method, where `newSize` specifies the new size of the committed physical memory to the chunk (starting from its base address). The committed memory can be expanded up to the reserved maximum size specified in the third argument of `CreateGlobal()`.

For example, say you create a chunk that has 0x1000 bytes of RAM committed to it, with a maximum size of 0x5000:

```
chk.CreateGlobal(KChunkName,0x1000,0x5000);
```

At this point you only have 0x1000 bytes of physical RAM assigned to your chunk to read and write. But you can expand the chunk later, for example by another 0x2000 bytes:

```
chk.Adjust(0x3000)
```

Now your chunk has 0x3000 bytes of memory assigned, and since you had reserved 0x5000 bytes of virtual memory, the chunk memory stays contiguous up to that maximum.

`RChunk Base()` is used to get a pointer to the chunk's memory area. This pointer can be used to write and read the chunk directly as needed (note however that it is the programmer's responsibility not to go out of bounds).

`RChunk OpenGlobal()` is used to open an already created global chunk for access. The first argument to `OpenGlobal()` is the full chunk name. The second argument is used to indicate if the chunk is read only (1) or writable (0). An `Open()` method also exists that uses the `TFindChunk` matching class to open the chunk by a partial name. `TFindChunk` operates similarly to `TFindProcess` and `TFindThread` (in fact you could easily convert Example 7.3 to output all the global chunk names in the system using `TFindChunk`).

When a process is finished with the chunk, the `RChunk Close()` is called. When the last reference to the global chunk is called, the global chunk itself is automatically deleted.

7.4.1 Local Memory Chunks

In addition to global memory chunks, Symbian OS also provides local memory chunks. Local chunks are similar to global chunks except that they can only be accessed by the process that created them. Therefore, local chunks are not useful for sharing data between processes.

To create a local chunk, you use the `CreateLocal(TInt aSize, TInt maxSize)` function of `RChunk`, where the sizes represent the committed memory and reserved memory of the chunk. Note that, in this case, there is no name associated with the chunk, and you just access the chunk via the `RChunk` handle that was used to create it.

You will rarely, if ever, need to use a local chunk yourself. However, Symbian OS does make extensive use of them internally, as will be discussed further in the next section.

7.4.2 Chunks – Background Information

This section includes some detailed information for those interested in the inner workings of how the kernel handles memory chunks.

In addition to any memory chunks that you create yourself, memory chunks are also used internally by Symbian OS to manage process memory. In fact, chunks are the basic atoms of memory management used by the kernel for representing the various data areas belonging to processes and threads – including stacks, heaps and other writable data areas. (Actually, by default a stack and a heap are combined in a single chunk.)

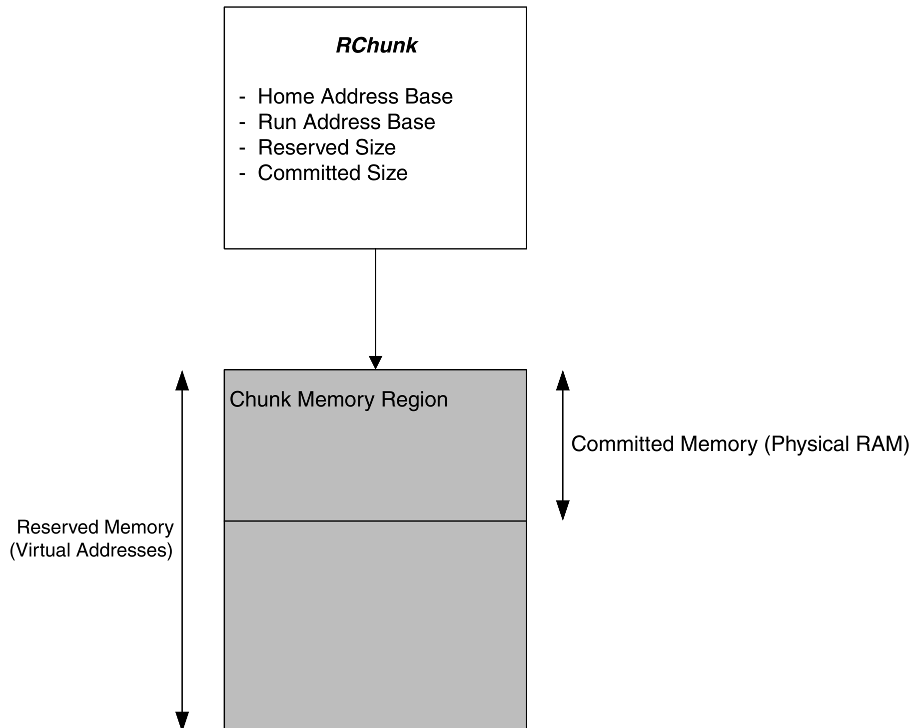

Figure 7.5 Global Chunk

Upon creation, the kernel assigns a memory chunk two addresses: a unique base address in the home section of memory – where the chunk resides when no one is using it, and an address in the Run area – where the chunk is mapped to when a running process needs to access it. So when you call the `Base()` method, you get the Run area address, since the chunk is always mapped to the Run area when a process with an open handle to the chunk becomes active.

If a process has handles to one or more global chunks (via `RChunk OpenGlobal()`, `Open()`, or the original `CreateGlobal()` methods), the kernel will remap these chunks to the run area, along with the rest of the data for that process, when it is switched to. Otherwise, global chunks remain in their Home areas of memory (see Figure 7.5).

Local chunks are associated with one only process. The kernel creates local chunks for a process's static data and for its stack/heap. As mentioned earlier, you can also create your own local chunks.

When a local chunk is created by a process, the system will add it to the list of chunks that the process owns. As shown in Figure 7.6, all the chunks owned by a process are moved as a group to their individual Run area addresses whenever that process becomes active.

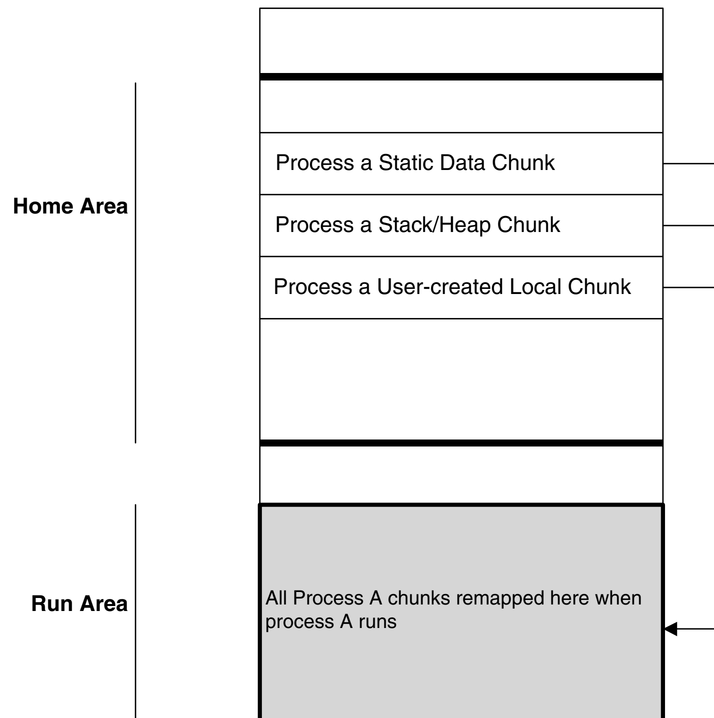

Figure 7.6 Local Memory Chunk

7.4.3 Code Chunks

Although not shown in Figure 7.6, a process also has a code chunk that contains the actual executable code of the process. Unlike with data chunks, a process executable only ever appears in a single code chunk, shared by all running instances of a process. This is because code is read-only, and thus we do not need a copy for each instance, unlike data chunks where each instance will have different data. Also unlike data chunks, a code chunk is not mapped to the Run area when the process is active.

7.5 Thread Synchronization

Being able to execute code in parallel, using threads, is a powerful feature, but it would not be very useful if there was no way to synchronize between them. After all, running in parallel is efficient – but without any coordination between the parallel strands of execution, chaos would result.

Symbian OS provides several API classes for synchronization of threads. In this section I will briefly cover three basic thread synchronization functions: semaphores, mutexes and critical sections.

7.5.1 Using Semaphores

You can use a `semaphore` either for sending a signal from one thread to another, or for protecting a shared resource from being accessed by multiple threads at one time.

A semaphore is created and accessed with a handle class called `RSemaphore`. You can create a global semaphore which can be opened and used by any process in the system, or you can create a local one that can only be used by the threads in your process.

The following is a simple example of using a semaphore. Assume you have two threads: Thread A and Thread B. Assume that Thread A needs to wait for a signal from Thread B before it can process some data. This can be accomplished with the following:

```
_LIT(KMySymName,"My Semaphore");

// Thread A:
/* ... */

RSemaphore sem;
TInt rc=sem.CreateGlobal(KMySymName,0);
if (rc != KErrNone)
  {
```

```
  /* error occurred creating semaphore, handle it */
  }

//have to wait for semaphore signal from ThreadB

sem.Wait();

/*... signal received, ok to process data */
```

Thread B signals Thread A when ready by:

```
//ThreadB:

RSemaphore sem;

TInt rc = sem.OpenGlobal(KMySymName);
if (rc != KErrNone)
  {
  /* error occurred opening semaphore, handle it */
  }

// do some stuff
// now send a signal to thread A so it knows it can continue

 sem.Signal();
```

`CreateGlobal()` indicates that a global semaphore is created. Since the semaphore was created using this function, Threads A and B in the example need not be in the same process. The first argument of `CreateGlobal()` is the name of the semaphore (like chunks, global semaphores have names, local ones do not). Once the global semaphore is created, it can be opened using the `OpenGlobal()` method as Thread B does.

The second argument to `CreateGlobal()` is a token count. Semaphores handle tokens as follows: a semaphore is created with an initial number of tokens. `Signal()` increments the semaphore's token count by one. `Wait()` decrements it by one. If `Wait()` finds that the decremented token count has become negative (i.e. there are no more tokens), then `Wait()` blocks, not returning until the token count is incremented by a `Signal()` call.

Since the initial token count in our example is 0, then if the `Wait()` in thread A happens before the `Signal()` in thread B, then `Wait()` will not return until the `Signal()` in thread B is called. If the `Signal()` in thread B occurs first, then the `Wait()` in thread A will return immediately since the token count was 1 before the call.

The preceding example used the semaphore as a straight signal – thread B sends a signal to thread A. You can also use a semaphore to protect a shared resource, as the following example shows:

Example 7.7. Semaphore used to Protect a Shared Resource

```
//ThreadA

RSemaphore sem;
TInt rc=sem.CreateGlobal(KMySymName,1);
if (rc != KErrNone)
  {
  /* error occurred creating semaphore, handle it */
  }

...

sem.Wait();
// access shared resource A

sem.Signal();  // signal that access is finished

//ThreadB
RSemaphore sem;
TInt rc=sem.OpenGlobal(KMySymName);
if (rc != KErrNone)
  {
  /* error occurred opening semaphore, handle it */
  }

sem.Wait();
// access shared resource A

sem.Signal();

..
```

For this example, assume that the two threads should never access resource A at the same time. To guard against this the example uses a semaphore. Thread A creates a semaphore and takes the single semaphore token (note the second argument of the `CreateGlobal()` call in Thread A) by calling the `Wait()` method before accessing resource A. If thread B gets to the code that accesses Resource A, it will block at the `Wait()` function until thread A replaces the `semaphore` token with the `Signal()` function, at which time the `Wait()` call in Thread B will take the token and return, allowing access to resource A.

In the preceding example, only one token exists in the semaphore so only one access to the shared resource is allowed. In some cases, however, you may want to allow multiple accesses of a resource up to a limit. In that case you would initialize the token count to the maximum number of parallel accesses you want to permit. For example, if a semaphore was initialized with a token count of five, then areas protected by the semaphore can be entered up to five times without waiting for one to exit. A sixth one however will block at `Wait()` until one of the other five leaves the area (indicated by calling `Signal()`).

When you are finished with an `RSemaphore` handle, call the `Close()` method.

7.5.2 Creating and Opening Semaphores

The syntax for creating a global semaphore is:

```
TInt RSemphore::CreateGlobal(const TDesC& aName,TInt aCount, TOwnerType
        aType=EOwnerProcess)
```

- `aName` specifies the name of the global semaphore.
- `aCount` is the initial token count for the semaphore.
- `aType` specifies the ownership of this handle and can be `EOwnerProcess` (the default) or `EOwnerThread`. `EOwnerProcess` indicates that this semaphore handle can be accessed anywhere in the process, whereas `EOwnerThread` indicates that it can be accessed only by the creating thread.

A global semaphore can be opened by name from any process or thread in the system using either:

```
RSemaphore::OpenGlobal(const TDesC& aName,TOwnerType aType=EOwnerProcess)
```

or:

```
RSemaphore::Open(const TFindSemaphore& aFind,TOwnerType
        aType=EOwnerProcess)
```

The first function will open the semaphore by its full name. The second will open it by a partial name, containing wildcard characters, using the `TFindSemaphore` class.

`TFindSemaphore` should look familiar to you – it works like `TFindProcess`, `TFindThread` and `TFindChunk`.

You can also create a local semaphore by using:

```
TInt CreateLocal(Tint aTokenCount, TOwnerType aType=EOwnerProcess)
```

In this case, the semaphore has no name and so cannot be opened by another process. Thus, you do not open a local semaphore – you simply access it via the `RSemaphore` handle that was used to create it.

Furthermore, if you specify `aType` as `EOwnerThread`, but want to use the semaphore in another thread, you must use the `Duplicate()` method to create a copy of the handle for that thread (for further information, refer to the SDK documentation for `RHandleBase::Duplicate()`).

7.5.3 Symbian OS Usage of Semaphores

Symbian OS automatically creates a semaphore, known as a request semaphore, for each thread on creation. This request semaphore is the basis of the Symbian asynchronous request functionality used by the active object framework which is in turn used in the client/server framework. When an asynchronous function is launched, the calling program's request semaphore is used to signal to the calling program that the function (which is running in a separate process/thread) has completed. The function `User::WaitForRequest()` includes the execution of a `Wait()` on this request semaphore.

It is not, therefore, very common to have to use semaphores directly in your programs. But you will, almost certainly, use semaphores indirectly, through asynchronous functions and active objects. Chapter 8 discusses these asynchronous functions in more detail and describes how the request semaphore is used.

7.5.4 Mutexes

A mutex is used to protect a shared resource that can only be accessed by one thread at a time. It acts like a semaphore that's been initialized with a token count of one.

A mutex is represented by the handle class `RMutex`. Aside from the fact that you can't specify a specific token count upon creation of a mutex, the `RMutex` class is otherwise equivalent to `RSemaphore`.

7.5.5 Critical Sections

Critical sections are regions of code in a process that should not be entered simultaneously by multiple threads. An example is a code region that manipulates static data, since it can obviously cause problems if multiple threads are accessing the static data simultaneously. Symbian provides the `RCriticalSection` class for this purpose. `RCriticalSection` is very similar to `RMutex` except that it is always local to the process. A critical section is created and used as shown in the following lines:

```
RCriticalSection crit;
crit.Wait();

// non reentrant code section.

crit.Signal();
```

In this example, the non-reentrant code section will only be able to be accessed by one thread at a time. A second thread that attempts to execute this same code region will block at the `Wait()` call until the first thread has finished executing the region and calls the `Signal()` method.

Incrementing and decrementing global integer variables is a common situation in which critical sections are needed. Since it is awkward to create and surround these simple operations with critical section calls, Symbian provides some static functions in the `User` API class as a convenience. The functions are:

```
User::LockedInc(TInt& aValue)

User::LockedDec(TInt& aValue)
```

These functions will respectively increment and decrement the static value whose reference is passed in `aValue` in a safe way – without requiring you to explicitly use an instance of `RCriticalSection`.

8

Asynchronous Functions and Active Objects

Although Symbian OS allows you to create preemptively scheduled threads via `RThread`, you'll find that in most programs you write you rarely need to (or should) create threads yourself – even where you would normally create a thread in another operating system (e.g. Unix or Linux). Instead, the preferred option is to have your program run as a single, event-driven thread using *asynchronous functions* and *active objects*.

Most functions are considered synchronous in that they return only after they complete. Asynchronous functions, on the other hand, are functions that return immediately and execute in parallel with the calling program (they run in separate threads in the background), sending an event to your calling program when execution is complete. Many of the Symbian OS API functions are asynchronous functions, and using them provides you with parallel operation in your program, since you can have multiple asynchronous functions executing at the same time. Active objects are classes used to invoke an asynchronous function, and to handle the completion of the asynchronous function via a callback.

Asynchronous functions and active objects are the foundation of the event-driven operation of Symbian OS, and mastering their use is essential to becoming a good Symbian programmer. This chapter looks at asynchronous programming in Symbian OS and describes how to use and implement its asynchronous functions and active objects.

8.1 Asynchronous Functions

You can identify an asynchronous function in Symbian OS by its inclusion of an argument of type `TRequestStatus`. `TRequestStatus` is a type class (mapped as a simple integer) that represents the status of the asynchronous function – that is, whether the function is in progress or has finished and, if it has finished, what its final status is. Many of the Symbian OS API functions come in both synchronous and asynchronous versions.

As an example, consider a traditional synchronous function called MyFunc():

```
TInt MyFunc(TAny *someArg);
```

This function returns after it is completely finished and the return value indicates the status of the function.

An asynchronous version of MyFunc() would look like:

```
void MyFunc(TAny *someArg, TRequestStatus& aStatus);
```

This function may perform the same functionality as the synchronous version, but instead of returning when it is completed, the function returns immediately with aStatus set to KRequestPending to indicate that MyFunc() is executing in parallel with the calling thread. When the asynchronous function completes, aStatus changes to the function's final status (KErrNone if successful).

At some point you will need to know when the asynchronous function has completed. You could wait for the function to complete as follows:

```
TRequestStatus status;
MyFunc(anArg, status);
// Code may be here which executes in parallel to MyFunc()
while (status == KRequestPending) ;
```

But polling the TRequestStatus variable like this is clearly wasteful. A more efficient option is to use User::WaitForRequest(TRequest-Status&aStatus), which will block your calling thread until the asynchronous function has completed (the name WaitForRequest means that it is waiting for the asynchronous request to complete) but your thread will yield control to other threads while it is waiting.

For example:

```
MyFunc is complete, status now contains the function status
```

How is this blocking possible? User::WaitForRequest() waits at a special semaphore owned by the calling thread, known as the *request semaphore*. Each thread has a request semaphore associated with it that is created automatically for you by the operating system. When an asynchronous function has completed processing, it first sets the TRequestStatus variable which was passed to the function to its final state, and then signals the calling thread's request semaphore. User::WaitForRequest(status) returns when the calling thread's

request semaphore is signaled, and `status` has a value other than `KRequestPending`.

`User::WaitForRequest()` is sometimes the most convenient way to wait for an asynchronous function to complete. For example, you may want to call a function in a synchronous way, yet only an asynchronous version of the function exists. However, in most cases, `User::WaitForRequest()` should not be used since it causes your thread to stop executing until the called function is completed, and thus defeats the whole purpose of using an asynchronous function.

The best way of using asynchronous functions (and what they were really designed for) is through active objects.

8.2 Introducing Active Objects

Active objects are classes derived from `CActive`. You use an active object to invoke an asynchronous function (via a method implemented in the `CActive` derived class). Then, when the asynchronous function completes, a system component known as an *active scheduler* invokes the `RunL()` method of the active object. `RunL()` is a virtual method in `CActive` that is implemented in the derived active object. You can have more than one active object active at a time, processing asynchronous completion events as they occur.

Figure 8.1 shows a high level view of the functionality of active objects.

A thread using active objects consists of one or more active objects and an active scheduler. The active scheduler is an instance of a class called `CActiveScheduler`. Active objects are added to the active scheduler using `CActiveScheduler::Add()`.

The active scheduler implements an event loop that waits on the thread's request semaphore and, when an event is received, invokes the `RunL()` method of the active object that the event belongs to. `RunL()` handles the asynchronous function's completion event and could, in turn, invoke further asynchronous functions. The active scheduler then waits on the request semaphore for the next event. The event loop is invoked by the `CActiveScheduler::Start()` method and, once it is running, everything in the thread is executed through the `RunL()` functions of the active objects.

8.2.1 The Non-Preemptive Multitasking Model

The active scheduler's event loop, along with its active object's `RunL()` invocations, all occur in the same thread, implementing what is known as non-preemptive multitasking. This means that, unlike with threads, one active object cannot start running while another one's `RunL()` is in progress, since they are executed as part of a loop in a single thread. The

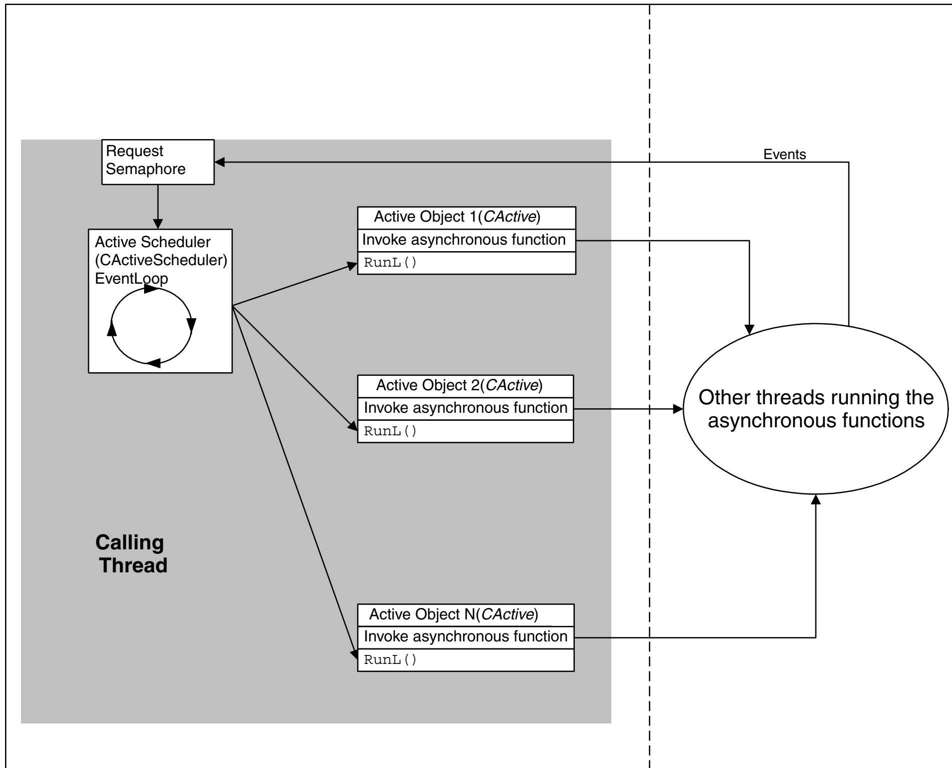

Figure 8.1 Active Object High-Level View

`RunL()` of the currently running active object must completely finish and return before the active scheduler can check for another event and call another active object's `RunL()` method.

The important thing to remember from the model just described is that you should not spend much time inside your active object's `RunL()` function, since it prevents all other active objects in your thread from running. For example, if you called `User::After()` in one of your active objects, no other active object would be able to run until that one had completed.

8.2.2 Creating an Active Object Class

To create an active object, declare a class and derive it from `CActive`, as in the following example:

```
class CMyActive : public CActive
  {
  public:
  static CMyActive* NewL();
```

```
  CMyActive();
  ~CMyActive();

  void ConstructL();

  void InvokeAsyncFunc();  // Some method to invoke the active object's
                           // associated asynchronous function.

// Overriden from CActive

   virtual void RunL(); // handles asynchronous function completion
   virtual void DoCancel();  // Cancels an outstanding
                             // asynchronous function call
   virtual TInt RunError(TInt err); // overriden if desired
                                    // to handle a leave that
                                    // occurs in RunL()
  }
```

Then, implement your active object class using the following steps:

1. Implement functions to construct your active object (i.e. the class constructor, `static NewL()`, and `ConstructL()`).
2. Register the active object with your thread's active scheduler (usually done as part of Step 1).
3. Implement one or more requestor functions, each of which invokes an asynchronous function.
4. Override `CActive`'s `RunL()` to handle each asynchronous function's completion event.
5. Override `CActive`'s `DoCancel()` function to cancel an outstanding asynchronous function call.
6. Optionally override `CActive`'s `RunError()` method to handle errors that occur in your `RunL()` event handler.
7. Create your active object's destructor, which should include calling the `CActive Cancel()` method.

The next sections look at these steps in more detail.

Constructing an Active Object

In the constructor for your active object, you must call the base constructor `CActive::CActive(TInt aPriority)`, passing it the active object's priority value (see Section 8.5). This is normally done in the initialization list, as the following example shows:

```
CMyActive::CMyActive()
    : CActive(CActive::EPriorityStandard)
  {

  }
```

Normally you also create a static `NewL()` function for your active object and a `ConstructL()` second stage constructor function.

Adding an Active Object to the Active Scheduler

In order for the active scheduler to know about your active object, as a potential source of events from the thread's request semaphore and to distribute events to it, your active object needs to be added to the thread's active scheduler. This is done via a call to the static function `CActiveScheduler::Add(CActive *ao)`, where `ao` is a pointer to your active object. In many cases this is done in `ConstructL()` as follows:

```
CMyActive::ConstructL()
  {
  /*  ...  */
  CActiveScheduler::Add(this);
  }
```

Alternatively `CActiveScheduler::Add()` could be done in the active object constructor (since it does not leave), in a static `NewL()` class for your active object, or anywhere where you can pass a pointer to your active object. In general though, it should be in your class so that the user of the active object does not need to be concerned with adding it to the active scheduler.

Implementing Requestor Functions

You need to implement at least one method in your active object that starts an asynchronous function. When you call the asynchronous function, you pass the active object's `TRequestStatus` member, `iStatus`, to the asynchronous function. After that, you call `CActive`'s `SetActive()` method to indicate that the active object has an asynchronous function call outstanding.

As an example, if the asynchronous call associated with the active object is `AnAsyncCall(TAny *Arg1, TRequestStatus&aStatus)`, then your active object's invocation method could look as follows:

```
void CMyCActive::InvokeAsyncFunc(TAny *aArg)
  {
  /*  ...  */
  AnAsyncCall(aArg,iStatus);
  SetActive();
  }
```

Implementing `RunL()`

You must override `CActive`'s `RunL()` method to handle the completion of the asynchronous function. The `iStatus` member (which was

passed to the asynchronous function call) can be checked in RunL() to determine the function's status. The active object can then, if desired, issue another asynchronous function request in the RunL() (from the current, or another, active object).

The following is an example RunL() method:

```
void CMyActive::RunL()
  {
  if (iStatus == KErrNone)
    {
    // add code to handle the event
    }
  else
    {
    // add error handling code
    }
  }
```

Implementing a DoCancel() Function

DoCancel() is a pure virtual method of CActive and you must implement this in your derived class to cancel your asynchronous function. DoCancel() is never called directly, but is called through the CActive::Cancel() method. Cancel() only calls DoCancel() if your active object is waiting for its asynchronous function to complete (it calls CActive::IsActive() to check if this is the case) – so don't worry about your DoCancel() canceling a function that is not in progress.

Overriding the RunError() Method

You can optionally override RunError(TInt aErr) to handle leaves that occur when the active scheduler invokes your RunL() function – aErr is the leave code. You should return KErrNone once the error is handled to prevent the active scheduler from attempting to handle it. The default implementation of RunError() returns the leave code, passing the responsibility for handling the error to the active scheduler (see Section 8.4).

8.3 The Active Scheduler

As described in Section 8.2, the active scheduler is the event handler for the thread. It receives an event at the request semaphore, determines which active object it belongs to, and invokes that active object's RunL() function. To use active objects in a thread there must be an active scheduler – a class of type CActiveScheduler – installed in your thread.

Often the thread you want to use active objects in will already have an active scheduler installed, and, in that case, you do not need to install your own. GUI applications already have an active scheduler installed and running, because the GUI framework itself uses active objects to process GUI events. You can add your own active objects and they will become part of the existing pool of active objects.

8.3.1 Installing and Starting an Active Scheduler

An active scheduler will need to be created, installed and started for threads that you create yourself, if you want to use active objects in them. This applies to threads created directly by `RThread` or as a result of creating your own `exe` executable which has no active scheduler installed by default.

Setting up an active scheduler is straightforward, as shown in the following example:

```
CActiveScheduler *mySched=new (ELeave) CActiveScheduler;
// install it
CActiveScheduler::Install(mySched);
// Add at least one active object here and invoke a request
CActiveScheduler::Start();
```

First you create the scheduler object itself, `CActiveScheduler`. Then you call the static `CActiveScheduler::Install()` function to install your scheduler as the active scheduler of the thread – making it responsible for waiting on, and distributing events from, the thread's request semaphore. If the thread already has an active scheduler installed, then you will get an exception (`E32USER-CBase 43`).

Although your active scheduler is now installed, it does not begin processing events until you call `CActiveScheduler::Start()`. Once you call `CActiveScheduler::Start()`, the scheduler is in its event loop – waiting at the request semaphore, invoking the appropriate active object's `RunL()` when an event is received, and then waiting for the next event.

So `CActiveScheduler::Start()` will, in effect, block your thread at the point it was called, and all code execution will now occur inside the `RunL()` functions of the active objects, since everything is now executing in response to events. `CActiveScheduler::Start()` returns only after you stop the scheduler using `CActiveScheduler::Stop()` (which will have to be called in an active object's `RunL()` method). Once the scheduler's stop method is called, the event loop is exited, and code execution will resume immediately after the `CActiveScheduler::Start()` function (usually to clean up and exit the thread).

Before you start the scheduler, thus putting it in the event handling loop, you need to have at least one active object added to the scheduler,

and have an outstanding asynchronous function active. If you don't, when you call `CActiveScheduler::Start()`, you will be stuck forever, waiting for an event that will never occur!

8.3.2 Background Information

There will be times when you'll find it useful to understand the details of how the active scheduler works, in order to really understand how your program behaves when using active objects. Let's look at the event loop implemented in `CActiveScheduler::Start()` in more detail.

The first thing `CActiveScheduler::Start()` does is to block at the thread's request semaphore. When a signal is received from the request semaphore, it determines which active object the signal belongs to by looking for an active object that has both its `iActive` variable (the `TBool` member of `CActive` that is set when you call `SetActive()`) set to `ETrue` and its `iStatus` set to some value other than `KRequestPending`.

Once an active object that meets this condition is found, the active scheduler invokes that active object's `RunL()` method, after which it goes back to waiting for the next signal at the request semaphore. This event loop repeats until the `ActiveScheduler::Stop()` function is invoked (in some active object's `RunL()` function) at which time the event loop is exited and `ActiveScheduler::Start()` returns.

Pseudo-code for the active scheduler event loop (`CActiveScheduler::Start()`) is shown in Example 8.1.

Example 8.1. Pseudo-code for Active Scheduler Event Loop

```
DO
{
WaitForAnyRequest();   // block at the thread's request semaphore,
                       // return when one is received
// signal received
FOR (I=0;I<NUMBER_OF_ACTIVE_OBJECTS;I++)
  {
  IF ( ACTIVE_OBJECT[I].iActive &&
         (ACTIVE_OBJECT[I].iStatus != KRequestPending) )
    {
    ACTIVE_OBJECT[I].RunL();   // invoke the target
                               // active object's RunL()
    break;
    }
  }
  IF (I==NUMBER_OF_ACTIVE_OBJECTS) // If it did not find an active
                                   // object signal belonged to
    {
    GENERATE_STRAY_SIGNAL_EXCEPTION();
    }

} WHILE (ActiveScheduler::Stop() not called)
```

NUMBER_OF_ACTIVE_OBJECTS represents the number of active objects added to the active scheduler, and ACTIVE_OBJECT[] represents the list of those active objects.

Remember (see Section 8.1) that when an asynchronous function starts, it sets the TRequestStatus argument (in the case of an active object, the iStatus member variable) to KRequestPending. When the asynchronous function completes, it writes the completion status into the TRequestStatus value and then signals the calling thread's request semaphore.

As you can see from the pseudo-code, when the active scheduler gets a semaphore event, it scans through its list of active objects, and invokes the RunL() method if an active object's iActive is set, and iStatus is a value other than KRequestPending. If the active scheduler does not find an active object that meets these conditions then a stray signal exception occurs for the thread.

The pseudo-code shown in Example 8.1 is simplified – for example, the event loop does not show the priority handling that takes place when multiple active object asynchronous events occur (instead, it just runs the first active object that is found to be ready to run), nor does it show RunL() error handling – but it should give you a basic idea of how the event loop works.

8.3.3 CActiveScheduler Methods

Many of the methods of CActiveScheduler are static, and operate on the currently installed active scheduler for the thread (or are used to install the scheduler itself).

The static methods of CActiveScheduler are:

- void Add(CActive* aActiveObject) adds aActiveObject to the currently installed scheduler, to register for receiving events. An active object usually adds itself to the active scheduler during its construction (by invoking it as CActiveScheduler::Add(this)).
- void Install(CActiveScheduler* aActiveScheduler) installs the specified CActiveScheduler object as the current thread's active scheduler. An exception is generated if one is already installed.
- void Replace(CActiveScheduler* aActiveScheduler) is similar to Install(), except that if an active scheduler is already installed, then the specified active scheduler object will be installed in place of the currently installed one (as opposed to generating an exception, as Install() would).
- void Start() contains the active scheduler's event loop. Once you call Start() the active scheduler will continually process events

from the thread's request semaphore and invoke the appropriate active object's `RunL()` method in response to them. `Start()` will return when an active object calls the active scheduler `Stop()` method. Make sure you have an event that will occur (i.e. at least one active object added to it, and with an event pending) or `Start()` will hang indefinitely. (See Section 8.3.1.)

- `void Stop()` causes the current active scheduler's event loop to exit.
- `CActiveScheduler* Current()` returns a pointer to the thread's currently installed active scheduler.

8.3.4 Customizing the Active Scheduler

`CActiveScheduler` is a concrete class and is normally created and used directly, without derivation. However, in some cases you may want to derive your own active scheduler so that you can customize the event loop or its start and stop functionality, and provide customized error handling for the scheduler.

The following virtual methods are used when deriving your own active scheduler from `CActiveScheduler`:

- `virtual void OnStarting()` is called from the `Start()` method of the `CActiveScheduler` base class before the event loop is started. The default implementation of `OnStarting()` does nothing, but your derived class can implement customized code that you want to execute before starting the event loop.
- `virtual void OnStopping()` is called from the `Stop()` method of the `CActiveScheduler` base class. The default implementation of `OnStopping()` does nothing, but your derived class can implement customized code you want executed upon stopping the active object's event loop.
- `virtual void Error(TInt aErr)` is invoked by the active scheduler when a leave occurs within an active object's `RunL()` function, and the active object itself did not handle the error in its own `RunError()` method. The argument to `Error()` contains the leave code. Your derived scheduler object can override this function to handle leaves that occur in a `RunL()` and are not handled by the active object itself (see Section 8.4). The default implementation of `Error()` is to invoke an `E32USER-CBase 47` exception.
- `virtual void WaitForAnyRequest()` is called in the active scheduler's event loop (i.e. it is initiated from the `Start()` method) and is used when waiting for an asynchronous function to complete. See the pseudo-code in Example 8.1. The default implementation of this function is to call `User::WaitForRequest()`,

which blocks and waits at the current thread's request semaphore. A derived `CActiveScheduler` can override this by implementing its own `WaitForAnyRequest()`. Normally this still involves calling `User::WaitForRequest()`, but customized pre- or post-processing of the event can be implemented here. Of course, you could also handle events via a method other than the thread's request semaphore (such as from a communication port, or as a network message). However, this would also require the attached active objects to use a set of customized asynchronous functions which were compatible with the customized event-handling method.

8.4 Active Scheduler Error Handling

A leave can occur in your active object's `RunL()` method (as indicated by the `L` suffix) to indicate an error. By default, a leave in `RunL()` will generate an exception, but you can change this behavior for your active object by overriding `CActive::RunError()`. Additionally, you can override the default error handling of all active objects belonging to the active scheduler by customizing the active scheduler `Error()` method – however, this is not often done.

`CActive::RunError()` is prototyped as:

```
TInt RunError(TInt aErr)
```

where `aErr` contains the leave code. The default base class implementation of `RunError()` returns the leave code passed to it.

A customized `RunError()` should return `KErrNone` to indicate it has handled the error. If `RunError()` returns a value other than `KErrNone` the active scheduler assumes that the error has not been handled, and invokes its own error-handling method, `Error()`.

`Error()` is an customizable method of `CActiveScheduler`, prototyped as:

```
void Error(Tint aErr)
```

where `aErr` is the value returned from the active object's `RunError()` method.

The following pseudo-code illustrates how the active scheduler handles `RunL()` leaves:

```
// invoke RunL() for target active object and handle error
TRAPD(LeaveCode,target_active_object->RunL());
if (LeaveCode != KErrNone)
  {
```

```
TInt rc = target_active_object->RunError(LeaveCode);
if (rc !=KErrNone)
  {
  Error(rc);    // active object did not handle the
                // error, so call the active
                // scheduler's own Error() function
  }
}
```

The default implementation of `CActiveScheduler::Error()` is to generate an `E32USER-CBase 47` exception. Since the default implementation of `CActive::RunError()` is to return the `RunL()` leave code, then `E32USER-CBase 47` is what you will get on a leave if you do not override any of the error-handling functions.

8.5 Active Object Priorities

When an active object is constructed, a priority value is passed to `CActive`'s constructor. The possible priority values (lowest to highest) are: `EPriorityIdle`, `EPriorityLow`, `EPriorityStandard`, `EPriorityUserInput`, `EPriorityHigh`. The priority value is normally specified in your derived active object's constructor as follows:

```
CDerivedActiveObject::CDerivedActiveObject()
: CActive(CActive::EPriorityStandard)
  {
  }
```

If multiple active objects have outstanding asynchronous functions in progress, and two or more of these functions complete at the same time, then the scheduler will see multiple active objects with `iActive` set to `ETrue` and `iStatus` not equal to `KRequestPending` upon the next semaphore event. In this case, the `RunL()` for the higher-priority active object is invoked. The scheduler then checks the request semaphore, sees the next event and executes the `RunL()` of the next-highest priority active object that is still active, and so on.

As an example, if three asynchronous functions produce events at the same time – one high, one medium, and one low priority – then three tokens are added to the calling thread's request semaphore. The scheduler processes these three events, one at a time, invoking first the high-priority active object's `RunL()`, then the medium-priority one, and finally the low-priority one. It does not really matter which semaphore signal (token) originally corresponded to which active object – they are all handled.

8.6 Canceling Outstanding Requests

Each asynchronous function has a cancel function associated with it. This cancel function causes its corresponding asynchronous function to

abort its operation and complete right away, with its `TRequestStatus` variable set to `KErrCancel`.

For example, the API class `RTimer` provides the asynchronous function method `RTimer::After(TRequestStatus&aStatus, TTimeIntervalMicroseconds32 aWait)`. After the specified time has elapsed, this function will change `aStatus` to `KErrNone` and send the completion event (`aStatus` will be `KRequestPending` during the time interval). `RTimer::Cancel()` cancels the timer. Calling `RTimer::Cancel()` while the time interval is in progress, causes `RTimer::After()` to complete right away, with its `TRequestStatus` variable set to `KErrCancel`.

As mentioned previously (see Section 8.2.7), your derived active object class must override `DoCancel()` to cancel any pending asynchronous requests, and `DoCancel()` is never called directly, but is called through `CActive`'s `Cancel()` function. `Cancel()` checks if the active object has an outstanding request, and if it does, it will invoke your overridden `DoCancel()` method. After `DoCancel()` exits, `Cancel()` will do a `User::WaitForRequest()` to consume the `KErrCancel` message. Therefore, do not wait for the message in `DoCancel()` itself (or expect `RunL()` to be invoked in response to it).

`Cancel()` should always be called in your active object's destructor. If it is not, and you delete your object with an asynchronous request in progress, you will get a stray signal panic.

A common error, when canceling an asynchronous function, is to call `User::WaitRequestForRequest()` from the `DoCancel()`, in order to consume the `KErrCancel` event directly. If you do this, you will see `Cancel()` (and thus your destructor) hang, since `CActive::Cancel()` does its own `User::WaitForRequest()` to consume the cancel event.

Figure 8.2 shows how `Cancel()` and `DoCancel()` work.

For example, if your active object uses the `RTimer` functions for its asynchronous events, the `DoCancel()` could look as follows:

```
void CDerivedActiveObject::DoCancel()
  {

  // Cancel the timer
  iTimer.Cancel();
  }
```

8.7 Removing an Active Object

The base class destructor removes the active object from the active scheduler's list. However, the base destructor requires that no asynchronous event should still be pending – if there is, it generates an `E32USER-CBase 40` exception. Therefore the destructor in your derived

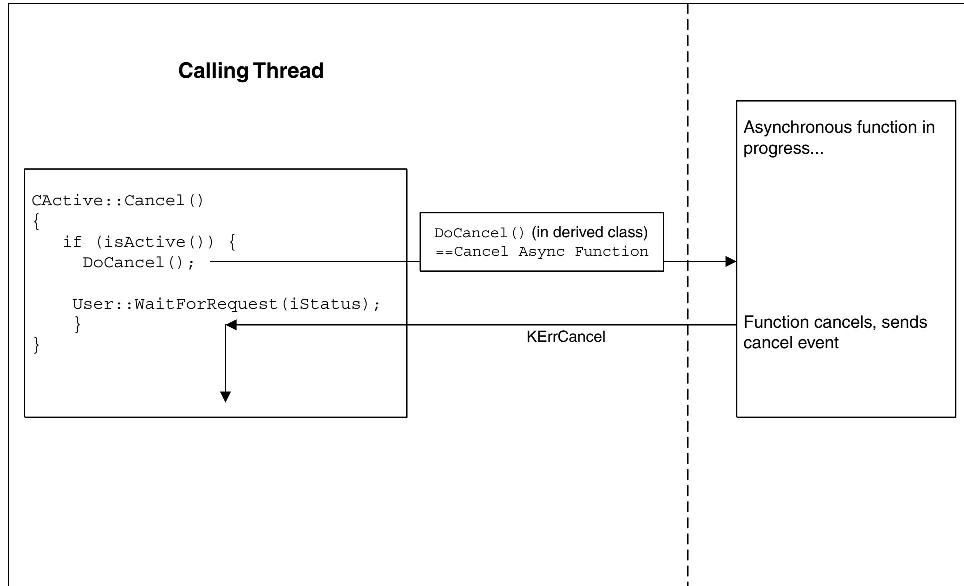

Figure 8.2 `Cancel()` and `DoCancel()` Operation

class should call the `Cancel()` method (as well as any other cleaning up it needs to do).

If you want to remove an active object from the active scheduler, but not destroy it, call `CActive::Deque()`. This will call the active object's `Cancel()` method to cancel any outstanding requests, and then remove the active object from the active scheduler.

8.8 Active Object Example

To illustrate active objects, I'll present an example based on the `SimpleEx` program presented in Section 2.3.2. The original `SimpleEx` displays 'Simple Example' in the middle of the screen. When you select a menu item called 'Start' it displays an alert dialog indicating that Start was selected.

The example will be expanded here to include an active object. The Start menu item is changed so that, when selected, it starts a countdown of 10 seconds – displaying the progress of the countdown on the screen – at the end of which it pops up the alert dialog. Also, a Stop item is added to the menu and, when Stop is selected, it will stop a countdown, if one is in progress. (Since active objects are used, the GUI stays responsive during the countdown.) The next time Start is selected, it starts the countdown from the beginning.

8.8.1 CCountdown Active Object

Let's begin by showing the active object that was added to SimpleEx to implement the countdown. Example 8.2 shows the declaration of our active object class, called CCountdown.

Example 8.2. CCountdown Active Object Class

```
class CCountdown : public CActive
  {
public:
  static CCountdown* NewL(CSimpleExAppView* aAppView);
  CCountdown();
  ~CCountdown();
  void ConstructL(CSimpleExAppView* aAppView);

  void StartCountdown();

  void Stop();
  void RunL();

  void DoCancel();

private:
  TInt iCount;
  RTimer iTimer;
  TInt iInterval;
  CSimpleExAppView* iAppView;
  };
```

Example 8.3 shows the implementation of the CCountdown active object.

Example 8.3. CCountdown Implementation

```
CCountdown* CCountdown::NewL(CSimpleExAppView *aAppView)
  {
  CCountdown* self = new(ELeave) CCountdown;
  CleanupStack::PushL(self);
  self->ConstructL(aAppView);
  CleanupStack::Pop(self);
  return self;
  }

CCountdown::CCountdown()
    : CActive(CActive::EPriorityUserInput)
       // Construct high-priority active object
  {
  }

void CCountdown::ConstructL(CSimpleExAppView* aAppView)
  {
  iCount=0;
  iAppView = aAppView;
  User::LeaveIfError(iTimer.CreateLocal());
```

```
  iInterval = 1000000;  // 1 second interval
  // Add to active scheduler
  CActiveScheduler::Add(this);
  }

void CCountdown::StartCountdown()
  {
// This method is invoked when user selects the start menu item to start
// the countdown.
  if (iCount == 0)
    {
    iCount=10;
    iTimer.After(iStatus,iInterval);
    SetActive();
    }
  }

CCountdown::~CCountdown()
  {
  // Make sure we're cancelled
  Cancel();
  iTimer.Close();
  }

void CCountdown::RunL()
  {
  TBuf<50> buff;
  buff.Format(_L("-%d-"),iCount);
  iAppView->UpdateScreenText(buff);

  if (iCount)
    {
    iTimer.After(iStatus,iInterval);
    SetActive();
    iCount--;
    } else
    {
    iAppView->UpdateScreenText(KSimpleExText);
    CEikonEnv::Static()->AlertWin(_L("Start Selected!"));
    }
  }

void CCountdown::Stop()
  {
  iCount=0;
  iAppView->UpdateScreenText(KSimpleExText);
  Cancel();
  }

void CCountdown::DoCancel()
  {
  iTimer.Cancel();
  }
```

Let's step through the methods. `CCountdown::NewL(CSimpleExAppView *aAppView)` is the static `NewL()` method that constructs the active object and calls the secondary constructor for it. A pointer to the `CSimpleExAppView` is passed to the active object so that it

can draw the countdown text to the application screen area, via the screen update method UpdateScreenText() that was added to the application view class.

The CCountdown constructor does nothing except set the active object's priority to EPriorityStandard (which is suitable for most active objects you will write). The secondary constructor, ConstructL() sets up a countdown interval of one second, and initializes the timer via the RTimer class. ConstructL() then adds the active object to the active scheduler. Note that, since this is a GUI application, the active scheduler is already installed.

CCountdown::StartCountdown() starts the actual countdown process, and is called when the user selects the Start menu item. Start-Countdown() calls RTimer::After() with the one-second interval, passing it the active object's TRequestStatus iStatus member variable. It then calls SetActive() so that the active scheduler knows that the object is waiting for an asynchronous event. Note that if (iCount==0) is there to prevent the countdown from being started when it is already in progress.

When the one-second timer has expired, the timer function sets the iStatus variable to KErrNone and signals the calling thread's request semaphore. This causes the active scheduler to invoke the CCount-down's RunL() method. CCountdown::RunL() calls a new method, implemented in the application view class, called UpdateScreen-Text() to display the current count on the screen (see Section 8.8.2). If zero has not yet been reached, the count is decremented by one and RTimer::After() is reissued along with the SetActive(). This will cause RunL() to be reentered after one second. If the count has reached zero, the timer function is not reissued, and the Alert Box that reads 'Start Selected' is displayed.

Note that RunL() does not check the value of iStatus. This is because the RTimer::After() function is very simple and no error conditions exist. However, for most other asynchronous functions, you'll want to check iStatus for the status of the asynchronous call.

The destructor for CCountdown simply calls Cancel(), which in turn calls DoCancel() if an outstanding request exists, which could happen if you exit the application while the count is in progress.

When modifying SimpleEx for this example, I put the active object class declaration directly in SimpleEx.h and its source in SimpleEx_ui.cpp, but you can create separate files for these.

8.8.2 Modifications to SimpleEx

Now, I'll show the modifications to the existing SimpleEx GUI classes. To simplify, I present the Series 60 program only. However, active objects are platform independent, so the same modifications can be made to UIQ and Series 80 programs.

In the `SimpleEx` header file, I modified the declarations of `CSimpleExAppUi` and `CSimpleExAppView`.

In `CSimpleExAppUi`, I added a pointer to the active object as follows:

```
class CSimpleExAppUi : public CAknAppUi
  {
public:
  void ConstructL();
  ~CSimpleExAppUi();

private:
  void HandleCommandL(TInt aCommand);

public:

private:
  CSimpleExAppView* iAppView;
  CCountdown *iCountdown;

  };
```

In `CSimpleExAppView`, I added the method `UpdateScreenText()` which is used by the active object to write its countdown to the screen. This method takes a descriptor string as an argument, and writes that string to the center of the screen.

Here is the modified application view class declaration:

```
// The Application View Class

class CSimpleExAppView : public CCoeControl
  {
public:
  static CSimpleExAppView* NewL(const TRect& aRect);
  static CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect);
  void ConstructL(const TRect& aRect);
  void UpdateScreenText(const TDesC16& aText);

private:
  TBuf<100> iScreenText;
  void Draw(const TRect&) const;
  };
```

The rest of the `SimpleEx` classes remain the same.

Example 8.4 shows the modified resource file, which adds the Stop item to the menu.

Example 8.4. `SimpleEx` Modified Resource File

```
RESOURCE TBUF r_default_document_name
  {
  buf="";
  }
```

```
RESOURCE EIK_APP_INFO
  {
  menubar = r_SimpleEx_menubar;
  cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
  }

RESOURCE MENU_BAR r_SimpleEx_menubar
  {
  titles =
    {
    MENU_TITLE
      {
      menu_pane = r_SimpleEx_menu;
      }
    };
  }

RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      },

      MENU_ITEM
        {
        command = ESimpleExStop;
        txt = "Stop";
        }
    };
  }
```

The `ESimpleExStop` command is added to the command `enum` in `SimpleEx.hrh` as follows:

```
#ifndef __SimpleEx_HRH__
#define __SimpleEx_HRH__

// SimpleEx enumerate command codes
enum
  {
  ESimpleExCommand = 1,  // start value must not be 0
  ESimpleExStop
  };

#endif // __SimpleEx_HRH__
```

Two existing `SimpleEx` source files are changed – `simpleEx_view.cpp` and `simpleEx_ui.cpp`. Example 8.5 shows the modified `CSimpleExAppView` in `simpleEx_view.cpp`.

Example 8.5. Modified `simpleEx_view.cpp`

```
#include "eikenv.h"
#include <coemain.h>

#include "SimpleEx.h"

CSimpleExAppView* CSimpleExAppView::NewL(const TRect& aRect)
  {
  CSimpleExAppView* self = CSimpleExAppView::NewLC(aRect);
  CleanupStack::Pop(self);
  return self;
  }

CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect)
  {
  CSimpleExAppView* self = new (ELeave) CSimpleExAppView;
  CleanupStack::PushL(self);
  self->ConstructL(aRect);
  return self;
  }

void CSimpleExAppView::ConstructL(const TRect& aRect)
  {

  CreateWindowL();
  SetRect(aRect);
  ActivateL();
  }

void CSimpleExAppView::UpdateScreenText(const TDesC16& msg)
  {
  iScreenText.Copy(msg);
  DrawNow();
  }

void CSimpleExAppView::Draw(const TRect& ) const
  {
  CWindowGc& gc = SystemGc();
  const CFont*   font;
  TRect drawRect = Rect();

  gc.Clear();

  font = iEikonEnv->TitleFont();
  gc.UseFont(font);
  TInt   baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;
  gc.DrawText(iScreenText,drawRect,baselineOffset,
      CGraphicsContext::ECenter, 0);

  gc.DiscardFont();
  }
```

The changes to `CSimpleExAppView` were to add the `Update-ScreenText()` method as well as modify `Draw()` to support it. `UpdateScreenText()` simply writes the text passed to it to the `iScreenText` descriptor and forces a screen draw. `Draw()` will call

`gc.DrawText()` to write whatever is in this descriptor to the center of the screen.

Example 8.6 shows the modified `CSimpleExAppUi` (in `simpleEx_ui.cpp`).

Example 8.6. Modified `CSimpleExAppUi`

```
void CSimpleExAppUi::ConstructL()
  {
  BaseConstructL();

  iAppView = CSimpleExAppView::NewL(ClientRect());
  iCountdown = CCountdown::NewL(iAppView);
  iAppView->UpdateScreenText(KSimpleExText);
  }

CSimpleExAppUi::~CSimpleExAppUi()
  {
  delete iAppView;
  iAppView = NULL;
  delete iCountdown;
  }
void CSimpleExAppUi::HandleCommandL(TInt aCommand)
  {
  switch(aCommand)
    {
    case EEikCmdExit:
    case EAknSoftkeyExit:
      Exit();
      break;

    case ESimpleExCommand:
      iCountdown->StartCountdown();
      break;

    case ESimpleExStop:
      iCountdown->Stop();
      break;
    }
  }
```

The main change here was to invoke the active object's methods to start and stop the countdown in response to the Start and Stop menu item commands. Also in the header file, I defined `KSimpleExText` as:

```
_LIT(KSimpleExText,"Simple Example AO");
```

`ConstructL()` sets this text to the screen (and the active object restores it when the countdown is terminated).

8.9 Active Object Issues

Using active objects is the best way to handle events from multiple asynchronous functions and to implement multiple (though non-preemptive) threads in Symbian OS. However, there are issues that can occur, and

many programmers struggle with active objects at first. Having a good understanding of how they work helps avoid problems.

8.9.1 Do Not Block in an Active Object

Once the active scheduler is started, the only safe blocking that can occur in the thread is when the active scheduler event loop waits at the thread's request semaphore for an event. However, if you do an operation that blocks inside an event handler (i.e. in your `RunL()` function), such as waiting for a semaphore, or calling a lengthy blocking function (such as `User::After()`) then not only do you block your `RunL()`, but you also block the entire thread, including the active scheduler and all the rest of the thread's active objects. That's not to say that you cannot block at all – sometimes a very small thread-blocking operation is needed – but you have to keep in mind that, during that block, no other asynchronous function events are being handled.

This is why you should not block in GUI programs. The GUI event handler, `HandleCommandL()`, is actually invoked from a `RunL()` of an active object the application UI class inherits from. So if you delay for some reason when handling a GUI event, your GUI will become nonresponsive during this delay since the GUI program's active scheduler cannot process further events.

It is possible to block in an active object, yet still have events processed using nested `CActiveScheduler::Start()` and `Stop()` calls. Nesting active scheduler calls can be very complex to implement correctly though, and its use is beyond the scope of this book.

8.9.2 Avoid Stray-Signal Exceptions

Any Symbian programmer who has developed active objects will probably have encountered the exception: `E32USER-CBase 46`. This is the stray-signal exception and it is invoked by the active scheduler's event loop when it receives a signal at the thread's request semaphore, but cannot find an active object that it belongs to (i.e. no active objects with `iActive==ETrue` AND `iStatus!=KRequestPending`). There are many situations that can cause this. Here are just a few:

- Not adding your active object to the scheduler (via `CActiveScheduler::Add()`)

  If you invoke the asynchronous function in your active object, yet it is not added to the scheduler, then a stray-signal exception will occur when the function completes, since the scheduler does not know about your active object.

- Not calling `SetActive()`

  If you do not call `SetActive()` in your active object when invoking the asynchronous function, then, when the signal comes in, the scheduler does not see your active object's `iActive` flag set, and therefore does

not consider it a target for the event. It finds that the event belongs to no active object and generates the stray-signal exception.

- Not calling `User::WaitForRequest()` on an asynchronous function not associated with an active object

  In some cases, you may want to use `User::WaitForRequest()` to wait for an asynchronous function that is not associated with an active object to complete (although remember that this also blocks your active object's event loop). If you call such an asynchronous function, but forget to consume its event via `User::WaitForRequest()`, then the active scheduler will eventually get the event and, since it will not find the active object the event belongs to, the scheduler will generate the stray-signal.

A common cause of a stray-signal exception is when you cancel an asynchronous function that is not associated with an active object. Remember that a cancel also generates an event (it's a common error that this is not considered) and a stray-signal results.

8.9.3 Have One Outstanding Event at a Time

On the surface it may seem that you can call as many asynchronous functions as you like in an active object (calling `SetActive()` after each) and assume that your `RunL()` is invoked as each asynchronous function completes. However, this is not true. An active object can only have one outstanding request at a time, and if you try to have more, a stray signal will result.

When you think about how the scheduler works, you will see why this is so. The only thing that `SetActive()` does is to set your active object's `iActive` flag to `ETrue`. So when you invoke the first asynchronous function and call `SetActive()`, you have `iActive==ETrue` and `iStatus==KRequestPending`. If you then invoke a second asynchronous function from your active object, and call `SetActive()`, `iActive` is already set from the previous request (it just sets it again). The first asynchronous function that completes generates a signal at the request semaphore and the active scheduler calls the active object's `RunL()` function. However, after that `RunL()` completes, the `iActive` flag is cleared by `CActive` – but there is still an outstanding function call since you invoked multiple asynchronous function calls. So when the second asynchronous function completes, the active scheduler sees your `iActive` flag set to zero, and, finding no other active object for it to belong to, generates the stray-signal exception.

8.10 Using Active Objects as Threads

Thus far we have discussed active objects as being simply a way to start asynchronous functions and handle their completion, but you can also

use them to perform background processing as you can with threads. This example illustrates that concept. It also illustrates other concepts mentioned earlier, such as creating, starting and stopping the active scheduler.

The example creates two active objects whose `RunL()` functions are invoked at specified intervals. This example uses the console and is built as an `exe` executable that can be run on the emulator (see Section 6.1 for how to create a console program).

8.10.1 CTimer

Instead of deriving our active objects straight from `CActive`, this example derives them from an API class called `CTimer` which is itself derived from `CActive`. `CTimer` uses `RTimer` to generate events after a user-defined period of time. Using `CTimer` is more convenient than implementing `RTimer` within our own active object.

`CTimer` implements the following method, which I will use in the example:

```
void After(TTineIntervalMicroSeconds32, aInterval)
```

`CTimer::After()` simply invokes its associated `RTimer`'s `After()` method and calls `CActive::SetActive()`. `RunL()` (implemented in `CTimer`'s derived class) is called when the timer expires.

Example 8.7 shows the header file.

Example 8.7. Example Header for `CTimer`

```
#ifndef _ACTIVEH
#define _ACTIVEH

#include <e32base.h>
#include <e32cons.h>

CConsoleBase* console;

class CPrimaryTask  : public CTimer
  {
public:
  static CPrimaryTask* NewLC();
  void StartRunning(TInt aRepeat, TInt aInterval);

  void RunL();
  CPrimaryTask() : CTimer(EPriorityStandard){ }

private:
  TInt iInterval;
  TInt iRepeatCnt;
  };
```

```
class CBackground : public CTimer
  {
public:
  static CBackground* NewLC();
  void StartBackground(TInt aInterval);

  void RunL();
  CBackground() : CTimer(EPriorityStandard){ }

private:
  TInt iInterval;
  };

_LIT(KActiveExName,"Active Object Example");

void activeExampleL();

#endif
```

There are two active objects in this example – a primary task and a background task. Example 8.8 shows their implementation, and the main program.

Example 8.8. Active Object Example Implementation

```
#include <e32base.h>
#include <e32cons.h>
#include "active.h"
#include "f32file.h"
#include "flogger.h"

RFileLogger iLog;

/*-------------------------------------------------------
    Active Object: CPrimaryTask
--------------------------------------------------------*/

CPrimaryTask* CPrimaryTask::NewLC()
  {
  CPrimaryTask* self=new (ELeave) CPrimaryTask;
  CleanupStack::PushL(self);
  self->ConstructL();
  CActiveScheduler::Add(self);
  return self;
  }

void CPrimaryTask::StartRunning(TInt aRepeat, TInt aInterval)
  {
  iInterval=aInterval;
  iRepeatCnt=aRepeat;
  CTimer::After(iInterval);
  }

void CPrimaryTask::RunL()
  {
  if (iRepeatCnt--)
```

```
    {
    LIT(KFirstActiveWaiting,"CPrimaryTask: RunL");
    console->Printf(_L("%S\n"),&KFirstActiveWaiting);
    iLog.Write(KFirstActiveWaiting);

    // do task processing...here
    // ...

    After(iInterval);  // reissue request
    } else
    CActiveScheduler::Stop();   // repeat count expired, end event
                                // loop to exit program
  }

/*-------------------------------------------------------
    Active Object: CBackground
--------------------------------------------------------*/

void CBackground::StartBackground(TInt aInterval)
  {
  iInterval=aInterval;
  CTimer::After(iInterval);
  }

CBackground* CBackground::NewLC()
  {
  CBackground* self=new (ELeave) CBackground;
  CleanupStack::PushL(self);
  self->ConstructL();
  CActiveScheduler::Add(self);
  return self;
  }

void CBackground::RunL()
  {
  _LIT(KBackgroundTaskMsg,"CBackground RunL");
  console->Printf(_L("%S\n"),&KBackgroundTaskMsg);
  iLog.Write(KBackgroundTaskMsg);

  // do task processing...here
  // ...

  After(iInterval);
  }

/*-------------------------------------------------------
    E32Main() Entry point
--------------------------------------------------------*/

TInt E32Main()
  {
  CTrapCleanup* cleanStack=CTrapCleanup::New(); // create a clean-up stack
  TRAPD(leaveCode,activeExampleL()); // more initialization, then do
                                     // example
  if ( leaveCode != KErrNone)
    User::Panic(KActiveExName,leaveCode);
  delete cleanStack; // cleanup cleanup stack
```

```
  return 0; // and return
  }

/*-----------------------------------------------
    Active Object Example main function
-----------------------------------------------*/

void activeExampleL()
  {
 /*------------------------------------------------------------
     Create a full screen console
 ------------------------------------------------------------*/

  console=Console::NewL(_L("Console"),TSize(KConsFullScreen,
  KConsFullScreen));
  CleanupStack::PushL(console);

  console->Printf(_L("Active Object Example"));
  console->Printf(_L("Press Key to Begin"));
  console->Getch();

/*----------------------------------------------------------
    Initialize file logging, need to create dir on phones
    c:\Logs\ActiveLogging for log to be created
------------------------------------------------------------*/

  iLog.Connect();
  iLog.CreateLog(_L("ActiveLogging"),_L("ActiveLog.txt"),
          EFileLoggingModeOverwrite);

/*-------------------------------------------------
    Create and install Active Scheduler
-------------------------------------------------*/

  CActiveScheduler*  mySched = new (ELeave) CActiveScheduler;
  CleanupStack::PushL(mySched);

  CActiveScheduler::Install(mySched);

/*-----------------------------------------------
    Start active objects
------------------------------------------------*/

  CPrimaryTask *firstAct = CPrimaryTask::NewLC();

  CBackground *backAct = CBackground::NewLC();

  backAct->StartBackground(4000000);  // Run every 4 seconds

  firstAct->StartRunning(20,1000000);  // Run for 20 times, 1 sec interval

/*-----------------------------------------------
    Start the active scheduler event loop.
-----------------------------------------------*/

  CActiveScheduler::Start();
```

```
/*-----------------------------------------------
    Event loop exited, cleanup
-----------------------------------------------*/

  CleanupStack::PopAndDestroy(3);  // two active objects and scheduler

  _LIT(KActiveComplete,"Scheduler exited, cleanup");
  console->Printf(_L("%S\n"),&KActiveComplete);
  iLog.Write(KActiveComplete);
  iLog.Close();

  console->Printf(_L("[Press any key] to exit"));
  console->Getch();

  CleanupStack::PopAndDestroy(console);
  }
```

This example starts two repeating tasks, implemented via active object classes `CPrimaryTask` and `CBackground`. Each task simply prints a message every time its `RunL()` runs and then restarts the timer. `CPrimaryTask`'s method `StartRunning()` begins the task. The first argument is a count that specifies how many times `RunL()` is invoked before the `exe` will exit. The second argument is the time interval (in microseconds) after which the task's `RunL()` is invoked. `CBackground` is similar to `CPrimaryTask` except you start it via a method called `StartBackground()`, specifying only the task interval (it does not control when the `exe` exits).

When `CPrimaryTask::RunL()` has executed for the number of times specified in the `StartRunning()` method, it calls `CActiveScheduler::Stop()`. This causes the active scheduler to exit, and control returns to `ActiveExampleL()` past the `CActiveScheduler::Start()` statement. In other words, `CActiveScheduler::Start()` returns since the event loop is finished.

Here is the `mmp` file for the example:

```
TARGET     active.exe
TARGETTYPE exe
UID 0

SOURCEPATH .
SOURCE     active.cpp

USERINCLUDE.
SYSTEMINCLUDE     \Epoc32\include

LIBRARY    euser.lib efsrv.lib flogger.lib
```

8.10.2 `RFileLogger`

This example uses a class called `RFileLogger` to write the program output and attach a timestamp. `RLogger` is a useful class for debugging, and can be used both on the emulator and on the phone itself.

To use `RFileLogger` you need to include `flogger.h` and link to `flogger.lib`. To use the logging, you need to initialize it as the example shows (`iLog` is the `RFileLogger` instance):

```
iLog.Connect();
iLog.CreateLog(_L("ActiveLogging"),_L("ActiveLog.txt"),
        EFileLoggingModeOverwrite);
```

The first argument to `CreateLog()` is the directory to contain the log file, relative to `c:\Logs`. The second argument is the name of the log file. `EFileLoggingModeOverwrite` means that the log is replaced on every run, as opposed to appending data to an existing file.

The `RFileLogger::Write()` method is then used to write to the log. `Write()` prints a nicely formatted timestamp to indicate the time each log is output. Refer to the SDK documentation for other `RFileLogger` methods to write to the log file (methods exist for formatted output and even hexadecimal dumps).

Note that the log is only written if the directory specified in `CreateLog()` exists. This provides a way to turn logging on and off (by having the directory there, or not).

In this example, before running on the emulator for the first time, you need to create an `epoc32\wins\c\Logs\ActiveLogging` directory in your SDK directory (on the phone itself it would be `c:\Logs\Active-Logging`). Then, after the program is run, the log file, `activelog.txt`, is in this directory.

An example of the log file (`activelogging.txt`) output from this example is shown below:

```
10/04/2005 9:03:31CPrimaryTask: RunL
10/04/2005 9:03:32CPrimaryTask: RunL
10/04/2005 9:03:33CPrimaryTask: RunL
10/04/2005 9:03:34CBackground RunL
10/04/2005 9:03:34CPrimaryTask: RunL
10/04/2005 9:03:35CPrimaryTask: RunL
10/04/2005 9:03:36CPrimaryTask: RunL
10/04/2005 9:03:37CPrimaryTask: RunL
10/04/2005 9:03:38CBackground RunL
10/04/2005 9:03:38CPrimaryTask: RunL
10/04/2005 9:03:40CPrimaryTask: RunL
10/04/2005 9:03:41CPrimaryTask: RunL
10/04/2005 9:03:42CBackground RunL
10/04/2005 9:03:42CPrimaryTask: RunL
10/04/2005 9:03:43CPrimaryTask: RunL
10/04/2005 9:03:44CPrimaryTask: RunL
10/04/2005 9:03:45CPrimaryTask: RunL
10/04/2005 9:03:46CBackground RunL
10/04/2005 9:03:46CPrimaryTask: RunL
10/04/2005 9:03:47CPrimaryTask: RunL
10/04/2005 9:03:49CPrimaryTask: RunL
```

```
10/04/2005 9:03:50CPrimaryTask: RunL
10/04/2005 9:03:50CBackground RunL
10/04/2005 9:03:51CPrimaryTask: RunL
10/04/2005 9:03:52CPrimaryTask: RunL
10/04/2005 9:03:53Scheduler exited, cleanup
```

9

Client/Server Framework

Symbian OS uses servers to centrally manage resources on behalf of one or more clients. A server does not normally have a graphical user interface (GUI), and, in many cases, runs its own process to provide protection and modularity. Fundamentally, a server is simply a command-processing engine – it waits for a command from a client, executes the service that corresponds to the command, returns the results to the client, and then waits for the next command. (Servers and clients are always in different threads, although you can have multiple servers in the same process, or even in the same thread.)

A client program uses a server through a client class. The client class handles the details of establishing a session with its associated server, as well as sending commands to the server and receiving responses from it – the programmer invokes the functions of the server through methods in the client class. Each method sends the appropriate command to the server and gets the command results back to return to the caller.

Symbian uses servers to implement much of its functionality. In fact, many of the Symbian OS API classes are client classes for servers. Also, many asynchronous functions are implemented within a server since the server runs within a separate thread from the client.

Here are just a few examples of servers in Symbian OS:

- The window server provides centralized access to the phone's screen, as well as user input devices such as the keyboard and pointer device. GUI applications are clients of this server, which allows them to concentrate on their own implementation, and let the server coordinate display usage between the various applications that are running, and ensure that input events are routed to the appropriate application.
- The file server handles all aspects of managing the files on the phone's storage devices on behalf of client programs. This includes creating directories and files, reading and writing files, file access control and copying and renaming. Clients interface to the server through client-side API classes such as `RFs` and `RFile`.

- The socket server manages the creation of network sockets, as well as sending and receiving data through them. The socket API classes (e.g. `RSocket`) act as clients to this server.
- The font and bitmap server provides central control over fonts and bitmaps and allows them to be efficiently accessed (for example, by only loading one copy of a ROM-based bitmap in to memory and providing shared access to all the clients that need it).

Symbian OS provides a client/server framework so that you can implement your own servers, along with client-side classes. This chapter describes this client/server framework in Symbian OS as well as describing all the applicable classes, using two example servers. Even if you do not implement your own server, understanding this information will help when writing software that uses servers.

9.1 Client/Server Overview

Servers process messages one at a time – i.e. further messages are not processed until the currently-executing command is completed. This is because a server is actually implemented as an active object and the server command handler is called from the server's `RunL()` method. So, as is the case for active objects in general, executing a server command ties up the entire thread – including any other active objects in that thread – while that command is executing. In consequence, server commands should be short and not block the thread for a long time.

For each server, there is a corresponding client-side class, derived from `RSessionBase`, that client programs use to invoke the server's functions. The client-side class handles the details of starting the server, and sending messages to and receiving messages from the server.

In order for a client to use a server, the client first establishes a communication context – known as a *session* – with the server. The client then sends commands to the server through this session. On the server side, a session is represented by an instance of a session class (derived from `CSharableSession`), and this allows context, such as state information, to be saved for that client while commands are being processed.

9.2 A Look at the Client/Server Classes

First, let's look at the key framework classes you need to use for developing both a server and its corresponding client class.

Figure 9.1 shows the basic classes used in the client/server framework.

On the client side, `RSessionBase` is the class from which you derive your server's client-side class. It represents a session with the server. `RSessionBase` methods exist to establish a session with the server and send commands to it. Key methods of `RSessionBase` are:

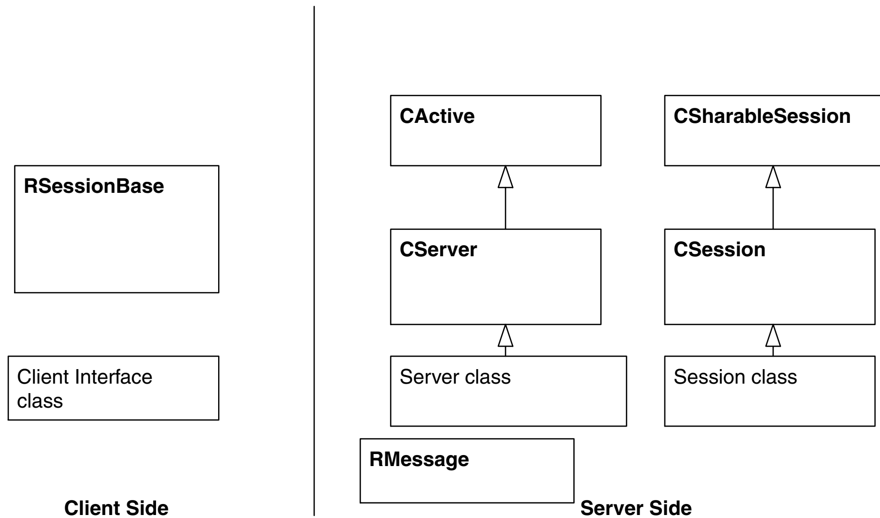

Figure 9.1 Key Client/Server Classes

- `CreateSession()`

  Your derived class calls this protected function to create a session with the server.

- `SendReceive()` and `Send()`

  Use these methods in your derived class to send messages to the server through the created session.

On the server side, you need to create classes derived from `CServer`. This is the main server class that initially receives all server messages, establishes new sessions, and routes received commands to the appropriate session. `CServer` is derived from `CActive`. There is just one instance of your `CServer`-derived class for each server. The key methods of `CServer` are:

- `NewSessionL()`

  Your `CServer`-derived class should implement this virtual function to create a new session (i.e. to instantiate and return an instance of your `CSession`-derived class). `NewSessionL()` is called when a client requests a session with the server (via the client method `CreateSession()`).

- `StartL()`

  This `CServer` method should be called when creating the server (usually from within a static `NewL()` function in a derived class). `StartL()` registers your server with the active scheduler and assigns it a name.

- `CSharableSession`

  This is the base class for a session object and is, itself, derived from `CSession`. An instance of a `CSharableSession`-derived class is created for each client session (by `CServer`'s `NewSessionL()`). The session class handles the commands sent from the client. You override `CSharableSession`'s virtual `ServiceL()` method to handle the commands.

- `RMessage`

  This is a concrete server-side class that represents the message from the client, with methods to access it. The message consists of a command code and four 32-bit arguments.

9.3 Client/Server Example

The simplest way to explain the client/server framework is by walking through an example. This section steps through a basic client/server example called `TextBuffServ`.

`TextBuffServ` maintains a text buffer for each client session and allows the client to append text to the session buffer, retrieve the text buffer, clear the buffer and backspace from its current position. In this case, the server is created as an independent process.

The example consists of the following components:

- a client-side interface class
- a process `TextBuffServ.exe` that contains the server.

Note that the purpose of this example is purely to illustrate client/server concepts – clearly, there are better ways to implement the example's functionality without using a server (such as the direct use of descriptors within your application's code).

9.3.1 Client-side Class

First let's look at the client-side class declaration (I put this in an include file named `textbuffclient.h` which all client applications would include):

```
class RTextBuff : public RSessionBase
  {
public:
  RTextBuff() { };
  TInt Connect();
  TInt AddText(const TDesC& aText);
  void GetText(TDes& aText);
```

```
  void Reset();
  void BackSpace(TInt anumChars);

  TVersion Version() const;

private:
  TInt StartServerProcess();
  };
```

The client class is derived from `RSessionBase`, which inherits from `RHandleBase`.

Starting the Server

The `Connect()` method starts the server, if it's not already started, and creates a session with the server. Applications always call this method first. When the application is finished with the session, it calls the `Close()` method, which `RTextBuff` inherits from `RHandleBase`.

Example 9.1 shows the `Connect()` method for our example.

Example 9.1. `Connect()` Function

```
_LIT(KTextBuffServerProcess,"c:\\system\\programs\\textbuffserver.exe");

TInt RTextBuff::Connect()
  {
  TInt res;

  res=CreateSession(KTextBuffServerName,Version());
  if ( res == KErrNotFound )
    {
    res = StartServerProcess();


    if (res==KErrNone)
      {
// Server coming up, try up to 10 times to create a session
      TInt retries=10;
      do
        {
        User::After(1000);
        res=CreateSession(KTextBuffServerName, Version());
        } while ( (res==KErrNotFound) && (retries--) );

      }
    }

  return(res);
  }
TInt RTextBuff::StartServerProcess()
  {
  RProcess proc;
  TInt res = proc.Create(KTextBuffServerProcess,KNullDesC);
```

```
  if  (res == KErrNone)
    {
    proc.Resume(); // start the process running
    proc.Close();  // finished with the handle
    }
  return(res);
  }
```

We create our client session by calling:

```
res=CreateSession(KTextBuffServerName,Version());
```

where the first argument is a name (defined in our client header file) that we have assigned to the server, and the second argument is the return value of `Version()`. `Version()` returns a `TVersion` object that contains a major version number, a minor version number and a build version number that are defined in our header.

Our implementation of `Version()` for the example is as follows:

```
TVersion RTextBuff::Version(void) const
  {
  return(TVersion(KTextBuffMajorVersionNumber,
      KTextBuffMinorVersionNumber, KTextBuffBuildVersionNumber));
```

And the version definitions in our header file are as follows:

```
const TUint KTextBuffMajorVersionNumber=0;
const TUint KTextBuffMinorVersionNumber=1;
const TUint KTextBuffBuildVersionNumber=1;
```

(See Example 9.3 in Section 9.3.2 for the complete header.)

What is the version argument used for when creating a session? In brief, it indicates the earliest version of the server that the client will work with. So if you change your client code so that it will no longer work with older versions of the server, then you would increase the version number – for example, by incrementing the minor number – in the header file. Then, if this new version of the client is used with an older, incompatible version of the server, the version number that the client passes in the `CreateSession()` call is higher than the version number built into the server, causing the server to return an error.

As you will see in the server-side example code, the version number from `CreateSession()` is passed to the `NewSessionL()` function of your `CServer` class, which is where the actual version comparison is made, using a call to `User::QueryVersionSupported()`. If the comparison fails, `NewSessionL()` leaves with the error `KErrNotSupported` (causing `CreateSession()` – and thus the client `Connect()` method – to return `KErrNotSupported`).

If CreateSession() returns KErrNotFound, then the system could not find a server with the specified server name, indicating that our server had not been started yet, and that we need to load and start it. We do this by calling StartServer() for our class, which uses RProcess to launch the server process. After a small delay, CreateSession() is again called. CreateSession(), and hence Connect(), will either return KErrNone once the server is up and running, or it will return an error.

Another version of CreateSession() exists that has the form:

```
TInt CreateSession(TDesC& aName,TVersion aVer, TInt aNumSlots)
```

aNumSlots determines how many messages can be queued from the client to the server at one time (a space for one message being a slot). The advantage of using this version is that the slots are pre-allocated when you create the session and thus memory use is more controlled.

If this version of CreateSession() is used without this last aNumSlots argument (as in the example), then memory on the kernel heap is allocated for each message as it is queued up. This will allow for a large queue, but it sacrifices some control over memory allocation, since kernel memory could become exhausted if the queue of messages builds up for too long.

Repeatedly calling CreateSession() after starting the process, as in the example's Connect() method, may appear a bit inefficient at first glance, but this is the simplest approach and is reasonably robust. In any case, it is likely that the server will be up by the time the first CreateSession() is called, provided that your server initialization process is short.

I presented the example this way for simplicity, but there are other more efficient ways to do it. For example, you can create a semaphore that the server would signal once fully initialized, and have your Connect() function wait at this semaphore before calling CreateSession(). Although this would work in most cases, a major disadvantage is that if the process does not initialize correctly due to an error, then your Connect() function could hang forever on the semaphore Wait() (semaphores have no timeouts you can use while waiting).

A more correct approach would be to pass a TRequestStatus variable along with your thread identifier to the server as an argument and have your server signal the client when the server is initialized (using the RThread:RequestComplete() function). Then, in your Connect() function, you can call the RProcess::Logon() function and wait for either the process to die or the server to be initialized using User::WaitForRequest().

This chapter will not go into detail on these methods, but concentrates on the client/server framework itself. The simple Connect() I presented should be adequate for most servers you are likely to need.

Invoking the Server's Services

The following four methods of `RTextBuff` invoke the server's services:

- `AddText()` appends the specified text to the session's text buffer
- `GetText()` retrieves the text from the session's text buffer
- `Reset()` clears the session's text buffer
- `BackSpace()` decrements the text buffer position by the indicated number of characters.

When the application is finished with the session, it calls `Close()`.

Example 9.2 shows the code for the `AddText()`, `GetText()`, `BackSpace()` and `Reset()` functions.

Example 9.2. Client Methods for invoking `TextBuffServ` services

```
TInt RTextBuff::AddText(const TDesC& aText)
  {
  TInt rc;
  TAny *p[KMaxMessageArguments];
  p[0]= (TAny*) &aText;
  rc = SendReceive(ETextBuffAddText,&p[0]);
  return rc;
  }

void RTextBuff::GetText(TDes& aText)
  {
  TAny *p[KMaxMessageArguments];
  p[0]= (TAny*) &aText;
  SendReceive(ETextBuffGetText,&p[0]);
  }

void RTextBuff::Reset()
  {
  TAny *p[KMaxMessageArguments];
  SendReceive(ETextBuffReset,&p[0]);
  }

void RTextBuff::BackSpace(TInt aNumChars)
  {
  TAny *p[KMaxMessageArguments];
  p[0]= (TAny*) aNumChars;
  SendReceive(ETextBuffBackSpace,&p[0]);
  }
```

The methods in Example 9.2 are simple wrappers that send commands (along with command arguments) to the server, using the `SendReceive()` method of `RSessionBase`, leaving the server to do the actual work. `SendReceive()` is a *protected* function of `RSessionBase` and is defined as follows:

```
void SendReceive(TInt aCommand, TAny *args)
```

`aCommand` indicates the command that the server session is to process.

args is an array of four 32-bit arguments, sent to the server along with the command. The 32-bit arguments must be cast to type TAny*, from either of two forms:

- A 32-bit integer (Backspace() uses the first argument in this way).
- A pointer to a descriptor (used by AddText() and GetText() in the example).

After sending the command, SendReceive() will wait for the server to complete the command's execution before returning. However, there is also an asynchronous version of SendReceive() that is defined as follows:

```
void SendReceive(TInt aCommand, TAny *args, TRequestStatus &aStat)
```

This version of SendReceive() does not block execution of the calling thread, but instead signals an asynchronous event to the calling thread when the server has completed execution of the command. You can use this version to create asynchronous client functions. For example, an asynchronous version of function DoSomeFunction() can be implemented as follows:

```
void RMyClient::DoSomeFunction(TInt arg, TRequestStatus& aStat)
  {
  TAny *p[KMaxMessageArguments];
  p[0]=(TAny *) arg;
  SendReceive(ESomeFunction,&p[0],aStat);
  }
```

The client program can then invoke DoSomeFunction() from an active object followed by a call to the active object's SetActive(). The active object's RunL() function is invoked when the server actually completes the command.

RSessionBase also provides a Send() method. Unlike SendReceive(), this function does not indicate when the server completes the command, but simply returns once the message is sent.

The commands sent to the server are defined in an include file that needs to be included by both the server and client (I named it textbuff.h). It contains the server name, the client/server commands and the version information for the server, and is defined as follows:

```
#include <e32base.h>

_LIT(KTextBuffServerName,"TextBuffServer");

//The server version. A version must be specified when creating a session
        with the server.
```

```
const TUint KTextBuffMajorVersionNumber=0;
const TUint KTextBuffMinorVersionNumber=1;
const TUint KTextBuffBuildVersionNumber=1;

enum TTextBuffSrvCmds
  {
  ETextBuffReset,
  ETextBuffAddText,
  ETextBuffGetText,
  ETextBuffBackSpace,
  ETextBuffCloseSession   // Used later in this chapter.
  };
```

9.3.2 Server Implementation

Example 9.3 shows the server's include file (`tbuffserver.h`), which contains the server and session class definitions.

Example 9.3. Server and Session Classes for `TextBuffServ`

```
#include "textbuff.h"

enum TTextBuffPanic
  {
  EInvalidCommand,
  EInvalidDescriptor,
  EServerInitError
  };

#define MAX_BUFFER_SIZE 4096

class CTextBuffServer : public CServer
  {
public:
  static CTextBuffServer* NewL();
  CTextBuffServer();
  CSharableSession* NewSessionL(const TVersion &aVersion) const;
  };

class CTextBuffSession : public CSharableSession
  {
public:
  static CTextBuffSession* NewL(CTextBuffServer* aServer);
  CTextBuffSession() {};
  ~CTextBuffSession();
  void ConstructL(CTextBuffServer* aServer);
  void ServiceL(const RMessage& aMessage);
  void DispatchMessageL(const RMessage& aMessage);

  TInt AddText(TDesC& txt);
  void BackSpace(TInt chars);
  TPtrC GetText();
  void Reset();

  void ClientPanic(TInt aPanicCode) const;

private:
```

```
HBufC* iTextBuff;
};
```

First, note that I have included the common client/server header file `textbuff.h`, since the server also needs the server name, version numbers, and the client/server commands.

The server class is derived from `CServer`, and there is only one instance of this when the server is running. The session class is derived from `CSharableSession`, and an instance of this class is created for each session a client opens.

The server is created and started by calling the static `NewL()` function of our `CTextBuffServ` class. `CTextBuffServ::NewL()` and the `CTextBuffServ` constructor are shown below:

```
CTextBuffServer* CTextBuffServer::NewL()
  {
  CTextBuffServer* self=new (ELeave) CTextBuffServer;
  self->StartL(KTextBuffServerName);
  return self;
  }

CTextBuffServer::CTextBuffServer()
: CServer(EPriorityStandard,ESharableSessions)
  {
  }
```

The constructor passes the server's priority to the `CServer` base class as the first argument. This becomes the priority of the underlying server's active object. The second argument means that the session is sharable between multiple threads in the same process.

Note that the server priority does not necessarily define the priority of the server in relation to other servers, but defines its priority in relation to all active objects (which could include other servers) connected to the thread's active scheduler.

`CTextBuffServ::NewL()` calls the base class method, `CServer::StartL()` to register the server and assign it a name. `StartL()` adds the server to the active scheduler and registers it with the name passed to it (`KTextBuffServerName` in this case). This name is referenced from a client when the client requests a connection to the server. `StartL()` does not begin the server's message processing – this occurs after the thread's active scheduler is started.

Example 9.4 shows the startup code for the `textbuffserv` process.

Example 9.4. Server Process Startup

```
static void StartServerL()
  {
// create and install an active scheduler
```

```
  CActiveScheduler *pA=new (ELeave) CActiveScheduler;
  CActiveScheduler::Install(pA);
  CleanupStack::PushL(pA);

  // create server and install
  CTextBuffServer *serv;
  serv = CTextBuffServer::NewL();

  CActiveScheduler::Start();

  delete serv;
  CleanupStack::PopAndDestroy(); // delete pA
  }

static TInt StartServer()
  {
  CTrapCleanup* cleanup=CTrapCleanup::New();

  // create a cleanup stack

  TRAPD(res,StartServerL());
  if (res)
    {
    _LIT(KTxtBuffServer,"TextBuffServer");
    User::Panic(KTxtBuffServer,EServerInitError);
    }

  delete cleanup;
  return(res);
  }
GLDEF_C TInt E32Main()
  {
  TInt res = StartServer();
  return(res);
  }
```

When the server process is started, control goes to `E32Main()`. Upon entry, the program calls `StartServer()` which creates a cleanup stack and calls `StartServerL()`. `StartServerL()` creates and installs the thread's active scheduler and creates the server class by calling `CTextBuffServ::NewL()`. The active scheduler is then started with `CActiveScheduler::Start()`. The server is now waiting for a client to request a session with the server, and we signal the semaphore we have created to indicate this to the client.

When the server receives a command to create a session (via a client calling `RSessionBase::CreateSession()`), `CServer` invokes its virtual `NewSessionL()`. This function is implemented in the `CServer`-derived class and its purpose is to create, and return a pointer to, an instance of the server's session class.

The following code shows `TextBuffServ`'s implementation of `NewSessionL()`:

```
CSharableSession* CTextBuffServer::NewSessionL(const TVersion &aVersion)
        const
  {
  // check version is ok
  TVersion v(KTextBuffMajorVersionNumber,KTextBuffMinorVersionNumber,
      KTextBuffBuildVersionNumber);
  if (!User::QueryVersionSupported(v,aVersion))
      User::Leave(KErrNotSupported);

  return CTextBuffSession::NewL((CTextBuffServer*)this);
  }
```

It calls `CTextBuffSession::NewL()` to create the `CSession`-based object (in turn based on `CSharableSession`, a class that allows multiple threads in the same process to access a single client session). Thereafter all client messages, through the session that created this session class instance, will go to this instance.

Also `NewSessionL()` checks the version number passed to the function and, if the version required by the client is higher than that of the server, it will leave with the error `KErrNotSupported`, causing the `CreateSession()` call on the client-side to return this error.

Below is the `CTextBuffSession::NewL()` function, along with the secondary constructor and destructor for the class:

```
CTextBuffSession* CTextBuffSession::NewL(CTextBuffServer* aServer)
  {
  CTextBuffSession* self=new (ELeave) CTextBuffSession();
  CleanupStack::PushL(self);
  self->ConstructL(aServer);
  CleanupStack::Pop();
  return self;
  }

// second-phase C++ constructor
void CTextBuffSession::ConstructL(CTextBuffServer* aServer)
  {
// second-phase construct base class
  CSharableSession::CreateL(*aServer);
  iTextBuff = HBufC::NewL(MAX_BUFFER_SIZE);
  }

CTextBuffSession::~CTextBuffSession()
  {
  delete iTextBuff;
  }
```

`CTextBuffSession::NewL()` calls the secondary constructor (`ConstructL()`) and uses the cleanup stack to safely handle any leave that may occur. The `CSession::CreateL()` function must be called to complete the session creation. Its default operation is to store the pointer to the server that created this session, passed to it as an argument.

For this example server, a text buffer is associated with each session, and the `ConstructL()` function creates the buffer as a heap-allocated `HBufC` descriptor.

Processing Messages from the Client

When a server receives a message from a client, the server creates an instance of a class called `RMessage` to hold the message contents. Then the server invokes the `ServiceL()` method of the appropriate session object – supplying the `RMessage` object as its argument.

Example 9.5 shows the session command handler for the `textbuff-serv` example.

Example 9.5. Handling the Server Commands

```
void CTextBuffSession::ServiceL(const RMessage& aMessage)
  {
  TRAPD(err,DispatchMessageL(aMessage));
  aMessage.Complete(err);
  }

void CTextBuffSession::DispatchMessageL(const RMessage& aMessage)
  {
// check for session-relative requests
  switch (aMessage.Function())
    {
    case ETextBuffAddText:
      {
      const TAny* pD=aMessage.Ptr0();
      TBuf<200> tmp;
      TRAPD(res,aMessage.ReadL(pD,tmp));
      if (res!=KErrNone)
        ClientPanic(EInvalidDescriptor);
      User::LeaveIfError(AddText(tmp));
      break;
      }
    case ETextBuffGetText:
      {
      TPtrC buff = GetText();
      TRAPD(res,aMessage.WriteL(aMessage.Ptr0(),buff));
      if (res!=KErrNone)
        {
        ClientPanic(EInvalidDescriptor);
        return;
        }
        break;
      }
    case ETextBuffReset:
      Reset();
      break;
    case ETextBuffBackSpace:
      BackSpace(aMessage.Int0());
      break;
    case ETextBuffCloseSession:
```

```
      CActiveScheduler::Stop();
      break;
    default:
      ClientPanic(EInvalidCommand);
      break;
    }
  }

void CTextBuffSession::Reset()
  {
  iTextBuff->Des().Zero();
  }

TInt CTextBuffSession::AddText(TDesC& aText)
  {
  TInt rc=0;
  if ( (aText.Length() + iTextBuff->Des().Length()) >
    iTextBuff->Des().MaxLength())
    return KErrTooBig;
  else
    iTextBuff->Des().Append(aText);
  return(rc);
  }

void CTextBuffSession::BackSpace(TInt chars)
  {
  if (chars <= iTextBuff->Des().Length())
    {
    TInt newLength = iTextBuff->Des().Length() - chars;
    iTextBuff->Des().SetLength(newLength);
    }
  }

TPtrC CTextBuffSession::GetText()
  {
  return *iTextBuff;
  }
```

`ServiceL()` invokes another method, `DispatchMessageL()`, to handle the message. `RMessage` has the following methods for accessing the command code and arguments of the message sent:

- `Function()` returns the command code that was specified via the first argument of the client object's `SendReceive()`/`Send()` function.
- `Int0()`, `Int1()`, `Int2()` and `Int3()` return, as integers, the four 32-bit values passed in the second argument of the `SendReceive()` and `Send()` function.
- `Ptr0()`, `Ptr1()`, `Ptr2()` and `Ptr3()` return, as `TAny *` pointers, the four 32-bit values passed in the second argument of the `SendReceive()` and `Send()` functions. Although these values are typed as `TAny *`, the server expects them to be pointers to descriptors. This is because client memory is accessed only through inter-thread

function calls, which take descriptors as arguments. Client pointers are never accessed directly by the server, since the client will usually reside in a separate process.

- `Panic(TDesC& aCategory,TInt aCode)` panics the client-side thread that sent the message to this session. This is usually done when the server detects coding errors in the client.
- `Complete(TInt status)` is called by the server when it has completed processing of the message. The passed value is the status returned by the `SendReceive()` method that sent the message. `ClientPanic()` is implemented as follows:

```
void CTextBuffSession::ClientPanic(TInt aPanic) const
  {
  _LIT(KTextBuffServSess,"CTextBuffSession");
  Message().Panic(KTextBuffServSess,aPanic);
  }
```

- `Message()` is a session base class method that returns the current `RMessage` being handled by the session, as a convenient alternative to passing the `RMessage` to other session functions. Calling its `Panic()` function causes a panic on the client thread that sent the message, as described earlier.

Using Pointers to Transfer Data Between the Client and Server

In many cases, a client specifies a buffer in the client memory space as an argument to the command that it sends to the server. This could be a buffer for the server to either read input from (`AddText` uses this in our example), or write output to (as our `GetText` command does). Since the client and server reside in different threads and, more importantly, could also reside in different processes, the server must use inter-thread data accesses rather than direct access through the client pointers.

We saw in Chapter 7 that, from the current thread, you can open a handle to another thread with `RThread`, and then use `RThread`'s `ReadL()` and `WriteL()` methods to read and write data to that other thread's address space. For convenience, `RMessage` provides its own `ReadL()` and `WriteL()` functions so you can access the address space of the client thread that sent the message. This is possible because `RMessage` contains an `RThread` handle to the sending client. (In fact you can call `RMessage`'s `Client()` method to return a handle to the client thread if you need to operate on the thread directly.)

Also, note that inter-thread read and write functions always use pointers to descriptors in the target thread's address space, and these descriptors, in turn, describe the target thread's memory buffer. Thus a pointer passed from a client to a server must always point to a valid descriptor in the

client address space, and never directly to the client buffer itself (see Chapter 7).

Returning to the example server, `DispatchMessageL()` calls `RMessage::Function()` to determine which command code was sent by the client, and then handles the command appropriately.

For the command `ETextBuffAddText`, the first argument of the message is a pointer to the client descriptor that contains the text to be added to the session's text buffer. The text is read using `RMessage::ReadL()`. This is an inter-thread read and thus will work properly when reading from a client address space in either the same or (as in this case) a different process. `RMessage` can do this because it has a handle to the client thread that sent the message – internally it just calls the client thread's `RThread::ReadL()` method.

If the pointer in the message does not point to a valid descriptor, an error will occur in the `ReadL()` function. The error is handled by a utility function which concludes by calling `CSession::Panic()` to panic the client thread.

Once the text has been read, `AddText()` is called to append the text to the text buffer associated with that session.

In the case of `ETextBuffGetText`, the first argument is a pointer to the client-side descriptor to which the text is to be written. The text is written using the inter-thread `RMessage::WriteL()` method.

`ETextBuffBackSpace` is an example of a case where the first argument is an integer rather than a pointer. This integer, read by the `RMessage::Int0()` method, indicates the number of characters to backspace in the text buffer.

9.3.3 Example Use of `TextBuffSrv`

Here is an example of how a client program might use `TextBuffSrv`:

```
LOCAL_C void ClientProgL()
  {
  RTextBuff textbuff;

  TBuf<100> t;
  TInt ret=textbuff.Connect();
  User::LeaveIfError(ret);

  textbuff.Reset();
  textbuff.AddText(_L("Hello"));
  textbuff.GetText(t);

  console->Printf(_L("GetText text=%S\n"),&t);

  textbuff.AddText(_L("Again"));
  textbuff.GetText(t);

  console->Printf(_L("GetText text=%S\n"),&t);
```

```
textbuff.BackSpace(3);
textbuff.AddText(_L("xxxx"));
t.Zero();
textbuff.GetText(t);

console->Printf(_L("GetText text=%S\n"),&t);

textbuff.Reset();
textbuff.AddText(_L("Start"));
textbuff.GetText(t);
console->Printf(_L("GetText text=%S\n"),&t);

textbuff.Close();
}
```

The output of this would be:

```
GetText text=Hello
GetText text=HelloAgain
GetText text=HelloAgxxxx
GetText text=Start
```

9.3.4 Shutting down the Server

In our example the server is never shut down, and many system servers in Symbian OS behave in this way. However, for applications, it is more common for the servers to be *transient* – i.e. they are shut down when they are not being used. This saves on system resources.

If our server always has just one client, and we want the server to be shut down once the client has finished with it, we can override the `RSessionBase Close()` method in our client class as follows:

```
void RTextBuff::Close()
  {
  TAny *p[KMaxMessageArguments];
  SendReceive(ETextBuffCloseSession,&p[0]);
  RHandleBase::Close();
  }
```

Then in the server, you can include an additional case in `ServiceL()` to handle this close command:

```
case ETextBuffCloseSession:
    CActiveScheduler::Stop();
    break;
```

Stopping the active scheduler would cause `CActiveScheduler::Start()` to return (in `StartServer()`) and shut down the server, and additionally clean up and exit the process.

Why would we ever want a server that just has one client associated with it? This is commonly done when you want to break up an application into an engine (which is the server) as a separate exe and a GUI *application*. One reason to put the bulk of your functionality in a server exe, is that you can have writable global variables in the exe, and this simplifies the porting of code from other systems that use writable global variables.

Using this GUI/engine model, the server is really just a component of your application, and only your GUI application will connect to it. Therefore, the close functionality just presented will work fine, with the server being shut down immediately. An application may shut down this type of server at any time, but would normally do so just before the application exits.

However, if you are servicing multiple clients (which is what servers are really meant for anyway), then you do not want to implement the Close() in this way, since you do not want one client to be able to close the server.

A good way to implement the shutdown in this case is to keep a reference count that tells you how many clients currently have open sessions with the server (you can increment/decrement a reference count variable in your server class as sessions are created and deleted). When the reference count reaches zero, start a shutdown timer, using a CTimer class. When the timer expires, close the server by stopping the active scheduler. If another session is opened while the timeout is in progress, cancel the timeout.

9.3.5 Running the Server on the Emulator

As discussed in Chapters 5 and 7, the emulator does not support multiple processes, and on Symbian OS versions before v8.0, RProcess is not supported at all (RProcess Create() returns KErrNotSupported). In pre-v8.0 versions of Symbian OS, to run your server on the emulator, you must build your exe as a DLL, and use RThread instead of RProcess. Versions 8.0 and later, while still not supporting processes, do provide emulation, through RProcess, to mimic the device more closely.

On the client side, to be able to run the example on pre-v8.0 emulators, you need to replace the implementation of the StartServer-Process() function, as in the following example (which includes both versions):

```
#ifdef __WINS__

_LIT(KTextBuffServerProcessDll,"textbuffserver.dll");
```

```
TInt RTextBuff::StartServerProcess()
  {
  RLibrary lib;
  TInt res;

  res = lib.Load(KTextBuffServerProcessDll);
  if (res == KErrNone)
    {
    TLibraryFunction func1=lib.Lookup(1);
    TThreadFunction threadFunc =
        reinterpret_cast<TThreadFunction>(func1());

    TName name(KTextBuffServerName);
    RThread thd;

    // Create thread.

    res = thd.Create(KTextBuffServerName,threadFunc,
        KDefaultStackSize,NULL,NULL);

    if (res == KErrNone)
    thd.Resume();
    }

  return(res);
  }

#else

TInt RTextBuff::StartServerProcess()
  {
  RProcess proc;
  TInt res = proc.Create(KTextBuffServerProcess,KNullDesC);
  if  (res == KErrNone)
    {
    proc.Resume(); // start the process running
    proc.Close();   // finished with the handle
    }

  return(res);
  }

#endif
```

And on the server side:

```
#ifdef __WINS__

static TInt ServerThreadFunction()
  {
  TInt res = StartServer();
  return(res);
  }

EXPORT_C TInt WinsMain()
  {
```

```
  return reinterpret_cast<TInt>(&ServerThreadFunction);
  }

// DLL entry point
TInt E32Dll(TDllReason /*aReason*/)
  {
  return(KErrNone);
  }

#else
GLDEF_C TInt E32Main()
  {
  TInt res = StartServer();
  return(res);
  }

#endif
```

On the server side, in the case of compiling for the emulator, `WinsMain()` is the first exported function instead of `E32Main()`. `WinsMain()` returns a function pointer to `ServerThreadFunction()` which in turn runs the server. On the client side, `StartProcess()` uses `RLibrary` to load your server `DLL`. It then looks up and calls the first exported function, `WinsMain()`. `StartProcess()` then creates a thread, using `RThread`, and specifies the function pointer returned by `WinsMain()` as the thread entry point. Once the `Resume()` function is called on the thread, the thread starts executing at `ServerThreadFunction()` and runs the server.

In your server `mmp` file, you should use `TARGETTYPE EPOCEXE`. This causes your process code to be built as a `DLL` for the emulator and an `exe` for target phone builds.

9.4 Subsessions of the Server

The client/server framework provides the ability to have subsessions within existing sessions, and this is an efficient solution for functionality that requires many client sessions. This section gives only a general overview of subsessions, and you should refer to the SDK documentation for more details of creating subsession-based servers.

Subsessions have fewer overheads than full sessions – but they are more complex to implement on the server side, since `CServer` and `CSharableSession` provide no base class functionality for subsessions.

Subsessions are represented on the client-side by a class derived from `RSubSessionBase`. A subsession class must be associated with a parent client session, and thus there are a minimum of two client-side classes for a server that supports subsessions – a session class, derived from `RSessionBase`, and a subsession class, derived from `RSubSessionBase`.

9.4.1 Example of an API Based on Subsessions

Many of the Symbian OS APIs use servers that support subsessions. A good example is the file server.

For example, look at the following code, which writes data to a file:

```
void FileWriteExL()
  {
  RFs fsSession;   // File server session
  fsSession.Connect();
  CleanupClosePushL(fsSession);  // this will close the session in case
                                 // we leave

  RFile f1;   // Represents a file, a subsession class

  _LIT(KFileName,"c:\\test.txt");

  TInt err=f1.Open(fsSession,KFileName,EFileWrite);
  if (err==KErrNotFound) // file does not exist - create it
    {
    err=f1.Create(fsSession,KFileName,EFileWrite);
    }
  User::LeaveIfError(err);

  _LIT8(KDataToWrite,"Test data to write");

  f1.Write(KDataToWrite);
  f1.Close();

  CleanupStack::PopAndDestroy(&fsSession);
  }
```

This example opens a file (first creating it if it does not already exist), and writes some data to it. `RFs` is a client session class (derived from `RSessionBase`), to represent a session to the file server. `RFile` is a subsession class, and you pass the parent session class as the first argument to the `Open()` and `Create()` functions to associate it with the parent session. You can open as many files as you like using this parent session class, although we open only one here.

The approach of implementing the file system API using subsessions in Symbian OS is much more efficient than the alternative of using a full session (and thus a `CSharableSession` instance on the server side) for each file you open.

You must include file `f32file.h` and library `efsrv.lib` to use the file system API.

Another example of an API that supports subsessions is the network socket API (see Chapter 10).

9.4.2 How Subsessions Work

While each session has its own CSharableSession instance on the server side, subsessions do not. Subsession commands go to the parent session and it is this session's responsibility to determine (by means of a handle sent with the command) the subsession to which the command belongs, and execute accordingly. Although not strictly necessary, subsessions are most often represented by instances of classes derived from the reference counting CObject class.

As mentioned earlier, all subsession commands, including the command to create a subsession itself, are routed to the ServiceL() method of the parent session's CSharableSession-based object. It is the responsibility of the session's ServiceL() implementation to create a subsession object, return a unique handle for it, then forward further subsession commands to the correct subsession object, based on the handle.

Most servers use CObject as the base class to implement subsession classes. A key reason for this is the ability to identify an instance of a CObject by means of its unique integer handle. All subsession commands will supply this handle, and it is up to the ServiceL() to use the handle to locate the correct CObject and invoke the subsession command on it. Refer to Figure 9.2 for this flow.

From Figure 9.2, you can see that the ServiceL() method of the session receives the commands for both the session itself and the subsessions belonging to that session. In the case of subsession commands, the handle (accessed by RMessage::Int3() of the command) is used to locate the correct instance of the subsession class and pass the message to it.

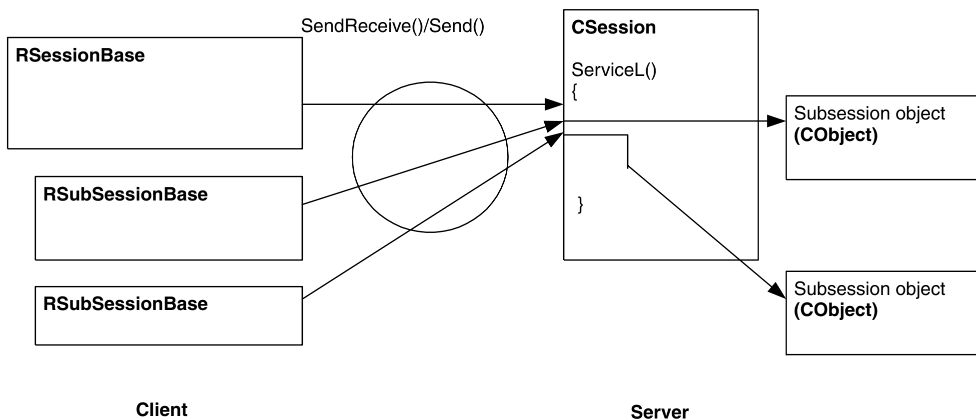

Figure 9.2 Session and Subsession Command Flow

9.4.3 Creating a Subsession

Subsessions on the Client Side

On the client side, RSubSessionBase is similar to RSessionBase in that RSubSessionBase provides methods to create a subsession and to send commands to the server. As with RSessionBase, these methods are protected, and thus the user of the subsession class does not directly call them. They are used for implementing the richer subsession interface methods in the RSubSession-derived class.

The following method creates a subsession within the specified server session, and associates this subsession with the class:

```
TInt CreateSubSession(RSessionBase& aSession,TInt aFunction, TAny *aArgs)
```

aSession is a reference to an already connected client session class. aFunction is the command that is sent to the server's session to create a subsession. On the server side, the ServiceL() for the session class should handle this command by creating the subsession, as described in the following section. aArgs is an array of four 32-bit values. Only the first three can be used.

The arguments in aArgs are passed to the server session along with aFunction, and are available for access through the RMessage object passed to ServiceL(). The following methods send commands to the server's subsession.

```
void SendReceive(TInt aCommand, TAny *args)
void SendReceive(TInt aCommand, TAny *args, TRequestStatus &aStat)
void Send(TInt aCommand, TAny *args)
```

Their formats are identical to the equivalent functions provided in RSessionBase (except you can only use three arguments in args since the fourth is always the handle). The commands are routed to the ServiceL() method of the server's CSession-based object that is associated with the subsession's session.

Subsessions on the Server Side

When the client subsession calls RSubSessionBase::CreateSubSession(), this function sends the command code for creation of the subsession to the server. The server, in turn, calls the ServiceL() function in the parent session's CSession-based object, to create a subsession object. The handle of this subsession is written to the client,

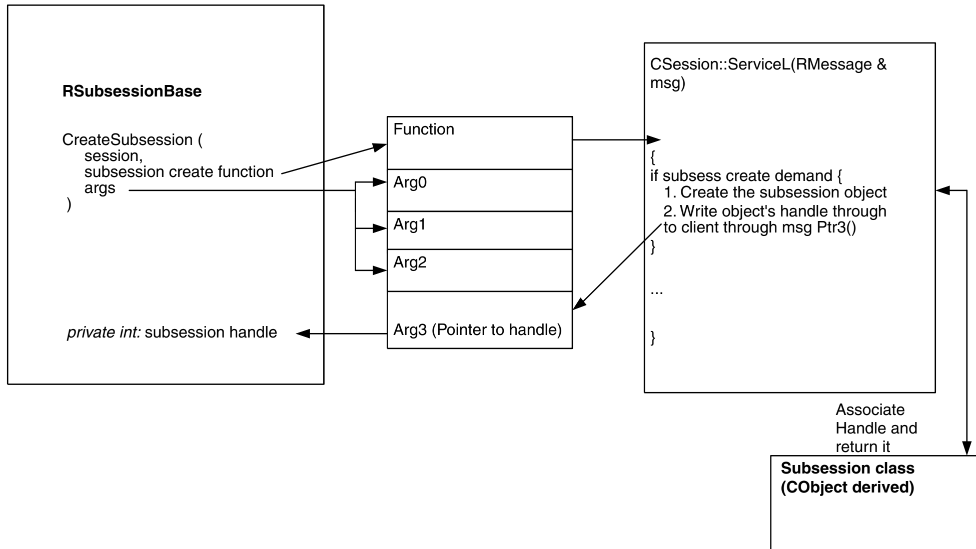

Figure 9.3 Creating a Subsession

through the pointer sent in the fourth argument of the server message. The flow of this process is illustrated in Figure 9.3.

Once the subsession has been created, commands are sent from your subsession client class via `RSubSessionBase`'s `Send()` and `SendReceive()` functions. Remember that only three arguments can be supplied to `SendReceive()` and `Send()` when using subsessions since `RSubSessionBase` transparently supplies the subsession's handle (set during the `CreateSubSession()`) as the fourth argument of the command. The `ServiceL()` of the parent session looks up the subsession class instance that corresponds to that handle, and invokes the function using that instance.

9.4.4 Using `CObject` as your Subsession Base Class

Just as each session is represented by a separate `CSession`-based class instance, every subsession is also represented by a class instance. But, while session classes are derived from `CSharableSession`, the client/server framework does not specify the derivation of a subsession class (since your session's `ServiceL()` completely determines subsession implementation).

When creating a subsession object, your session must return a unique subsession handle to the client class. Subsequent commands to this subsession will supply this handle, which is used to route the command to the correct subsession instance. This means that the `CObject` class, with its associated unique identifier, is ideally suited to representing a subsession, and subsession classes are normally derived from `CObject`.

Unfortunately, you can't just instantiate a `CObject`-derived class and retrieve its handle – you need to deal with a few other associated classes:

- `CObjectCon` is a `CObject` container, which must be used to create `CObject` instances.
- `CObjectConIx` is a class that needs to be used to create `CObject` containers.
- `CObjectIx` is an index to `CObject` instances, which may themselves be held in one or more containers.

The details of the `CObject` class and of `CObject` containers and indexes are beyond the scope of this chapter. However, the SDK provides good information on these topics.

10

Symbian OS TCP/IP Network Programming

The ability to communicate data is a feature that differentiates smartphones from traditional voice-only mobile phones. Smartphones can connect to a network through cellular technologies such as GPRS and EDGE and perform a variety of tasks normally associated with networked PCs. Some phones also have WiFi capability (e.g. the Nokia 9500 Communicator) allowing them to connect to a Wireless LAN.

Here are just some of the smartphone applications made possible by data communication:

- browsing (HTML, WAP)
- email
- instant messaging
- streaming media (mobile video services, etc)
- multiplayer network-connected games.

The TCP/IP protocol suite is used for most networked services including the examples just given. In fact, TCP/IP is the de facto standard for communicating on the Internet (it's almost synonymous with the Internet itself), and is used in most private data networks as well.

Symbian OS provides full TCP/IP networking support as well as a socket-based API to allow developers to write their own communication software. This chapter introduces TCP/IP on a Symbian OS device and shows how to use the socket API to write your own TCP/IP networking applications. The most popular network API for programming in TCP/IP is the Berkley Unix (or BSD) C-based socket API and it is presented in this chapter for comparison (Symbian supports a version of it). The Symbian OS native C++ socket API is then presented, and compared to the BSD socket API.

We begin with a generic introduction to TCP/IP programming for those new to network programming. If you are already familiar with the general principles of TCP/IP and socket programming you can skip the preliminary TCP/IP and BSD sections, and go directly to Section 10.2.

At the end of the chapter is a comprehensive example that enables you to expand `SimpleEx` to retrieve the current temperature from a weather network server and display it on the screen.

10.1 Introduction to TCP/IP

TCP/IP refers to a suite of protocols as opposed to a single protocol (i.e. TCP over IP). This protocol suite, sometimes referred to as the Internet Protocol Suite, serves as the foundation for communication on the Internet, as well as most local networks. TCP/IP, like any other communication protocol suite, insulates network applications from the underlying hardware and low-level software so that they can concentrate on sending and receiving application data over the network.

Let's look at the main protocols that are included in the TCP/IP suite – TCP, UDP and IP. These protocols are the most important ones for the network application programmer and are the ones that are covered in this chapter.

- TCP – Transmission Control Protocol

  TCP is a transport-level, connection-oriented protocol and provides built-in flow control for reliable transfer of data between network nodes. TCP is packaged and sent over the network layer IP protocol.

  TCP is the protocol most used by network applications including the World Wide Web.

- UDP – User Datagram Protocol.

  UDP, like TCP, is a transport-level protocol; however, UDP is connectionless and thus more lightweight than TCP. UDP is basically a 'fire-and-forget' protocol in that it does not provide any built-in confirmation that the packet has arrived at its destination, nor does it perform any retransmissions on errors, or handshaking of any kind. UDP is used where speed is most important and it doesn't matter if a few packets are lost.

  UDP is not as commonly used as TCP, except for programs (e.g. games) where speed is more important than reliability. UDP is also used for some network services, such as DNS (Domain Name Service) which resolves a host name into an IP address.

- IP – Internet Protocol

  IP is the network-level protocol over which both TCP and UDP (as well as other Internet suite protocols) are layered. TCP and UDP packets reside within the data area of an IP packet – like an envelope within an envelope. When a destination node receives an IP packet, it looks at a protocol field in the header to determine the protocol of the data it contains.

IP is connectionless and data is transferred via packets that flow from a source to a destination. IP defines how these packets are routed and delivered from the source to the destination by the various routers and switches, using a quad byte address – known as the IP address – assigned to each communication point on the network. IP also handles functions such as fragmenting large packets into multiple smaller packets, and limiting the lifetime of a packet in the event of a router setup problem (to avoid having a packet roaming around the Internet forever).

At the application level there is seldom a need to deal with IP directly, applications use TCP and UDP instead.

Figure 10.1 shows a simple diagram of TCP and UDP layered on top of IP.

After reading the descriptions of UDP and IP, you may wonder what advantages UDP provides over IP, since both are connectionless with no flow control built in. The answer is: not a lot. However, UDP adds a port address (as TCP does) to supplement the IP address and it includes a data integrity checksum (IP only checksums the IP header).

10.1.1 IP Addresses and Ports

In both TCP and UDP, data is transferred between two endpoints of a network. An endpoint is uniquely identified by the combination of an *IP address* and a *port address*. The IP address (which has the form xx.xx.xx.xx – e.g. 10.1.2.3) identifies a particular machine, and the port address – which specifies one of 65,536 possible ports – identifies a particular endpoint on that machine.

In the case of TCP, a virtual connection is first established between the two endpoints (via a special handshake), and data is then sent through this virtual connection. In UDP, no virtual connection is established and packets are simply sent from one endpoint to another.

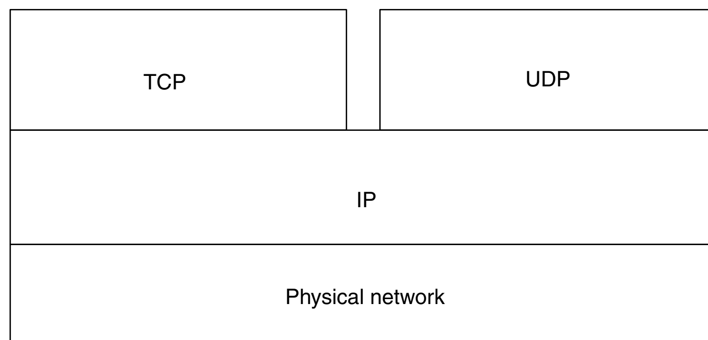

Figure 10.1 TCP, UDP, IP layering

10.1.2 Client/Server Network Model

Both TCP and UDP use a client/server model for network communications. A server provides a service, and makes it available to multiple clients (and this can be a very large number of clients), by creating a network endpoint with a well-known port address, and waiting at that endpoint for a client request to use the service. In the case of TCP, these requests are connection requests to establish a virtual connection where the server and client can exchange data. In the case of UDP, it will simply receive packets from the client and process them on its behalf.

10.1.3 Well Known Server-side Port Addresses

Since clients initiate network connections, they must not only know the IP address of the server, but also the port number of the service on that server. To make this possible, services are assigned fixed port addresses, which are well known, so that a client always knows where to find a service on a server.

Below are some examples of network services, along with the port numbers they use:

Service	**Port**
Echo	7
FTP	21
Telnet	23
HTTP/World Wide Web	80
POP3 Email	110
IMAP Email	143
Streaming Media	537
Doom	666
MSN Instant Messenger	1863

Figure 10.2 illustrates an example of client/server network communications and shows the relation of IP and port addresses.

10.1.4 Client-side Port Addresses

Although the server-side port addresses must be known so that the client can find the service, the port address on the client side does not need to be generally known. This is because the server simply sends its responses back to the IP address/port that sent it the data. Therefore, in most cases a client endpoint is assigned a random unused port address by the operating system.

10.2 Network Programming Using Sockets

The goal of a good networking API is to hide the underlying details of the network and allow you to connect to a remote host and transfer data

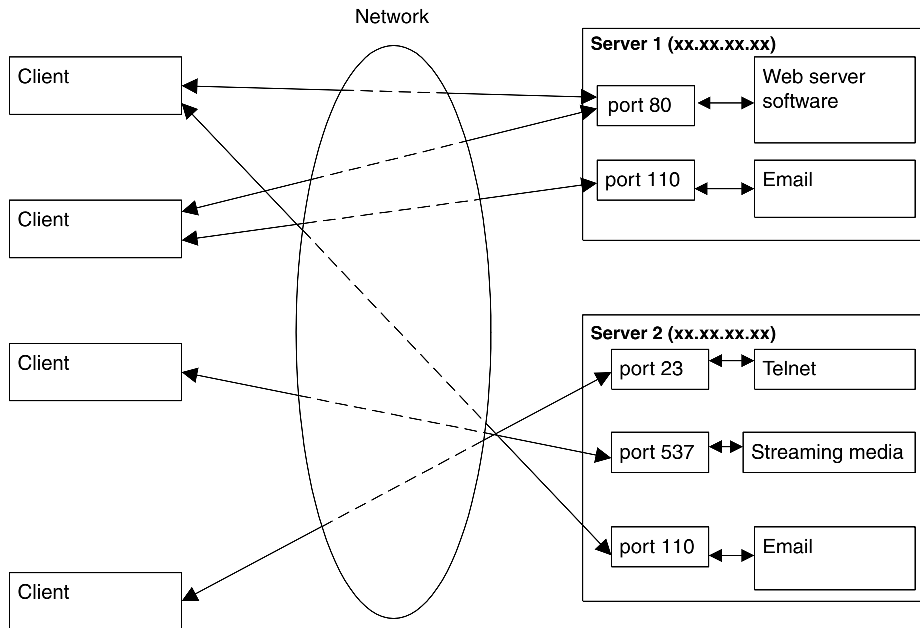

Figure 10.2 Client and Server Communications

easily. This section introduces the concepts of network sockets and how to program using them.

The socket programming interface originated from BSD Unix as a generic API for Inter-Process Communications. Since then it has become the standard for network programming. Virtually all operating systems that support TCP/IP have some sort of socket API, including Symbian OS.

A socket represents an endpoint of a communication path in a network. The endpoints are identified by a machine's IP address in combination with a port address and the communication between them represents a channel. So a session between two nodes on a network consists of a socket pair – one socket for each end of the communication pipe.

The socket API functions provide a generic interface for transferring data between the two endpoint sockets, independent (for the most part) of the underlying protocols. The programmer creates a socket, establishes a connection with the remote endpoint, and then transfers data over the socket using read and write commands.

10.2.1 Client/Server Socket Flow

The flow of creating and using a socket depends on whether you are on the client or the server side, and whether you are using TCP or UDP. Here are the basic steps for creating and using a socket on the client and server sides.

Client-side Code

1. Create the client-side socket and get a handle to it.
2. Connect the client socket to a destination endpoint.

   The socket must be connected to a destination endpoint for TCP but it is optional for UDP.

   A TCP connection is established between the endpoints by exchanging a set of packets to establish a virtual connection.

   Since UDP is connectionless, there is no handshaking between the endpoints to establish a connection. An explicit connection need not even be made since the destination endpoint can be defined, packet by packet, when data is sent (unlike for TCP, where all socket transfers occur between the endpoints of the established connection).
3. Transfer data between client and server.

   The client can now read and write data to the communication path using the socket.
4. Close the socket.

   When the communication session is completed, the socket is closed.

Note that, in the case of a client, the programmer need not explicitly assign a port number. The client address is automatically assigned a random port number during the connect phase (for TCP) or send (for UDP). This automatic assignment works since the specific client port number does not really matter; the server simply returns data to the source port it is communicating with.

Server-side Code

1. Create the server-side socket.
2. Bind an endpoint address to the server socket.

   Unlike in the case of a client, the programmer must make an explicit call to a `bind()` function to assign the endpoint address. The socket is assigned to the known port number for the particular service offered and, of course, the IP address must be one that is assigned to that machine.
3. Process client connections.

   For TCP, the server-side socket will get connection requests from clients. For each client connection request, a new socket handle is created to represent that particular client connection. The originally opened socket is still maintained though, to continue looking for new

client connections. Usually the server software will create a separate thread for each client connection it receives, and that thread uses the newly created connection socket handle to communicate with the client.

For UDP, unlike with TCP, there is no automatic connection capability for sockets, so UDP server programming is more involved. The server receives all raw UDP packets that clients send to that IP address/port. It is up to the server program to set up data structures and logic to filter the data and create connections, as well as doing anything else that is required for that service.

4. Transfer data with the client.

   The server transfers data to and from the client via the socket's send and receive functions.

5. Close the socket.

   When the connection is complete, the socket is closed.

10.2.2 BSD C Socket API

BSD Unix defines a set of C socket functions for creating and using sockets. Many operating systems use this API for network communication, and it has practically become a de facto standard for network programming.

Before looking at the Symbian OS socket API, let's look at a simple example using the BSD C API and then go through some of the functions. I will then refer to this when describing the Symbian OS C++ native socket API, which is similar to the BSD API in many ways. I will concentrate on client-side software only, since not many servers are implemented on a smartphone!

Note that Symbian OS also supports the BSD C-based socket API as part of its standard C library support. This can be convenient for porting network code from other operating systems to Symbian OS; however, in most cases you'll want to use the native Symbian C++ socket API described in the next section.

10.2.3 BSD C API Socket Client Example

Example 10.1 shows a simple program to fetch a web page using TCP on a server's port 80 (web server port) and to print the HTTP data to the screen.

Example 10.1. Getting a Web Page

```
int OutputWebPage( char *servName, char* urlDoc)
  {
  int sock;
```

```
  struct sockaddr_in server;
  struct hostent *hp;
  char buffer[1024];

  /* create socket */
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
    fprintf(stderr,"Error opening stream socket\n");
    return -1;
    }
/* connect socket using name specified by command line */

  server.sin_len = sizeof(server);
  server.sin_family = AF_INET;
  hp = gethostbyname(servName);
  if (hp == 0)
    {
    fprintf(stderr, "%s: unknown host\n", argv[1]);
    return(-1);
    }

  memcpy((char *)&server.sin_addr, (char *)hp->h_addr, hp->h_length);
  server.sin_port = hton(80); // set to well-known HTTP server port 80

  if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
    fprintf(stderr,"Error connecting stream socket");
    return(-1);
    }

  // send a HTTP GET to web server

  sprintf(buffer, "GET %s\n", urlDoc);
  nRet = send(sock, buffer, strlen(buffer), 0);
  if (nRet <= 0)
    {
    printf(stderr,"Error on send().\n");
    close(Socket);
    return -1;
    }

  // Receive the file contents and print to stdout

  while(1)
    {
    // Wait to receive, nRet = NumberOfBytesReceived
    nRet = recv(sock, buffer, sizeof(buffer), 0);
    if (nRet <= 0)
      break;

    puts(buffer);
    }
  close(sock);
  return 0;
  }
```

In this example, the function `OutputWebPage()` retrieves a web page from the server specified in the first argument, accessing the source with

the specified URL document name (blank just gets the home page). An example invocation is:

```
OutputWebPage(www.yahoo.com," ");
```

which retrieves and prints the HTML source of Yahoo's home page.

Creating the Socket

First, the client socket is created by the line:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
```

This creates a TCP socket to be used in communicating with the web server. The function `socket()` is prototyped as follows:

```
int socket(int domain, int type, int protocol)
```

where `domain` is `AF_INET` for TCP/IP, `type` is `SOCK_STREAM` for TCP or `SOCK_DATAGRAM` for UDP, and `protocol` indicates the specific protocol for the type. In this case, `protocol` can be set to zero, since TCP and UDP are the only protocols in `AF_INET` for those types.

`socket()` returns an integer handle for the socket which is used as a reference in subsequent socket function calls.

Converting Domain Names to IP Addresses

Next, the server name is converted to an IP address as follows:

```
server.sin_len = sizeof(server);
server.sin_family = AF_INET;
hp = gethostbyname(servName);
if (hp == 0)
  {
  fprintf(stderr, "%s: unknown host\n", argv[1]);
  return(-1);
  }

memcpy((char *)&server.sin_addr, (char *)hp->h_addr, hp->h_length);
server.sin_port = hton(80); // set to well-known HTTP server port 80
```

IP addresses are hard for people to remember, so ASCII names – known as domain names – are used instead. It's much easier to remember ***www.yahoo.com*** for example than it is to remember 216.109.118.77.

The BSD function `gethostbyname()` converts the server name to an IP address using what is known as the Domain Name System (DNS). DNS is a service used in TCP/IP networks which translates human-readable

domain names to IP addresses. DNS is a complex system due to the billions of IP addresses in use, which change everyday – but fortunately as a network programmer, it is easy to use.

The port address (`server.sin_port`) is set to 80, which is the port number for HTTP web pages.

Connecting the Socket

Now that the software has the IP address and port, it performs a TCP connection to the server's HTTP endpoint as follows:

```
if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0)
  {
  fprintf(stderr,"Error connecting stream socket");
  return (-1);
  }
```

The function `connect()` connects a socket whose handle is specified as the first argument, to a destination endpoint whose address is specified in the second argument (with the address structure's size specified in argument three). For TCP this consists of a packet exchange between the endpoints to establish a virtual connection.

In the case of UDP, this `connect()` just associates the socket with the destination address so that the programmer need not supply the address on each send. In this example, we are establishing a TCP connection, however.

The hardest part about using `connect()` is setting up the data structures to specify the endpoint address to connect to. I will not go into the data structure in detail here, but the form of address setup shown in the example is fairly typical.

Sending Data

Next, the HTTP GET request is sent to the server through the connected socket as follows:

```
sprintf(buffer, "GET %s\n", urlDoc);
nRet = send(socket, buffer, strlen(buffer), 0);
```

`send()` is used to send a buffer to the remote endpoint through the socket whose handle is passed as the first argument. It has the form

```
int send(int socket, const void *buff, size_t len, int flags);
```

`send()` returns the number of bytes sent. If it is a negative number, then an error occurred.

For UDP sockets, you can use `sendto()` for sending UDP packets. It is the same as `send()` except you specify the address of the endpoint to send the data to. It is defined as:

```
int sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
```

Receiving Data

When the server gets the HTTP GET request, it will start sending the web page to the client. The example retrieves this data and prints it to the screen, using the following lines of code:

```
while(1)
  {
  // Wait to receive, nRet = NumberOfBytesReceived
  nRet = recv(sock, buffer, sizeof(buffer), 0);
  if (nRet <= 0)
    break;

  puts(buffer);
  }
```

`recv()` is used for TCP sockets (or UDP sockets in which `connect()` was called) to receive data. The data is then placed in a supplied buffer. `recv()` returns the number of bytes received (if this is zero it means the connection was terminated, if negative an error occurred). It has the form:

```
int recv(int socket, void *buffer, size_t length, int flags);
```

In the case of UDP sockets, you usually use `recvfrom()`. It's the same as `recv()` except it returns the address of the endpoint that sent the packet. It has the form

```
int recvfrom(int s, void *buf, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen);
```

Cleaning Up the Connection

Once the web page retrieval is complete, the socket is cleaned up as follows:

```
close(sock);
```

`close(int socket)` closes the socket and shuts down the connection. A function called `shutdown()`, prototyped as `int shutdown(int s,`

`int how`), also exists to shut down a specific direction of the session. The parameter `how` has three possible values: `SHUT_RD` disallows further reception, `SHUT_WR` disallows further transmission and `SHUT_RDWR` disallows both reception and transmission.

10.3 Symbian OS Socket API

Symbian OS provides a C++ socket API which, as previously mentioned, is in many ways similar to the BSD C-based socket API. In addition to allowing TCP/IP communication, the Symbian OS socket API allows for other types of communication as well, including Bluetooth, USB and IR (although I will be covering only TCP/IP in this chapter). Underlying layers in the communication architecture handle these communication differences, and the socket API can be used in a transport-independent way.

10.3.1 Socket API Classes

First, let's briefly look at the key classes of the Symbian C++ Socket API that you'll use for TCP/IP communication:

- `RSocketServ` is the client-side class for the socket server and must be created and connected in order for your program to establish a session with the socket server. There is no equivalent to this class in the BSD socket API.
- `RSocket` represents a single socket in much the same way as does the handle returned by the `socket()` function in the BSD API. The other methods of `RSocket` correspond, for the most part, with the BSD network API functions.
- `RHostResolver` provides methods for both getting an IP address from a given domain name, and getting a domain name from a given IP address. DNS (Domain Name Service) is used in the case of TCP/IP. `RHostResolver` methods `GetByName()` and `GetByAddr()` provide the same functionality as the C API socket functions `gethostbyname()` and `gethostbyaddr()` respectively.

`RSocketServ` is a client session class to the socket server and is derived from `RSessionBase`. `RSocket` and `RHostResolver` are subsessions to an established `RSocketServ` session and are derived from `RSubSessionBase`.

10.3.2 HTTP Example Using Symbian OS Socket API

Now let's look at the `OutputWebPage()` program from Example 10.1, rewritten to use the Symbian OS socket API. The code is shown in Example 10.2.

Example 10.2. HTTP Example Using Symbian OS API

```
#include <in_sock.h>

TInt OutputWebPage(const TDesC& aServerName, const TDesC& aDoc)
  {
  RSocketServ sockSrv;
  sockSrv.Connect();

  RSocket sock;

  TInt res = sock.Open(sockSrv,KAfInet,KSockStream, KProtocolInetTcp);
  if (res != KErrNone)
    {
    sockSrv.Close();
    _LIT(KSockOpenFail,"Socket open failed");
    PrintError(KSockOpenFail);
    return res;
    }

  TNameEntry nameEntry;
  RHostResolver resolver;
  res = resolver.Open(sockSrv, KAfInet, KProtocolInetTcp);
  if (res != KErrNone)
    {
    sockSrv.Close();
    _LIT(KResvOpenFail,"host resolver open failed");
    PrintError(KResvOpenFail);
    return res;
    }

  TRequestStatus status;
  resolver.GetByName(aServerName, nameEntry, status);
  User::WaitForRequest(status);
  resolver.Close();

  if (status != KErrNone)
    {
    _LIT(KDnsFail,"DNS lookup failed");
    PrintError(KDnsFail);
    sockSrv.Close();
    return res;
    }

  TInetAddr destAddr;
  destAddr = nameEntry().iAddr;  // set address to DNS returned IP
                                 // address
  destAddr.SetPort(80); // Set to well-known HTTP port

  // Connect to the remote host
  sock.Connect(destAddr,status);
  User::WaitForRequest(status);
  if (status != KErrNone)
    {
    _LIT(KSocketConnectFail,"Failed to connect to server");
    PrintError(KSocketConnectFail);

    sockSrv.Close();
    return res;
```

```
  }

// Assemble HTTP GET command

TBuf8<300> getBuff;
getBuff.Copy(_L8("GET"));
getBuff.Append(aDoc);
getBuff.Append(_L("\xD\xA"));

// Send HTTP GET

sock.Send(getBuff,0,status);
User::WaitForRequest(status);

TBuf8<200> buff;
do
  {
  TSockXfrLength len;
  sock.RecvOneOrMore(buff,0,status,len);
  User::WaitForRequest(status);
  PrintOutput(buff);  // some generic 8-bit output-to screen or file
  } while (status == KErrNone);

sock.Close();
sockSrv.Close();
return (KErrNone);
}
```

The first thing to note is that many of the socket functions are asynchronous functions, and I use `User::WaitForRequest()` to wait for them to complete. This is for simplicity in showing the API; however, these functions are most effectively used in active objects (see Section 10.3.3).

Connecting to the Socket Server

Before using the socket API, you must first establish a session with the socket server. This is done by instantiating and connecting `RSocketServ` and calling its `Connect()` function, as shown below:

```
RSocketSrv sockSrv;
sockSrv.Connect();
```

Socket handling, like many other functions in Symbian OS, is best performed by means of a server process, along with client-side interface classes to access the server's services. The client-side classes for the socket server comprise the socket API.

The socket server handles all the details of creating sockets, connecting them to the client and server, and communicating through sockets in a transparent fashion. At this level of network programming, you don't need to know the details of the Symbian OS network communication

architecture but, if you are interested, they are covered in greater depth in Chapter 3 (see Section 3.11).

Creating the Socket

To create and open a socket, you instantiate an `RSocket` class and call its `Open()` method. This is done in the example as:

```
TInt res = sock.Open(sockSrv,KAfInet,KSockStream, KProtocolInetTcp);
```

`Open()` has the following form:

```
TInt Open(RSocketServ& aServ,TUint addrFamily, TUint socketType, TUint
        protocol).
```

The first argument is the connected `RSocketServ` class – this is needed because each `RSocket` is a subsession of the client socket server session established by `RSocketServ`.

The last three arguments are similar to those for the C `socket()` call. `KAfInet` specifies the TCP/IP version 4 protocol suite. `socketType` is set to `KSockStream` for TCP, or `KSockDatagram` for UDP.

`protocol` should be:

- `KProtocolInetTcp` for TCP
- `KProtocolInetUdp` for UDP

Unlike in the BSD socket API, `protocol` cannot be zero.

Setting the Destination Address

The class `TInetAddr` represents an endpoint's IP address and port, which can be set up using `SetAddress()` and `SetPort()`, respectively. For example, the following code sets up a `TInetAddr` to represent IP address 10.1.2.3, port 80

```
TInetAddr addr;
addr.SetAddress(INET_ADDR(10,1,2,3));
addr.SetPort(80);
```

`INET_ADDR` is a macro that writes the quad address into a 32-bit value that contains the four address bytes.

In our HTTP example, we are passed the web server name, so we need to use the `RHostResolver` class to contact DNS and look up the IP address associated with that name. To use the `RHostResolver`, you open it for the appropriate protocol (in this case TCP) and then call `RHostResolver`'s `GetByName()` method to look up the corresponding IP address.

In the example, this is accomplished by:

```
TNameEntry nameEntry;
RHostResolver resolver;
res = resolver.Open(sockSrv, KAfInet, KProtocolInetTcp);
if (res != KErrNone)
  {
  sockSrv.Close();
  _LIT(KResvOpenFail,"host resolver open failed");
  PrintError(KResvOpenFail);
  return res;
  }

TRequestStatus status;
resolver.GetByName(aServerName, nameEntry, status);
User::WaitForRequest(status);
resolver.Close();

if (status != KErrNone)
  {
  _LIT(KDnsFail,"DNS lookup failed");
  PrintError(KDnsFail);
  sockSrv.Close();
  return res;
  }
```

As previously mentioned (see Section 10.2.3, Converting Domain Names to IP Addresses), `GetByName()` converts the server name in `aServerName` to an IP address. The first argument of `GetByName()` is the server name you want translated. The results of the lookup are put in `nameEntry` upon return. When you assign `nameEntry().iAddr` to a `TInetAddr`, you'll set the IP address associated with the server name, then you just need to set the port (port 80 in our case for HTTP).

You can now set up the destination address that you will use to connect to the server:

```
TInetAddr destAddr;
destAddr = nameEntry().iAddr;  // set address to DNS returned IP address
destAddr.SetPort(80); // Set to well-known HTTP port
```

Connection to the Remote Server

The `RSocket Connect()` is used to establish a connection with the remote web server:

```
sock.Connect(destAddr,status);
User::WaitForRequest(status);
if (status != KErrNone)
  {
  _LIT(KSocketConnectFail,"Failed to connect to server");
  PrintError(KSocketConnectFail);
```

```
sockSrv.Close();
return res;
}
```

Sending a Packet

Once the socket is created and connected, packets can be sent (remember, a connection is not required in the case of UDP). `RSocket` provides the `Send()` method to send data through the socket. `Send()` has the following form:

```
void Send(const TDesC8& aBuffer, TUint aFlags, TRequestStatus& aStatus)
```

The buffer to send is specified as an 8-bit descriptor, and all bytes in the descriptor are sent through the socket.

The `HTTP GET` command in our example is sent to the web server as follows:

```
TBuf8<300> getBuff;
getBuff.Copy(_L8("GET "));
getBuff.Append(aDoc);
getBuff.Append(_L("\xD\xA"));

// Send HTTP GET

sock.Send(getBuff,0,status);
User::WaitForRequest(status);
```

`RSocket` also has a `SendTo()` method that is used to send UDP data. `This method` has the same form as `Send()` except an extra argument is added to specify the remote address that the packet should go to. Example 10.3 shows a way of sending data via UDP using `SendTo()`.

Example 10.3. Sending UDP Data

```
RSocket sock;
RSocketSrv sockSrv;

sockSrv.Connect();

sock.Open(socksvr,KafInet,KSockDatagram,0);

TInetAddr destAddr;
destAddr.SetAddr(INET_ADDR(10,1,2,3);
destAddr.SetPort(80);

TBuf8<300> buff;
buff.Copy (_L("Some stuff to send over UDP"));

sock.SendTo(buff,destAddr,0,iStatus);
User::WaitForRequest(iStatus);
```

You can use also use `Send()` to send UDP data, provided you first call `Connect()`, to connect to the remote address. `Connect()` is a convenience method for UDP, you call it so that the remote address need not be specified every time you send a UDP packet.

Receiving Packets

The web page is retrieved in our HTTP example as follows:

```
TBuf8<200> buff;
do
  {
  TSockXfrLength len;
  sock.RecvOneOrMore(buff,0,status,len);
  User::WaitForRequest(status);
  PrintOutput(buff);  // some generic 8-bit output-to screen or file

  } while (status == KErrNone);
```

The example uses the `RSocket::RecvOneOrMore()` method to retrieve the data sent from the server. `RecvOneOrMore()` has the following form:

```
void RecvOneOrMore(TDes8& aDesc, TUint flags, TRequestStatus& aStatus,
        TSockXfrLength& aLen)
```

`RecvOneOrMore()` acts like the BSD `recv()` socket call in that it completes when *any* data is available from the connection. The receive buffer (`aDesc`) is specified as an 8-bit descriptor and the received data is added to this buffer. The size of the descriptor is updated to match the number of bytes received (`aLen` also returns the number of bytes that were received).

Another `RSocket` method to receive data exists, which is called `Recv()`. You may be tempted to use `Recv()` instead of `RecvOneOrMore()` due to the name matching the BSD `recv()` call. However, there is a big difference between these receive calls when using TCP. Unlike `RecvOneOrMore()`, which completes when any amount of data is received, `Recv()` will not complete until the *entire* descriptor (specified by the maximum length of the receive descriptor) is filled with data. So, unless you know exactly how many bytes you will receive from the server, do not use `Recv()` for TCP.

`Recv()` is usually used for UDP. It behaves differently for UDP in that `Recv()` returns the data from a received UDP datagram even if it is below the maximum length of the descriptor (bear in mind that the `Recv()` method can only be used for UDP if `Connect()` was called first). So, `Recv()` acts the same for UDP as `RecvOneOrMore()` does for TCP.

`RSocket` also provides a method called `RecvFrom()` for receiving UDP data. This method is equivalent to the BSD `recvfrom()` function.

It receives a UDP packet and also the address of the host that sent it. `RecvFrom()` has the following form:

```
void RecvFrom(TDes8& aDesc, TSockAddr& anAddr, TUint flags,
        TRequestStatus& aStatus)
```

This function receives UDP data and supplies not only the data received but also the address of the endpoint that sent the data. `TSockAddr` is the base class for `TInetAddr`, so a `TInetAddr` can be passed here to obtain the sending node's address.

Closing the Socket and Socket Server

The example cleans up the socket and socket server connection with:

```
sock.Close();
sockSrv.Close();
```

10.3.3 Network Programming Using Active Objects

As you've seen, I used `User::WaitForRequest()` to wait for the asynchronous socket functions to complete in the previous section. However, the better way to call these socket functions is within active objects, letting the active object's `RunL()` method handle the socket function's completion. So the `Connect()` call, for example, would look something like:

```
void CMyActiveObject::DoNetworkStuff()
  {
  iSock.Connect(destAddr,iStatus);
  SetActive();
  }

CMyActiveObject::RunL()
  {
  // Invoked when the asynchronous function Connect completes.
  // iStatus contains the completion status
  }
```

With active objects, you can have your program continue to process other, nonnetwork, events while your network communication is taking place. For example, if you invoke networking functionality in a GUI application using an active object – say in response to some user selection – your GUI program can continue to process other user events while the network communication is in progress.

Normally you will want to have a sequence of networking calls performed in the background, started by a single active object method

(e.g. a connect, followed by a send, followed by one or more receives). To do this, you create a simple state machine in an active object, including a method that makes the first network call in the sequence (e.g., resolving the host name). Then your `RunL()` method would invoke the rest of the network calls, in response to completion events from previous network activity.

For the example of loading a web page, an active object can be declared as follows:

```
class CWebPage : public CActive
  {
public:
  static CWebPage* NewL();
  CWebPage();
  void ConstructL();

  ~CWebPage();

  TInt OutputWebPage(const TDesC& aServ, const TDesC& aDoc);

  enum TLoadStates
    {
    EResolvingName,
    EConnecting,
    ESending,
    EReceiving
    };

  void RunL();

  void DoCancel();

private:
  TInt iState;
  TBuf8<100> iUrlDoc;
  RSocketServ iSocketSrv;
  RSocket iSocket;
  TNameEntry iNameEntry;
  RHostResolver iResolver;
  TBuf8<20000> iWebBuff;
  TSockXfrLength iLen;

  };
```

The `OutputWebPage()` method would initialize the socket and start the first asynchronous function in the sequence (resolving the host name) as follows:

```
TInt CWebPage::OutputWebPage(const TDesC& aServerName, const TDesC& aDoc)
  {
  iSocketSrv.Connect();
  iUrlDoc.Copy(aDoc);

/* Resolve name, rest handled by RunL() */
```

```
  iState=EResolvingName;
  TInt res =
      iSocket.Open(iSocketSrv,KAfInet,KSockStream,KProtocolInetTcp);
  if (res != KErrNone)
    {
    iSocketSrv.Close();
    _LIT(KSockOpenFail,"Socket open failed");
    PrintError(KSockOpenFail);
    return res;
    }

      res = iResolver.Open(iSocketSrv, KAfInet, KProtocolInetTcp);
  if (res != KErrNone)
    {
    iSocketSrv.Close();
    _LIT(KResvOpenFail,"host resolver open failed");
    PrintError(KResvOpenFail);
    return res;
    }

  TRequestStatus status;
  iResolver.GetByName(aServerName, iNameEntry, status);
  SetActive();

// first asyncronous function started, RunL() takes over from here.
  }
```

The active object `RunL()` can then be implemented to process the event and start the next socket call in the sequence, based on the state value in `iState`, as shown in Example 10.4.

Example 10.4. `RunL()` State Machine

```
void CWebPageActiveObject::RunL()
  {
  if ( (iStatus != KErrNone) && (iStatus != KErrEof) )
    {
    // error happened, abort sequence, no further RunL()s will be invoked
    _LIT(KWebPageFail,"error loading web page");
    PrintError(KWebPageFail);
    iSocketServ.Close();
    iSocket.Close();
    iResolver.Close();
    } else
    {
    // walk through state machine to load the web page.
    switch(iState)
      {
      case EResolvingName:
        {
        TInetAddr destAddr;
        destAddr=iNameEntry().iAddr;
        destAddr.SetPort(80);

        // Connect to the remote host
```

```
        iState=EConnecting;
        iSocket.Connect(destAddr,iStatus);
        SetActive();
        break;
        }
      case EConnecting:
        {
        // Send GET packet
        TBuf8<300> getBuff;
        getBuff.Copy(_L8("GET "));
        getBuff.Append(iUrlDoc);
        getBuff.Append(_L("\xD\xA"));

        iState=ESending;
        iSocket.Send(getBuff,0,iStatus);
        SetActive();
        break;
        }
      case ESending:
        {
        //  Start receiving web page now

        iState=EReceiving;

        iSocket.RecvOneOrMore(iWebBuff,0,iStatus,iLen);
        SetActive();
        break;
        }
      case EReceiving:
        if (iStatus != KErrEof)
          {
          /* Web data received */
          WriteTextOutput(iWebBuff); // write data to
          // console or file, whatever.
          iSocket.RecvOneOrMore(iWebBuff,0,iStatus,iLen);
          SetActive();
          } else
          {
          // End of file, page load complete
          iSocket.Close();
          iResolver.Close();
          iSocketServ.Close();
          }
          break;
        }
      }
  }
```

`iState` is a `TInt` that determines what the active object's `RunL()` should do next in response to a completion event. Our `RunL()` is first invoked when the `RHostResolver` `GetByName()` method completes, at which time `RunL()` calls the next call in the sequence – `Connect()` – based on the `iState` value. When `Connect()` completes, `RunL()` calls the socket's `Send()` method to send the `HTTP GET` command. When `Send()` completes, `RunL()` invokes the socket `RecvOneOrMore()` method, at which time `RunL()` is invoked on each `Recv()` completion to reissue the

RecvOneOrMore() and print the retrieved web output. When iStatus returns KErrEof (the code assumes no other call besides RecvOneOrMore() will return this status) the server has finished, so RunL() cleans up and no further commands are reissued.

10.4 Example: Retrieving Weather Information

This section presents an example program using the Symbian OS socket API to retrieve the current temperature from the weather server *wunderground.com*. The example consists of an active object which steps through the various socket calls needed to collect the data from the server. The active object provides a method called GetTemperatureL(const TDesC& aCity) – where aCity is a descriptor that contains the airport code for the US city whose temperature you want. When this function is called, the data is collected from the weather server and parsed. Then an info message is displayed on the screen that shows the information in the form of Temperature=XX where XX is the last reported temperature for the specified city.

10.4.1 wunderground.com

www.wunderground.com is a web site that provides weather information. In addition to its HTTP web site, wunderground also provides a telnet interface (through rainmaker.wunderground.com, port 3000). Using the telnet interface, you can enter a three-letter US City code, and retrieve the current and forecast weather conditions for that city in a simple text format. Since this text is easier to parse than HTML, the example here uses the telnet interface.

First, let's run the telnet manually, so we can see what this server outputs. Figure 10.3 shows the output when the following is typed at a command prompt:

telnet rainmaker.wunderground.com 3000

and then AUS (for Austin, TX) is typed in answer to the city code prompt.

Notice that, in Figure 10.3, the current temperature follows the end of the line filled with '=' characters. I will use this fact to retrieve the temperature in the example code.

Example 10.5 shows the active object class definition for the example.

Example 10.5. CWeatherInfo Class Definition

```
#include <in_sock.h> // needed to use socket API

class CWeatherInfo : public CActive
```

Figure 10.3 Output of *wunderground* telnet session

```
  {
public:
  static CWeatherInfo* NewL();
  CWeatherInfo();
  ~CWeatherInfo();

  void GetTemperatureL(const TDesC& aCity);
  void RunL();

  void DoCancel();

  enum TLoadStates
    {
    EResolvingName,
    EConnecting,
    ESending,
    EReceiving
    };

private:
  TInt iCommState;
  RSocketServ iSocketSrv;
  RSocket iSocket;
  TNameEntry iNameEntry;

  RHostResolver iResolver;

  TBuf8<20000> iNetBuff;
  TSockXfrLength iLen;

  TBuf<16> iCityCode;
  };
```

Example 10.6 shows the code for the `NewL()`, constructor, destructor and `DoCancel()` functions.

Example 10.6. Construction and destruction and cancel functions

```
CWeatherInfo* CWeatherInfo::NewL()
  {
  CWeatherInfo* self = new(ELeave) CWeatherInfo;
  CActiveScheduler::Add(self);
  return self;
  }

CWeatherInfo::CWeatherInfo(): CActive(CActive::EPriorityStandard)
  {
  }

CWeatherInfo::~CWeatherInfo()
  {
  // Make sure we're cancelled
  Cancel();
  }

void CWeatherInfo::DoCancel()
  {
  iSocket.CancelAll();
  }
```

`NewL()` is a static function that creates the `CWeatherInfo` active object, adds it to the current active scheduler, and returns a pointer to the created instance. The `CWeatherInfo` constructor passes the active object priority to the base constructor. The destructor calls `DoCancel()`, which cancels any asynchronous call in progress so that the active object can safely be destroyed.

Example 10.7 shows the implementation of `GetTemperatureL()`. You call this function to start the process of collecting the weather information from which the temperature will be extracted and displayed.

Example 10.7. `GetTemperatureL()` Method

```
void CWeatherInfo::GetTemperatureL(const TDesC& aCity)
  {
// if we are already in the middle of getting the
// temperature, then return.
  if (IsActive())
    return;

  iSocketSrv.Connect();
  iCityCode.Copy(aCity);

  TInt res =
  iSocket.Open(iSocketSrv,KAfInet,KSockStream,KProtocolInetTcp);
  User::LeaveIfError(res);

/* Resolve name, rest handled by RunL() */
```

```
iCommState=EResolvingName;
res = iResolver.Open(iSocketSrv, KAfInet, KProtocolInetTcp);
User::LeaveIfError(res);

_LIT(KWeatherServerName,"rainmaker.wunderground.com");
iResolver.GetByName(KWeatherServerName, iNameEntry, iStatus);
SetActive();
}
```

GetTemperatureL() first checks if the active object is currently active and exits if it is. This prevents a panic E32USER-CBase 42 (setting an active object to the active state while it is already active) which could happen if GetTemperatureL() was called while a temperature retrieval is already in progress. If the active object is not active, GetTemperatureL() copies the city code, which has been passed to it, to a member variable for later sending to the server, and then opens a TCP socket. It creates an RHostResolver so that it can do the first step in the sequence – getting the weather server's IP address. The active object maintains the sequence state in iCommState, and the first state is EResolvingName. The RHostResolver's GetByName() is called to begin the DNS name lookup, and then SetActive() is called. Recall that SetActive() is a method that returns immediately, but sets a flag indicating to the active scheduler that this active object is now expecting a asynchronous function event. The result is that the active object's RunL() function will be called when the GetByName() function completes (either by getting the name, or following an error).

The RHostResolver GetByName() lookup of rainmaker.wunderground.com was done as an illustration. However, you could have looked up the IP address for rainmaker.wunderground.com (it's 66.28.69.161) manually up-front using a program like ping from a PC command line – then hardcoded this address and passed it to the socket Connect() method, thus skipping the DNS look up and making the program faster.

The call to GetTemperatureL() returns after initiating GetByName(). The remaining sequence of socket calls used to retrieve the weather information is performed in response to asynchronous events handled in the active object's RunL() – the first event being the GetByName() completion event.

Example 10.8 shows the active object's RunL() function.

Example 10.8. Example's RunL() Function

```
void CWeatherInfo::RunL()
  {
  if  (iStatus != KErrNone)
```

```
    {
    iSocket.Close();
    iSocketSrv.Close();
    _LIT(KErrorMsg,"Error getting temperature");
    User::InfoPrint(KErrorMsg);
    }
    else
    {
    switch(iCommState)
      {
      case  EResolvingName:
        {
        TInetAddr destAddr;
        destAddr=iNameEntry().iAddr;
        destAddr.SetPort(3000);

        // Connect to the remote host
        iCommState=EConnecting;

        iSocket.Connect(destAddr,iStatus);
        SetActive();
        break;
        }
      case EConnecting:
        {
        TBuf8<300> getBuff;
        getBuff.Copy(_L("\xD\xA"));
        getBuff.Append(iCityCode);
        getBuff.Append(_L("\xD\xA"));

        iCommState=ESending;
        iSocket.Send(getBuff,0,iStatus);
        SetActive();
        break;
        }
      case ESending:
        {
        // Start receiving

        iCommState=EReceiving;

        iSocket.RecvOneOrMore(iNetBuff,0,iStatus,iLen);
        SetActive();
        break;
        }
      case EReceiving:
        {
/*--------------------------------------------------------------------
   The rainmaker.wunderground.com line with the temperature starts
   after a line filled with '=' s.
--------------------------------------------------------------------*/
        TInt pos = iNetBuff.FindF(_L8("=\xA"));
        TBuf<100> temp;
        if (pos != KErrNotFound)
          {
          temp.Copy(iNetBuff.Mid(pos+2,10));
          temp.Trim();
          temp.Insert(0,_L("Temperature = "));
```

```
          User::InfoPrint(temp);
          iSocket.Close();
          iSocketSrv.Close();
          } else
          {
          iSocket.RecvOneOrMore(iNetBuff,0,iStatus,iLen);
          SetActive();
          }
        break;
        }
      }
    }
  }
```

`RunL()` is invoked on completion of each socket call, and the sequence of network operations needed to collect the temperature from *wunderground.com* is accomplished through a state machine, illustrated in Figure 10.4.

The first event processed by `RunL()` is the resolution of the host name to an IP address. In response to this, the `Connect()` is performed on the socket to hook it to the rainmaker.wunderground.com server at port address 3000 and the state changes to `EConnecting`.

Upon the `Connect()` completion, `RunL()` is called again, invoking the `RSocket Send()` method to send the city code. The state is changed to `ESending`.

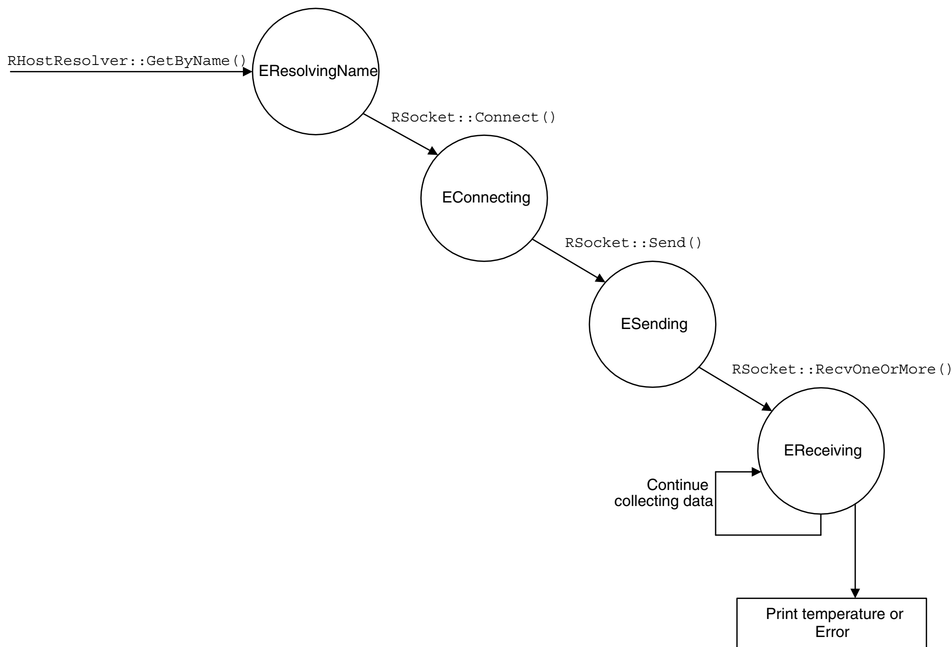

Figure 10.4 Weather Example State Machine

Once the `Send()` completes, the `RunL()` calls `RecvOneOrMore()` to start getting the weather data. The state changes to `EReceiving` and remains in this state as long as the data keeps coming from the server (although, in this case, you will probably get all the data on the first call). `RunL()` looks for the end of the lines of '=' characters (see Figure 10.3) which immediately precede the temperature. Once the temperature is found, it is printed as a message and the communication session is closed.

10.4.2 Adding this Code to `SimpleEx`

Chapter 2 contains an example called `SimpleEx` (see Section 2.3.2). You can add the weather collection code to this program by doing the following:

1. Add a new menu item called 'Get temperature' to the resource file, changing the menu definition as follows:

```
RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      },
    MENU_ITEM
      {
      command = ESimpleExTemperatureCommand;
      txt = "Get temperature";
      }
    };
  }
```

2. Add the `ESimpleExTemperatureCommand` to the command `enum` in the program's `simpleEx.hrh` file.

3. Add a private member variable, `CWeatherInfo *iWeather`, to the `CSimpleExAppUI` class.

4. Include the previously listed example source (either in separate files or in the existing source and include files).

5. In the `CSimpleExAppUi::ConstructL()` function, add the statement:

   `iWeather = CWeatherInfo::NewL();` to create the example's active object.

6. In the `CSimpleExAppUi` destructor add: `delete iWeather;`.

7. In the `CSimpleExAppUi::CommandHandlerL()`, add a case for the command `ESimpleExTemperatureCommand`. Mine looks as follows:

```
case ESimpleExTemperatureCommand:

 /* Display the temperature in Austin, TX */

  iWeather->GetTemperatureL(_L("AUS"));
    break;
```

8. In your mmp file, add `insock.lib` and `esock.lib` to the `LIBRARY` line, to include the socket calls.

Once you have updated the program (which may be the UIQ, Series 60 or Series 80 version), build and run the updated `SimpleEx` program on the phone (Chapter 2 shows how to use `makesis` to create a `sis` file to enable you to install and run `SimpleEx` on the phone (see Section 2.4)). When you select the `Get Temperature` menu option, a network connection will be established. Once the communication is complete, the temperature of the city you requested is displayed on the screen.

Since the example uses an active object, you'll note that the GUI is still responsive while the communication is taking place. For example, the `Start` menu item can be selected during the network communication and it will still display its dialog boxes. So, in effect, the entire network sequence to collect the weather information is running in the background.

10.5 Making a Network Connection

Up to this point, I have not discussed the network connection itself on a Symbian OS smartphone – how it is selected and established. This section gives a brief overview of this functionality.

Symbian OS smartphones have multiple ways of reaching the Internet, depending on the particular phone you have, the wireless data services available to you, and which services are included in your service plan. The different means of communication for Symbian OS smartphones, including GPRS, EDGE and CSD (as well as WiFi for some newer models) are discussed in Chapter 1 (see Section 1.5).

10.5.1 Internet Access Points

On a Symbian OS smartphone, network connections are represented by *Internet Access Points* (IAPs). IAPs can be created from the phone's control panel, usually during initial setup. The information in an IAP

includes the physical connection type (such as GPRS or CSD), and the specific attributes applicable to the selected connection type (such as the phone number; user's identifier and password for a dial-up server, for CSD-type connections; or APN for a GPRS connection). You assign a name to the IAP when creating it, and that name is used when establishing the connection.

For example, you can create an IAP called `T-Mobile GPRS` to use the APN provided by T-Mobile (e.g. *Internet2.voicestream.com*). Then, when you are prompted for a connection by the phone, you select `T-Mobile GPRS` to use the GPRS connection.

Note that IAP setup varies from phone to phone, and also depends on the particular service you intend to access. In many cases the IAP creation is done automatically for you (e.g. through a service SMS message).

10.5.2 Establishing a Connection for a Program

The example Symbian OS socket code, presented in Section 10.3, created and used a socket as if the smartphone connection was already established. In fact, this is often done in Symbian OS and is known as an *implicit connection*. When a socket operation is performed that requires data to be sent on the network – for example an `RSocket Connect()` on a TCP socket, or `sendto()` for UDP – the operating system establishes the connection for you, if it is not already established. This involves handling the details of starting up a connection on the phone (i.e. an IAP), or finding one that is already connected, and then the socket will communicate using that connection.

Alternatively, your program can connect to a specific IAP for the socket to use – this is known as an *explicit connection*. The APIs to start a specific IAP differ from phone to phone. This will not be discussed here, but you need to be aware that it can be done. As an example, the API class for Symbian 7.0s (used in the Nokia 6600 and Nokia 9500) to explicitly start a specific IAP is `RConnection`. `RGenericAgent` is used for earlier versions of Symbian OS. Refer to the SDK for more details on these classes.

An example of a program that uses explicit connections is email. The user can specify a particular IAP to use for an email account, through the email application settings. When sending or retrieving email for that account, an explicit connection to the configured IAP is performed.

10.5.3 Automatic Connection Selection

An implicit network connection leaves it to the operating system to select the best connection to use. As mentioned in the last section, an implicit connection occurs if you start using a socket when there is, as yet, no established connection.

How does the operating system determine the best connection? The smartphone's control panel provides a way for the user not only to create, but also to prioritize the phone's IAPs. When an implicit connection is requested, the operating system tries the IAPs in the specified order until a successful connection is established. In other words, the highest priority IAP is tried first, and if that connection fails (e.g. you are out of range of GPRS) it goes to the next priority IAP, and so on.

The exception to this automatic selection is when the connection prompting option (selectable under connection control panel settings) is enabled. In this case, the operating system will prompt the user to select an IAP to use for the connection instead of automatically selecting one based on its priority.

For example, when you run the weather example program with connection prompting on, as soon as the user selects the `Get Temperature` menu option, the operating system prompts the user to select an IAP (caused by the first network operation that sends data, in this case `GetByName()`). The prompting occurs only if a connection is not yet established at the time the selection is performed.

For the most part, if an implicit connection is requested and a connection is already established, then the socket will use that existing connection. However, operating system versions from Symbian OS v7.0s onwards support multi-homing so, for these operating systems, each program can have its own connection to the network. For example, the browser could be using a WiFi IAP, while the weather example you just wrote is using GPRS.

The methods of creating, editing and prioritizing IAPs, as well as specifying if connection prompting is on or off, differ between phones. The concepts are the same, however.

11

GUI Application Programming

Smartphones are capable of running complex and powerful applications, but due to the smartphone's small size, it can be challenging to make these applications intuitive and easy to use. Symbian OS addresses this well by providing a full-featured graphical user interface (GUI) environment especially suited for the hardware characteristics of smartphone devices.

To take full advantage of the GUI environment, your application's interface should be consistent with those of other applications, so that the smartphone's user does not get confused (follow the GUI guidelines for the specific device type) – yet it should also creatively provide good presentation and user control of the features unique to your application. Your program's GUI is its front to the world, and creating a good one can greatly enhance, if not totally determine, the success of your smartphone application.

This chapter covers Symbian OS GUI programming. First, there is a discussion of the different Symbian GUIs specific to smartphone device types – UIQ, Series 60 and Series 80. Then I describe the creation of GUI applications – how to use the GUI framework to write your application, and how to use the various components of the GUI such as dialog boxes, menus, buttons, list boxes, etc. I also cover other GUI application aspects, including icons and internationalization – everything needed to create a complete and ready to use application.

11.1 Symbian OS User Interfaces

First, let's take a quick tour of the UIQ, Series 60 and Series 80 user interfaces from the user's point of view. These interfaces differ in order to provide the best user experience for specific smartphone form factors; however, programming for these different GUI platforms is not all that different, as you will see in the following sections.

11.1.1 UIQ Phones

UIQ was designed for smartphone devices with the following characteristics:

- Quarter VGA (240x320) portrait screen
- Touch screen
- No hardware keyboard
- Hardware keys for up, down and confirm

Examples of UIQ phones include the Sony Ericsson P910 and Motorola A920.

Figure 11.1 shows the calendar UIQ application as an example.

The screen is divided into the following areas:

- Application selector

  This is the topmost row of tabs, used for easy access to commonly used applications (these can be customized by the user). The rightmost tab always represents the application launcher screen, which displays a desktop with icons for all the applications on the smartphone, so that any application can be launched.

- Menu bar

  The next row, below the application selector, is the menu bar. Each application has its own menu bar and the menu bar is always shown (unlike in Series 60 and Series 80). UIQ menus behave a lot like menus on a Microsoft Windows PC, in that the top-level menu items

Figure 11.1 UIQ Application Screen

are displayed in the bar and, once one is selected, submenu items can be selected.

- View selector

  UIQ applications typically have more than one application view. The view selector is a pull-down list box at the far right end of the menu bar. Views are different screen representations of an application's data. For example, an application that involves a grouping of items could have a list view and a detailed view. Alternatively, the views could represent filters that define what application data is displayed in the view. For example, in the UIQ Agenda application you can have views that display only business or personal calendar entries – the default view being to display all calendar items.

- Application area

  The large central area, below the menu bar, is the application area. This is where the application-specific data and controls that make up an application view are displayed.

- Button bar

  Below the application area is an optional button bar. The button bar is specific to the application and an application may or may not have one. The button bar often contains convenient global controls that select specific views, or perform some sort of global operation specific to that application.

- Status bar

  The status bar appears at the bottom of the screen and contains various phone-specific statuses such as the battery power indication and connection status. This is customizable only by the phone manufacturer.

User Input

UIQ phones have a touch screen and a keypad, but typically have no QWERTY hardware keyboard (although the Sony Ericsson P910 UIQ phone has a very small one on the inside of the flip). GUI controls are selected on UIQ via the touch screen. ASCII input is accomplished either via a virtual keyboard or by use of handwriting recognition. To use the virtual keyboard, you click the virtual keyboard icon in the status bar and a keyboard is displayed on the screen. You can then type text with the touch screen, by selecting the appropriate keys. To use handwriting recognition, draw the characters you want to enter on the touch screen, and the software interprets the characters, and inputs them.

UIQ Paper Metaphor

When you start an application, it remains running and persistent. Applications normally do not provide an exit option; you just switch away

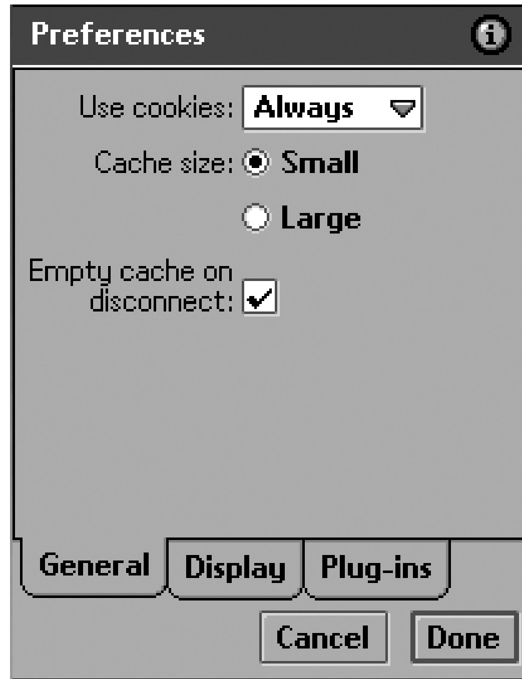

Figure 11.2 UIQ Dialog

from them when you want to use something else. Any data that you have entered into the application remains intact and saved – when you go back to your application, it looks as it did when you switched away. UIQ exposes none of the content of a file system to the user and does all storage of data behind the scenes.

UIQ Dialogs

As with all of the Symbian user interface platforms, much of a GUI application's functionality is performed in dialog boxes. Dialogs display a title bar, a set of controls and a row of one or more buttons, which are used to save, cancel, or perform some other application-specific operation. One of the buttons can be set as the default, so that the phone's Confirm hardware key will activate it. UIQ supports simple, single-page dialogs and multipage dialogs, in which each page can be selected via a row of tabs. Figure 11.2 shows an example of a multipage dialog (this is the browser's Preferences dialog).

11.1.2 Series 60 Phones

The Series 60 user interface is made for smaller devices, with smaller screens (176 × 208) and no touch screen. Unlike UIQ, the Series 60 UI is specifically designed and optimized for one-handed operation.

Figure 11.3 Series 60 Application Screen

Figure 11.3 shows an example Series 60 application screen.

The entire screen is devoted to displaying information related to the current application, and is divided into the following areas:

- Status pane

  The status pane is at the top of the screen and displays the current application's title, and system status information, such as the signal strength shown in Figure 11.3. As with UIQ, the layout and system status content of this pane is determined by the phone manufacturer.

- Main pane

  The main pane is situated below the status pane and occupies the bulk of the screen. As with UIQ, it is dedicated to displaying the data and controls that make up an application's view.

- Control pane

  The control pane occupies the bottom area of the screen. It contains two softkey tabs, which are selected by depressing the hardware buttons aligned below them. Unlike UIQ, an application menu bar is not displayed on the screen. Instead, the control pane displays an Options softkey which, when selected, presents the application's menu. Series 60 menus can be multilayered, with each menu item optionally expanding to display a sub-menu.

User Input

Instead of selecting controls by a touch screen, a thumb device is used to navigate between screen fields. Text input is accomplished through the phone's numeric pad, which, if you are not familiar with it, takes some getting used to – but once learned is efficient.

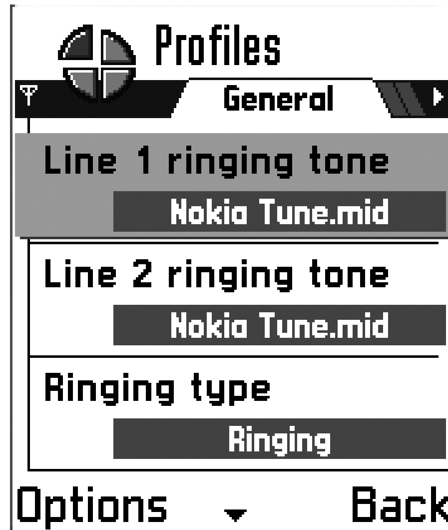

Figure 11.4 Series 60 Dialog

Series 60 Dialogs

Like in UIQ, Series 60 dialogs are popup windows that overlay the application's view. Dialogs usually consist of a set of GUI controls, but they can also be a simple message pop-up. Within a dialog, individual controls are selected by using the hardware thumb pointer, as opposed to tapping on a touch screen. Also, Series 60 dialogs do not contain a row of GUI exit buttons, but use the two softkeys instead. Figure 11.4 shows an example of a Series 60 dialog.

11.1.3 Series 80 Phones

Series 80 phones – also known as communicators – fold out to expose a full keyboard and a large landscape display (640 × 200). As a result, these phones are the easiest smartphones to control, at the expense of size (although they are getting smaller) and cost. Examples of Series 80 phones are the Nokia 9200 series and the newer Nokia 9300 and 9500 communicators.

Series 80 phones do not have touch screens, and controls are traversed by a rocker key on the keyboard.

Figure 11.5 shows an example application screen on a Series 80 phone.

The screen is divided into the following areas:

- Indicator area

  The indicator area is on the left of the screen and is reserved for displaying information about the currently-active application. Depending on the application, it may show the application's name and icon, the date and time, and/or other status information. This area is normally

Figure 11.5 Series 80 Application Screen

92 pixels wide, but an application may reduce it to a width of 32 pixels if it needs additional space to display its data.

- Command button array

  This is situated at the right-hand-side of the screen, and is used to label the four adjacent hardware keys. An application will label these keys as appropriate, depending on the context.

- Application area

  The application area occupies the remaining central area of the screen and is reserved for the application's use. The application is entirely responsible for the layout of this area.

The menu bar is not displayed on the screen unless it's invoked by depressing the menu key on the keyboard. The keyboard has a row of keys at the top to directly invoke popular applications. The rightmost of these keys can be customized by the user to go to an application of their choosing.

Series 80 dialogs, like those in UIQ and Series 60, are pop-up windows that overlay the main application screen. They usually consist of one or more rows of controls that are navigated by means of the rocker key. Dialogs can be either a single page or multipage, where the page is selected by labeled tabs located on the upper portion of the dialog – or they can be simple pop-ups that display a message or confirmation. As in Series 60, dialogs do not include any exit buttons; Series 80 dialogs use the command button array for this purpose, together with the Escape key on the keyboard, which is used to cancel the dialog. Figure 11.6 shows an example of a Series 80 dialog.

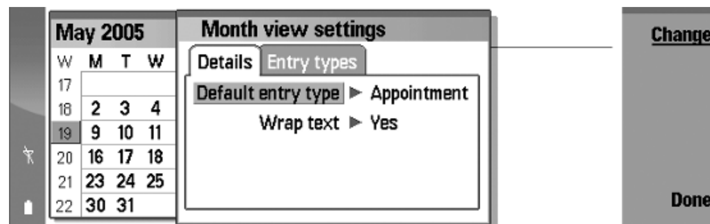

Figure 11.6 Series 80 Dialog

11.2 Anatomy of a GUI Application

In Chapter 2, I presented a basic example GUI application, and provided steps to build it and load it to the phone (see Section 2.3). To recap, a basic GUI application project consists of the following parts:

- The `mmp` and `bld.inf` files needed to build the application (or appropriate IDE project file if using an IDE instead of the command line)
- A resource file (`rss` file) to define the various GUI components, dialogs and text strings your application uses
- The application source code
- A `pkg` file to build a `sis` file that can be stored on the phone.

In addition, an application will usually also have:

- A set of bitmaps to define the application's icon (at various sizes, as defined by the UI platform used)

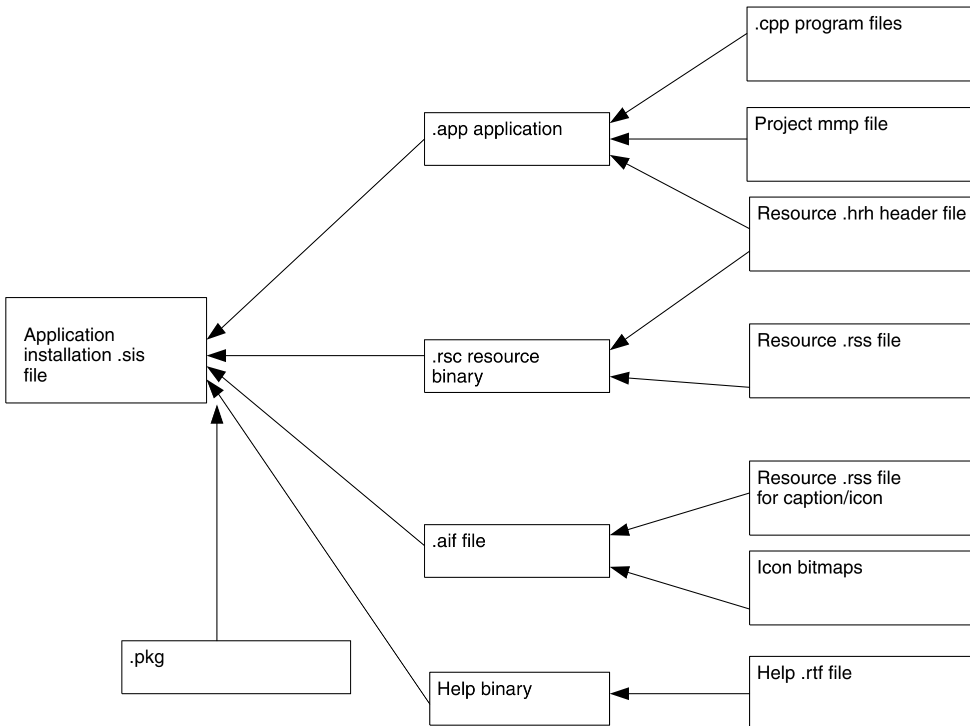

Figure 11.7 Application Project Elements and Build Flow

- An application information resource file that defines the application's caption, number of icons and other information (this, together with the icon bitmaps, is used to generate the `aif` file for download to the phone along with the application)
- An online help file in Rich Text Format (RTF) to define the on-line help available to the application user (generated into a `hlp` file using the help compiler).

Figure 11.7 shows how all of these elements come together to build an application.

11.3 Application Classes

In Chapter 2, I also gave a basic tour of the key functions and classes of a GUI application, using the `SimpleEx` example (see Section 2.3). Let's look at this again, but go into more detail, as needed.

11.3.1 `E32Dll()` and `NewApplication()`

All Symbian OS GUI applications have to implement the two functions: `E32Dll()` and `NewApplication()`.

`E32Dll()` is the entry function required by all DLLs in Symbian OS and is always implemented as follows:

```
GLDEF_C TInt E32Dll(TDllReason )
  {
  return KErrNone;
  }
```

`NewApplication()` is the first exported function of the application DLL. When a GUI application is started, `apprun.exe` runs, which in turn brings up the application architecture (APPARC) component (i.e. the GUI application framework) of Symbian OS. This framework, in turn, loads your application DLL and calls its `NewApplication()` function to create, and return a pointer to, your application's instance of its Application class.

Here is the implementation of the `NewApplication()` function from Chapter 2's `SimpleEx` example:

```
EXPORT_C CApaApplication* NewApplication()
  {
  return (new CSimpleExApplication);
  }
```

The next section describes the basic classes used to implement your application including this Application class.

11.3.2 Examining the GUI Application Classes

Here are the classes each application defines as a minimum:

1. Application class

   As mentioned above, this is the first thing the GUI framework creates. This class is used to identify the application (by returning the application's UID) and to instantiate, and return a pointer to, your application's document class.

2. Document class

   This class represents the application's data model. It is also responsible for instantiating, and returning a pointer to, an instance of your application's UI class.

3. UI class

   This class handles all UI events, and is responsible for creating the application's default view.

4. View class

   The view class implements the application's screen display, including drawing the window and the creation of the initial screen controls. An application can have multiple view classes.

These classes contain the basic functionality of your application, as well as providing the interface needed for the GUI framework to start, and drive, your application. The nice thing about the GUI framework in Symbian OS is that you implement these application classes in the same way across all the different Symbian OS GUI platforms. The only difference is that (with the exception of the view class) they are derived from different platform-specific classes (see Table 11.1).

Table 11.1 Platform-specific classes inherited from

Class	UIQ	Series 60	Series 80
Application	`CQikApplication`	`CAknApplication`	`CEikApplication`
Document	`CQikDocument`	`CAknDocument`	`CEikDocument`
Application UI	`CQikAppUi`	`CAknAppUi`	`CEikAppUi`
View	`CCoeControl`	`CCoeControl`	`CCoeControl`

Note that, in the simplest case, application views are pure controls, which inherit directly from `CCoeControl`. However, there are more powerful view classes (known simply as 'views') that allow you to switch easily between multiple views. These are platform-specific and will be discussed in more detail later (see Section 11.8).

The Application Class

The declaration for the UIQ version of `SimpleEx`'s application class is as follows:

```
class CSimpleExApplication : public CQikApplication
  {
private:
  CApaDocument* CreateDocumentL();
  TUid AppDllUid() const;
  };
```

Series 60 and 80 declare it in the same way, except that the class is derived from `CAknApplication` and `CEikApplication` respectively. Here is the implementation of this class for `SimpleEx`:

```
const TUid KUidSimpleExApp = {0x10005B94};
    TUid CSimpleExApplication::AppDllUid() const
  {
  return KUidSimpleExApp;
  }

CApaDocument* CSimpleExApplication::CreateDocumentL()
  {
  return new(ELeave) CSimpleExDocument(*this);
  }
```

After the framework has obtained a pointer to the application object, via the call to `NewApplication()`, it calls that object's `AppDllUid()` method to get the application's UID. It does this as a sanity check to make sure that the UID matches the one it expects. Thus, you must ensure that `AppDllUid()` returns the same UID as the one that you specify in the application's `mmp` file. Once the framework has verified the UID, it calls the application object's `CreateDocumentL()` method, which returns a pointer to your application's document class instance.

The Document Class

The document class has two purposes, the first of which is to represent your application's persistent data. The other is to create the Application UI instance in which this data (if any) can be manipulated. For applications that have no persistent data, and therefore are not file based, the document class simply implements the `CreateAppUiL()` function to return the application's UI object, as the following code from `SimpleEx` shows:

```
class CSimpleExDocument : public CQikDocument
  {
public:
  CSimpleExDocument(CEikApplication& aApp) : CQikDocument(aApp) { };
```

```
private:
  CEikAppUi* CreateAppUiL();
  };

CEikAppUi* CSimpleExDocument::CreateAppUiL()
  {
// Create the application user interface, and return a pointer to it;
// the framework takes ownership of this object

  return new (ELeave) CSimpleExAppUi;
  }
```

An application's data is said to be persistent if it remains in existence beyond the time when the application is closed down. This implies that the data is stored externally to the application, and it would normally be stored in a file.

An application that has data it wants to save when the application closes, and reload when the application starts up again, can provide implementations for the document class's `StoreL()` and `RestoreL()` functions, which are prototyped as:

```
void StoreL(CStreamStore& aStore, CStreamDictionary& aStreamDic) const;
```

and:

```
void RestoreL(const CStreamStore& aStore, const CStreamDictionary&
        aStreamDic);
```

The application framework calls these functions as necessary to create, save and reload the file.

To use these functions, you need to understand the concepts of streams and stores. A Symbian OS stream is a sequence of data items that supplies an external representation of a class instance. The external form is one that is free of any peculiarities of the internal storage format, such as byte order or data alignment. Symbian OS supports the conversion of data between the two formats by means of internalization (>>) and externalization (<<) operators, and a class's implementation of the `InternalizeL()` and `ExternalizeL()` functions. You should refer to the SDK documentation for further information.

A Symbian OS store is a collection of streams, and is normally used to implement persistent data. Such data is usually, but not always, stored in a file, with the aid of either the `CDirectFileStore` or `CPermanentFileStore` classes. You can use the `CBufStore` class to store data in a memory buffer, but such storage will, of course, not be persistent. Other classes of interest are `CSecureStore`, to store encrypted data, and `CEmbeddedStore` which, as its name suggests, allows you to create more complex stores, where one store may be

embedded within another. All these store classes inherit from an abstract `CStreamStore` base class (which is used as the type of the first parameter for the `StoreL()` and `RestoreL()` functions), Each store contains a root stream, which is the first stream to be read on opening the store. The root stream contains an index (which is an instance of the `CStreamDictionary` class) that provides access to the other streams within the store. Refer to the SDK documentation for examples, and further information about creating and using stores.

The persistence mechanism described above results in the whole file being read on application startup and written when the application closes. In consequence, it is not suited to applications that display and manipulate one or more records from a (potentially large) database file. Such an application should not implement the document's `StoreL()` and `RestoreL()` functions, but should provide its own mechanisms for updating the database file as and when necessary.

Since most Series 60 applications fall into this category, the default behavior of the Series 60 document class is to disable the automatic saving and loading of persistent data. If necessary, you can re-enable it by supplying the following implementation of your document's `OpenFileL()` function:

```
CFileStore* CMyDocument::OpenFileL(TBool aDoOpen, const TDesC& aFilename,
        RFs& aFs)
  {
  return CEikDocument::OpenFileL(aDoOpen, aFilename, aFs);
  }
```

The Application UI Class

Your application's UI class supplies the logic that directs the action of your application in response to user actions and other events, including the command handler. It is also responsible, upon construction, for creating the application's default view.

The declaration for the UIQ version of SimpleEx's application UI class is as follows:

```
class CSimpleExAppUi : public CQikAppUi
  {
public:
  void ConstructL();
  ~CSimpleExAppUi();
private:
  void HandleCommandL(TInt aCommand);
private:
  CCoeControl* iAppView;
  };
```

The application framework calls the document's `CreateAppUiL()` function, to create the application's instance of its application UI class,

and then invokes the application UI's `ConstructL()` function. This function must contain all the logic to initialize your GUI application, including the creation of the application view. A pointer to this view should be assigned to `iAppView` for future reference.

The following is the `SimpleEx` application UI class's `ConstructL()` and its destructor.

```
void CSimpleExAppUi::ConstructL()
  {
  BaseConstructL();

  iAppView = CSimpleExAppView::NewL(ClientRect());
  }
CSimpleExAppUi::\simCSimpleExAppUi()
  {
  delete iAppView;
  }
```

In particular, the application UI's `ConstructL()` must call `BaseConstructL()`. This will invoke CEikAppUi's `BaseConstructL()` function, which will open the application's resource file and construct items such as the application's menu.

The application UI implements the `HandleCommand()` function, to process the application's menu command events. Below is `SimpleEx`'s implementation of it.

```
void CSimpleExAppUi::HandleCommandL(TInt aCommand)
  {
  switch(aCommand)
    {
    case EEikCmdExit:
      Exit();
      break;
    case ESimpleExCommand:
      {
      _LIT(KMessage,"Start Selected!");
      iEikonEnv->AlertWin(KMessage);
      }
      break;
    }
  }
```

The events handled by `HandleCommandL()` are identified by 32-bit integers that are defined (typically in an `enum`) in an include file which is included in both your resource file and your source code. Depending on the platform, these events may originate from a variety of sources, including menu bars, keyboard hotkey combinations, tool bars and command button arrays.

The Application View Class

The application view class handles the presentation of your application on the smartphone's screen, as well as allowing the user to interact

with your program. In Symbian OS all objects drawn to a screen are controls – including the application view, which is a custom control.

As was mentioned earlier, in the simplest case, the application's view is implemented as a single control derived from the `CCoeControl` control base class. For more complex applications, Symbian OS supplies a view architecture that allows you to create multiple application views. Custom controls and views will be discussed in more detail in Sections 11.7 and 11.8 respectively. For now let's look at the `SimpleEx` application's view implementation and briefly discuss its key points.

The declaration for `SimpleEx`'s application view class is as follows:

```
class CSimpleExAppView : public CCoeControl
  {
public:
  static CSimpleExAppView* NewL(const TRect& aRect);
  void ConstructL(const TRect& aRect);

private:
  void Draw(const TRect&) const;
  };
```

and the implementation of the view is:

```
CSimpleExAppView* CSimpleExAppView::NewL(const TRect& aRect)
  {
  CSimpleExAppView* self = new (ELeave) CSimpleExAppView;
  CleanupStack::PushL(self);
  self->ConstructL(aRect);
  CleanupStack::Pop(self);
  return self;
  }

void CSimpleExAppView::ConstructL(const TRect& aRect)
  {
  CreateWindowL();
  SetRect(aRect);
  ActivateL();
  }

void CSimpleExAppView::Draw(const TRect& ) const
  {
  CWindowGc& gc = SystemGc();
  const CFont*     font;
  TRect drawRect = Rect();

  gc.Clear();

  font = iEikonEnv->TitleFont();
  gc.UseFont(font);
  TInt   baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;
  gc.DrawText(_L("Simple Example"),
      drawRect,baselineOffset,CGraphicsContext::ECenter, 0);

  gc.DiscardFont();
  }
```

The view's `NewL()` static constructor both instantiates the application view class and calls its `ConstructL()` secondary constructor. The application UI calls this `NewL()` function, passing the application's client area (which is supplied by the `ClientRect()` method in the application UI's base class). This area specifies the region that the application has at its disposal to display its data and is, in turn, passed to the view's `ConstructL()` function.

In `ConstructL()`, `CreateWindowL()` is called to create the control's associated window (a view is always what is known as a window-owning control (see Section 11.7). `SetRect()` sets the area on the screen that the control will occupy. `ActivateL()` is called to mark the control as being ready to draw itself.

The application view, being a custom control, implements its own `Draw()` function by overriding `CCoeControl::Draw()`. This function is called whenever the application view has become invalid, either because it needs to show new data, or because some other object which overlaid the area has been dismissed.

Later, I will discuss more of the details of drawing, but by looking at the `SimpleEx` view's `Draw()` function, you should be able to see that it is simply clearing the screen and drawing the text 'SimpleEx'.

11.4 Resource Files

The application resource file defines a significant part of how your application will appear and function. The resource file is a text file whose name ends in `rss`, and is compiled into a binary form by the SDK's resource compiler. This compiled version of the resource file is loaded onto the phone along with the application executable and is accessed during application execution.

11.4.1 Resource File Format

A resource file consists of data constructs that begin with an uppercase *keyword*. There are only a few keywords used in resource files. The main ones are:

- `NAME`

  `NAME` defines a name, of between one and four upper case characters, that is used by the resource compiler to generate a 20-bit number that it prefixes to resource identifiers to ensure they are distinguishable from the identifiers of other resources used by the application. Note that this means that the name need not be globally unique, it just needs to be different from system resource file names – so avoid starting the name with `EIK`, or using component names like `CONE`. Also, if your application uses multiple resource files, define a unique name for each.

In SimpleEx, the name is defined in the resource file as:

```
NAME SIMP
```

- CHARACTER_SET

  CHARACTER_SET specifies if your resource file is to use either code page 1250 or the UTF-8 character set. If CHARACTER_SET is not specified, it defaults to code page 1250. To specify that your resource file is in UTF-8 format, add the following line to your resource file:

  ```
  CHARACTER_SET UTF8
  ```

- STRUCT

  The STRUCT keyword is used to define a data structure that consists of a sequence of items, with each item being specified by its name and its data type.

You won't often need to define your own STRUCTs, since there is a wide variety of existing ones, for use by all the different GUI elements, and defined in the system's various resource header files (e.g. eikon.rh, uikon.rh). However, it is helpful to see what a STRUCT looks like, in order to better understand the resource file format.

- RESOURCE

  The RESOURCE keyword is used to create an instance of a data structure.

- ENUM

  The ENUM keyword defines an enumeration with the same syntax as in C/C++. This is used for constants such as control identifiers and event codes.

The following is a simple STRUCT definition:

```
STRUCT MYDATA
  {
  WORD value=0;
  LTEXT main_text;
  LTEXT text_items[];
  }
```

WORD and LTEXT are built-in data types that represent a 16-bit word and a Unicode text string (with leading length byte) respectively. Other common data types are:

BYTE	8-bit signed value
LONG	4-byte value
BUF	Unicode string with no leading length byte
LLINK	Link to a resource that contains a resource identifier
STRUCT	Use a STRUCT within a STRUCT

Arrays can also be defined by appending [] to the attribute name. In the MYDATA structure, text_items is defined as an array of text strings.

Also, attributes can be assigned default values within the STRUCT definition. In the example above, value is assigned a default of 0. So, if the programmer does not assign an explicit value to the attribute in a RESOURCE definition, it is automatically assigned the default value.

You use the RESOURCE keyword to create an instance of a STRUCT, as illustrated below for MYDATA:

```
RESOURCE MYDATA r_mydata_res
  {
  value=3;
  main_text="some text string";
  text_items={"text item1", "some other item","other item"};
  }
```

This resource creates an instance of the MYDATA structure with a resource identifier of r_mydata_res (which actually represents a 32-bit integer). You access the resource from within program code by using this identifier in upper case: R_MYDATA_RES.

It's worth pointing out that you can include STRUCT members within STRUCTs since this is commonly used for predefined resources. I illustrate this below, using an additional STRUCT called WIDGETDATA:

```
STRUCT WIDGETDATA
  {
  LTEXT widget_caption;
  STRUCT main_data;
  }
```

main_data is specified as a STRUCT but does not indicate the structure type. It's up to the programmer to know what type of structure to use. In this case, WIDGETDATA expects it to be of our MYDATA type. It's initialized in the following way:

```
RESOURCE WIDGETDATA r_my_widget
  {
  widget_caption="Widget Name";
  main_data=MYDATA {3,"main data", { "item1", "item2"} };
  }
```

11.4.2 SimpleEx's Resource File

Armed with the knowledge in the above section, let's look at the resource file again (originally presented in Chapter 2 (see Section 2.3.4)).

```
NAME SIMP

#include <eikon.rh>
#include "SimpleEx.hrh"

RESOURCE RSS_SIGNATURE
  {
  }

RESOURCE TBUF r_default_document_name
  {
  buf="";
  }

RESOURCE EIK_APP_INFO
  {
  menubar = r_SimpleEx_menubar;
  }

RESOURCE MENU_BAR r_SimpleEx_menubar
  {
  titles =
    {
    MENU_TITLE
      {
      menu_pane = r_SimpleEx_menu; txt="Simple Menu";
      }
    };
  }

RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
      MENU_ITEM
        {
        command = ESimpleExCommand;
        txt = "Start";
        }
    };
  }
```

The `RSS_SIGNATURE` resource is used to validate the file and must appear, exactly as shown above, as the first resource in every application resource file.

The next resource:

```
RESOURCE TBUF r_default_document_name
  {
  buf="";
  }
```

defines the file name of your application's default document. A document is not used in SimpleEx, so it is blank.

Note that TBUF is defined as:

```
STRUCT TBUF
  {
  BUF buf; // non-zero terminated string
  }
```

in baded.rh, included from ukon.rh.

```
RESOURCE EIK_APP_INFO
  {
  menubar = r_SimpleEx_menubar;
  }
```

The resource structure for EIK_APP_INFO is defined as:

```
STRUCT EIK_APP_INFO
  {
  LLINK hotkeys=0;
  LLINK menubar=0;
  LLINK toolbar=0;
  LLINK toolband=0;
  LLINK cba=0;
  LLINK status_pane=0;
  }
```

So, although we only specify the application's menu bar (and the softkeys via the cba attribute for the Series 80 example), you can see that this is the place to associate other things with the application, such as hot keys, the tool bar and the status pane.

> Recall from the section on the resource file format that the LLINK type indicates a link to another resource (see Section 11.4.1).

The menu is constructed using resource structures MENU_BAR, MENU_TITLE, MENU_PANE, and MENU_ITEM, all of which are defined in uikon.rh. Let's look at MENU_BAR:

```
STRUCT MENU_BAR
  {
  STRUCT titles[]; // MENU_TITLEs
  LLINK extension=0;
  }
```

titles[] is expected to be an array of structures of type MENU_TITLE, which is defined as follows:

```
STRUCT MENU_TITLE
  {
  LLINK menu_pane;
```

```
  LTEXT txt;
  LONG flags=0;
  LTEXT bmpfile="";
  WORD bmpid=0xffff;
  WORD bmpmask=0xffff;
  LLINK extension=0;
  }
```

So let's look again at the SimpleEx menu definition:

```
RESOURCE MENU_BAR r_SimpleEx_menubar
  {
  titles =
    {
    MENU_TITLE
      {
      menu_pane = r_SimpleEx_menu; txt="Simple Menu";
      }
    };
  }
RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      }
    };
  }
```

We can see that this instantiates a `MENU_BAR` structure, associating it with resource identifier `r_SimpleEx_menubar`. It initializes its `titles` attribute (which is an array of `STRUCT`s) to a single `MENU_TITLE` structure, which points to our menu pane resource.

11.4.3 Localizing a Resource File

While you can put text strings directly within the resource file as I have done in the examples, this is not recommended if you need to support different language translations. Symbian recommends that you put all your strings into a `rls` file (a `loc` file for Series 60), and then include this string file using `\#include` in your `rss` file. There should be a separate `rls` file for each language you support.

Each string in the `rls` file is defined using the `rls_string` keyword. For example:

```
rls_string STRING_r_example_start “Start”
```

Then in your `rss` file, you supply the keyword `STRING_r_example_start_selected`, instead of putting in the string directly. For example:

```
RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = STRING_r_example_start_selected;
      }
    };
  }
```

At the top of the `rss` file you need to include the proper language `rls` file and this is normally none by using `#ifdef` s. Below is an example of including `rls` files in your resource file for a program that supports English, French and German:

```
#ifdef LANGUAGE_EN
    #include "strings_en.rls"
#elif defined LANGUAGE_FR
    #include "strings_fr.rls"
#elif defined LANGUAGE_DE
    #include "strings_de.rls"
#endif
```

How are the language definitions set? Symbian OS provides support for this in the project `mmp` file by means of the `LANG` keyword. Here are the example lines in the `mmp`:

```
LANG EN FR DE
RESOURCE    MyApp.rss
```

The build script will compile the resource file, defined on the `RESOURCE` line, once for every language specified in the `LANG` line. During each resource compilation, `LANGUAGE_` <language> is defined, where <language> is the current item on the `LANG` line. So it will compile `SimpleEx.rss` once with `LANGUAGE_EN` defined (with the output going to `SimpleEx.ren`), then with `LANGUAGE_FR` defined (with the output going to `SimpleEx.rfr`), and, finally, with `LANGUAGE_DE` defined (with the output going to `SimpleEx.rde`). During each compilation, the correct language `rls` file is included in the resource as defined in the `#ifdef` structure.

There are no rules for the format of the language identifiers on the `LANG` line, each compilation stage simply `#define`s a variable with the specified name, prefixed with `LANGUAGE_`. You can use letter codes as I did, or numeric codes (e.g. 01, 02) to represent different languages.

You typically have all the supported languages of an application contained in a single `sis` file. The user can then select the language they

need when they install that file to their phone (sometimes the system will install the language that matches the one defined in the phone).

Here are the pkg files needed to support having these languages in a single sis:

```
&EN,FR,GE
...
{
"c:\symbian\uiq_70\epoc32\data\z\system\apps\simpleEx\SimpleEx.ren"
"c:\symbian\uiq_70\epoc32\data\z\system\apps\simpleEx\SimpleEx.rfr"
"c:\symbian\uiq_70\epoc32\data\z\system\apps\simpleEx\SimpleEx.rde"
}-"!:\system\apps\simpleEx\SimpleEx.rsc"
```

The & line contains a list of language codes that determine which language options are offered to the user during installation. Unlike the LANG statement in the mmp file, you must use predefined two-letter language codes that correspond to the languages you are supporting (see Section 5.9.4).

The package line in the above example will cause the correct resource file to be installed on the phone (as \system\apps\simpleEx\SimpleEx.rsc) based on the language that the user selected. It is important that the files to be installed should be listed in the same order as the list of specified languages in the & line.

11.4.4 Reading Resource Strings From Code

You should try to avoid using strings directly in the code, since this makes localizing your applications very difficult. For example, I hard-coded a string in the SimpleEx example in the following lines:

```
_LIT(message,"Start Selected!");
iEikonEnv->AlertWin(_L("Start Selected!"));
```

The recommended way of doing this is to define a TBUF resource in your resource file as follows:

```
RESOURCE TBUF r_start_selected {buf="Start Selected!";}
```

Better still, define an rls_string in your rls file (one for each supported language):

```
rls_string STRING_start_selected "Start Selected!"
```

and use that in your rss file:

```
RESOURCE TBUF r_start_selected { STRING_start_selected;}
```

Then in the code, read the resource string in the following way:

```
TBuf<256> message;
iCoeEnv->ReadResource(message, R_START_SELECTED);
```

This will read the string defined in the `TBUF` resource whose resource identifier is `R_START_SELECTED` into the descriptor `message`.

11.5 Dialogs

Much of your GUI application programming will be concerned with creating and managing either dialogs or other forms of GUI controls. This section discusses both and I'll also present a simple dialog you can add to `SimpleEx` to illustrate how to create and manage a dialog box and its associated controls.

11.5.1 Creating a Basic Dialog

Dialogs exist on all Symbian OS platforms although some of the details of their usage vary from platform to platform. For example, UIQ dialogs contain dialog exit buttons, whereas Series 60 and Series 80 dialogs use labeled hardware keys.

A dialog is a pop-up window that has a title, one or more buttons to dismiss the dialog, and one or more lines containing controls that display information and allow the user to set application-specific parameters. Dialogs are almost always *modal*, meaning that the user can interact only with the dialog, and not the rest of the application, until the dialog is dismissed.

Creating a dialog typically consists of the following steps:

1. Create a `DIALOG` resource in your resource file to define the dialog's title and set of dialog lines, where each line contains a control and a text prompt.
2. Create a class derived from `CEikDialog` that, at a minimum, initializes the controls when the dialog is started up and processes/saves the control values when the dialog is dismissed.
3. Implement code to launch the dialog by calling the dialog's `ExecuteLD()` (implemented in the `CEikDialog` base class) specifying the resource identifier of your `DIALOG` resource.

To best explain the process, let's look at an example of a simple dialog box. This dialog is added to `SimpleEx`, and allows you to set the text that's displayed in the middle of the screen. For the Series 80 and UIQ

versions, I have added an extra line with a choice list so that you can also select the color of the displayed text.

The first line of the dialog has a text edit control to specify the display text, and the second line is a choice list control, where you select the color of the text to be one of black, red, green or blue. Figures 11.8 and 11.9 show this dialog for UIQ and Series 80 respectively. When the user selects OK the text in the center of the screen is changed to reflect the string and color that was specified in the dialog.

The choice list control is not supported in Series 60, and list boxes or pop-up controls are normally used instead. Since using these controls in Series 60 involves some extra considerations, let's just use the control to set the text for now. The Series 60 dialog is shown in Figure 11.10.

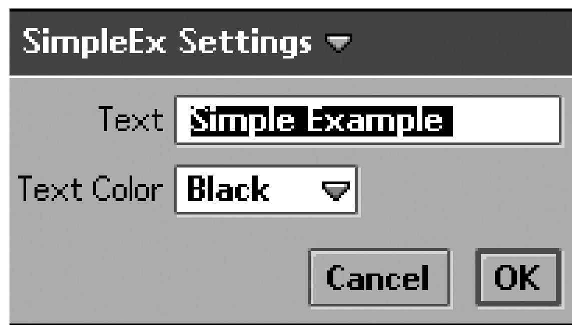

Figure 11.8 `SimpleEx` Dialog in UIQ

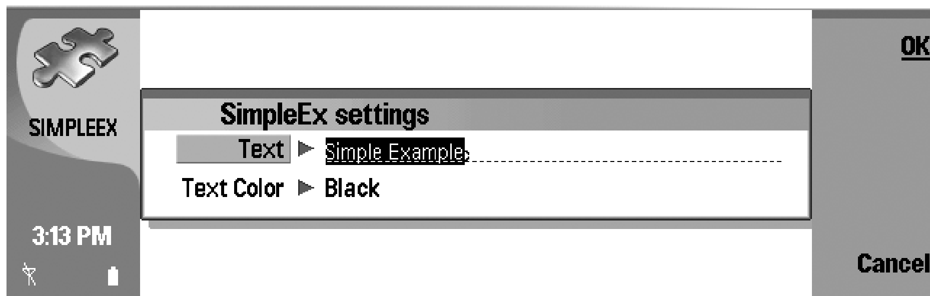

Figure 11.9 `SimpleEx` Dialog for Series 80

Defining the Dialog Resource

First let's create the dialog resource structure for UIQ and Series 80, and add it to the `SimpleEx` resource file:

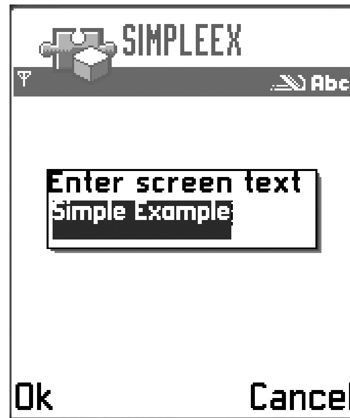

Figure 11.10 `SimpleEx` Dialog for Series 60

```
RESOURCE DIALOG r_SimpleEx_dialog
  {
  title="SimpleEx Settings";
  buttons=R_EIK_BUTTONS_CANCEL_OK;
  flags=EEikDialogFlagWait;
  items=
    {
    DLG_LINE
      {
      type=EEikCtEdwin;
      prompt="Text";
      id=ESimpleExText;
      control=EDWIN {width=25; maxlength=256;};
      },
      DLG_LINE
        {
// This dialog line is the choice list to pick the color.
// Note that this line will not work on Series 60 since
// choice lists are not supported there.

        type=EEikCtChoiceList;
        prompt="Text Color";
        id=ESimpleExColor;
        control=CHOICELIST
          {
          array_id=r_color_list;
          }; // End of control.
        }
    };
  }
```

And here is the Series 60 dialog structure, containing only the text box:

```
RESOURCE DIALOG r_SimpleEx_dialog
  {
  buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
```

```
flags = EGeneralQueryFlags;
items=
  {
  DLG_LINE
    {
    type=EEikCtLabel;
    id=ESimpleExTextLabel;
    control = LABEL
      {
      txt="Enter screen text";
      };
    },

    DLG_LINE
      {
      type=EEikCtEdwin;
      id=ESimpleExText;
      control=EDWIN {width=10; maxlength=256;};
      }
  };
}
```

To simplify the example, I still define the strings directly in the resource file. However, as discussed in the last section, the more correct way is to define them in an `rls` file using the `rls_string` keyword.

`DIALOG` is a `STRUCT` defined in `eikon.rh`. In the example, I create a `DIALOG` resource with identifier `r_SimpleEx_dialog. title` is an `LTEXT` type attribute that specifies the caption on the title bar when displaying the dialog. In this case the title is 'SimpleEx Settings'.

`buttons` is of type `LLINK`, meaning it points to another resource. In this case, it points to a `DLG_BUTTONS` resource that specifies the set of buttons to appear on the dialog, such as OK and Cancel, or any other button you want on the dialog. You can create your own custom set of buttons by creating a `DLG_BUTTONS` resource (which in turn contains one or more `DLG_BUTTON` resources) and specifying it in the dialog's `buttons` attribute. Or, as is commonly done, you can use one of the predefined button resources from `eik-core.rsg` – I used `R_EIK_BUTTONS_CANCEL_OK` which puts 'Cancel' and 'OK' buttons in the dialog. Other predefined dialog button attributes include:

- `R_EIK_BUTTONS_CONTINUE` displays a single 'Continue' button.
- `R_EIK_BUTTONS_CANCEL` displays a single 'Cancel' Button
- `R_EIK_BUTTONS_NO_YES` displays 'Yes' and 'No' buttons.

Series 60 defines its own set of predefined softkey resources for the `buttons` attribute, including:

- `R_AVKON_SOFTKEYS_OK_EMPTY` displays an 'OK' softkey and a blank softkey.

- R_AVKON_DONE_CANCEL displays 'Done' and 'Cancel' softkeys.
- R_AVKON_YES_NO displays 'Yes' and 'No' softkeys.
- R_AVKON_OK_BACK displays 'OK' and 'Back' softkeys.

Refer to avkon.rsg for the complete list.

The dialog's flags attribute (type LONG) defines various characteristics of the dialog. These flags can vary with the platform.

In the example dialog, I only set the EEikDialogFlagWait flag. This makes the dialog modal (meaning that the dialog's ExecuteLD() will not return until the user dismisses the dialog).

Some other flags are:

- EEikDialogFlagNoTitleBar – do not display the title bar.
- EEikDialogFlagDensePacking – minimize the spacing between lines.
- EEikDialogFlagFillAppClientRect – expand the dialog to fill the client rectangle.

Series 60 provides additional platform-specific flags (in avkon.hrh) that are implemented as a combination of other flags to define a specific type of dialog that conforms to the Series 60 GUI guidelines. For example, the dialog flag value EAknDialogSelectionList specifies a combination of flags appropriate for a dialog that presents a list of options in a choice list. See avkon.hrh for the complete list of these Series 60 specific flags.

UIQ and Series 80 typically use only the EEikDialogFlagWait flag in their dialogs.

The items attribute of the DIALOG structure must be an array of DLG_LINE structures, to specify what each line of a dialog contains. The key attributes of a DLG_LINE are type, prompt, id, and control.

type specifies the type of control to be used for that dialog line. This tells the UI's control factory what control class to construct. The first dialog line in the example specifies a text box, with type set to EEikCtEdwin. This tells the control factory to construct the control class CEikEdwin.

prompt specifies the text that is displayed before the control.

id specifies an identifier for the dialog line, and is used by your code to access the line's control. CEikDialog's Control(aId) method is used to convert the id to a pointer to the control.

control is an attribute of type STRUCT. It is expected to be a structure appropriate to the type of control indicated by the type attribute,

containing attributes that are specific to that control. In the first dialog line of our example, the type is EEikCtEdwin, and this means that control structure should be of type EDWIN.

EDWIN is defined in uikon.rh as:

```
STRUCT EDWIN
  {
  LONG flags=0;
  WORD width=0;
  WORD lines=1;
  WORD maxlength=0;
  }
```

In the example, I define the field width (width) as 25 and the maximum length of the entered string (maxlength) as 255.

See the SDK documentation for lists of all the predefined control resource structures, and descriptions of their attributes.

For Series 60, instead of using the prompt attribute of the DLG_LINE, I included another dialog line containing a label control (control type EEikCtLabel). A label is simply a read-only text line, whose text is specified by the txt attribute of a LABEL structure. This displays the edit box prompt above the edit control.

For the second line of the dialog, I have the choice list where the user can select the color of the screen text. The control type for a choice list is EEikCtChoiceList and the corresponding control structure type is CHOICELIST.

CHOICELIST is defined in eikon.rh as:

```
STRUCT CHOICELIST
  {
  WORD flags=0;
  WORD maxdisplaychar=0;
  LLINK array_id=0;
  }
```

The important attribute here is array_id, which points to an array resource that contains the text for each choice in the choice list. In the example, I assign array_id to r_color_list, and define r_color_list in the resource file as follows:

```
RESOURCE ARRAY r_color_list
  {
  items=
    {
    LBUF { txt="Black"; },
    LBUF { txt="Red"; },
```

```
    LBUF { txt="Green"; },
    LBUF { txt="Blue"; }
    };
  }
```

ARRAY is defined in badef.rh as:

```
STRUCT ARRAY
  {
  STRUCT items[];
  }
```

Finally, LBUF is defined as:

```
STRUCT LBUF
  {
  LTEXT txt; // leading-byte counted text string
  }
```

As you can see, for a given control, you have to know three things: the type of control to assign to the DLG_LINE type attribute, the expected control structure to assign to the control attribute of the DLG_LINE, and the name of the control class to use in the source code. Refer to Section 11.6, as well as the SDK, for the list of these identifiers for the different controls.

The Dialog Class

Now let's look at the dialog class in the source code:

```
class CSimpleExDialog : public CEikDialog

  {
public:
  CSimpleExDialog(TDes& aText,TRgb& aColor);

private:
// Inherited from CEikDialog
  void  PreLayoutDynInitL();
  TBool OkToExitL(TInt aKeycode);

  TDes& iText;
  TRgb& iColor;
  };
```

You derive dialog classes from CEikDialog. When the dialog is launched, PreLayoutDynInitL() is called – you override this method in your dialog class to set the initial values of the dialog controls. OkToExitL() is called when the dialog is dismissed (an exception is that it is not

called when Cancel or the Escape key is selected). In our case, it's called when the OK button is selected.

Now let's look at the dialog class implementation for UIQ and Series 80:

```
CSimpleExDialog::CSimpleExDialog(TDes& aText, TRgb& aColor) :
        iText(aText),iColor(aColor)
  {
  }

const TRgb colorList[4]={KRgbBlack,KRgbRed,KRgbGreen,KRgbBlue};

void CSimpleExDialog::PreLayoutDynInitL()
  {
  STATIC_CAST(CEikEdwin*,Control(ESimpleExText))->SetTextL(&iText);
  TInt currColorIndex=0;
  for (TInt i=0;i<4;i++)
    {
    if (iColor==colorList[i])
      {
      currColorIndex=i;
      break;
      }
    }
  STATIC_CAST(CEikChoiceList*,
      Control(ESimpleExColor))->SetCurrentItem(currColorIndex);
  }

TBool CSimpleExDialog::OkToExitL(TInt /*aKeycode*/)
  {
  TInt cIndex = STATIC_CAST(CEikChoiceList*,
      Control(ESimpleExColor))->CurrentItem();
  iColor = colorList[cIndex];

  STATIC_CAST(CEikEdwin*,Control(ESimpleExText))->GetText(iText);

  return ETrue;
  }
```

Let's start with the constructor – it is passed references to the text descriptor that determines the text drawn on the main screen, and to the variable that indicates the current text color. The text and text color are assigned to the dialog's `iText` and `iColor` member data items, respectively.

In `PreLayoutDynInit()`, I get a pointer to the text edit box and set its initial text with:

```
STATIC_CAST(CEikEdwin*,Control(ESimpleExText))->SetTextL(&iText);
```

This calls `CEikDialog`'s `Control()` method, passing the identifier of the dialog line to it that contains the control (specified in `DLG_LINE`'s `id` attribute). In this case, it's the line with the edit box, whose identifier is set to `ESimpleExText`. I use `STATIC_CAST()` to cast this pointer to type

CEikEdwin, the edit window control class. The SDK documentation describes all the methods available to the various control classes and, of course, these can vary greatly between controls. In this case, I use CEikEdwin::SetTextL() to set the initial text in the edit box to that contained in iText.

Next, for the UIQ and Series 80 versions of the example, I want the choice list to be initialized to the current text color, whose value is in iColor. First I determine the choice list index of the current color (using a lookup array I created called colorList). Then I set the initial index value to the choice list with:

```
STATIC_CAST(CEikChoiceList*,
        Control(ESimpleExColor))->SetCurrentItem(currColorIndex);
```

This retrieves the choice list control in the DLG_LINE, whose identifier is ESimpleExColor, casts it to the choice list class CEikChoiceList, then calls the SetCurrentItem() method to set the current list choice.

After PreLayoutDynInit(), the dialog is now displayed with the initial values of the controls corresponding to the values of iText and iColor. The user can change the control values as desired on the dialog box. When satisfied, the user selects OK, and the dialog class method OkToExitL() is invoked, and is passed the identifier of the button selected by the user (we ignore it here, because it will always be the OK button in our case).

In OkToExitL() I do the reverse of PreLayoutDynInit(), and get the values of the text box and the color choice list and write these to iText and iColor, respectively. I then return ETrue, so that the dialog will actually be dismissed when the function exits. If you return EFalse, the dialog remains displayed upon exit of OkToExitL(). This is useful when you have multiple dialog buttons and you want to perform an operation upon selection of a specific button (which you can determine by checking the value of aKeycode), but do not want to dismiss the dialog yet.

Also, remember that if the Cancel button is selected then the dialog exits without calling OkToExitL().

For Series 60, since we have only the edit control, the dialog class is simpler and defined as follows:

```
CSimpleExDialog::CSimpleExDialog(TDes& aText, TRgb& aColor) :
        iText(aText),iColor(aColor)
  {
  }

void CSimpleExDialog::PreLayoutDynInitL()
  {
  STATIC_CAST(CEikEdwin*,Control(ESimpleExText))->SetTextL(&iText);
  }
```

```
TBool CSimpleExDialog::OkToExitL(TInt /*aKeycode*/)
  {
  STATIC_CAST(CEikEdwin*,Control(ESimpleExText))->GetText(iText);
  return ETrue;
  }
```

Launching the Dialog

To launch the dialog, I added a menu item called `Settings` to the main menu in the resource file:

```
RESOURCE MENU_PANE r_SimpleEx_menu
  {
  items =
    {
    MENU_ITEM
      {
      command = ESimpleExCommand;
      txt = "Start";
      },
      MENU_ITEM
        {
        command = ESimpleExDialog;
        txt = "Settings";
        }
    };
  }
```

I added the `ESimpleExDialog` command code, as well as the dialog line identifiers, to `SimpleEx.hrh`:

```
enum
  {
  ESimpleExCommand = 1,  // start value must not be 0
  ESimpleExDialog,
  ESimpleExText,
  ESimpleExColor
  };
```

I then added the following code to the `CSimpleExAppUi::Handle-CommandL()` command `switch` statement, to handle the `ESimpleEx-Dialog` command:

```
case ESimpleExDialog:
  CEikDialog* dialog = new (ELeave) CSimpleExDialog
      (iAppView->iDisplayText,iAppView->iTextColor);
  dialog->ExecuteLD(R_SIMPLEEX_DIALOG);
  break;
```

This code constructs the dialog class, passing the current display text and display color to the constructor. The code then launches the dialog

by calling the dialog's `ExecuteLD()` method – passing it the resource identifier of the dialog. I do not check the return value here, since for this example it's not needed, but `ExecuteLD()` returns `ETrue` when OK is selected, and `EFalse` if the dialog was canceled.

Although I have not done so here, it's common for a dialog class to implement a static `RunDlgLD()` method that will both construct the dialog class instance and call the `ExecuteLD()` function.

Other Modifications to SimpleEx

Let's look at the modifications made to the `SimpleEx` view class in order to have the text displayed using the text and color variables manipulated by the dialog.

In the class declaration, added members `iDisplayText` and `iTextColor` to specify the text string and color:

```
class CSimpleExAppView : public CCoeControl
  {
public:
  static CSimpleExAppView* NewL(const TRect& aRect);
  static CSimpleExAppView* CSimpleExAppView::NewLC(const TRect& aRect);
  void ConstructL(const TRect& aRect);

  TBuf<100> iDisplayText;
  TRgb iTextColor;

private:
  void Draw(const TRect&) const;
  };
```

And I modified the view's secondary constructor to initialize these variables to a default.

```
void CSimpleExAppView::ConstructL(const TRect& aRect)
  {
  CreateWindowL();
  SetRect(aRect);
  iTextColor=KRgbBlack;
  iDisplayText.Copy(_L("Simple Example"));
  ActivateL();
  }
```

I then modified the `Draw()` function to use `iDisplayText` and `iTextColor` when drawing the centered text on the view as follows:

```
void CSimpleExAppView::Draw(const TRect& ) const
  {
  CWindowGc& gc = SystemGc();
  const CFont* font;
  TRect drawRect = Rect();
```

```
gc.Clear();

font = iEikonEnv->TitleFont();
gc.UseFont(font);
TInt   baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;

gc.SetPenColor(iTextColor);
gc.DrawText(iDisplayText,drawRect,baselineOffset,
    CGraphicsContext::ECenter, 0);

gc.DiscardFont();
}
```

Also, in your mmp file, you need to add the following:

```
LIBRARY   eikdlg.lib eikctl.lib eikcoctl.lib
```

since these libraries are needed for the dialog class, and the controls it uses.

11.5.2 Multipage Dialogs

A multipage dialog is a dialog that allows you to switch between multiple dialog pages via tabs. Multipage dialogs are useful for dialogs which contain a large number of related controls. All of the UI platforms support multipage dialogs, although their appearance and navigation methods can vary slightly.

To define a multipage dialog, instead of putting your DLG_LINEs directly in the DIALOG resource structure, you set the pages attribute of your DIALOG structure to point to an ARRAY resource, which specifies a list of PAGE structures. In each PAGE structure, you define the text to appear on the tab associated with the page (text attribute), an identifier for the page (id attribute) and a pointer to an array (another ARRAY resource) that contains the list of your DLG_LINEs that make up that page. Here is an example resource definition of a three-page multipage dialog:

```
RESOURCE DIALOG r_my_multi_page_dialog
  {
  title="Multipage dialog;

  pages=r_my_pages;
  flags=EEikDialogFlagWait;
  buttons= R_EIK_BUTTONS_CANCEL_OK;
  }

RESOURCE ARRAY r_my_pages
  {
  items=
```

```
    {
    PAGE { text="Page 1"; id=EMyPage1; lines=r_my_page_1; },
    PAGE { text="Page 2"; id=EMyPage2; lines=r_my_page_2; },
    PAGE { text="Page 3"; id=EMyPage3; lines=r_my_page_3; },
    };
  }

RESOURCE ARRAY r_my_page_1
  {
  items=
    {
    DLG_LINE  { ...  },
    DLG_LINE  { ...  }
    };
  }

RESOURCE ARRAY r_my_page_2
  {
  items=
    {
    DLG_LINE  { ...  },
    DLG_LINE  { ...  }
    };
  }

RESOURCE ARRAY r_my_page_3
  {
  items=
    {
    DLG_LINE  { ...  },
    DLG_LINE  { ...  }
    };
  }
```

In your dialog class code, the control values are set and retrieved in the same way as on a single-screen dialog. In other words, the division of controls between pages on the dialog affects the resource file only, and your code still retrieves controls by the `Control()` method in the same way, regardless of what page it is on.

11.5.3 Series 60 Specifics

CAknDialog Class

Series 60 provides a class called `CAknDialog` that extends `CEikDialog` by allowing you to associate a menu with the dialog This allows the dialog to have a variable number of 'button' options, instead of just the `OK` and `Exit` softkeys that are the only ones available from `CEikDialog`. You can pass the identifier of the required `MENU` resource to `CAknDialog::ConstructL()`. In programming terms, apart from the

addition of this optional menu, these dialogs behave in the same way as those derived from CEikDialog.

Series 60 Forms

Series 60 provides another type of dialog, which is known as a form. Forms derive from CAknForm, which itself extends CAknDialog. A form is the preferred type of dialog to use in Series 60 applications it guarantees that the dialog conforms to the Series 60 UI guidelines.

A form displays a set of data fields in the form of a list, with each data field in the list consisting of a label and a control. The label can be on the same line as the control, or it can be on a separate line, with the control below it. In addition, a form dialog is automatically associated with a standard menu that supplies the options: Add field, Edit label, Delete field, Save and, optionally, Edit. Selecting one of the first four of these options results in a call to the appropriate one of the CAknForm functions: AddItemL(), EditCurrentLabel(), DeleteCurrentItem() and SaveFormDataL().

A Series 60 form has two modes: in 'view' mode it acts as an application view that displays a list of data items, and 'edit' mode can be used to modify the data items displayed. By default, it starts up in 'view' mode and you can switch to 'edit' mode by selecting the Edit menu option. When you have finished editing the data, you press the right softkey (temporarily labeled Done) to return to the 'view' mode.

A form is actually more powerful than a dialog. If, for example, the data items it is displaying are the fields of a database record, you can implement the commands described above to add, delete or modify entire records.

You can specify that the form should be edit-only (via a flag in the FORM resource), so that the form is always in 'edit' mode, and in this case, the Edit menu option does not appear. Also, you can override its DynInitMenuPaneL() to disable some or all of the other menu options.

You specify a form in the resource file by creating a FORM resource and assigning it to the form attribute of a DIALOG resource (or a PAGE resource for multipage dialogs). The FORM resource contains the list of DLG_LINES that specify the label and control for each field in the form's list.

The above description of forms is, of necessity, brief. You should refer to the SDK documentation for examples, and a more complete explanation.

For now, let's look at a form-based implementation of the SimpleEx dialog that modifies both the displayed text and its color. This example illustrates many of the features of a form, and displays the dialog shown in Figure 11.11:

Figure 11.11 Form-based SimpleEx Dialog for Series 60

Here is the form's definition in the resource file:

```
RESOURCE DIALOG r_SimpleEx_dialog
  {
  flags=EEikDialogFlagNoDrag|EEikDialogFlagFillAppClientRect|
      EEikDialogFlagNoTitleBar|EEikDialogFlagWait|
      EEikDialogFlagCbaButtons;
  buttons=R_AVKON_SOFTKEYS_OPTIONS_BACK;
  form=r_SimpleEx_form;
  }

RESOURCE FORM r_SimpleEx_form
  {
  flags = EEikFormEditModeOnly | EEikFormUseDoubleSpacedFormat;
  items=
    {
    DLG_LINE
      {
      type=EEikCtEdwin;
      prompt="Text";
      id=ESimpleExText;
      control=EDWIN { width=10; maxlength=256;};
      },
    DLG_LINE
      {
      type=EAknCtPopupFieldText;
      prompt="Color";
      id=ESimpleExPopup;
      itemflags=EikDlgItemTakesEnterKey|
          EEikDlgItemOfferAllHotKeys;
      control = POPUP_FIELD_TEXT
        {
        popupfield=POPUP_FIELD
          {
          width=10;
```

```
        };
      textarray=r_color_list;
      };
    }
    };
  }

RESOURCE ARRAY r_color_list
  {
  items=
    {
    LBUF { txt="Black"; },
    LBUF { txt="Red"; },
    LBUF { txt="Green"; },
    LBUF { txt="Blue"; }
    };
  }
```

As you can see, the `DIALOG` resource defines the flags and softkeys, as in the earlier examples, and the `form` attribute points to a `FORM` resource. This resource specifies the dialog's content which, in this case, consists of two dialog lines: the text edit box and a Series 60 specific control, known as a pop-up field (type `EAknCtPopupFieldText` and control structure `POPUP_FIELD`) which is used to select the text color. The `FORM` resource has an additional `flags` attribute, which is used here to set each control and its prompt to be displayed on separate lines, and to set the 'edit only' mode that was mentioned earlier.

Here is the dialog class definition:

```
class CSimpleExForm : public CAknForm
  {

public:
  static CSimpleExForm* NewL(TDes& aText,TRgb& aColor);

private:
  CSimpleExForm(TDes& aText,TRgb& aColor);
  // Inherited from CAknForm
  void DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* aMenuPane);
  TBool SaveFormDataL();
  void  PreLayoutDynInitL();

private:
  TDes& iText;
  TRgb& iColor;
  };
```

The corresponding implementation is:

```
#include "SimpleEx.h"
#include "SimpleEx.hrh"
```

```
#include "eikedwin.h"
#include <AknPopupFieldText.h> // CAknPopupFieldText
#include <avkon.rsg> // R_AVKON_FORM_MENUPANE
#include <eikmenup.h> // CEikMenuPane

const TRgb colorList[4] =
  {
  KRgbBlack, KRgbRed, KRgbGreen, KRgbBlue
  };

CSimpleExForm* CSimpleExForm::NewL(TDes& aText,TRgb& aColor)
  {
  CSimpleExForm* self = new (ELeave) CSimpleExForm(aText, aColor);
  CleanupStack::PushL(self);
  self->ConstructL();
  CleanupStack::Pop(self);
  return self;
  }

CSimpleExForm::CSimpleExForm(TDes& aText, TRgb& aColor) :
        iText(aText),iColor(aColor)
  {
  }

void CSimpleExForm::PreLayoutDynInitL()
  {
  STATIC_CAST(CEikEdwin*,Control(ESimpleExText))->SetTextL(&iText);
  TInt currColorIndex = 0;
  for ( TInt i = 0; i < 4; i++ )
    {
    if ( iColor == colorList[i] )
      {
      currColorIndex = i;
      break;
      }
    }
  CAknPopupFieldText* popupFieldText = static_cast
      <CAknPopupFieldText*>(ControlOrNull(ESimpleExPopup));
  if (popupFieldText)
    {
    popupFieldText->SetCurrentValueIndex (currColorIndex);
    }
  }

void CSimpleExForm::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane*
        aMenuPane)
  {
  CAknForm::DynInitMenuPaneL(aResourceId,aMenuPane);

  if (aResourceId == R_AVKON_FORM_MENUPANE)
    {
    aMenuPane->SetItemDimmed(EAknFormCmdLabel, ETrue);
    aMenuPane->SetItemDimmed(EAknFormCmdAdd, ETrue);
    aMenuPane->SetItemDimmed(EAknFormCmdDelete, ETrue);
    }
  }

TBool CSimpleExForm::SaveFormDataL()
```

```
{
CEikEdwin* nameEditor = static_cast <CEikEdwin*>
    (ControlOrNull(ESimpleExText));
if (nameEditor)
  {
  nameEditor->GetText(iText);
  }

CAknPopupFieldText* popupFieldText = static_cast
    <CAknPopupFieldText*>(ControlOrNull(ESimpleExPopup));
if (popupFieldText)
  {
  iColor = colorList[popupFieldText->CurrentValueIndex()];
  }

  return ETrue;
}
```

`PreLayoutDynInit()` is overridden, exactly as in our earlier dialog example, to set the initial values of the controls in the form. As before, we use `CEikEdwin` for the text editor control. For the color list, we use the `CAknPopupFieldText` class, which corresponds to the `EAknCtPopupFieldText` control type that is specified in the resource file.

In a form, you do not override `OkToExitL()` to have the settings take effect. Instead, you override `CAknForm`'s `SaveFormDataL()` function which is called when the user selects the `Save` option on the menu. In this case, the code is similar to that used in the earlier example's `OkToExitL()`; it reads the control's values and assigns them to the text and color references passed to our form, thus enabling them to take effect.

Since we do not want the user to be able to add or delete data, or edit the field labels for our simple example, we override `DynInitMenuPaneL()` and disable the corresponding menu items as follows:

```
void CSimpleExForm::DynInitMenuPaneL(TInt aResourceId,
    CEikMenuPane* aMenuPane)
  {
  CAknForm::DynInitMenuPaneL(aResourceId,aMenuPane);

  if (aResourceId == R_AVKON_FORM_MENUPANE)
    {
    aMenuPane->SetItemDimmed(EAknFormCmdLabel, ETrue);
    aMenuPane->SetItemDimmed(EAknFormCmdAdd, ETrue);
    aMenuPane->SetItemDimmed(EAknFormCmdDelete, ETrue);
    }
  };
```

Calling `SetItemDimmed()` on these menu items has the effect of removing them from the form's menu. We don't need to dim the `Edit` menu item (which normally switches the form from 'view' mode to 'edit' mode) since it is automatically suppressed for an 'edit only' form.

In HandleCommand(), the form is launched in a similar way to launching a standard dialog:

```
case ESimpleExDialog:
  CSimpleExForm* form = CSimpleExForm::NewL(iAppView->iDisplayText,
      iAppView->iTextColor);
  form->ExecuteLD(R_SIMPLEEX_DIALOG);
  break;
```

11.5.4 Additional Dialog Methods

Here are some additional methods of CEikDialog you can take advantage of to create more advanced dialog boxes.

```
void MakeWholeLineVisible(TInt aControlId, TBool aVisible)
```

This method will make the dialog line with identifier aControlId visible or invisible depending on whether aVisible is set to ETrue, or EFalse respectively. For example:

```
MakeWholeLineVisible(ESimpleExColor,EFalse);
```

would make the dialog line containing our color choice list field in our example dialog invisible.

Alternatively you can use MakeLineVisible(TInt aControlId, TBool aVisible) which affects only the control's visibility and not the rest of the dialog line.

```
void SetDimmedNow(TInt aControlId, TBool aDimmed)
```

This method will dim or undim the dialog line with the specified control identifier. Set aDimmed to ETrue to dim, or EFalse to undim. Dimming a dialog line causes it to be read only.

```
void SetTitleL(TDesC& aTitle)
```

This method sets the dialog title to aTitle. There is also a version that reads the title from a text resource, specified by its resource identifier.

```
void SetControlCaptionL(TInt aControlId, TDesC& aCaption)
```

You can use this to change the prompt displayed before the control in the dialog line, identified by aControlId to the text specified in aCaption. Again, there is also a version that reads the text from a resource.

```
void DeleteLine(TInt aControlId)
```

This deletes the dialog line identified by aControlId.

```
void InsertLineL(TInt aIndex,TInt aResourceId,TInt aPageId=0)
```

This inserts a dialog line in your dialog box, at the line number specified in aIndex. aResourceId is the resource identifier of a dialog line resource (defined with RESOURCE DLG_LINE). In a multipage dialog, aPageId specifies the page in which the insertion will be made (for single page dialogs, it's always 0).

11.5.5 Additional CEikDialog Methods to Override

In the example we overrode CEikDialog's PreLayoutDynInitL() and OkToExitL() methods to pre-initialize the dialog controls upon display and to save/process the dialog settings on exit, respectively. Here are some other CEikDialog methods you can override to make more advanced dialog boxes:

```
virtual void HandleControlStateChangeL(TInt aControlId)
```

The framework calls this method when one of the dialog's components reports a change of state, which happens, for example, when there is a change in the selected item in a choice list. The value of aControlId identifies the control whose state has changed. You can override this CEikDialog method, for example, if you want to make a line visible or not (using, for example, MakeWholeLineVisible()) depending on the state of another control. That way, as the control state changes, some other dialog field can be made to appear or disappear, based on its applicability.

```
virtual void LineChangedL(TInt aControlId)
```

This method is called when the dialog line focus changes. aControlId indicates which line has gained focus. Override this if you want to do something special (such as put up special softkeys) when a particular dialog line obtains focus.

```
virtual void PageChangedL(TInt aPageId)
```

This function is called when the user changes pages on a multipage dialog. aPageId indicates the identifier of the page that was changed to (as defined by the id attribute of the PAGE resource structure). You can override this, for example, if you want to display a special set of softkeys when the user switches to a particular dialog page.

11.5.6 Using Stock Dialogs

Symbian OS has some predefined dialogs that you can use for convenience, and some of these vary from platform to platform. I will briefly cover some of the main ones in this section.

Dialogs Common to All UI platforms

The following methods of CEikonEnv are a quick way to put up some simple, commonly used dialogs without defining a class or creating a resource. They are available to all UI platforms.

```
static void CEikonEnv::InfoWinL(const TDesC& aLine1, const TDesC& aLine2)
```

This is a static function that displays an information dialog with the specified lines of text.

```
void CEikonEnv::AlertWin(const TDesC& aLine1, const TDesC& aLine2)
```

This displays an alert dialog with the specified lines of text.

```
TBool CEikonEnv::QueryWinL(TInt aFirstLineId, TInt aSecondLineId=0)
```

This displays a Yes/No query dialog with the specified lines of text, supplied as resource identifiers. QueryWinL() returns ETrue if Yes was selected, and EFalse if No.

UIQ Stock Dialogs

These dialogs are straightforward to use, in that no resources need be defined. You just call the class's static function RunDlgLD() (with the appropriate arguments) and the dialog will be constructed and displayed. RunDlgLD() will return when the dialog is dismissed, and the dialog data will be returned in the appropriate parameters passed to the function.

For example, CEikTimeDialogSetTime displays a dialog allowing the user to enter the time and date. Below is an example of the code to display this (you need to include file eiktime.h):

```
 TTime currentDateTime;
// Launch the dialog and get the date and time from the dialog.
// if(CEikTimeDialogSetTime::RunDlgLD(currentDateTime))
  {
  // currentDateTime now contains the date and time entered.
  }
```

The `TTime` class holds the time and date, as well as providing many formatting functions for text display. It also supplies comparison functions for calculating intervals between two `TTime` instances.

This date and time dialog for UIQ is shown in Figure 11.12.

Other UIQ stock dialog classes include:

- `CEikSetPasswordDialog` sets a password, providing both password and confirmation fields.
- `CEikTimeDialogSetCity` allows the user to set allocation, by selecting a country and city.
- `CEikEdwinFindDialog` allows the user to find text.
- `CQikZoomDialog` allows the user to set the zoom level.

To use these dialogs, be sure to include the appropriate include file. Also, in the `LIBRARY` section of your `mmp` file, you need to add `eikcdlg.lib` to use dialogs that begin with `CEik`, and `qikdlh.lib` to use dialogs that begin with `CQik`.

Series 80 Predefined Dialogs

Series 80 provides a set of predefined dialogs that begin with `CCkn` (part of the Series 80 `Ckon` layer). To use them, include the appropriate include file and add the `ckndlg.lib` library to your `mmp` file. No resource definition is needed. There are many variations of these dialogs, with options to supply icons, or to use other ways to customize them. Refer to the Series 80 SDK documentation for the complete list of dialog classes, as well as the various options that they have. Here are some of them:

- `CCknConfirmationDialog` – include `cknconf.h`

```
CCknConfirmationDialog::RunDlgWithDefaultIconLD(_L("Do you want to do
        this?"), R_EIK_BUTTONS_NO_YES );
```

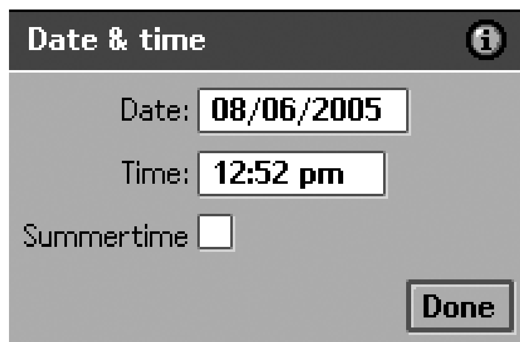

Figure 11.12 Date and Time Dialog for UIQ

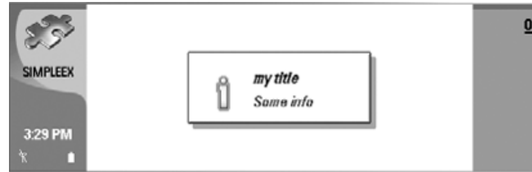

Figure 11.13 Info dialog for Series 80

- `CCknInfoDialog` – include `ckninfo.h`

```
CCknInfoDialog::RunDlgLD( _L("My title"), _L("Some info"));
```

This displays the dialog shown in Figure 11.13.

- `CCknOpenFileDialog` – include `ckndgopn.h`.

```
_LIT(KPath,"\\");
TBuf<255> path = KPath;
CCknOpenFileDialog::RunDlgLD(path);
```

This displays a full-featured dialog to browse the directory structure, to locate and select a file for opening. On exit from `RunDlgLD()`, `path` contains the full path of the file the user selected.

- `CCknNewFileDialog`, `CCknSaveAsDialog` – include `ckndgsve.h`

These display dialogs to create a new file and to save a file to a specific path, respectively.

Series 60 Predefined Dialogs

Series 60 provides a set of standard query dialogs to obtain specific types of data from the user. Each of these dialogs requires that a `DIALOG` resource be created, with its flags attribute set to `EGeneralQueryFlags`, plus a `DLG_LINE` with control `type` set to `EAknCtQuery` and `id` set to `EGeneralQuery`. Here are a couple of examples.

Confirmation Query Dialog

A confirmation query dialog uses class `CAknQuieryDialog`. Below is a resource defined for it:

```
RESOURCE DIALOG r_yes_no_dialog
  {
  flags = EGeneralQueryFlags;
```

```
  buttons = R_AVKON_SOFTKEYS_YES_NO;
  items =
    {
    DLG_LINE
      {
      type = EAknCtQuery;
      id = EGeneralQuery;
      control = AVKON_CONFIRMATION_QUERY
        {
        layout = EConfirmationQueryLayout;
        label = "";
        };
      }
    };
  }
```

And the code to invoke it is:

```
CAknQueryDialog* dlg = CAknQueryDialog::NewL(aQueryTxt, aSound ?
        CAknQueryDialog::EConfirmationTone : CAknQueryDialog::ENoTone);
return dlg->ExecuteLD(R_YES_NO_DIALOG);
```

Although not used in this example, `AVKON_CONFIRMATION_QUERY` also has bitmap and animation attributes.

Text Query

`CAknTextQueryDialog` is a class used to display a dialog that prompts the user for a text entry. Below is an example resource definition for it:

```
RESOURCE DIALOG r_demo_data_query
  {
  flags = EGeneralQueryFlags;
  buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
  items =
    {
    DLG_LINE
      {
      type = EAknCtQuery;
      id = EGeneralQuery;
      control = AVKON_DATA_QUERY
        {
        layout = EDataLayout;
        label = ""; // prompt text
        control = EDWIN
          {
          width = 5;
          lines = 1;
          maxlength = 15;
          };
        };
      }
    };
  }
```

And the code to invoke it is:

```
TBuf<128> text; //where the user text will be placed
TBuf<128> prompt(_L("Enter text:"));

CAknTextQueryDialog* dlg =
new( ELeave ) CAknTextQueryDialog( text, prompt );
// Prepares the dialog, constructing it from the specified resource
// Sets the maximum length of the text editor
dlg->SetMaxLength(20);
// Launch the dialog
if (dlg->ExecuteLD(R_DEMO_DATA_QUERY))
  {
  // ok pressed, text is the descriptor containing
  // the entered text in the editor.
  }
```

This displays a text edit box with the prompt 'Enter text:'.

Other Series 60 query dialogs include:

- `CAknNumberQueryDialog` – Used for entering an integer.
- `CAknTimeQueryDialog` – Used for entering a date/time query.
- `CAknListQueryDialog` – Used for entering a query that requires a selection from a list (setting `DLG_LINE` control attribute to `AVKON_LIST_QUERY` for a single selection list, and `AVKON_MULTI-SELECTION_LIST_QUERY` to allow multiple selections).

Below are classes that are referred to as wrappers in Series 60, since they require no resource to be defined, and thus are simple to use.

- `CAknConfirmationNote` – Allows the user to confirm an action, as in the following example.

```
TBuf<256> msg;
CCoeEnv::Static()->ReadResource(msg, R_MSG_DELETE_PENDING);
CAknConfirmationNote* note = new (ELeave) CAknConfirmationNote();
note->ExecuteLD(msg);
```

- `CAknInformationNote` – Used in a similar way to the above dialog, but displays an information note.
- `CAknErrorNote` – Displays an error message.
- `CAknWaitDialog` – Used to wait for a process to complete.
- `CAknProgressDialog` – Indicates the progress of a long-running process.

11.6 Symbian OS Controls

Symbian OS has a large variety of GUI controls available. Many are common to all platforms, and others are specific to the particular UI

platform you are using. As we saw in the dialog example, you need to know the following to use a control:

- The RESOURCE structure to use for the control, and the appropriate attributes to set. The resource structures of all the common controls are in uikon.rh and eikon.rh. UIQ specific control structures are in Qikon.rh, Series 60 specific controls are in avkon.rh and Series 80 specific controls are in cknctl.rh.
- The name of the control type identifier to use as the type attribute of the DLG_LINE (e.g. type=EEikCtEdwin for a text edit control).
- The class of the control, and its class-specific methods, including those to write data to and retrieve data from the control.
- The header file to include for using the control class, and the library to link to.

Note that, although it is most common to implement controls using resource definitions within a dialog resource's DLG_LINE structure, there are other options as well. For example, your control could have its own, separate RESOURCE identifier, and you can use code to insert it into a dialog, or directly into a view. Alternatively, you can incorporate the control into your program without using a resource definition at all, using control-specific methods to set the control's attributes. This is common, for example, when you implement a list box to be a part of your main view.

11.6.1 Types of Control

This section gives a tour of some of the main control types. It is not exhaustive, but the SDK documentation contains a complete list of available controls.

Editor Controls

Edit controls allow the user to enter a piece of data of a specific type. A wide variety of edit controls exist, including text editors, number editors, calendar editors, date and time editors, duration editors, secret editors, PIN editors, color editors, IP address editors, and more.

For example, we have seen the text editor, which uses resource structure EDWIN, control type identifier EEikCtEdwin, and class CEikEdwin. Its key methods are SetText() to write initial text into the editor and GetText() to retrieve the contents of the editor.

List Boxes

List boxes are very common in Symbian OS software, and a wide variety of list box types exist. These include, amongst others, single and multiple section boxes, list boxes with graphics, numbered list boxes, double list boxes, and settings list boxes (where you can change the value of each entry).

List boxes use the `LISTBOX` resource structure, which is defined as:

```
STRUCT LISTBOX
  {
  BYTE version=0;
  WORD flags = 0;
  WORD height = 5;        // in items
  WORD width = 10;        // in chars
  LLINK array_id = 0;
  }
```

The `flags` attribute specifies the characteristics of the list (for example, `EEikListBoxMultipleSelection` means that the user can select multiple options in the list), and `height` and `width` specify the list's size. `array_id` points to an array resource of text items to display in the list. Alternatively, you can leave this `array_id` at 0 and build your own array in code, using class methods (such as `Model()->SetTextItemArray()`) to associate the array with the list box.

List box classes are ultimately derived from `CEikListBox`. The main classes available are:

- `CEikTextListBox` (control type `EEikCtListBox`) – This is the basic text list box.
- `CEikColumnListBox` (control type `EEikCtColListBox`) – Displays cells that are grouped in columns.
- `CEikHierarchicalListBox` – Displays a hierarchical list, where items can be expanded or collapsed.

In UIQ and Series 80, you usually use the above classes directly. In these platforms, list boxes are rarely used in dialogs (choice lists are used instead), and list boxes are most often added directly to your application view.

In Series 60, you can use list box query dialogs as an alternative to including a list box control in a dialog. Series 60 provides a number of specific classes for the various types of list boxes. For example:

- `CAknSingleStyleListBox` (control type `EAknCtSingleListBox`) is a standard single-selection list box.

- `CAknSingleGraphicsStyleListBox` (control type `EAknCtSingleGraphicsListBox`) allows you to include graphics in a single-selection list box.

There are many more list box classes to choose from – refer to the SDK documentation for the complete list.

Progress Bars

A progress bar control uses a `PROGRESSINFO` resource structure, together with a control type of `EEikCtProgInfo` and class `CEikProgressInfo`. You use this to provide user feedback on the progress of a long-running process. The control gives a graphical representation of progress, as well as text showing the percentage (or fraction) completed. In Series 60, you usually display the progress bar using a dedicated dialog (class `CAknProgressDialog`) known as the progress note dialog.

Option Buttons

Also known as radio buttons, option buttons allow you to select one option from a small list of choices.

UIQ has a vertical option list (class `CQikVertOptionButtonList`, resource structure `QIK_VERTOPBUT`, type `EQikCtVertOptionButtonList`) and a horizontal option list (class `CQikHorOptionButtonList`, resource structure `QIK_HOROPBUT`, type `EQikCtHorOptionButtonList`).

You can also use option buttons in menus. In this case, you need to indicate the position of each button, relative to the other buttons in the sequence, by specifying one of `EEikMenuItemRadioStart`, `EEikMenuItemRadioMiddle` or `EEikMenuItemRadioEnd` in the `flags` field of each button's `MENU_ITEM` structure. Use `CEikMenuPane::SetItemButtonState()` to set which button is selected.

Check Boxes

Check boxes allow you to select or deselect a particular item by checking it or not. You can use these to allow multiple selections of options from a list, or as a single enable/disable type field for a particular application setting.

Check box controls use a control type of `EikCtCheckBox` and class `CEikCheckBox` (methods `SetState()` and `State()` respectively set and get the state of the checkbox). No resource structure is needed.

Choice List

A choice list allows you to select an item from a list of text options. The control displays a single item at a time, and pops up the list only while

the user is changing the control value. Choice lists are supported in UIQ and Series 80, but not in Series 60.

The choice list uses a control type of `EEikCtChoiceList`, class `CEikChoiceList`, and resource structure `CHOICELIST`. As with list boxes, a choice list allows you either to specify the array of choices in the resource file or to attach a list programmatically, using `CEikChoiceList::SetArrayL()`.

Combo Box

A combo box is similar to a choice list, except that the user can also enter text into the control field, as well as select an item from the list (hence the name: the control is a combination text editor and choice list).

The combo box uses control type `EEikCtComboBox`, class `CEikComboBox` and resource structure `COMBOBOX`.

As with choice lists and list boxes, you can set the array either in the resource or via `EEikComboBox::SetArrayL()`. Use class methods `SetTextL()` and `GetTextL()` to set text to and retrieve text from the control.

11.6.2 What Is a Control?

When implementing dialogs or using stock controls, you do not need to know the details of how the controls are implemented, or how they work. However, this knowledge is needed if you write your own custom controls and, since your application view is actually a custom control, you always have at least one (it can be a very simple one, but more complex ones are common) per application.

A control is a rectangular region on the screen that can receive user input and, in many cases, encapsulates some kind of user-editable data Every display region in an application is a control, including the main application view window. A control is implemented by a class that inherits from `CCoeControl`. The derived class overrides the methods of `CCoeControl` to draw itself on the screen and to handle any user input events that are destined for that control.

A control is considered a custom control if it is not one of the predefined control classes that are supplied with Symbian OS but is, instead, created by an application programmer by deriving from `CCoeControl`. The application view is always a custom control – and, in most cases, the only custom control in the application. The application view can be very simple (as it is in `SimpleEx` which simply displays a window with text in it) or it can be complex, possibly containing one or more child controls.

In addition to the application view, you may also want to create custom controls to allow the user to manipulate or display data in a way that none of the Symbian OS predefined controls do. Although the predefined

controls provide most of the functionality you will need, in some cases a custom control may be necessary.

11.6.3 Anatomy of a Control

A control has the following characteristics:

1. It inherits from `CCoeControl`.

   All controls inherit, directly or indirectly, from `CCoeControl`.

2. It draws itself.

   Controls are responsible for drawing themselves in response to redraw events. This is done by overriding `CCoeControl`'s `Draw()` method.

3. It can handle key events.

   A control may optionally handle user key events by implementing the `OfferKeyL()` method.

4. It can handle pointer events.

   Controls can optionally handle pointer events (like a touch on the touch screen) by implementing the `HandlePointerEvent()` function.

5. It encapsulates and provides access to user data.

   The control class usually (although not always) encapsulates some user data which can be simply displayed, or may be modifiable by the user via the control. The class derived from `CCoeControl` will provide its own specific methods to set and retrieve this data

Although many controls share all of these characteristics, only the first one is an essential requirement. For example, a simple application view, as in `SimpleEx`, is a window with text displayed in it – and thus only needs to implement the `Draw()` method to draw itself in response to redraw events.

11.6.4 Window-Owning Versus Lodger Controls

In order for its content to be displayed, each control must be associated with one of the windows that are managed by the window server. In the most straightforward case, a control is directly associated with a window, and is responsible for maintaining the whole of the window's area. Such a control is said to be a window-owning control. An application has at least one window owning control – the application view.

In their `ConstructL()` function, window-owning controls call `CreateWindowL()`, which causes the window to be created and registered with the window server. In addition, the control environment (CONE)

associates the window with the control whose call to `CreateWindowL()` created it.

For example, the application view in `SimpleEx` (as with all application views) is a window-owning control. The `ConstructL()` of `SimpleEx`'s view is as follows:

```
void CSimpleExAppView::ConstructL(const TRect& aRect)
  {
  CreateWindowL();
  SetRect(aRect);
  ActivateL();
  }
```

In addition to calling `CreateWindowL()`, `ConstructL()` calls `SetRect()` to specify the size of the control – and thus the window – which, in this case, ends up filling the entire application space. `ActivateL()` must always be called to indicate that the control is ready to draw itself.

In many cases, a control occupies – and is responsible for drawing – only a portion of a window, sharing the window's area with one or more other controls. Such controls are said to be non-window-owning, or lodger, controls. Instead of calling `CreateWindowL()` in their `ConstructL()`, lodger controls call `SetContainerWindowL()`, passing a reference to their parent window (which must, in turn, be owned by a window-owning control).

With the exception of application views, window-owning controls should be avoided where possible. Your application should minimize the number of windows it owns, since each window adds to the overhead of client/server messages between the application and the window server. Lodger controls are much more efficient in this regard, since only a single redraw event needs to be sent from the window server to the parent window. The control environment can then ensure that all controls that share that window are called upon to draw themselves, without any further client/server traffic.

11.6.5 Compound Versus Simple Controls

A compound control is a control that contains – and owns – other controls (known as child, or component, controls) within its region. A simple control contains no other controls.

A lodger control, by definition, is always a child of a compound control. Since a lodger can itself be a compound control, it is possible for a lodger's immediate parent control to be a lodger, and so not own a window. However, the ultimate parent of a compound control tree must always be a window-owning control. Although most compound controls contain only lodger controls, you can, if you have a good reason to do so, use component controls that themselves own windows.

A compound control always overrides the following functions from `CCoeControl`:

```
TInt CountComponentControls()
```

This function should return the number of controls that make up the compound control. The default base class implementation is to return 0.

```
CCoeControl* ComponentControl(TInt index)
```

This function should return the child component that corresponds to each index number passed to it.

The following shows an example implementation of these functions for a compound control that contains two other controls – a list box and an edit box:

```
TInt MyCompoundControl::CountComponentControl()
  {
  return 2;
  }

CCoeControl *ComponentControl(TInt index)
  {
  switch(index)
    {
    case 0:
      return iMyListbox;
    case 1:
      return iMyEditBox;
    }
  }
```

This code is seen most frequently in an application view, since the application view is usually the only custom control in the application. It is common for an application's main view to contain at least one child control (a list box in many cases) – which means that it is a compound control and would therefore need to override the above functions.

11.6.6 Drawing Controls

All controls are responsible for drawing themselves, by overriding the `Draw()` method of `CCoeControl`. For predefined Symbian OS controls, `Draw()` is implemented for you, but custom controls need to provide their own implementation.

Before looking at how to implement the `Draw()` function, let's take a brief look at how draws occur in the system, so that you can better understand when and how the system invokes your `Draw()` method.

Flow of a Redraw Event

As mentioned earlier, every application has a control environment, often referred to as a CONE, associated with it (it's an instance of a class called `CCoeEnv` that is created automatically for your application). One of a CONE's principal tasks is to interface with the window server. It is, for example, responsible for receiving a window draw event that is directed at your application, and ensuring that all the controls within the invalid region of the window are redrawn.

The following steps occur when a window managed by the window server needs to be redrawn:

1. The window server sends a redraw event.

   When a window needs to be redrawn, either because it has just been created, or because a region has become invalid, the window server sends a redraw command to the window server client that owns the window. In the case of an application, this client is the target application's control environment instance.

2. CONE calls the control's `HandleRedrawEvent()` method.

   On receipt of the redraw event from the window server, the CONE will look up the window-owning control that is associated with the window that needs to be redrawn (this will usually be a view or a dialog box). It then invokes that control's `CCoeControl::HandleRedrawEvent()` method, passing it the area of the window that needs redrawing.

3. The window control and its lodgers are drawn.

   `HandleRedrawEvent()` is implemented completely in the `CCoeControl` base class – the application programmer does not need to, and should not, override it. `HandleRedrawEvent()` will first call the control's `Draw()` method (which is overridden in a concrete control) passing it the rectangle that needs redrawing. `HandleRedrawEvent()` then calls the `Draw()` method of any non-window-owning component controls (and their components, if they are compound controls) whose areas overlap the region that needs redrawing. `HandleRedrawEvent()` iterates through these component controls by calling the parent control's `CountComponentControls()` and `ComponentControl()` functions.

Note that if a component control owns a window, it is not redrawn by this mechanism. This is for efficiency since, if the window associated with that control needs updating, the window server will send a separate redraw event.

Application-Initiated Drawing

An application will often need to force the drawing of one or more of its controls, for example, when the application knows that a control's content has changed. It should do this by calling the control's `DrawNow()` function, which invokes the control's `Draw()` method, along with the `Draw()` of all its component controls (in this case, including any window-owning controls).

An application should never call a control's `Draw()` method directly.

In complex cases, such as where the changes to a control's content come from more than one source, calling `DrawNow()` after each change has occurred may result in the control being redrawn unnecessarily often. In such a situation, you should consider calling the control's `DrawDeferred()` function, rather than `DrawNow()`. `DrawDeferred()` works by getting the window server to mark the control's window as needing to be redrawn. The window server will then generate a redraw event, and the control will be redrawn by the mechanism described in the previous section. The advantage of this technique is that there is an opportunity for multiple calls to `DrawDeferred()` to be handled in a single redraw of the control (at the expense of some additional client/server messaging overhead).

`SimpleEx` Draw()

Let's look again at the `Draw()` function in `SimpleEx`'s view class, and describe it in more detail:

```
void CSimpleExAppView::Draw(const TRect& ) const
  {
  CWindowGc& gc = SystemGc();
  const CFont* font;

  gc.Clear();

  font = iEikonEnv->TitleFont();
  gc.UseFont(font);
  TRect drawRect = Rect();

  TInt   baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;
  gc.DrawText(_L("Simple Example"),drawRect,
      baselineOffset,CGraphicsContext::ECenter, 0);

  gc.DiscardFont();
  }
```

`Draw()` takes a single argument of type `TRect&` that specifies the rectangular area of the screen that requires redrawing. In this case, I ignore it and just redraw the entire control (i.e. the view window). Of

course, if there is a significant advantage in doing so, you can use the passed rectangle to redraw only what is needed.

The first thing `Draw()` does in the example is to get a reference to the window's graphics context. The graphics context provides the necessary functions both to initialize the context and to draw to the window. It is discussed in more detail in the following section. The example's `Draw()` then clears the window, using the graphics context's `Clear()` function. It then creates a `TRect` data structure to represent the window's area (`Rect()` returns the region occupied by the control, in this case the entire view window), and the text 'Simple Example' is displayed in the center of that rectangle (the vertical justification being previously calculated and assigned to `baselineOffset`). Afterwards, to prevent a memory leak, the font is discarded.

Although this example is very simple, you can do much more complex screen drawing using the `CWindowGc` graphics context class, such as drawing points, lines, bitmaps and a variety of filled shapes. The next section gives a very basic overview of the drawing you can perform with the aid of the graphics context.

11.6.7 Drawing Using the Graphics Context

Like Windows, Symbian OS has the concept of a graphics context object that is used when drawing graphics and/or text to a window. In Symbian OS, this graphics context to write to a window is represented by the `CWindowGC` class. The first thing any control's `Draw()` method should do is get a reference to this context, using the control's `SystemGc()` method.

Coordinates and Areas Types

Before examining the context and drawing functions, let's look at some key classes that will be used extensively when drawing to the screen.

- A `TPoint` specifies an x and y location on the screen, via its `iX` and `iY` members. Screen coordinates in a control are relative to the top-left corner of the window that the control resides in.
- `TSize` is a simple class that contains integer members `iHeight` and `iWidth` to specify a height and width.
- `TRect` specifies a rectangular region on the screen. It has public members `iTl` and `iBr`, which are type `TPoint`, and define the top-left and bottom-right corners of the rectangle. `TRect` also has a few useful member functions that perform commonly needed operations on the rectangle.

Some useful functions of `TRect` include:

- `Move()` – This adds specified delta x and delta y values to the top-left and bottom-right coordinates.

- `Grow()` and `Shrink()` – Expand or contract the rectangle by specified amounts.
- `Intersects()` – This returns `ETrue` if the passed rectangle intersects this rectangle, `EFalse` otherwise.
- `Contains()` – This returns `ETrue` if the specified point is contained within the rectangle, `EFalse` otherwise.

Setting up to Draw

The graphics context provides methods to set up the characteristics of what will be drawn. This includes defining a pen or a brush, to be used when drawing points, lines and shapes. It also provides a method for assigning a font for the text-drawing functions to use.

Pen

The pen defines the color, style and thickness used when drawing points, lines, text and the outline of filled shapes. These pen characteristics are set using the graphics context methods `SetPenColor()`, `SetPenStyle()` and `SetPenSize()`.

For example:

```
CWindowGc& gc=SystemGc();
gc.SetPenColor(KRgbRed);  // set pen to red (predefined colors in gdi.h)
gc.SetPenStyle(ESolidPen);  // sets the pen to solid lines (the default)
TSize penSize(3,3);
gc.SetPenSize(penSize);  // set pen to be 3x3 pixels thick.
```

Brush

The brush defines the background color or pattern that is used to fill areas on the screen. The color and style of the brush are set using `SetBrushColor()` and `SetBrushSyle()`. If the brush style is set to use a pattern – that is, by calling `SetBrushStyle(TBrushStyle::EPatternedBrush)` – then a call to `UseBrushPattern()` will specify the bitmap to use as the fill pattern.

Font

A font is set using the `UseFont()` function, and from then on is used when drawing text. After you are finished with the font, you must call `DiscardFont()` to avoid a memory leak. Fonts are of type `CFont`.

Drawing

Drawing points and lines

You can draw points, lines and groups of lines using `CWindowGC` functions, which include `Plot()`, `DrawLine()`, `DrawLineTo()` and

DrawPolyLine() functions. These functions draw the points and lines using the context's pen.

Setting Current Positions

MoveTo() is used for setting the current position. Some functions use the current position as the starting point for their action. For example, DrawLineTo() draws a line from the current position to a single specified point, in contrast with DrawLine(), which takes both endpoints of the line as its arguments.

Drawing Filled shapes

Functions to draw filled shapes include DrawRect(), DrawEllipse(), DrawRoundRect(), DrawPolygon() and DrawPie(). A filled shape's outline is drawn using the characteristics of the pen, and filled using the characteristics of the brush.

Drawing Text

To draw text on the screen you need to call UseFont() to set the font, set the pen color/style needed to draw the text and call DrawText() to output the text.

Two forms of DrawText() exist. One draws text justified within a rectangular region. This is the one used in SimpleEx. The other draws the text at a specific position on the screen, as the following example shows:

```
font = iEikonEnv->TitleFont();
gc.UseFont(font);
TPoint position(10,10);
gc.DrawText(KSomeText,position);
```

This will draw the string in KSomeText, with its bottom-left corner at position 10,10 in the window.

Drawing Bitmaps

You can draw bitmaps to the screen using the DrawBitmap() function. Also, as mentioned earlier, you can specify a bitmap to be used as the fill pattern for shapes by setting the brush style to TBrushStyle::EPatternedBrush and making a call to UseBrushPattern().

11.6.8 How User Input Is Handled

A control can handle two types of user input: keyboard and pointer. Keyboard events occur when a hardware key is pressed or, in the case of

UIQ, a key icon on the virtual keyboard is selected. Pointer events occur when the touch screen is touched (this only applies to UIQ, since Series 60 and Series 80 devices have no touch screen).

The Control Stack for Key Handling

The control stack is a data structure within an application's control environment. It lists, in priority order, all controls within the application that should be offered key events. An application adds controls to this control stack using the `AddToStackL()` method of the `CCoeAppUi` application UI base class, and removes them by calling `CCoeAppUi`'s `RemoveFromStackL()` method.

The control stack lists everything that can receive user-key input for your application – including your application's views, dialog boxes and menus (for hotkeys). However, you need only worry about adding and removing your application view controls to the control stack, since the framework automatically handles the addition and removal of the control stack entries for dialogs and menus.

Handling Keys in a Control

When an application receives a key-press event, the control environment invokes `CoeControl`'s virtual `OfferKeyEventL()` method in every control on the control stack, in priority order, until the key is consumed.

A control should override `OfferKeyEventL()` to perform any needed key handling. If the particular key passed to the method is handled by that control, then the method should return `EKeyConsumed`. Otherwise, it should return `EKeyNotConsumed`, so that another control can be given the chance to handle the event.

In a view control, it is usual to add only the view control itself to the control stack. The view's `OfferKeyEventL()` should, in turn, invoke the `OfferKeyEventL()` method of each relevant component control, in order to distribute the handling of the key event in a manner that is appropriate for the application.

Virtual Keyboard and Handwriting Input

UIQ phones, such as the Sony Ericsson P900, allow text input by means of handwriting recognition, or via an optional on-screen keyboard. These are handled by what are known as front end processors (FEPs). In order for a custom control – including your application view – to receive input from these FEPs, your control must override `CCoeControl`'s `InputCapabilities()` method, to return the types of input it can handle. An FEP may call this function to determine what types of event it can send to the control. The default implementation of `InputCapabilities()`, in

CCoeControl, is to return TCoeInputCapabilities::ENone – so, by default, your control will not receive input from such FEPs.

Refer to your SDK documentation for the various types of input you can specify via the InputCapabilities() function. The following example returns TCoeInputCapabilities::EAllText to request all types of text input from both the virtual keyboard and the handwriting recognition FEPs:

```
TCoeInputCapabilities CMyView::InputCapabilities() const
  {
  TCoeInputCapabilities
  capabilities(TCoeInputCapabilities::EAllText);

  return capabilities;
  }
```

Pointer Events

To handle pointer events in your custom control, override HandlePointerEventL() in CoeControl This method is defined as:

```
virtual void HandlePointerEventL(const TPointerEvent& aPointerEvent)
```

where TPointerEvent is a class that contains the type and position of the pointer event. For example, the following code will display the message 'Hit' when the user selects a point within the (previously defined) TRect named iTarget:

```
void CMyView::HandlePointerEventL(const TPointerEvent& aPointerEvent)
  {
  TInt dx=0, dy=0;

// aPointerEvent.iPosition is a TPoint class
// aPointerEvent.iPosition.iX contains x coordinate of pointer selection
// aPointerEvent.iPosition.iY contains x coordinate of pointer selection
// Assume iTarget is a TRect of some target square, the below will print
// an info message
// if you tap the touch screen.

  if (iTarget.Contains(aPointerEvent.iPosition))
    {
    User::InfoPrint(_L("Hit"));
    return;
    }
  }
```

In addition to the coordinates of the screen touch contained in TPointerEvent's iPosition member, member iType contains the type of touch screen event that occurred. For example, iType will be equal to

EButton1Down when the pen makes contact, EButton1Up when the pen breaks contact, and EDrag if the stylus is moved while in contact with the screen.

11.7 View Architecture

Up to this point, I have described applications that have only a single, CCoeControl-derived view, created and started by the application UI class upon construction. Symbian OS, however, permits you to define multiple views for a single application, and allows you to switch between them. This means that you can, for example, present application data to the user in various different ways. Each view is registered with its own identifier, and then the application can switch between them in response to user events. It is also possible for one application to switch to the view of another.

In this section, I'll briefly introduce the view architecture, showing only the basic principles of how to create and use views. You should refer to the SDK for further information and examples.

11.7.1 How to Create Views

For each application view, you still need a class derived from CCoeControl (i.e. a custom control) to display the data. To create views that can be used with the view architecture, your view class needs to inherit from an interface class called MCoeView. MCoeView provides virtual functions that are called by the view architecture to register a view, get its identifier and switch from one view to another. In most cases, you define each view to inherit from both CCoeControl and MCoeView (although you can, if necessary, derive a class from MCoeView only, and keep your view control as a separate object).

Creating an application with multiple views will normally involve the following steps:

- Create your view custom controls and additionally derive them from MCoeView.
- In each of your view classes, override the MCoeView virtual functions: ViewID(), ViewActivatedL(), ViewDeactivated() and ViewConstructL().
- In your application, register all your views using the application UI class's RegisterViewL() method.
- Set a default view using UI class's SetDefaultView() method.
- Switch between views as needed with the application UI class ActivateViewL().

MCoeView Methods to Override

Let's now look at each of the MCoeView virtual functions that you need to override:

```
virtual TVwsViewId ViewId() const=0
```

Override this class to return the identifier assigned to your view. This identifier is a combination of your application UID and a UID assigned to the view itself. For example:

```
const TUid KUidMyApp = { 0x01001000 };
const TUid KUidMyAppView1 = { 0x0100A000 };

TVwsViewId CMyAppView1::ViewId() const
  {
  TVwsViewId KViewId {KUidMyApp,KUidMyAppView1 } ;
  return KViewId;
  }
```

```
virtual void ViewConstructL()=0
```

This function is called to construct your view. Normally this is where you actually construct your view's component controls, although you do not yet make them visible.

```
virtual void ViewActivatedL(const TvsViewId& aPrevView, TUid aMessageId,
        const TDesC8& aViewMessage)=0
```

This is called by the view architecture to display your view, passing (in aPrevView) the identifier of the view just switched from, as well as a possible message (for example, to specify a particular data record) that the view can use to help it work out what to display.

In most cases, this method will make the control visible (by calling CoeControl::MakeVisible()), add the view's control to the control stack, and draw the view.

```
virtual void ViewDeactivated()=0
```

Override this function to do what is necessary to deactivate your view, before the view architecture switches to another view. Usually, you make your control invisible here, as well as removing the view's control from the control stack.

Registering Your Views and Setting a Default

In the UI class (usually in ConstructL()), you register all your views using CEikAppUi's RegisterView(MCoeView& aView) method and

call SetDefault() to set one of your views as the default active view. For example:

```
void CMyViewAppUi::ConstructL()
  {
  BaseConstructL();
  ...
  iMyView1 = new (ELeave) CMyView1;
  iMyView2 = new (ELeave) CMyView2;

  RegisterViewL(*iMyView1);
  RegisterViewL(*iMyView2);

  SetDefaultViewL(*iMyView1);
  }
```

RegisterViewL() will call your view's ViewId() method and register its identifier. It also calls your view's ViewConstructL() method.

Switching Between Views

Use the ActivateViewL() method, in the application UI's CCoeAppUI base class, to switch to a view. This is normally done in response to a user event, such as a menu selection or a button press.

There are two versions of ActivateViewL(), the first being prototyped as:

```
void CCoeAppUi::ActivateViewL(const TVwsViewId& aViewId)
```

The following example illustrates one of the more common ways it is used:

```
const TVwsViewId KMyViewId1 {KUidMyApp,KUidMyAppView1 };
const TVwsViewId KMyViewId2 {KUidMyApp,KUidMyAppView2 };

void CMyAppUi::HandleCommandL(TInt aCommand)
  {
  switch(aCommand)
    {
    ...
    case EMyAppDisplayView1:
      ActivateViewL(KMyViewId1);
      break;
    case EMyAppDisplayView2:
      ActivateViewL(KMyViewId2);
      break;
    ...
    }
  }
```

The second version of ActivateViewL() is prototyped as:

```
void CCoeAppUi::ActivateViewL(const TVwsViewId& aViewId, TUid aMessageId,
        const TDesC8& aMessage)
```

and additionally passes the view a message, defined in the last two arguments.

Both versions of ActivateViewL() call your view's ViewActivatedL() method.

Deregistering a View

Upon closing your application, you should deregister your views using the application UI's DeregisterView() method. For example:

```
CMyAppUi::~CMyAppUi()
  {
  DeregisterView(*iMyView1);
  DeregisterView(*iMyView2);
  delete iMyView1;
  delete iMyView2;
  }
```

11.7.2 Series 60 Views

The previous sections discussed the Symbian OS generic view architecture, and you can use this directly in UIQ and Series 80. However, Series 60 provides another layer of classes, in its Avkon UI layer, that you use to implement views.

In Series 60, you should derive your application UI class from CAknViewAppUi when using views. Also, the views themselves are derived from CAknView. I will not go into the details of implementing views on Series 60, but the concepts are basically the same as those described above. Also, a resource structure exists for views in Series 60, defined as below in avkon.rh:

```
STRUCT AVKON_VIEW
  {
  LLINK hotkeys=0;
  LLINK menubar=0;
  LLINK cba=0;
  }
```

As you can see, you can associate hotkeys, a menu bar and softkeys with a view, and they will be enabled whenever the view becomes active. You associate a particular view with an AVKON_VIEW resource by calling CAknView's BaseConstructL from your view's ConstructL() function. For example, if you have a view resource defined as:

```
// In resource file
// Assume the hot key, menu and cba resources
// are defined elsewhere in the file

RESOURCE AVKON_VIEW r_my_view1
```

```
  {
  hotkeys=r_my_hot_keys_for_view1;
  menubar=r_my_menu_for_view1;
  cba=r_my_softkeys_for_view1;
  }
```

then you can associate it with your `CAknView`-derived view by:

```
void CAknMyView1::ConstructL()
  {
  BaseConstructL(R_MY_VIEW1);
  }
```

The Series 60 SDK has many examples that use this view architecture and it is very instructive to review them.

11.8 Application Icon and Caption

Symbian OS associates an icon as well as a caption with your application. You can supply an application caption and one or more icon bitmaps to represent the application by including an application information (`aif`) file in the target's application directory, along with the `app` file. If an application's `aif` file does not exist, the system uses a default icon and caption.

This section provides only a brief introduction to using `aif` files to supply your application with icons and captions. You should refer to the SDK documentation for more detail, as well as examples.

To generate an `aif` file, you need to do the following:

- Create your icon bitmaps using an image editor.
- Write an `aif` resource file defining the caption, number of icons and other information.
- Add an `aif` line to your `mmp` file to specify which `aif` resource file and icon bitmaps should be used to generate the `aif` file.

11.8.1 Creating the Icon Bitmaps

To cope with the different situations in which application icons may be displayed, each application needs to be supplied with icon bitmaps in two or more sizes. The required number and sizes (in pixels) vary with the UI platform, and are listed below.

- UIQ: 32×32, 20×16
- Series 60: 44×44, 42×29
- Series 80: 64×50, 50×50, 25×20

You can, in some cases, replace the multiple icons by a single icon of intermediate size. If you do this, the system will dynamically scale the

icon to fit the different situations, but this is not recommended, since the appearance of your application will suffer.

Icon bitmaps can be created with any paint program. Each icon image in Symbian OS consists of two bmp files, one defining the image itself, and the other defining a two-color mask for the bitmap. For every black pixel in the mask bitmap, the corresponding pixel in the image bitmap is displayed – the rest are not displayed, thus giving a transparent effect for those pixels. So, for every icon size, you need both an image bitmap and a mask bitmap.

11.8.2 Creating the AIF Resource File

An aif resource file is simply an rss file containing a single AIF_DATA resource structure. It specifies the application's caption (in one or more languages), the UID and the number of icons, as well as a few other settings. Here is an example:

```
#include <aiftool.rh>
RESOURCE AIF_DATA
  {
  app_uid = 0x01001000;
  caption_list =
    {
    CAPTION
      {
      code = ELangEnglish;
      caption = "My Application";
      }
    };
  num_icons = 2;
  }
```

Note that caption_list is an array, so you can specify additional CAPTION items, each containing the appropriate language code and caption text for a different language.

AIF_DATA has a few more attributes – whether or not the application can be embedded; whether it is to be launched in the foreground or not; and you can associate your application with MIME types. You can find a description of their use in the SDK documentation. The definition of the AIF_DATA structure is in aiftool.rh.

11.8.3 Building the AIF file

Symbian OS provides an AIF build tool called aiftool which takes the aif resource file and bitmaps as input and creates an aif file to be downloaded to the phone, along with the application. Although you can use this tool directly, it is better to add an AIF line to your mmp file, so that the tool is invoked automatically, as part of the normal build process.

Here is an example `AIF` line for Series 60:

```
AIF MyApp.aif ..\aif MyAppAif.rss c12 myapp_l.bmp myapp_lm.bmp \
    myapp_s.bmp myapp_sm.bmp
```

The first argument specifies the name of the output `aif` file, the second, the location of the source files and the third, the name of the `aif` resource file. The value `c12` indicates that the icon bitmaps use a color depth of 12 bits. Following that is a list of bitmap pairs (image and mask) for each of the differently sized icons (the number of pairs should match the number of icons specified by the `num_icons` attribute in the AIF resource file).

In your package file, make sure you copy the `aif` file to the phone's application directory, along with the `app` file.

Appendix 1

Specifications of Symbian OS Phones

This appendix contains notes on the UI, screen size, and other attributes relevant to application developers of currently available, open Symbian OS phones. Further technical information and an up-to-date list of phones can be found at: ***www.symbian.com/phones***.

Please note that this is a quick guide to Symbian OS phones, some of which are not yet commercially available. The information contained within this appendix was correct at the time of going to press. For full, up-to-date information, refer to the manufacturer's website.

Arima U300

OS Version	Symbian OS v7.0
UI	UIQ 2.1
Built-in memory available	32 MB
Storage media	Mini SD/MMC
Screen	208×320 pixels 65,536 colors
Data input methods	Keypad Pointing device
Camera	1280×960 resolution 4x digital zoom
Network Protocol(s)	GSM E900/1800/1900 HSCSD GPRS
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML

BenQ P30

OS Version	Symbian OS v7.0
UI	UIQ 2.1
Built-in memory available	32 MB
Storage media	MMC and SD
Screen	208×320 pixels 65,536 colors
Data input methods	Keypad Pointing device
Camera	640×480 resolution
Network Protocol(s)	GSM E900/1800/1900 HSCSD GPRS (Class 10, B)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML

Motorola A920/A925

OS Version	Symbian OS v7.0
UI	UIQ 1.0
Built-in memory available	8 MB
Storage media	MMC and SD
Screen	208×320 pixels 65,536 colors
Data input methods	Small number of keys Pointing device
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 4) 3G
Connectivity	Infrared Bluetooth (A920 No/A925 Yes) USB Serial
Browsing	xHTML

Motorola A1000

OS Version	Symbian OS v7.0
UI	UIQ 2.1
Built-in memory available	24 MB
Storage media	Triflash-R
Screen	208×320 pixels 65,536 colors
Data input methods	Small number of keys Pointing device
Camera	1280×960 resolution 4x digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2100 HSCSD GPRS (Class 10, B) EDGE 3G
Connectivity	Bluetooth USB
Browsing	WAP xHTML

Nokia 3230

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	6 MB
Storage media	RS-MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	1.3 megapixel resolution 3x digital zoom
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10) EDGE
Connectivity	Bluetooth Infrared USB
Browsing	WAP 2.0 xHTML/HTML

Nokia 3600/3650

OS Version		Symbian OS v6.1
UI		Series 60 v1
Built-in memory available		3.4 MB
Storage media		MMC
Screen		176×208 pixels 4096/65,536 colors
Data input methods		Keypad
Camera		640×480 resolution
Network Protocol(s)	3600	GSM 850/1900
	3650	GSM 900/1800/1900
		HSCSD
		GPRS (Class 8; B)
Connectivity		Infrared Bluetooth
Browsing		WAP 1.2.1 xHTML

Nokia 3620/3660

OS Version		Symbian OS v6.1
UI		Series 60 v1
Built-in memory available		4 MB
Storage media		MMC
Screen		176×208 pixels 4096/65,536 colors
Data input methods		Keypad
Camera		640×480 resolution
Network Protocol(s)	*3620*	GSM 850/1900
	3660	GSM 900/1800/1900 HSCSD GPRS (Class 8; B)
Connectivity		Infrared Bluetooth
Browsing		WAP 1.2.1 xHTML

Nokia 6260

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	3.5 MB
Storage media	MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution 4x digital zoom
Network Protocol(s)	GSM 900/1800/1900 GSM 850/1800/1900 HSCSD GPRS (Class 6, B)
Connectivity	Infrared Bluetooth USB
Browsing	HTML xHTML WAP 2.0

Nokia 6600

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	6 MB
Storage media	MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution 2x digital zoom
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 8; B and C)
Connectivity	Infrared Bluetooth
Browsing	WAP 2.0 xHTML

Nokia 6620

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	12 MB
Storage media	MMC
Screen	176×220 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 850/1800/1900 GPRS (Class 8; B) HSCSD EDGE
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML

Nokia 6630

OS Version	Symbian OS v8.0
UI	Series 60 v2.6
Built-in memory available	3.5 MB
Storage media	MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	1280×960 resolution 6x digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2000 GPRS (Class 10, B) EDGE 3G
Connectivity	Bluetooth USB
Browsing	WAP 2.0 HTML xHTML

Nokia 6670

OS Version	Symbian OS v7.0s
UI	Series 60
Built-in memory available	8 MB
Storage media	RS-MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	1152×864 resolution 4x digital zoom
Network Protocol(s)	GSM 850/900/1800/1900 GPRS (Class 6, B) HSCSD
Connectivity	Bluetooth USB
Browsing	WAP 2.0 HTML xHTML

Nokia 6680/6681/6682

OS Version	Symbian OS v8.0
UI/Category	Series 60 v2.6
Built-in memory available	10 MB
Storage media	RS-MMC
Screen	176×208 pixels 262,144 colors
Data input methods	Keypad
Camera	*Front* 1280×960 resolution 6x digital zoom *Back* 640×480 pixels 2x digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2100 EDGE GPRS (Class 10, B)
Connectivity	Bluetooth USB
Browsing	WAP 2.0 xHTML/HTML

Nokia 7610

OS Version	Symbian OS v7.0s
UI/Category	Series 60 v2.1
Built-in memory available	8 MB
Storage media	RS-MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	1152×864 resolution 4x digital zoom
Network Protocol(s)	GSM 850/900/1800/1900 GPRS (Class 10; B)
Connectivity	Bluetooth USB
Browsing	WAP 2.0 xHTML

Nokia 7710

OS Version	Symbian OS v7.0s
UI	Series 90
Built-in memory available	80 MB
Storage media	MMC
Screen	640×320 pixels 65,536 colors
Data input methods	Keypad Pointing device
Camera	1152×864 resolution 2x digital zoom
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10) EDGE
Connectivity	Bluetooth USB
Browsing	HTML xHTML

Nokia 9300

OS Version	Symbian OS v7.0s
UI	Series 80
Built-in memory available	80 MB
Storage media	MMC
Screen	Two displays, both 65,536 colors *main screen*: 200×640 pixels *secondary screen*: 128×128 pixels
Data input methods	Keypad Full keyboard Customizable buttons beside screen
Camera	No
Network Protocol(s)	GSM E900/800/1900 EDGE GPRS (Class 10, B) HSCSD
Connectivity	Infrared Bluetooth USB
Browsing	HTML 4.01 xHTML

Nokia 9500

OS Version	Symbian OS v7.0s
UI	Series 80
Built-in memory available	80 MB
Storage media	MMC
Screen	Two displays, both 65,536 colors *main screen*: 200×640 pixels *secondary screen*: 128×128 pixels
Data input methods	Keypad Full keyboard Customizable buttons beside screen
Camera	640×480 resolution
Network Protocol(s)	GSM 850/900/1800/1900 HSCSD GPRS (Class 10, B) EDGE WiFi
Connectivity	Infrared Bluetooth USB
Browsing	HTML 4.01 xHTML

Nokia N-Gage

OS Version	Symbian OS v6.1
UI	Series 60 v1
Built-in memory available	4 MB
Storage media	MMC
Screen	176×208 pixels 4096 colors
Data input methods	Keypad
Camera	No
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 6, B and C)
Connectivity	Bluetooth USB
Browsing	WAP 1.2.1 xHTML

Nokia N-Gage QD

OS Version	Symbian OS v6.1
UI	Series 60 v1
Built-in memory available	3.4 MB
Storage media	MMC
Screen	176×208 pixels 4,096 colors
Data input methods	Keypad
Camera	No
Network Protocol(s)	GSM 850/900/1800/1900 HSCSD GPRS (Class 6, B)
Connectivity	Bluetooth
Browsing	WAP 1.2.1 xHTML

Nokia N70

OS Version	Symbian OS v8.1a
UI/Category	Series 60 v2.8
Built-in memory available	31 MB
Storage media	RS-MMC
Screen	176×208 pixels 262,144 colors
Data input methods	Keypad
Camera	*Front* 1600×1200 resolution 20x digital zoom *Back* 640×480 pixels 2x digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2100 EDGE GPRS (Class 10, B)
Connectivity	Bluetooth USB
Browsing	WAP 2.0 xHTML/HTML

Nokia N90

OS Version	Symbian OS v8.1a
UI/Category	Series 60 v2.8
Built-in memory available	30 MB
Storage media	RS-MMC
Screen	*Internal* 352×416 pixels 262,144 colors *Cover* 128×128 pixels 65,536 colors
Data input methods	Keypad
Camera	1600×1200 resolution 20x digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2100 EDGE GPRS (Class 10, B)
Connectivity	Bluetooth USB
Browsing	WAP 2.0 xHTML/HTML

Nokia N91

OS Version	Symbian OS v8.1a
UI/Category	Series 60 v2.8
Built-in memory available	30 MB (and) 4 GB devoted to media storage
Storage media	RS-MMC
Screen	176×208 pixels 262,144 colors
Data input methods	Keypad
Camera	1600×1200 resolution 20x digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2100 EDGE GPRS (Class 10, B)
Connectivity	Bluetooth USB
Browsing	WAP 2.0 xHTML/HTML

Panasonic X700

OS Version	Symbian OS v7.0s
UI/Category	Series 60
Built-in memory available	4 MB
Storage media	miniSD
Screen	176×280 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM E900/1800/1900 GPRS (Class 10; B)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML

Panasonic X800

OS Version	Symbian OS v7.0s
UI/Category	Series 60 v2.0
Built-in memory available	8 MB
Storage media	miniSD
Screen	176×280 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM E900/1800/1900 GPRS (Class 10)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML

Sendo X

OS Version	Symbian OS v6.1
UI	Series 60
Built-in memory available	12 MB
Storage media	MMC and SD
Screen	176×220 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 GPRS (Class 8; B)
Connectivity	Infrared Bluetooth USB Serial
Browsing	WAP 2.0 xHTML

Siemens SX1

OS Version	Symbian OS v6.1
UI	Series 60
Built-in memory available	3.5 MB
Storage media	MMC
Screen	176×220 pixels 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10; B)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML

Sony Ericsson P800

OS Version	Symbian OS v7.0
UI	UIQ 2.0
Built-in memory available	12 MB
Storage media	Sony MS Duo
Screen	208×320 pixels (Flip Open); 208×144 pixels (Flip Closed) 4,096 colors
Data input methods	Flip keypad Pointing device Jog dial
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 8, B)
Connectivity	Infrared Bluetooth USB support
Browsing	WAP 2.0 xHTML

Sony Ericsson P900

OS Version	Symbian OS v7.0 (+ security updates and MIDP2.0)
UI	UIQ 2.1
Built-in memory available	16 MB
Storage media	Sony MS Duo
Screen	208×320 pixels (Flip Open); 208×208 pixels (Flip Closed) 65,536 colors
Data input methods	Flip keypad Pointing device Jog dial
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10; B)
Connectivity	Infrared Bluetooth USB support
Browsing	WAP 2.0 xHTML

Sony Ericsson P910

OS Version		Symbian OS v7.0
UI		UIQ 2.1
Built-in memory available		64 MB
Storage media		Memory Stick Duo Pro
Screen		208×320 pixels 262,000 colors
Data input methods		Flip keypad Thumb keyboard Pointing device Jog dial
Camera		1152×864 resolution 4x digital zoom
Network Protocol(s)	*P910i*	GSM 900/1800/1900
	P910c	GSM 900/1800/1900
	P910a	GSM 850/1800/1900
Connectivity		Bluetooth Infrared USB support
Browsing		WAP 2.0 cHTML

Appendix 2

Security in Symbian OS v9

Version 9 of Symbian OS is a major progression, specifically geared to target hundreds of millions of mid-range phones. To facilitate this, some changes have been made to the very core of Symbian OS.

One aspect of these changes is the Platform Security enhancements. These represent an evolution of the existing perimeter security model of Symbian OS, adding support for vital concepts such as data protection, or 'caging', and restricting some API usage, thus helping to ensure the integrity of the platform.

Platform Security Concepts

The following sections explain concepts and terms that are central to an understanding of platform security.

Capabilities

A *capability* is a unit of protection. An API within Symbian OS that needs to be protected has a capability associated with it, to provide that protection.

The capabilities that an application requires must be listed in its project definition (`mmp`) file. This data is used by the Software Install (`SWInstall`) component of Symbian OS, Symbian OS itself, and also certification programs such as Symbian Signed, to check what functionality an application should be allowed to access.

For details of the various capabilities, and the functionality that they protect, have a look at the Platform Security technical papers on the Symbian developer web site (***www.symbian.com/developer/index.html***).

Permissions

Permissions are used to determine whether an application can access a capability-protected API. Within Symbian OS, APIs divide into two distinct groups:

- unprotected APIs, which do not require permission to install applications that use them (some 60% of all APIs fall into this division)
- capability-protected APIs, which require permission in order to install applications that use them.

Where required, permission is granted when the application is installed. Two types of permission can be given to allow the application to use the capability-protected API: blanket permission or single-shot permission.

- *Blanket permission* for a particular capability allows the installed application to have unrestricted access to all APIs protected by that capability. Blanket permission may be obtained either by a certification program such as Symbian Signed or by user authorization at install time, depending on the capabilities for which the permission is required
- *Single-shot permission* to use a capability-protected API may be granted to an application that doesn't have blanket permission for that capability. Single-shot permission is granted on a one-off basis, permission being requested directly from the end-user each time that action is performed. For example, this may be used when sending a text message.

Authorization

Authorization is the process of confirming that an application is trusted enough to use the APIs protected by a particular capability. Authorization is carried out by a certification process, normally Symbian Signed (***www.symbiansigned.com***).

For some capability-protected APIs, Symbian OS will only grant access permission to applications that are authorized. The remaining capability-protected APIs (called 'Unsigned Sandboxed' APIs) do not require authorization, but permission is still needed to access them. This permission can be obtained in two ways:

- The application can be authorized anyway, in which case blanket permission is granted to the application for the authorized capabilities.
- The user can be asked to give their authorization for blanket or single-shot permission to access that capability at load time.

This is shown in the diagram below.

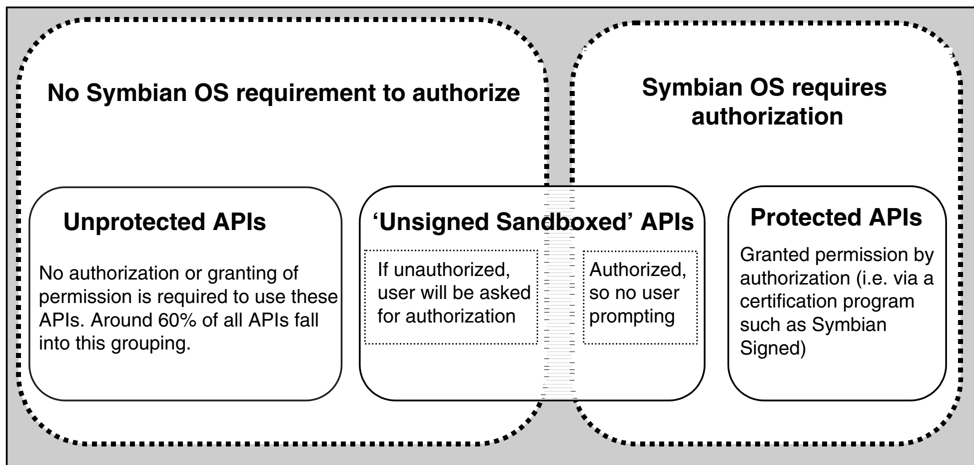

Figure A2.1 Authorization and Permissions for Symbian OS APIs

Note that Symbian OS licensees may choose to implement single-shot permissions for selected capabilities on their devices.

Secure Identifier

In a secure environment, a server needs to know which applications are permitted to access its APIs. The capabilities model avoids the need for specific identification of applications, so a server can generally control access to its APIs without having to know who is calling them. However,

there is sometimes a need to uniquely identify an application, for example when data needs to be tied to a specific application. To achieve this, the Secure Identifier (SID) has been defined, allowing Symbian OS to:

- enable data caging (see below)
- differentiate between signed and unsigned applications
- protect access to file system areas on the phone that are used when upgrading content.

From Symbian OS v9, all executables must contain a Secure Identifier that is guaranteed to be locally unique. By default, this matches the UID3 value which has always been required, but it is possible to explicitly set a SID for the application by specifying it in the application's `mmp` file using the keyword SECUREID. UIDs will be issued by ***www.symbiansigned.com***.

Data Caging

Platform Security not only implements protection for APIs, but also provides a facility for protecting private data. Coupled with the necessity to separate code and data (preventing code in the data space being executed), this has resulted in a reorganization of the file system to implement data partitioning. By 'caging' processes into a specific part of the filing system, privacy of data is ensured.

The filing system now has the following structure:

`/sys/`	Restricted system area, accessible only to applications with Trusted Computing Base (TCB) capability, which provides access to all the hardware and software on the platform. It is granted only to the kernel, the file server and SWInstall. Executables are placed in, and can only be run from, `/sys/bin/`.
`/private/`	Contains the private data for all applications, held in the directory `/private/<SID>/`. If the subdirectory `import/` already exists, SWInstall may write data into this directory during installation of any application.
`/resource/`	Contains public data, but is read-only for applications without TCB capability.

This structure is enforced by the software installer when new applications are installed, or existing applications are updated. Since only the software installer may change executables in `/sys/bin/` on the internal drive, executables cannot be tampered with by other software.

Note that a signed application may still choose to place its related data in the public area, to make it accessible to other applications.

Using Capabilities

As we've seen, capabilities ensure that only authorized applications can access protected APIs, and the APIs that the application can access are determined by which capabilities have been granted to the application.

For example, an application with LocalServices and Messaging capabilities would be able to request the messaging server to send data by infrared, but (without the NetworkServices capability) would be prevented from sending an SMS.

Capabilities are granted to applications (i.e. processes) only. Where an application (an `exe` file) needs to load a library (a `DLL` file), the library is loaded only if it has been authorized for (at least) the same capabilities that the calling application has been granted. Once loaded, the `DLL` inherits the capabilities of the calling executable.

The following example illustrates this.

The application `P.EXE` is linked to the library `L1.DLL`.

The library `L1.DLL` is linked to the library `L0.DLL`.

Case 1:

`P.EXE` holds Cap1 and Cap2

`L1.DLL` holds Cap1 and Cap3

`L0.DLL` holds Cap1 and Cap2.

The load fails because `P.EXE` cannot load `L1.DLL` (no Cap2).

Case 2:

`P.EXE` holds Cap1 and Cap2

`L1.DLL` holds Cap1, Cap2 and Cap3

`L0.DLL` holds Cap1, Cap2 and Cap3 and Cap4

The load succeeds and the new process is assigned Cap1 and Cap2.

Index

1G networks 3
2G networks 3–5
2.5G networks 3, 5–6
3G network protocol 398–406
3G networks 3, 5–6
4G networks 3
a (function arguments) prefixes, concepts 110–11
abld 25–9, 50–1, 104, 120–5, 129–31, 132–6
abld freeze 121–3, 129–36
abstract classes, C++ concepts 81–2, 86
ActivateL 339–40, 358–9, 378
ActivateViewL 387–91
active objects
  see also asynchronous functions
  active scheduler 69, 237–65
  blocks 257, 268
  CActive 69, 237–63
  cancellations 239, 241, 247–9, 251–8
  class creation 238–41, 250–6
  complexity problems 256–8
  concepts 69, 212–13, 214–19, 232, 237–65, 305–24
  construction 239–40, 250–6
  creation 238–41, 250–6
  CTimer 259–60, 285
  definition 237
  examples 249–56, 305–24
  high-level view 237–8
  issues 256–8
  network programming 305–24
  non-preemptive multitasking model 237–8, 256
  outstanding requests 258
  priorities 247
  removal 248–9
  requestor implementation 240–1, 250–6
  servers 267–8, 275–6, 277–83
  sockets 305–24
  stray-signal exceptions 257–8
  thread uses 258–65
  tips 256–8
  uses 237–8, 258–9
active scheduler
  concepts 69, 237–65, 277–83, 305–24
  customization 245–6
  definition 237, 241–2
  detailed workings 243–4
  error handling 245, 246–7, 257–8
  event-loop pseudo-code 243–4
  examples 249–56, 277–83
  GUI applications 242, 257, 311–12, 322
  high-level view 238
  installation 242–5, 277–83
  leave/trap mechanism 246–7, 250–6, 257–8, 278–80
  starting 242–5, 277–83
ActiveLogging 264
activelog.txt 264
ActivePerl 518 21
Add 200–1, 237–41, 244–5, 257–8
Add Field 361–6
AddFirst 200
AddItemL 361–6
AddLast 200
addresses
  IP (Internet Protocol) 295–6, 301–4, 307–8, 318–20
  memory 61–2, 223–8
  MMU 62, 64–5, 224–5
  physical/virtual memory addresses 61–6, 224–8
AddText 274–87
AddToStack 385
Adjust 225
After 109, 214–15, 238, 257, 259–63
aif files 322–3, 391–3
AIF_DATA 392
aiftool.rh 392
aknapp.h 33–8
aknappui.h 33–8
akndoc.h 33–8
alarms 79
AlertWin 45–9, 368
Alloc 171–2
AllocL 172
AllocLC 98–9, 172
animation plug-ins
  see also graphics
  concepts 75
APIs see application programming interfaces

app files 70, 75, 122–3, 322–93
apparc.lib 121–3, 333–40
AppDllUid 49, 335–40
Append 157–8, 161, 164–6, 168–9, 179–81, 196–9, 309–24
AppendFill 179–80
AppendFormat 181
appendices 395–429
AppendJustify 179–80
AppendNum 179–80
application engines, concepts 79–80
application framework
  see also graphical user interface framework
  concepts 56, 72–5, 325–93
application programming interfaces (APIs) 55–6, 66–7, 76–80, 92–5, 107, 130–1, 151–201, 288, 325–93, 426–9
  base libraries 55–6
  BSD socket API 77, 293–4, 297–304, 310–11
  data organization classes 154–201
  platform security 426–9
  sockets 77, 293–4, 297–324
  subsessions 288
  TCP/IP 293–324
  types 66–7, 79–80, 92–5, 426
  User 82, 92–5, 104
application protocols, concepts 79–80
application services/engines/protocols, concepts 56, 79–80
application views 31–49, 326–32, 334–93
application-initiated drawing, drawing controls 381–2
application-specific cleanups, concepts 95–6
applications
  see also executables; processes; software
  availability 1, 10, 13
  captions 391–3
  classes 31–49, 332–93
  communications architecture 76–9
  components 31–49, 332–93
  concepts 1–2, 10, 13, 31–49, 72–5, 136–47, 332–93
  downloads 1, 8–9, 13–16, 136–7
  example applications 31–49, 332–93
  GUI programming 31–49, 325–93
  header files 32–49, 104, 119–23, 128–31, 138–47, 332–93
  icons 391–3
  installation 19–20, 32–49, 52–3, 136–47, 332–93
  package definition files 32–49, 51–3, 137–47, 332–93
  platform security 426–9
  project build files 47–9, 104–7, 119–49, 332–93
  resource files 38–49, 121–3, 128–31, 332–3, 340–93
  SimpleEx 31–49, 250–6, 321–2, 332–93
  source files 43–7, 121–3, 128–31, 332–93
  TCP/IP applications 293–324
  third-party suppliers 1, 11
  types 1, 2–3, 10
  UIDs 31–49, 104, 106, 109, 121–4, 139–47, 334–93
  view architecture 387–91
apprun.exe 75, 108–9, 333–40
apr files 72
area types, graphics context (GC) 382–4
arguments, processes 205–7
Arima U300 396
ARM 20–2, 51–3, 61–2, 68, 108, 116, 119–20, 127
  see also CPUs
ARM4 119
armi 51–3, 104, 116–17, 119–20, 138–47
ARRAY resource 353–6, 359–60
arrays
  classes 191–9, 353–6
  concepts 156–9, 162–3, 185–6, 191–9, 341–2, 353–6
  data-finding method 197
  descriptors 156–9, 162–3, 185–6, 192–4
  dynamic arrays 194–9
  fixed arrays 191–2
  inserting/appending data 196
  methods 196–9
  RArray 195–9
  sorting method 197
  templates 191–2, 194–9
ASCII 207–8, 301–2, 327
assert macros, concepts 103
_ASSERT_ALWAYS 103
_ASSERT_DEBUG 103
asynchronous functions
  see also active objects; Logon
  cancellations 239, 241, 247–9, 251–8
  concepts 69, 212–13, 214–19, 232, 235–65, 275–6, 305–24
  definition 235
  examples 249–56, 305–24
  high-level view 238
  request semaphores 232, 236–65
  servers 267–8, 275–6, 277–83, 305–24
  sockets 305–24
  TRequestStatus 235–48, 305–24
AT&T 6
audio 3, 6–7
authorization process, platform security 427
automatic network connections, concepts 323–4
AVKON_CONFIRMATION_QUERY 371
avkon.lib 121–3
avkon.rh 38–49
avkon.rsg 38–49
AVKON_VIEW 390–1

BackPtr 187–91
BackSpace 274–87

badef.rh 344, 354
badesca.h 193
bafl.lib 193
bandwidth, concepts 2–9
BARN 133
Base 225, 227
base classes
  polymorphic DLL pointers 70–2, 104
  TDes... 157–61
base libraries
  see also libraries
  concepts 55–6
BaseConstructL 338–40, 389–91
basic data types, concepts 82–3, 110
batteries 9, 327–32
BenQ P30 23, 397
Berkley Unix see BSD
binary data
  see also descriptors
  concepts 154–8, 185–6
  strings 155
bind 298–9
bitmaps 268, 332–3, 371, 384, 391–3
blanket permissions, platform security 426
bld.inf 32–49, 119–23, 332–93
  see also Component Description File
bldmake 120–3, 129–31, 132–6, 153–4
blocks
  active objects 257, 268
  memory 62, 257
Bluetooth connectivity 1, 8–9, 10, 14, 15, 19, 56, 76–9, 136–7, 304, 396–424
bmp files 392–3
Borland C++ Builder 22–3, 25–6, 29, 49, 119
  see also Integrated Development Environments
browsing 2–3, 7–8, 13–16, 293–324
  see also Internet
  specifications 14–16, 396–424
brushes 383–4
BSD socket API 77, 293–4, 297–304, 310–11
  see also sockets
  concepts 293–4, 297–304, 310–11
  examples 299–304
bt.prt 76
buffer descriptors
  see also descriptors; TBuf...
  concepts 158–60, 162–6, 220, 343–8
  definition 158, 162
build targets 25–8, 29, 50–3, 104, 116–17, 121–3, 129–36, 140–7, 153–4
build tools 19–53, 115–49
buttons 325–72, 375
BWINS 133
BYTE 341–2, 374

C++ 13, 22, 25–9, 61–2, 69–72, 74–5, 81–113, 156–8, 293, 299, 304–24
  basic data types 82–3
  concepts 81–113, 156–8, 293, 299
  features 81–2
  nonstandard characteristics 82
  overload features 81–2, 96, 99–100, 205–6
  overview 81–2
  sockets 293, 299, 304–24
  STL 82
  strings 156–8
  Symbian OS 81–113, 156–8, 293, 299–324
  templates 81–2, 96–7
c: drive 61, 125–8
  see also internal flash disk
C (heap-allocated) classes, concepts 84–8, 110
caches 65–6
CActive 69, 237–63, 269–91
  see also active objects; client/server...
CActiveScheduler 237–8, 240–63, 278–83, 284–5
  see also active scheduler
  concepts 237–8, 240–63
  methods 237–41, 244–5, 257, 263, 278–83, 284–5
CActiveScheduler::Add 237–41
CActiveScheduler::Start 237–8, 257, 263, 278–83, 284–5
CAknApplication 33–49, 334–40
CAknAppUi 33–49, 253–6, 334–40
CAknConfirmationNote 372
CAknDialog 360–6
CAknDocument 33–49, 334–40
CAknErrorNote 372
CAknForm 361–6
CAknInformationNote 45–9, 372
CAknListQueryDialog 372
CAknNumberQueryDialog 372
CAknProgressDialog 372
CAknQueryDialog 370–2
CAknSingleGraphicStyle-ListBox 375
CAknSingleStyleListBox 374–5
CAknTextQueryDialog 371–2
CAknTimeQueryDialog 372
CAknView 390–1
CAknViewAppUi 390–1
CAknWaitDialog 372
calendars 8, 79, 326–32
cameras 2, 14–16, 396–424
Cancel 239, 248–9, 251–6, 317–22
capabilities
  example 429
  platform security 426–7
  TCB 428–9
Capacity 187–91
captions 391–3
  see also aif files
CArrayFix... 194–9
CArrayPtr... 194–9
case conversions, descriptors 155, 177–9, 181–2
CBA_BUTTON 41–9

CBase 84–8, 95–9, 110, 257
CBufBase 186–91
CBufFlat 186–91
CBufSeq 186–91
CBufStore 336–40
CCirBuf 200–1
CCkn... 369–70
CCoeAppUi 385, 389–91
CCoeControl 36–49, 253–6, 334–40, 376–80, 385–6, 387–91
CConsoleBase 151–4
CConsoleBase::Getch 152–4
CConsoleBase::Printf 152–4
CCountdown 250–6
CDesC8Array... 192–4
CDesC16Array... 192–4
CDesCArray... 192–4
CDirectFileStore 336–40
CDMA network protocol 3–4, 16
CEikApplication 35–49, 334–40
CEikAppUi 36–49, 334–40, 388–91
CEikChoiceList 376
CEikColumnListBox 374
CEikComboBox 376
CEikDialog 112, 348–61, 366–8
CEikDocument 36–49, 334–40
CEikEdwin 349–66, 373
CEikHierarchicalListBox 374
CEikonEnv 368
CEikProgressInfo 375
CEikSetPasswordDialog 369
CEikTextListBox 374–5
CEikTimeDialogSetCity 369
CEikTimeDialogSetTime 368–9
CEmbeddedStore 336–40
CFileStore 337–40
CFont 383–4
char 82–3
CHARACTER_SET keyword, resource files 341–2
check boxes 375
choice lists 353–6, 374, 375–6
CHOICELIST 353–6, 376
chunks
- see also RChunk
- concepts 62–6, 85, 223–33
- creation 62, 223–5
- detailed workings 226–8
- types 62–3, 85, 223–8

circuit-switched voice communication 3–5
circular buffers, concepts 200–1
cknctl.rh 373
ckndlg.lib 369
classes
- see also C...; M...; R...; T...
- active objects 238–41, 250–6
- applications 31–49, 332–93
- client/server model 267–76
- concepts 83–8, 104, 110–11, 154–201, 333–40
- controls 373
- descriptors 157–72
- dialogs 354–7
- DLLs 104, 333–40
- libraries 104
- naming conventions 83–4, 110–11
- programming basics 83–8
- Series 60 (Nokia) 33–49, 334–93
- Series 80 (Nokia) 35–49, 334–93
- sockets 85–8, 104, 268, 304–24
- Symbian OS 83–8, 110–11, 154–201, 333–40
- types 83–8, 110–11, 154–201
- UIQ 34–49, 334–93
- variable names 110–11

clean 121–3
cleanup, exception handling 89–90, 93–103, 152–3
cleanup stack
- complexities 95–6
- concepts 93–103, 152–3, 189–93
- object types 95–9

CleanupClosePushL 96–9, 288
CleanupDeletePushL 96–9
CleanupReleasePushL 96–9
CleanupStack 94–5
CleanupStack::Pop 94–102, 250–6, 279–83, 339–40
CleanupStack::PopAndDestroy 94–102, 189–91, 263, 278–80, 288
CleanupStack::PushL 94–102, 110, 111–12, 193–4, 250–6, 262–3, 278–83, 339–40
Clear 381–2
client classes, concepts 267–92
client-side code, sockets 298–324
client/server model
- see also CActive; CServer; CSharableSession; RSessionBase
- active objects 267–8, 275–6, 277–83
- classes 267–76
- concepts 59–60, 73–5, 220, 267–92, 296–324
- definition 267
- emulator 285–7
- examples 270–87
- implementation 276–83, 287–92
- message-processing example 280–2
- overview 268–9
- pointers 280–3
- service-invoking methods 274–6, 318–22
- shutdown issues 284–5, 303–4
- sockets 296–324
- starting 271–4, 277–80, 285–7, 288
- subsessions 287–92
- TCP/IP 296–324
- TextBuffServ example 270–87
- transient servers 284–5

ClientRect 339–40
Close 85–8, 96–9, 110, 195–9, 205–6, 207–8, 220, 225, 230, 251–6, 271–6, 284–5, 305–24
CMyPolyDll 107–8

CnvUtfConverter 186
CObject 289–92
  see also subsessions
CObjectCon 292
CObjectConIx 292
CObjectIx 292
code 13, 22, 25–9, 56–7, 60–4, 69–72, 74–5, 81–113, 228, 293
  see also threads
  C++ 13, 22, 25–9, 61–2, 69–72, 74–5, 81–113, 293
  chunks 63–4, 228
  critical sections 232–3
  naming conventions 58, 70, 83–4, 110–12
  start-up code 60–1
collation method, descriptor comparisons 173–6
colorList 356–7, 363–6
combo boxes 376
CommandLine 206–13
committed memory 224–8
communication database, communications architecture 76, 78–9
communication methods, smartphones 2–9
communications architecture
  see also local device communication features
  components 76–9
  concepts 2–6, 13–16, 56, 75–9, 293–324
  overview 75–9
Communicators (Nokia) 6, 12, 15–16, 23, 27, 148, 293, 330–1, 412
  see also Nokia
Compare 172–4
comparisons, descriptors 172–4
competitors, Symbian OS 11, 16–17
Complete 282
Component Description File 25–9, 32–49, 50–1, 104, 119–23
  see also bld.inf
ComponentControl 379–80
compound/simple controls, contrasts 378–9
Compress 187–91
computers 2, 6–9
  see also PCs
CONE (control environment), concepts 73–5, 340, 377–8, 380
cone.lib 121–3
configuration, emulator 24–5, 125–7
Connect 85–8, 271–4, 288, 305–24
connection agents, communications architecture 76, 77–9
connectivity features 1, 8–9, 14–16, 56, 75–9, 136–7, 429
  see also local device communication features
  concepts 8–9, 56, 75–9, 136–7
  specifications 14–16, 396–424
const 108, 161
constants, naming conventions 111
ConstructL 45–9, 101–2, 239–40, 250–6, 276–83, 337–40, 358–66, 377–8, 390–1
constructors 45–9, 101–2, 166–9, 238–41, 250–6, 276–83, 317–22, 337–40, 377–8, 390–1
  leaves 100–2
  two-phase constructors 100–2
contact entries 8, 79
Contains 383–4
context switches 65–6, 68–9
Control 355–7
control stack, keys 385
controls
  anatomy 377
  characteristics 377
  classes 373
  compound/simple contrasts 378–9
  concepts 73–5, 274–87, 340, 349–66, 372–87
  definition 376
  drawing controls 46–7, 358–9, 379–84
  GUI controls 355–7, 372–87
  handwriting recognition 385–6
  header files 373
  implementation options 373
  keys 384–7
  libraries 373
  lodger controls 377–9
  pointers 377, 384, 386–7
  redrawn windows 380–1
  requirements 373
  types 373–7
  user input 384–7
  window-owning/lodger contrasts 377–9
conversions, descriptors 155, 177–9, 181–4, 186
coordinates, graphics context (GC) 382–4
Copy 157–8, 164–6, 168–9, 177–9, 181, 186, 220, 309–24
copying data, descriptors 157–8, 164–6, 168–9, 177–9
CountComponentControls 379–80
CPermanentFileStore 336–40
cpp files 151–4, 252–6, 332–93
CPtrC8Array 192–4
CPtrC16Array 192–4
CPtrCArray 192–9
CPUs 20–2, 51–3, 61–2, 66–8, 119, 127–8, 223–5
  see also ARM...; x86...
CQikApplication 34–49, 334–40
CQikAppUi 34–49, 334–40
CQikDocument 35–49, 334–40
CQikZoomDialog 369
crashes 10, 155, 204
Create 205–6, 215–18, 288
CreateAppUiL 44–6, 335–40
CreateDocument 335–40
CreateGlobal 223–33
CreateLocal 226, 231

CreateLog 262–3
CreateSession 269–92
CreateWindowL 339–40, 358–9, 377–8
critical sections, concepts 232–3
cryptography 10
Crystal 12
  see also Series 80
CSD network protocol 4–5, 15, 322–4
  see also HSCSD...
CServer
  concepts 269–91
  methods 269–70
CSession 269–91
CSharableSession 268–91
  see also client/server...
CSimpleExApplication 37–49, 250–6, 321–2, 335–40
CSimpleExAppUi 37–49, 253–6, 321–2, 337–40, 357–8
CSimpleExAppView 37–49, 250–6, 339–40, 358–9, 378, 381–2
CSimpleExDialog 354–7
CSimpleExDocument 37–49, 335–40
CSimpleExForm 363–6
CStreamDictionary 337–40
CStreamStore 337–40
CSY modules, serial communications server 76, 78–9
CTextBuffServ 277–83
CTimer 259–60, 285
CTrapCleanup::New 95
Ctrl + F5 keys 29
Current 245
current position, drawing controls 384
CWeatherInfo 315–24
CWindowGC 382–4

d: drive 61
  see also removable memory cards
D suffixes, concepts 112
data buffers
  see also descriptors
  concepts 155, 158–201, 220
data chunks
  see also chunks
  concepts 63–5, 228
data collection classes, concepts 200–1
data input methods 11–16, 67, 73–5, 124–8, 267–8, 326–93, 396–424
  concepts 11–16, 67, 73–5, 124–8, 267–8, 326–72, 384–7
  controls 384–7
  platforms 11–16, 325–72, 384–7
  specifications 13–16, 396–424
data organization classes, concepts 154–201
data transfers, concepts 2–9
data types
  classes 84–8, 110, 154–201
  concepts 82–3, 110
data-caging concepts, platform security 428–9
debuggers 20–2, 103, 120, 124, 155, 263–5
  assert macros 103
  log files 21, 263–5
  Windows development tools 20–2, 120, 124
def files 132–6
  see also freezing
DEFNAME 133
Delete 182–3, 187–91
Delete Field 361–6
delete trap 95
DeleteCurrentItem 361–6
DeleteLine 366–8
Deque 249
DeregisterView 390–1
Des 169–71
descriptors
  see also HBuf...; TBuf...; TPtr...
  8/16 bit conversions 186
  advantages 155
  appending methods 157–8, 161, 164–6, 168–9, 179–80
  arrays 156–9, 162–3, 185–6, 192–4
  binary data 154–8
  buffer descriptors 158–60, 162–6, 220, 343–8
  case conversions 155, 177–9, 181–2
  class types 157–72
  comparisons 172–4
  concepts 154–201, 220, 270
  conversions 155, 177–9, 181–4, 186
  copying data 157–8, 164–6, 168–9, 177–9, 181
  definitions 154–5, 158, 162, 166, 169
  deletions 182–3
  examples 156–8
  exception handling 155
  fill method 178–9
  formatting data 180–1
  heap descriptors 158–60, 169–72, 279–80
  hierarchy 159–60
  importance 155
  lengths 158–85
  memory layouts 163–72
  memory overruns 155, 165
  methods 172–86
  modifiable/non-modifiable descriptors 158–66, 169–71, 172–85
  modifying methods 177–85
  non-modifying methods 172–7
  NULL-terminated string conversions 183–4
  pointer descriptors 158–60, 166–9, 282–3
  size-setting method 184–5
  sub-strings 174–7
  types 158–201
  wildcard searches 175–6
destructors 84–8, 95–6, 248–9, 316–22
development tools

see also software development kits; Windows development tools
basic pieces 19–20
components 19–24, 115–49
concepts 19–30, 115–49
examples 19–53
firing up 24–30
needs 19–20
overview 19–20, 117–18
problems 29–30
quick start guide 19–53
tips and traps 30
tools 10, 11–12, 19–53, 115–49
device contrasts, emulator 127–8
device drivers
concepts 60–1, 66–8, 76–9, 127–8
definition 79
emulators 127–8
ROM 60–1, 125–8
devices 24–30, 147–9
dial-up connections, drawbacks 4–5
DIALOG resource 348–72
dialogs 325, 328–31, 348–72
classes 354–7
creation 348–59
launching 357–8
list boxes 374
multipage dialogs 359–60
resource definition 349–54
Series 60 (Nokia) 329–30, 348–66, 367–72
Series 80 (Nokia) 112, 331, 348–59, 368–72
stock dialogs 368–72
UIQ 328, 348–59, 368–72
dir 26–7
direct screen access, APIs 75
directories 25–30, 51–3, 115–17, 137–47, 267–8
DiscardFont 339–40, 381, 383–4
DispatchMessageL 280–3
DLG_BUTTONS 351–9
DLG_LINE 349–73
dll files 43–9, 57–9, 67, 76, 122–3, 129–31, 333–93, 429
DLLs see dynamic link libraries
DNS see Domain Name System
DoCancel 239, 241, 248–9, 251–6, 316–22
document classes, applications 31–49, 334–40
documentation
OS requirements 10
SDK directories 117
Domain Name System (DNS) 301–4, 318–20
domain names, IP addresses 301–4, 307–8, 318–20
Doom network service 296
downloaded applications 1, 8–9, 13–16, 136–7
Draw 46–7, 253–6, 339–40, 358–9, 377, 379–84
DrawBitmap 384
DrawDeferred 381
DrawEllipse 384
drawing controls
application-initiated drawing 381–2
concepts 46–7, 358–9, 379–84
graphics context (GC) 46–7, 382–4
points and lines 382–4
redrawn windows 380–1
SimpleEx Draw 381–2
DrawLine 383–4
DrawLineTo 383–4
DrawNow 381
DrawPie 384
DrawPolygon 384
DrawPolyLine 384
DrawRect 384
DrawRoundRect 384
DrawText 46–7, 339–40, 381–2, 384
drive letters 60–1
Duplicate 231
dynamic arrays
see also arrays
concepts 194–9
dynamic buffers
see also CBuf...
area pointers 191
class diagram 187
concepts 186–91
inserting/deleting data 189–90
methods 188–91
reading/writing methods 188–9
size changes 190
types 187–8
uses 187
dynamic link libraries (DLLs) 43–9, 55, 57–9, 67, 69–72, 76–9, 103–8, 128–36, 333–40, 429
classes 104, 333–40
concepts 57–9, 67, 69–72, 76–9, 103–8, 128–36, 333–40
creation 104–6, 128–31
definition 57, 103
emulator 108, 128, 285–7
extension names 58, 70
freezing mechanism 131–6
GUI applications 57–8, 108–9, 333–40
mmp files 128–36
multiple DLLs 107–8
ordinals 130–6
programming basics 103–8
RLibrary 71–2, 107–8, 130–1, 286–7
rules 105–6
types 57–8, 67, 69–72, 76–9, 103–8
DynInitMenuPaneL 361–6

e: drive 61
see also removable memory cards
E (enumeration members) prefixes, concepts 111
e32base.h 151–4
e32cons.h 151–4
e32des8.h 158–9
e32des16.h 158–9
E32Dll 43–9, 105–7, 333–40
E32Main 109, 204–13, 261–3, 278–83, 286–7
see also processes

e32std.h 88, 115–16, 159
E32USER - CBase 42 318
E32USER - CBase 46 257
EAknSoftkeyExit 45–9
Echo network service 296
ECOM API, application protocols 79–80
EDGE network protocol 3–4, 6, 15, 293–4, 322–4, 399–400, 405–17
 see also GSM...
Edit 361–6
Edit Label 361–6
EditCurrentLabel 361–6
editor controls
 see also controls
 concepts 274–87, 349–66, 373–4, 391–3
editors, Windows development tools 20–2
EDWIN 349–66, 373
EIK_APP_INFO 39–49, 344–5
eikappui.h 35–49
eikcdlg.lib 369
eikcoct1.lib 359
eikcore.lib 121–3
eikct1.lib 359
eikdlg.lib 359
EikDll::StartExe 109, 206
eikdoc.h 35–49
eikmenup.h 35–49
eikon.rh 38–49, 341, 353, 373
ELeave 99–102, 107–8, 134–6
emails 2–3, 6–7, 10, 14–17, 56, 293
embedded sis files, installation 142–3
emulator
 see also epoc
 client/server model 285–7
 concepts 20–2, 24–30, 50–1, 108–9, 116–17, 119–23, 124–8, 147–9, 213, 285–7
 configuration 24–5, 125–7
 device contrasts 127–8
 DLLs 108, 128, 285–7
 exe files 109
 fonts 127–8
 memory capacity 126
 multiple processes 128, 213, 285
 pixels 127–8
 quick test 24–30
 running 125
 SDK 20–2, 24–8, 116–17, 119–23, 124–8, 147–9
 Series 60 20–1, 24–8, 50–1, 126–8, 147–9
 Series 80 (Nokia) 50–1, 148–9
 static data in DLLs 108, 128
 UIQ 50–1, 147–9
 virtual drives 125–8
encapsulation features, C++ 81–2
enum 111, 338
ENUM keyword, resource files 341–2
enumerations, naming conventions 111
EPOC 10–11
epoc 24–30, 125–8
 see also emulator
epoc32 30, 115–17, 120–3, 125–8
epoc32/build 116, 120–3, 125–8, 147
epoc32/data/z 116–17, 125–8, 138–47
epoc32/gcc 116–17, 149
epoc32/include 115–16, 121–3, 125–8, 158–9, 204
epoc32/release 116–17, 120–3, 125–8, 129–31, 138–47
epoc32/tools 116, 148–9
epoc32/wins 117
EPOC_DRIVE_D 126–8
epoc.exe 29
epoc.ini 24–5, 125–8
epocprocesspriority 211
EPOCROOT 51–3, 125, 147–9, 154, 158–9
EPriority... 217–18
Ericsson, Symbian ownership 11
Error 245–7
errors 10, 88–103, 155, 245, 246–7, 257–8
 see also exception...
 active scheduler 245, 246–7, 257–8
 assert macros 103
 concepts 88–103, 245, 246–7, 257–8
 leave/trap mechanism 89–103, 111–12, 169, 198, 241, 245–7, 250–6, 278–83
 panics 102–3, 158, 211–12, 277–83
 return codes 88
Escape key 331
ESimpleExCommand 45–9
ESimpleExDialog 357–8
ESimpleEx.hrh 43
esock.dll 76
ETEL server, communications architecture 76, 78–9
euser.lib 121–3, 204
event handlers 74–5, 237, 241
events, active objects 69, 212–13, 214–19, 232, 235–65
examples, quick start guide 19–53
Excel 17
exception handling 82, 88–103, 111–12, 155, 169, 198, 241, 245–7, 250–7
 see also errors
 assert macros 103
 cleanup 89–90, 93–103, 152–3
 concepts 88–103, 155
 descriptors 155
 leave/trap mechanism 89–103, 111–12, 169, 198, 241, 245–7, 250–6, 278–83
 panics 102–3, 158, 211–12, 277–83
 programming basics 88–103
 return codes 88
 stray-signal exceptions 257–8
exe files 29, 57, 59–60, 64–6, 75, 108–9, 122–3, 203–13, 429
 see also executables; processes
 emulator 109

programming basics 108–9
structure 108–9
executables 29, 57, 59–60, 64–8, 108–9, 116–17, 136–47, 203–13, 429
see also applications; exe files; processes
concepts 108–9, 116–17, 203–13, 215
epoc32/release 116–17, 120–3, 125–8, 129–31, 138–47
platform security 429
programming basics 108–9
sis files 19–20, 32–49, 51–3, 136–47, 332–93
threads 215
executed-in-place code, concepts 60–6
ExecuteLD 112, 348–59
ExitReason 211–12, 215, 218–19
ExitType 211, 218
Expand 187–91
explicit network connections, concepts 323–4
EXPORT_C 105–7, 130–1, 134–6
exports, libraries 105–6, 128–36
EXPORTUNFROZEN 106, 128–31, 132–6
extensions, kernel 66–8
ExternalizeL 336

f32file.h 288
fast executive kernel calls, concepts 68
fax 7
features, smartphones 1–9
FEPs see Front End Processors
file server
see also RF...; servers
concepts 59–60, 267–8, 288–92
subsessions 288–92
file system 11, 59–61, 267–8, 288–92, 428–9
structure 428–9
FILENULL 142
FILERUN (FR) 143
FILETEXT 141–2, 146
Fill 178–9
filled shapes, drawing controls 384
FillZ 179
Find 174–6, 197
'fire and forget' protocols 294
First 200
fixed arrays
see also arrays
concepts 191–2
fixed processes, concepts 66, 68
flash memory see internal flash disk
flat dynamic buffers, concepts 187–91, 192
flogger.h 264–5
flogger.lib 264–5
flushing costs, caches 66
_FOFF 196
folding method, descriptor comparisons 173–6
font and bitmap server
see also servers
concepts 268
fonts 127–8, 268, 339–40, 381, 383–4
foreign languages
installation support 144–7, 345–8
translators 13
FORM 361–6
Format 180–1
format, resource files 240–2
formatting data, descriptors 180–1
forms, Series 60 (Nokia) 361–6
freeware 1
freeze 121–3, 129–36
freezing
concepts 106, 128–36
def files 132–6
definition 131–2
disabling methods 106, 128–31, 132, 136
enabling methods 132
importance 131–2
libraries 29–30, 105–6, 121–3, 128–36
new-function inserts 135–6
violated interfaces 136
Front End Processors (FEPs) 385–6
FTP network service 296
Function 280–3
function arguments, naming conventions 110–11
function names, conventions 110–12

games 13–16, 293
GC see graphics context
generations, mobile communications 3–9
generic build system, SDK 20–2
GetByAddr 304–24
GetByName 304–24
Getch 152–4
gethostbyname 301–2, 304
GetMemoryInfo 209–10
GetTemperatureL 315–24
GetText 274–87, 373
GLDEF_C 134–6
global memory chunks
see also chunks
concepts 223–33
global variables, restrictions 108, 111
GNUPoc 24
GPRS network protocol 3, 5, 10, 14–17, 78–9, 293–4, 322–4, 396–423
see also GSM...
graphical user interface framework (GUI)
see also application framework; Series...; UIQ...
active scheduler 242, 257, 311–12, 322
anatomy 31–49, 332–3
application classes 31–49, 332–93
application programming 31–49, 325–93
concepts 11–16, 21–2, 31–49, 56–8, 72–5, 108–9, 117, 124–5, 151–2,

graphical user interface framework (GUI) (*continued*)
214, 242, 257, 267–8, 285, 311–12, 322, 325–93
controls 73–5, 274–87, 340, 349–66, 372–87
dialogs 325, 328–31, 348–72
DLLs 57–8, 108–9, 333–40
examples 31–49, 325–93
icons and captions 391–3
overview 31–49, 73–5, 325–32
resource files 38–49, 340–93
servers 267, 285
Symbian OS 11, 21–2, 31–49, 56–8, 72–5, 108–9, 117, 151–2, 325–93
types 11–16, 325–32
view architecture 387–91
Windows development tools 21–2
graphics
animation plug-ins 75
direct screen access 75
drawing controls 382–4
high performance graphics 75
graphics context (GC) 46–7, 382–4
Grow 383–4
GSM network protocol (Global System for Mobile Communication) 3–4, 5–6, 8, 10, 14–17, 78, 293–324, 396–423
see also EDGE...; GPRS...
GUI see graphical user interface framework

HAL see Hardware Abstraction Layer
handle classes 85–8, 205–13, 214–19, 228–9, 232, 282–3
see also RMutex; RProcess; RSemaphore; RThread
HandleCommandL 45–9, 253–7, 338–40, 357–8, 366, 389–91
HandleControlStateChangeL 367
HandlePointerEvent 377, 386–7
HandleRedrawEvent 380
handles, concepts 84–8
Handspring Treo 600 16–17
handwriting recognition 12–14, 385–6
Hardware Abstraction Layer (HAL), concepts 66–8
HBufC 158–93, 279–80
see also heap descriptors
concepts 158–93, 279–80
memory layout 169–71
TBufC 169
header files
concepts 32–49, 104, 119–23, 128–31, 138–47, 332–93
controls 373
heap chunk, concepts 62–3
heap classes, concepts 84–8
heap descriptors
see also descriptors; HBufC...
concepts 158–60, 169–72, 279–80
definition 158, 169
modifications 169–71
usage of other descriptors 171–2
help files 322–3
hierarchy, descriptor classes 159–60
high performance graphics 75
see also graphics
hlp files 332–3
home area, virtual memory map 63–6, 222–8
hot keys, emulator 126–8
hrh files 32–49, 332–93
HSCSD network protocol 5, 14–15, 396–423
see also CSD...
HTML 7–8, 14–16, 117, 293–4, 315–17, 403, 406–7, 410–12, 415–17
HTTP 56, 80, 296, 299–304, 308–10, 314–24
i (member variables) prefixes 110–11
IAPs see Internet Access Points
icons 391–3
see also aif files
IDEs see Integrated Development Environments
IMAP accounts 7, 296
see also emails
implicit network connections, concepts 323–4
import libraries
see also static libraries
concepts 105–7, 129–36
IMPORT_C 105–7, 130–1, 134–6
include 345–6
InfoPrint 109, 386–7
InfoWinL 368
infrared connectivity (IR) 8–9, 14, 19, 56, 77–9, 136–7, 304, 396–424, 429
inheritance features
C++ 81–2, 86–7
interface classes 86–7
InputCapabilities 385–6
Insert 187–91, 196–9
Install 242–5, 278–83
installation 19–20, 32–49, 52–3, 136–47, 242–5, 277–83, 332–93
see also sis files
active scheduler 242–3, 277–83
advanced pkg options 141–4
concepts 31–49, 136–47, 332–93
directories 137
file-specification methods 140–1
language support 144–7, 345–8
pkg files 32–49, 138–47, 332–93
requisite lines 143–4
running executables 143
runtime-generated file removal 142
text notices 141–2
instant messaging 6–7, 293–4

instantiated classes, concepts 82–4, 104, 110, 158–9, 214
int 82–3
Integrated Development Environments (IDEs) 19–53, 104, 119, 124–8, 332–3
  see also Borland...; Metrowerks...; Microsoft...
  concepts 19–22, 27–9, 49, 104, 119, 124–8, 332–3
  providers 21–2, 25, 49
  quick-start examples 19–22, 25, 27–9, 31–49, 332–3
  selection criteria 22
inter-thread communications, concepts 66–8, 207–8, 220–8, 282–3
interface classes
  concepts 84, 86–8, 110
  example 86–8
  inheritance features 86–7
interface freezing see freezing
internal flash disk
  see also c: drive; memory...
  concepts 61, 125–8, 136–47
InternalizeL 336
Internet 2–3, 6–8, 293–324
  see also browsing; TCP/IP...
Internet Access Points (IAPs) 15, 78–9, 322–4
Internet Protocol Suite see TCP/IP
interrupts
  concepts 68
  kernel executive 68
Intersects 383–4
IP (Internet Protocol) 294–324
  see also TCP...
  addresses 295–6, 301–4, 307–8, 318–20
  concepts 294–324
  domain names 301–4
  layering diagram 295
  port addresses 295–324
IR see infrared connectivity
ISPs 4–5

Java 13, 21, 88, 116
Java Runtime Environment 21

K (constants) prefixes, naming conventions 111
KDynamicLibraryUid 124
kernel 55–80, 210, 214–19
  architectural overview 66–7
  concepts 55–6, 62–8, 210–11, 214–19
  definition 55, 66
  executive 67, 68
  extensions 66–8
  HAL 66–8
  MMU 62, 64–5
  process priorities 210–11
  roles 55, 62, 64–5, 66–8
  server 66–8
  threads 214–19
  user library 67–8
KerrCancel 248–9
KErrEof 314–15
KErrNoMemory 88, 92–3
KErrNone 43–9, 88, 91, 106, 134–6, 199, 207–8, 216, 223–4, 228–9, 236, 241, 248, 252–6, 261–3, 273–4, 287, 305–24
KErrNotFound 88, 174–6, 197, 207, 273
KErrNotSupported 272–4, 279–87
KExecutableImageUid 124
keys 12–16, 74, 126–8, 267–8, 326–93, 396–424
  control stack 385
  controls 384–7
  emulator 126–8
  platforms 326–32
  virtual keyboards 12–14, 126–8, 385–6
Kill 211–12, 218–19
KRequestPending 236–7, 244, 248, 258
KUidApp 124

_L 156–8, 161–2, 171–2
L suffixes, concepts 93–103, 110, 111–12
LAF see Look and Feel
LANG keyword, resource files 346–8
LANGUAGE keyword, resource files 346–8
language support
  installation 144–7, 345–8
  mmp files 345–8
  resource files 345–8
laptops 2
Last 200
LBUF 353–6
LC functions, concepts 98–102, 111–12
Leave 90–103, 198–9, 205–6
leave/trap mechanism
  active scheduler 246–7, 250–6, 257–8, 278–80
  concepts 89–103, 111–12, 152–3, 169, 198, 241, 245–7, 250–6, 278–83
  constructors 100–2
  object creation 99–100
Left 176–7
Length 160–1
Lenovo P930 23
lib files 106–7, 121–3, 125–8
libraries 10–11, 20–2, 43–9, 55–6, 57, 67–8, 73–5, 103–8, 121–3, 128–36, 204, 263, 322, 359, 369, 429
  see also dynamic link...; middleware
  application protocols 79–80
  base libraries 55–6
  classes 104
  concepts 20–2, 55–7, 67–8, 73–5, 103–8, 128–36
  CONE 74
  controls 373
  freezing 29–30, 105–6, 121–3, 128–36
  OS requirements 10
  programming basics 103–8
  SDK 20–2, 128–36
  types 57, 67–8, 73–5, 103–8
  UIKON 73–5, 373
  user library 67–8
LIBRARY 121–3, 129–31, 204, 263, 322, 359, 369
LineChangedL 367
lines, drawing controls 382–4

linked lists
  see also TDblQue
  concepts 200
Linux 16, 17, 24, 56, 235
list boxes 325–32, 374–5
LISTBOX 374–5
_LIT 156–8, 161–9, 173–86, 189–91, 199, 205–9, 223–4, 228–9, 260–3, 271–6, 285–8
LLINK 341–5, 351–4, 374
Load 130–1
loading methods, polymorphic DLLs 71–2, 130–1
local device communication features
  see also connectivity features
  concepts 8–9, 56, 77–9
local memory chunks
  see also chunks
  concepts 225–8
local semaphores
  see also semaphores
  concepts 231–2
localization, resource files 345–7
lodger/window-owning controls, contrasts 377–9
log files 21, 263–5
log servers 79
Logon 212–13, 219–20, 273–4
  see also asynchronous functions
long 82–3, 341–2
Look and Feel (LAF), concepts 73–5
Lookup 130–1
LowerCase 155, 181–2
LTEXT, resource files 341–2

M (interface) classes, concepts 84, 86–8, 110
MACRO 123
macros
  assert macros 103
  naming conventions 111
  string literals 156–8, 161–2
make 117
makefiles, build system overview 117–18
MakeLineVisible 366–8
makeName 156–8
makesis 51–3, 137–46
MakeVisible 388–91
MakeWholeLineVisible 366–8
manufacturers 10–12
  see also individual manufacturers
Match 175–6
MCoeView 387–91
MegabytesOfFreeMemory 126–8
member variables, naming conventions 110–11
memory 9–10, 14–16, 55–6, 57, 60–6, 125–8, 136–47, 163–72, 223–8, 396–424
  see also Random Access...; Read Only...
  addresses 61–2, 223–8
  blocks 62, 257
  capacity specifications 14–16, 60–1, 126, 396–424
  chunks 62–6, 85, 223–8
  committed memory 224–8
  concepts 60–6, 125–8, 163–72, 223–8
  descriptors 163–72
  emulator configuration 125–8
  frugal requirements 9–10
  organization 61–6
  orphaned memory 93
  overrun problems 155, 165
  physical/virtual memory addresses 61–6, 224–8
  processes 62–6, 220–8
  shared memory 57, 103–8, 220–8
  types 60–2
memory cards see removable memory cards
memory leaks 93, 153, 383
Memory Management Unit (MMU)
  concepts 62–6, 213, 222–5
  page tables 63–5, 224–5
  protection role 62, 65
memory maps, concepts 61–6, 213, 224–8
menu/softkey items, resource files 38–49, 344–72
MENU_BAR 39–49, 344–5
MENU_ITEM 39–49, 344–5, 375
menu_pane 39–49
menus 38–49, 325, 326–93
MENU_TITLE 39–49, 344–5
messages, client/server model 267–92
messaging, smartphones 6–7, 14–16, 56, 293–4, 323–4
Metrowerks 21–3, 25–6, 28, 49
  see also Integrated Development Environments
Microsoft 11, 16–17, 22–3, 25–6, 28–9, 50–1, 117–19
  see also Integrated Development Environments; Windows
  Smartphone OS 11, 16–17
  Visual C++ 22–3, 25–6, 28–9, 50–1, 119
Mid 176–7
middleware
  see also libraries
  concepts 10, 56
Mixin 86–8
MMC storage media 14–15, 61, 396–421
  see also removable memory cards; storage media
mmp files 28, 32–49, 104, 106–7, 109, 119–24, 128–36, 153–4, 204–13, 263, 322, 332–93, 426
  see also project...
  concepts 32–49, 119–24, 128–36, 153–4, 204–13, 263, 322, 332–93
  DLLs 128–36
  language definitions 346–8
  processes 204–13, 332–93
MMS see Multimedia Messaging Service
MMU see Memory Management Unit
mobile phones
  see also smartphones
  concepts 1–17, 325

generations 3–9
hardware limitations 325
historical background 2–3, 9
network protocols 2–6, 8, 10, 14–17, 78, 293–324, 396–423
PDAs 2–3, 8–9, 16–17
platform security 425–9
specifications 13–16, 395–424
modem features 9
modifying methods, descriptors 177–85
Motorola
see also UIQ
A760 17
A920/A925 5, 12, 23, 326, 398
A1000 399
MPx200 17
Move 382–4
MPEG-4 video 14
MSN Instant Messenger 296
multi-homing features 15, 79
Multimedia Messaging Service (MMS) 6–7, 10, 14–16, 56, 79
multimedia support, Symbian OS 11, 13, 293
multipage dialogs
see also dialogs
concepts 359–60
multiple DLLs, concepts 107–8
multiple inheritance features, C++ 81–2, 86–7
multiple processes 128, 203–13, 285
see also processes
multiple threads 56–7, 69, 128, 203–4, 213–19
multitasking aspects, Symbian OS 11, 56–7, 128, 237–8
mutexes
see also synchronization
concepts 66–8, 85, 232–3

NAME keyword, resource files 340–2
naming conventions 58, 70, 83–4, 110–12
network connections, concepts 322–4
network interface manager (NIFMAN), communications architecture 76–9
network programming
see also sockets; TCP...
active objects 305–24
concepts 293–324
network protocols
concepts 2–6, 8, 10, 14–17, 76–9, 293–4, 322–4, 396–423
specifications 13–16, 396–424
network services, well-known server-side port addresses 296
New 84, 99–100, 107–8, 110, 169
NewApplication 43–9, 333–40
NewL 101–2, 110, 135–6, 187–91, 238–40, 251–6, 269–70, 277–83, 316–22, 339–40
NewLC 101–2, 110, 169, 253–6, 260–3
NewSessionL 269–91
Next 208–10
nif files 78–9
NIFMAN see network interface manager
nmake 117
NOCOMPRESS (NC) 139–47
Nokia 11–12, 22–3, 115–49
see also Series...
3230 23, 400
3600/3650 12, 22, 401
3620/3660 22, 402
6260 23, 403
6600 5, 12, 14, 23, 25, 61, 323, 404
6620 6, 405
6630 23, 406
6670 23, 407
6680/6681/6682 23, 408
7610 409
7650 12
7710 12, 410
9210 12, 15
9290 5, 12, 15
9300 Communicator 16, 27, 52, 330–1, 411
9500 Communicator 6, 12, 15–16, 23, 27, 52, 61, 293, 323, 330–1, 412
N-Gage 22, 413
N-Gage QD 414
N70 23, 415
N90 23, 416
N91 417
SDK 22–3, 25–8, 31–49, 115–49
Symbian ownership 11
non-modifying methods, descriptors 172–7
non-preemptive multitasking model, active objects 237–8, 256
nostrictdef 133
NULL 99–100, 156–7, 161, 177–9, 183–4, 216

object types, cleanup stack 95–100
object-oriented operating systems 10–11, 82
OEM hardware 11, 65, 67, 72–5
OfferKeyEventL 385
OfferKeyL 377
OkToExitL 354–7, 365–6, 367–8
OnStarting 245–6
OnStopping 245–6
Open 85–8, 102, 207–9, 213, 216–17, 231, 288, 307–24
'open' aspects, Symbian OS phones 1
Open Workspace 29
OpenFileL 337–40
OpenGlobal 225–9, 231
operating systems 1, 9–13, 23–4
see also Symbian OS
competitors 11, 16–17
historical background 9–13
Linux 16, 17, 24, 56
Microsoft Smartphone OS 11, 16–17
Palm OS 16, 56
requirements 9–10

operating systems (*continued*)
resource-limitations 9–10
robustness needs 9–10
option buttons 375
Orange SPV 17
Order 197
ordinal function references 130–6
orphaned memory, dangers 93
OSs see operating systems
out-of-memory situations 10, 82, 88
OutputWebPage 299–324
overload features, C++ 81–2, 96, 99–100, 205–6
overrun problems, memory 155, 165
owning manufacturers, Symbian OS 11

package definition files 32–49, 51–3, 137–47, 332–93
see also pkg files
packets
concepts 5, 293–324
TCP/IP 293–324
PAGE structures 359–66, 367
page tables, MMU 63–5, 224–5
Palm OS 16, 56
Panasonic
Symbian ownership 11
X700 23, 418
X800 23, 419
Panic 102–3, 211, 277–83
panics
concepts 102–3, 158, 211–12, 277–83
examples 102–3, 277–83
SDK list 102
PCs 2, 6–9, 14, 20–2, 24–30, 50–1, 108–9, 116–17, 119–23, 124–8, 326–7
see also Windows
emulator 20–2, 24–30, 50–1, 108–9, 116–17, 119–23, 124–8, 147–9, 213, 285–7
installation 136–7
multiple SDKs 147–9
PDAs 1, 2–3, 8–9, 16–17
PDF files 117
PE files 108
Pearl design 12
see also Series 60
pens 12–14, 383–4, 396–9, 410, 422–4
see also touch screens
performance issues
context switches 65–6, 68–9
switched processes 65–6, 68–9
Perl scripts 116
permissions, platform security 426–7
PETRAN 108
physical memory addresses, concepts 61–6, 224–8
pictures 2, 6–7, 9, 14–16
ping 318
pixels 127–8, 391–3
pkg files 32–49, 51–3, 137–47, 332–93
see also package definition files
advanced pkg options 141–4
concepts 32–49, 51–3, 137–47, 332–93
installation 32–49, 138–47, 332–93
language support 144–7, 347–8
platform security
authorization process 427
capabilities 426–7
concepts 62, 65, 68, 425–9
data-caging concepts 428–9
MMU 62, 65
permissions 426–7
SID 427–8
platforms, Symbian OS 11–16, 140–7, 325–32
Plot 383–4
plug-in DLLs
see also dynamic link libraries
concepts 58–9, 70–2, 75, 79, 104, 107–8, 124
Pocket PC OS 17
pointer descriptors
see also descriptors; TPtr...
concepts 158–60, 166–9, 282–3
definition 158, 166
pointers, controls 377, 384, 386–7
points and lines, drawing controls 382–4
polymorphic DLLs
see also dynamic link libraries; plug-in...
concepts 57–8, 70–72, 75–9, 104, 107–8, 124
examples 70–2
loading methods 71–2, 130–1
virtual declarations 70–2
Pop 94–102, 189–91, 250–6, 279–83, 339–40
pop-up fields 362–6
POP3 accounts 7, 296
see also emails
PopAndDestroy 94–102, 189–91, 263, 278–80, 288
port addresses
concepts 295–304
IP (Internet Protocol) 295–324
well-known server-side addresses 296
PPP module 77
pre-emptive multithreading, concepts 56–7, 235, 237–8
prefixes, naming conventions 83–4, 110–12
PreLayoutDynInitL 354–7, 363–6, 367–8
PrintDocument 70–2
printf 151–4, 173–4, 180–1
priorities
active objects 247
processes 210–11
threads 217–18
Priority 211
private data, data-caging concepts 428–9
private directory 428–9
prn files 70
processes
see also applications
arguments 205–7

chunks 223–8
code chunks 63–4, 228
concepts 56–7, 62–8, 203–13, 220–8, 426–9
critical sections 232–3
definition 57, 203–4
E32Main 109, 204–13
end-signaling method 212–13
examples 204
fixed processes 66, 68
inter-process communications 66–8, 207–8, 220–8
launching method 205–6, 216
memory 62–6, 220–8
mmp files 204–13, 332–93
multiple processes 128, 203–13, 285
names 208–9
performance issues 65–6
platform security 62, 65, 426–9
priorities 210–11
processes-running queries 209–10
protection 62, 65, 213
RProcess 85–8, 205–13, 220–8, 273–4, 285
running 205–6, 216
shared memory 220–8
status checks 211–12
switched processes 63–6, 68–9, 213, 222–3
terminations 211
virtual memory map 63–6, 213, 222–8
wildcard searches 208

programming basics 81–113, 235–65, 293–324, 325–93
asynchronous functions 235–65
basic data types 82–3
C++ in Symbian OS 81–2, 90–1
descriptors 154–201, 220, 270
DLLs 103–8
exception handling 88–103
executables 108–9
GUI applications 31–49, 325–93
libraries 103–8
naming conventions 58, 70, 83–4, 110–12
Symbian OS classes 83–8
TCP/IP applications 293–324

progress bars 375
PROGRESSINFO 375
project build files, concepts 47–9, 104–7, 119–49
project definitions 47–9, 104–7, 109, 119–24, 128–36, 153–4, 204–13, 345–8, 426
see also mmp files
concepts 119–24, 128–36, 204–13
definition 119–20
DLLs 128–36

project management tools, Windows development tools 20–2
protection
MMU role 62, 65
processes 62, 65, 213
semaphores 229–30

protocol 300–2
protocol modules
see also Bluetooth...; infrared...; TCP/IP
communications architecture 76–9, 293–324

protocols, interface classes 86–8
proxy servers, WAP 7
prt files 70, 76–9
Psion 10–11
Ptr 183–4, 187–91
PtrZ 183–4
PushL 94–102, 110, 111–12, 193–4, 250–6, 262–3, 278–83, 339–40

qikapplication.h 34–8
qikappui.h 34–8
qikdlh.lib 369
qikdocument.h 34–8
Quartz 12
see also UIQ
QueryWinL 368
quick start guide, Symbian OS development environment 19–53
QWERTY keyboards 327

R (resource) classes, concepts 84–8, 97, 110, 205
radio 2–9, 56
radio buttons 375
Random Access Memory (RAM)
see also memory...
capacity specifications 60
concepts 60–1, 64, 224–5

RArray 195–9
R_AVKON_DONE_CANCEL 352–4
R_AVKON_FORM_MENUPANE 365–6
R_AVKON_OK_BACK 352–4
R_AVKON_SOFTKEYS_OK_EMPTY 352–4
R_AVKON_SOFTKEYS_OPTIONS_EXIT 39–49
R_AVKON_SOFTKEYS_YES_NO 371–2
R_AVKON_YES_NO 352–4
RChunk 62, 223–8
see also chunks
RConnection 323–4
RCriticalSection 232–3
Read 85–8, 187–91, 220–8
Read Only Memory (ROM) 9, 60–6, 116–17, 125–8
see also memory; z: drive
capacity specifications 60
concepts 60–6, 116–17, 125–6
executed-in-place code 60–6

ReadL 220–8, 282–3
ReAlloc 169, 171
RecvFrom 310–11, 314–15
RecvOneOrMore 310–11, 314–22
redrawn windows, controls 380–1
reference platforms, Symbian OS 11–12, 325–32
RegisterViewL 387–91
R_EIK_BUTTONS_CANCEL 351–4

R_EIK_BUTTONS_CONTINUE 351–4
R_EIK_BUTTONS_NO_YES 351–4
relocated data, concepts 63–6, 222–3
removable memory cards
  see also memory...
  concepts 61
  MMC storage media 14–15, 61, 396–421
Remove 197, 201, 385
RemoveFromStack 385
Rename 208
Replace 244–5
request semaphores, asynchronous functions 232, 236–65
requisite lines, installation 143–4
Reset 187–91, 197, 274–87
Resize 187–91
resource classes
  concepts 84–8, 110
  example 85–6
resource directory 428–9
resource files
  see also rss files
  concepts 38–49, 121–3, 128–31, 332–3, 340–93
  definition 41, 340
  format 240–2
  language support 345–8
  localization 345–7
  SimpleEx example 342–5
  string-reading tips 347–8
RESOURCE keyword, resource files 341–2, 373
resource-limitations, smartphones 9–10
RestoreL 336–40
Resume 205–7, 215–17
return codes, errors 88
RFile 59–60, 85–8, 97–9, 102, 104, 267–8, 288
  see also file server
RFileLogger 263–5
RFs 267–8, 288
  see also file server
RGenericAgent 323
RHandleBase 231, 271–4
RHandleBase::Duplicate 231
RHostResolver 304–24
  see also sockets
Rich Text Format (RTF) 333
Right 176–7
RLibrary 71–2, 107–8, 130–1, 286–7
  see also dynamic link libraries
RLogger 263–5
rls files 345–8
RMessage 269–91
RMutex 232
robustness needs, smartphones 9–10
ROM see Read Only Memory
RPointerArray 195–9
RProcess 85–8, 205–13, 220–8, 273–4, 285
  see also processes
RProcess::Create 205–6
RProcess::Logon 212–13, 273–4
RSemaphore 85–8, 228–33
RSessionBase 268–91, 304
  see also client/server...
  concepts 268–91, 304
  methods 268–9, 284–5
r_SimpleEx_cba 40–9
r_SimpleEx_dialog 362–6
r_SimpleEx_form 362–6
r_SimpleEx_menu 39–49, 321
RSocket 85–8, 104, 268, 304–24
  see also sockets
RSocketServ 304–24
  see also sockets
rss files 32–49, 332–3, 340–93
  see also resource files
RSS_SIGNATURE resource 343–5
RSubSessionBase 287–92
RTextBuff 271–4
RTF see Rich Text Format
RThread 85–8, 214–19, 220–8, 235, 242, 273–4, 282–3, 285–7
  see also threads
RThread::Create 215–18
RThread::Kill 218–19
RThread::Logon 219–20
RThread::Open 216–17
RThread::RequestComplete 273–4
RTimer 248, 250–6, 259–60
run area, virtual memory map 63–6, 213, 222–8
Run Project 29
RUNBOTH (RB) 143
RunDlgLD 358, 368–72
RunError 239, 241, 246–7
RUNINSTALL (RI) 143
RunL 69, 219, 237–65, 268, 275–6, 311–24
  see also active scheduler
  concepts 237–65, 268, 275–6, 311–24
  implementation 240–1, 250–6, 316–22
RUNREMOVE (RR) 143
RUNWAITEND (RW) 143

Samsung, Symbian ownership 11
sandbox see run area
Save 361–6
SaveFormDataL 361–6
screens 11–16, 73–5, 124–5, 267–8, 325–93, 395–424
  see also graphical user interface framework
  concepts 11–16, 73–5, 124–5, 267–8, 325–42
  platforms 11–16, 325–32
  Series 60 (Nokia) 328–30
  Series 80 (Nokia) 330–1
  specifications 13–16, 395–424
  UIQ 326–8
SD memory cards 61
SDKs see software development kits
Secure Identifier (SID), platform security 427–8
security issues see platform security
segmented dynamic buffers, concepts 187–91, 192
semaphores
  see also synchronization
  asynchronous functions 232, 236–65

concepts 66–8, 69, 85–8, 228–33, 236–7
creation 231
opening 231
protection uses 229–30
Symbian OS 232
uses 229–30, 232, 236–7
Send 269–92, 309–24
Sendo X 12, 22, 420
SendReceive 269–92
SendTo 309–10
serial cable connection 8
serial communications server, communications architecture 76, 78–9
Series 60 (Nokia) 12–16, 24–8, 31–51, 72–5, 102–3, 126–8, 138–47, 325–93, 400–21
see also CAkn...
characteristics 328–30
classes 33–49, 334–93
control structures 373–87
data input 329–30, 348–59
dialogs 329–30, 348–66, 367–72
emulator 20–1, 24–8, 50–1, 126–8, 147–9
forms 361–6
GUI architecture 12–16, 31–49, 72–5, 325–93
header file 32–8, 138–47
icons and captions 391–3
package file 51–3, 138–47
panics 102
project build file 47–9, 121–3
quick-start development examples 20–3, 25–8, 31–49
resource file 38–43, 45, 121–3, 348–72
screens 328–30
SDK 22–3, 25–8, 31–49, 115–49
stock dialogs 368–72
view architecture 390–1
Series 80 (Nokia) 12, 22–3, 27, 31–49, 72–5, 148–9, 252, 325–93, 411, 412
see also CEik...; Communicator...
characteristics 330–1
classes 35–49, 334–93
control structures 373–87
data input 330–1, 348–59
dialogs 112, 331, 348–59, 368–72
emulator 50–1, 148–9
GUI architecture 12, 31–49, 72–5, 325–93
header file 35–8
icons and captions 391–3
package file 51–3
project build file 48–9
quick-start development examples 22–3, 27, 31–49
resource file 38–43, 348–72
screens 330–1
SDK 22–3, 27, 31–49, 148–9
stock dialogs 368–72
view architecture 387–90
Series 90 (Nokia) 12, 410
Series60Ex 25–30, 117
Series80Ex 27–30
servers
see also file...; font and bitmap...; socket...; window...
active objects 267–8, 275–6, 277–83
asynchronous functions 267–8, 275–6, 277–83, 305–24
client/server model 59–60, 73–5, 220, 267–92, 296
concepts 59–60, 73–5, 220, 267–92
CSharableSession 268–91
definition 267
emulator 285–7
ETEL server 76, 78–9
examples 270–87
execution flow 59–60, 267–8
GUI 267, 285
implementation 276–83, 287–92
kernel server 66–8
message-processing example 280–2
pointers 280–3
serial communications server 76, 78–9
service-invoking methods 274–6, 318–22
shutdown issues 284–5, 303–4
sockets 76–9, 85–8, 104, 268, 293–4, 296–324
starting 271–4, 277–80, 285–7, 288
subsessions 287–92
TCP/IP 296–324
TextBuffServ example 270–87
transient servers 284–5
types 59, 66–8, 73–5, 267–8
window server 73–5, 267–8, 380
servers-side code, sockets 298–324
ServerThreadFunction 287
ServiceL 270–91
sessions
client/server model 268–92
subsessions 287–92
SetActive 240–4, 251–6, 257–63, 275–6, 318–22
SetAddress 307–24
SetArrayL 376
SetBrushColor 383–4
SetBrushStyle 383–4
SetContainerWindowL 378
SetControlCaptionL 366–8
setdefault 148
SetDefaultView 387–91
SetDimmedNow 366–8
SetItemDimmed 365–6
SetLength 184–5, 201
SetMax 184–5
SetPenColor 383–4
SetPenSize 383–4
SetPenStyle 383–4
SetPort 307–24
SetPriority 210–11, 217–18
SetProtected 213
SetRect 339–40, 358–9, 378
SetReserveL 187–91

SetState 375
SetTextL 356–7, 373
SetTitleL 366–8
SetType 208–9
shapes, drawing controls 384
shared memory, concepts 57, 103–8, 220–8
Sharp 17
Short Messaging Service (SMS) 6–7, 10, 14–16, 56, 323, 429
Shrink 383–4
SHUTDOWNAPPS (SH) 139–47
SID see Secure Identifier
Siemens
 SX1 22, 421
 Symbian ownership 11
Signal 229–33
 see also synchronization
SIM see Subscriber Identification Module
simple/compound controls, contrasts 378–9
SimpleEx 31–49, 138–47, 249–56, 321–2, 332–93
 active objects example 249–56, 321–2
 class-hierarchy diagrams 36–8
 Draw 339–40, 358–9, 377, 379–84
 overview 31–2
SimpleEx_app.cpp 43–4, 49
SimpleEx.cpp 43–4
SimpleEx_Doc.cpp 44–6
SimpleEx.mmp 47–9, 121–3, 346–8
SimpleEx.pkg 51–3, 347–8
SimpleEx.rss 38–49, 342–8
SimpleEx.sis 53, 347–8
SimpleEx_UI.cpp 44–6, 252–6
SimpleEx_View.cpp 46–7, 254–6, 339–40
single-shot permissions, platform security 426
sis files 19–20, 32–49, 51–3, 136–47, 332–93
 see also installation
 concepts 19–20, 32–49, 51–3, 136–47, 332–93
 creation 137
 embedded sis files 142–3
 language support 144–5, 347–8
SISAPP 139–47
Size 160–1, 187–91
slow executive kernel calls, concepts 68
smartphones
 see also mobile phones
 benefits 2
 browsing 2–3, 7–8, 13–16
 communication methods 2–9
 concepts 1–17, 325
 connectivity features 1, 8–9, 14–16, 56, 75–9, 429
 features 1–9
 hardware limitations 325
 historical background 2–3, 9
 manufacturers 10–12
 messaging 6–7, 14–16, 56, 293–4, 323
 network protocols 2–6, 8, 10, 14–17, 78, 293–324, 396–423
 operating systems 1, 9–13
 PDAs 2–3, 8–9, 16–17
 resource-limitations 9–10
 robustness needs 9–10
SMS see Short Messaging Service
socket 300–2, 307
sockets
 see also RSocket; servers; TCP/IP
 active objects 305–24
 asynchronous functions 305–24
 BSD socket 77, 293–4, 297–304
 C++ 293, 299, 304–24
 classes 85–8, 104, 268, 304–24
 client-side code 298–304
 communications architecture 76–9
 concepts 76–9, 85–8, 104, 268, 293–4, 296–324
 connection 302–4, 305–24
 creation 297–304, 307–24
 destination addresses 307–8
 examples 299–324
 network programming 296–324
 receiving data 303, 310–11, 314–15, 321
 remote web servers 308–9
 sending data 302–3, 309–10, 313–22
 servers-side code 298–304
 shutdown issues 303–4
 Symbian OS API 304–24
 TCP/IP applications 293–324
 weather-information example 315–24
softkey items, resource files 38–49, 344–72
software
 see also applications; development...
 C++ 13, 22, 25–9, 61–2, 69–72, 74–5, 81–113, 156–8, 293, 299–324
 developer prospects 1, 10
 interrupts 68
 Java 13
 titles available 1, 10, 13
software development kits (SDKs) 11–12, 19–53, 79, 102–3, 115–49
 see also development tools
 build flow 32–49, 119–23
 components 20–1, 115–49
 concepts 11–12, 19–25, 115–49
 directory structure 115–17
 documentation directories 117
 examples 22–3, 25–30, 31–49, 115–49
 getting 21
 problems 29–30
 switched SDKs 147–9
 tips and traps 30
Sony Ericsson
 see also UIQ
 P800 12, 23, 422
 P900 12, 13–14, 23, 385, 423
 P910 326, 327, 424
 Symbian ownership 11
Sort 197

SOURCE 104, 106, 109, 121–3, 128–31, 204, 263
source files, concepts 43–7, 121–3, 128–31, 332–93
SOURCEPATH 104, 106, 109, 121–3, 128–31, 204, 263
special handshakes, TCP 295
specifications, mobile phones 13–16, 395–424
sprintf 180–1
src directory 43–4, 47
stack and heap chunk, concepts 62–3
Standard Template Library (STL) 82
Start 237–8, 242–5, 257, 262–3, 278–87
start-up code, ROM 60–1
StartBackground 261–3
StartExe 109, 206
StartL 269–91
StartRunning 259–63
StartServer 273–4, 277–80, 285–7
StartThreadL 214–15
State 375
state machines, active objects 312–22
static data chunks, concepts 63–5
static interface DLLs
  see also dynamic link libraries
  concepts 57–9, 70, 75, 103–8
static libraries
  see also import libraries
  concepts 57–9, 75, 103–8
  creation 104
STATIC_CAST 355–7
status bars 326–32
status checks, processes 211–12
STL see Standard Template Library
stock dialogs
  see also dialogs
  concepts 368–72
Stop 242–5, 250–6, 257
storage media 14–16, 61, 396–424
  see also MMC...; removable memory cards
  specifications 14–16, 396–424
StoreL 336–40
stray-signal exceptions, active objects 257–8
strcmp 173
streaming media 293–4
Streaming Media network service 296
string literals, concepts 156–8, 161–2
strings
  see also descriptors
  binary data 155
  concepts 154–8, 345–8
STRUCT keyword, resource files 341–54, 374
subsessions
  see also CObject
  client/server model 287–92
  creation 290–1
  example 288
  file server 288–92
  workings 289
sub-strings, descriptors 174–7
Subscriber Identification Module (SIM) 4
suffixes, naming conventions 93–103, 111–12
SWI instruction 68
SWInstall 426, 428
switched processes, concepts 63–6, 68–9, 213, 222–3
symbian 115
Symbian Ltd 1, 10–12
  see also UIQ Technology AB
Symbian OS
  see also operating systems; smartphones; sockets
  application engines/services/protocols 79–80
  architecture 55–80, 387–91
  basic data types 82–3
  C++ concepts 81–113, 156–8, 293, 299–324
  classes 83–8, 110–11, 154–201, 333–40
  client/server model, concepts 59–60, 73–5, 220, 267–92
  communications architecture 2–6, 13–16, 56, 75–9, 293–324
  competitors 11, 16–17
  components 55–6
  concepts 1, 10–17, 19–24, 55–80
  controls 355–7, 372–87
  development tools 19–53, 115–49
  DLLs 57–9, 67, 69–72, 76–9, 103–8, 128–31, 333–40
  emulator 20–2, 24–30, 50–1, 108–9, 116–17, 119–23, 124–8, 147–9, 213, 285–7
  flexible architecture 11–12
  GUI architecture 11, 21–2, 31–49, 56–8, 72–5, 108–9, 117, 151–2, 267–8, 325–93
  high performance graphics 75
  historical background 10–13
  kernel 55–6, 62–8
  memory 60–6
  multitasking aspects 11, 56–7, 128, 237–8
  naming conventions 58, 70, 83–4, 110–12
  network connections 322–4
  'open' aspects 1
  overview 11–13
  owning manufacturers 11
  phone specifications 13–16, 395–424
  platform security 425–9
  platforms 11–16, 140–7, 325–32
  programming basics 81–113
  quick start guide 19–53
  reference platforms 11–12, 325–32
  SDKs 11–12, 19–53, 115–49
  semaphores 232
  Socket API 304–24
  TCP/IP 11, 56, 76–9, 293–324
  text console 151–4, 259
  v6.0 15, 23
  v6.1 22, 413–14, 420–1

Symbian OS (*continued*)
v7.0 15, 23, 24, 79, 147–9, 210, 323–4, 396–424
v8.1 23, 213, 285–7, 415–17
v9 57, 58–9, 108–9, 425–9
Symbian_Base 25–30
synchronization
see also critical sections; mutexes; semaphores
concepts 228–33
threads 228–33
synchronous functions, concepts 235–6
SyncML, application protocols 56, 80
sys directory 428–9
system/apps directory 137–46
system/data directory 137–46
SystemGc 382–4
SYSTEMINCLUDE 106, 109, 121–3, 128–31, 204, 263
system/lib directory 130, 137–46

T (data type) classes, concepts 84–8, 97–9, 110
T-Mobile 323
tabs 326–32, 359–72
TAny 83, 95–6
TARGET 49, 104, 106, 109, 121–4, 128–31, 204, 263
TARGETPATH 121–4, 204
TARGETTYPE 49, 104, 106, 121–4, 128–31, 204, 263
TArrayFixed 195–9
TBool 83, 243
TBuf 156–94, 220, 251–6, 305–24, 343–8
see also buffer descriptors
concepts 156–94, 220, 343–8
memory layout 163
TBufBase 159–60
TBufC 156–94
see also buffer descriptors
concepts 156–94
HBufC 169
memory layout 165–6
TBufCBase 159–60
tbuffserver.h 276
TCB see Trusted Computing Base
TChar 83, 84, 178–9
TCleanupItem 96–9
TCoeInputCapabilities 386
TCP (Transmission Control Protocol) 293–324
TCP/IP
see also sockets
applications 293–324
client/server model 296–324
concepts 293–324
introduction 294–6
layering diagram 295
network connections 322–4
network programming 293–324
protocols 294–5
Symbian OS 11, 56, 76–9, 293–324
virtual connections 295–324
tcpip6.prt 76
TDblQue 200
see also linked lists
TDblQueLink 200
TDes base class 159–61, 166, 172–86, 221
TDesC base class 157–62, 165–6, 172–86
see also base classes; descriptors
Techview 12
telephony server see ETEL server
Telnet network service 296, 315–17
templates
arrays 191–2, 194–9
C++ 81–2, 96–7
terminations
processes 211
threads 218–19
text 46–7
TEXT... 141–2
text 141–2
TEXT... 146
text 146, 339–40, 381–2, 384
text console, concepts 151–4, 259
text notices, installation 141–2
textbuffclient.h 270
textbuff.h 276–7
TextBuffServ 270–87
TFindChunk 225
TFindHandleBase 208–9
TFindProcess 208–10, 225
TFindSemaphore 231
TFindThread 217, 225
TFixedArray 191–2
third-party suppliers 1, 11, 13
threads
see also RThread
active objects 232, 237–65
cautionary uses 214
chunks 223–8
client/server model 267–92
concepts 56–7, 64–5, 66–9, 85, 203–4, 213–19, 220–8
creation 214–18
definition 56
end-signaling method 219–20
executables 215
inter-thread communications 66–8, 207–8, 220–8, 282–3
multiple threads 56–7, 69, 128, 203–4, 213–19
opening methods 216–17
pre-emptive multithreading 56–7, 235, 237–8
priorities 217–18
running 214–16
starting 214–16
synchronization 228–33
terminations 218–19
throw/catch exception C++ feature 82, 90
THUMB 116–17, 119
thumb pointers 330
TIdentifyRelation 197
timers, concepts 66–8
TInt types 82–3, 91–2, 99–100, 198–9, 200–1, 208, 223–4, 228–30, 260
TitleFont 339–40
TLeave 99–100
TLibraryFunction 71–2
TLinearOrder 196, 199
TLitC 161–2
tool bars 325–40

touch screens 12–14, 326–32, 396–9, 410, 422–4
see also pens
TPoint 382–4
TPointerEvent 386–7
TProcessId 207–8
TProcessPriority 210–11
TPtr
see also pointer descriptors
concepts 158–94
memory layout 166–9
TPtrC
see also pointer descriptors
concepts 158–94
memory layout 166–9
training, OS requirements 10
transcoding features, WAP 7
TRAP 90–5
trap mechanism, concepts 89–103, 111–12, 152–3, 169, 198, 241, 245–7, 250–6, 278–83
TRAPD 91–5, 278–80
TReal types 83
TRect 381–4
TRequestStatus 212–13, 235–48, 273–4, 305–24
see also asynchronous functions
TRes 159–60
TrimAll 182–3
TrimLeft 182–3
TrimRight 182–3
Trusted Computing Base (TCB) 428–9
TSize 382–4
TSocketAddr 311
tsy files 78
TSY modules, ETEL server 76, 78–9
TText types 82–3
TTime 368–9
TUint types 82–3, 177–9, 185–6, 188–9
TVersion 272–4
two-phase constructors, concepts 100–2
typedefs 83
UART 78
UDEB 29, 116–28
UDP (User Datagram Protocol) 294–324
see also sockets
client/server model 296
concepts 294–5
layering diagram 295
_UHEAP_MARK 153
_UHEAP_MARKEND 153
UI see user interfaces
UI classes, applications 31–49, 332–93
UI control framework see CONE
UID 104, 106, 109, 121–4, 128–31, 208, 263
UID@symbiandevnet.com 124
UID1 123–4, 128–31
UID2 123–4
UID3 123–4
UIDs see unique identifiers
UIKON, concepts 73–5, 373
uikon.rh 353, 373
UIQ 12–16, 22–3, 26, 31–49, 72–5, 136–7, 140–7, 252, 325–93, 396–424
see also CQik...; Motorola; Sony Ericsson
characteristics 326–8
classes 34–49, 334–93
concepts 326–8
control structures 373–87
data input 327–8, 348–59
dialogs 328, 348–59, 368–72
emulator 50–1, 147–9
GUI architecture 12–16, 31–49, 72–5, 325–93
header file 34–8, 140–7
icons and captions 391–3
package file 51–3, 140–7
paper metaphor 327–8
project build file 48–9
quick-start development examples 22–3, 26, 31–49
resource file 38–43, 348–72
screens 326–8
SDK 22–3, 26, 31–49, 117–49
stock dialogs 368–72
view architecture 387–90
UIQ Technology AB 12
see also Symbian Ltd
UIQExamples 26, 117
ukon.rh 344, 353
UMTS network protocol 6
Unicode 83, 155, 159, 186, 341–2
unique identifiers (UIDs) 31–49, 104, 106, 109, 121–4, 139–47, 334–93
concepts 31–49, 121–4, 139–47, 334–40, 388–91
identifiers 123–4
sis files 139–47
unit of protection, platform security 426
Unix 235
UpperCase 181–2
UREL 51–3, 116–25, 138–47
URLs 7
USB connectivity 1, 8, 14, 16, 19, 56, 304, 396–424
UseBrushPattern 383–4
UseFont 383–4
User 82, 92–5, 104
user interfaces (UI)
see also graphical...; Series...; UIQ...
classes 31–49, 332–93
concepts 11–16, 31–49, 72–5, 124–5, 325–93
customization 72–5
OS requirements 10, 11–12, 72–5, 325–32
specifications 13–16, 395–424
types 11–16, 325–32
user library, concepts 67–8
User::After 109, 214–15, 238, 257, 259–63
User::AllocLC 98–9
user.dll 68, 151
USERINCLUDE 106, 109, 121–3, 128–31, 204, 263
User::InfoPrint 109, 214–15, 386–7
User::Leave 90–5, 198–9, 205–7, 215, 220, 250–6

User::LeaveIfError 92–3, 198–9, 205–7, 215, 220, 250–6, 280–3
User::LeaveIfNull 92–3
User::LeaveNoMemory 92–3
User::Locked... 233
User::Panic 102–3, 277–83
User::PrintInfo 162
User::QueryVersionSupported 272–4, 279–83
User::WaitForRequest 212, 219, 236–7, 245–6, 248, 258, 273–4, 305–24
UTF-8186, 341

variables
  global variables 108, 111
  naming conventions 110–11
vc6 28–9
Verizon Wireless network 4
Version 272–4
VGA screens 12–13, 326–8
video 3, 6–7, 14–16
video teleconferencing 3
view architecture
  concepts 387–91
  creation 387–90
  Series 60 (Nokia) 390–1
view classes 31–49, 326–32, 334–93
ViewActivatedL 387–91
ViewConstructL 387–91
ViewDeactivated 387–91
ViewId 387–91
virtual buttons, emulator 126–8
virtual connections, TCP/IP 295–324
virtual declarations, polymorphic DLLs 70–2
virtual drives, emulator 125–8
virtual functions 81–2, 86, 95, 107–8, 387–91
virtual keyboards 12–14, 126–8, 385–6
virtual memory addresses
  concepts 61–6, 222–8
  memory map 63–6, 213, 222–8
VirtualKey 126–8
voice transfers 2–3
void data type 83, 244–5

W-CDMA technology 6
Wait 229–33, 243–6, 248, 258
  see also synchronization
WaitForRequest 213, 219, 236–7, 243–4, 245–6, 248, 258, 273–4, 305–24
WAP browsers 7–8, 14–16, 293, 396–424
WCDMA network protocol 399, 408
web browsing see browsing
WiFi network protocol 15, 293, 322–4
wildcard searches
  descriptors 175–6
  processes 208
Win32 development tools 19–53
window server
  see also servers
  animation plug-ins 75
  concepts 73–5, 267–8, 380
  CONE 380
window-owning/lodger controls, contrasts 377–9
Windows 17, 19–53, 56, 104, 108–9, 116–17, 119–23, 124–8, 147–9, 213, 285–7
  see also Microsoft
  2000 19
  CE 17
  development package (Win32) 19–53
  emulator 20–2, 24–30, 50–1, 108–9, 116–17, 119–23, 124–8, 147–9, 213, 285–7
  Mobile family 17
  NT 19
  XP 19
Windows development tools 19–53, 124–8
  see also development tools
  components 20–2
  concepts 20–2, 25–9, 124–8
  debuggers 20–2, 120, 124
  examples 25–30, 31–49, 332–3
  getting 21–2
  monopoly situation 23–4
  problems 29–30
  providers 22
  quick test 25–9
  tips and traps 30
WindowTitle 126–7
wins 25–8, 50–1, 104, 116–17, 119–28, 133–6, 154, 286–7
winsb 26–8, 29, 51, 116–17, 119
winscw 26–8, 51, 116–17, 119, 154
WinsMain 286–7
WLAN 79
WML 7
Word 17
WORD, resource files 341–2, 374
World Wide Web 294
  see also browsing
wrapper classes 96–9, 106–7, 274–6, 372
Write 85–8, 187–91, 220, 264
WriteL 220–8, 282–3
www.epocware.com 1
www.forum.nokia.com 30
www.gnuPoc.sourceforge.net 24
www.handango.com 1
www.symbian.com/developer/index.html 426
www.symbian.com/developer/sdks.asp 21
www.symbian.com/phones 395
www.symbiansigned.com 427–8
www.wunderground.com 315–24

x86-based Windows binaries 119, 127
xHTML 7, 396–424

z: drive
  see also Read Only Memory
  concepts 60–1, 116–17, 125–8
Zero 184–5
ZeroTerminate 183–4