

Developing Software for Symbian OS

Second Edition



A Beginner's Guide to Creating Symbian OS
v9 Smartphone Applications in C++

symbian

Steve Babin

Developing Software for Symbian OS

A Beginner's Guide to Creating Symbian OS v9 Smartphone Applications in C++

Steve Babin

Reviewed by

Antony Pranata, Bruce Carney, Chris Notton, Douglas Feather, Freddie Gjertsen, Howard Sykes, Jehad Al-Ansari, Jo Stichbury, Laura Sykes, Lucinda Barlow, Mark Jacobs, Matthew O'Donnell, Neil Hepworth, Ricky Junday, Roderick Burns, Steve Rawlings, and Warren Day

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Developing Software for Symbian OS

**A Beginner's Guide to Creating
Symbian OS v9 Smartphone
Applications in C++**

Developing Software for Symbian OS

A Beginner's Guide to Creating Symbian OS v9 Smartphone Applications in C++

Steve Babin

Reviewed by

**Antony Pranata, Bruce Carney, Chris Notton, Douglas Feather,
Freddie Gjertsen, Howard Sykes, Jehad Al-Ansari, Jo Stichbury,
Laura Sykes, Lucinda Barlow, Mark Jacobs, Matthew O'Donnell,
Neil Hepworth, Ricky Junday, Roderick Burns, Steve Rawlings,
and Warren Day**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Copyright © 2007

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.wileyeurope.com or 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.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

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, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

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

Anniversary Logo Design: Richard J. Pacifico

British Library Cataloguing in Publication Data

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

ISBN: 978-0-470-72570-2

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 (Jo Stichbury)	ix
Foreword (Warren Day)	xi
Biography	xiii
Author Acknowledgments	xv
Symbian Press Acknowledgments	xvii
Symbian OS Code Conventions and Notations Used in the Book	xix
1 Smartphones and Symbian OS	1
1.1 Notes on this New Edition	1
1.2 Smartphone Concepts	2
1.3 Smartphone Features	3
1.4 The Mobile OS	11
1.5 Symbian OS – A Little History	12
1.6 Symbian OS Smartphones	15
1.7 Other Smartphone Operating Systems	20
2 Symbian OS Quick Start	23
2.1 What Do You Need to Get Started?	23
2.2 Firing Up the Development Tools	31
2.3 Simple Example Application	38
2.4 Building and Executing on the Emulator	56

2.5	A Carbide.c++ Project	58
2.6	Building for the Smartphone	59
3	Symbian OS Architecture	63
3.1	Components in Symbian OS	63
3.2	Multitasking in Symbian OS	64
3.3	Shared Code: Libraries, DLLs, and Frameworks	65
3.4	Client–Server Model	68
3.5	Memory in Symbian OS	70
3.6	The Kernel	77
3.7	Active Objects and Asynchronous Functions	81
3.8	GUI Architecture	83
3.9	High-Performance Graphics	85
3.10	The Communication Architecture	86
3.11	Application Engines and Services	90
3.12	Platform Security	90
4	Symbian OS Programming Basics	93
4.1	Use of C++ in Symbian OS	93
4.2	Non-standard C++ Characteristics	94
4.3	Basic Data Types	94
4.4	Symbian OS Classes	95
4.5	Exception Error Handling and Cleanup	101
4.6	Libraries	115
4.7	Executable Files	118
4.8	Naming Conventions	119
4.9	Summary	122
5	Symbian OS Build Environment	123
5.1	SDK Directory Structure	123
5.2	Build System Overview	126
5.3	Basic Build Flow	126
5.4	Build Targets	131
5.5	What is a UID?	135
5.6	The Emulator	137
5.7	Building Shared Libraries	141
5.8	DLL Interface Freezing	144
5.9	Installing Applications on the Smartphone	149
6	Strings, Buffers, and Data Collections	161
6.1	Introducing the Text Console	161
6.2	Descriptors for Strings and Binary Data	165
6.3	The Descriptor Classes	168
6.4	Descriptor Methods	186
6.5	Converting Between 8-Bit and 16-Bit Descriptors	198

6.6	Dynamic Buffers	199
6.7	Templates in Symbian OS	203
6.8	Arrays	205
6.9	Other Data Collection Classes	213
7	Platform Security and Symbian Signed	217
7.1	What is Platform Security?	217
7.2	What Platform Security is Not	218
7.3	What this Means to a Developer	219
7.4	Capabilities for API Security	219
7.5	Application Signing in Symbian	232
7.6	Getting Your Application Symbian Signed	238
7.7	Developer Certificates	244
8	Asynchronous Functions and Active Objects	247
8.1	Asynchronous Functions	247
8.2	Introducing Active Objects	249
8.3	The Active Scheduler	254
8.4	Active Scheduler Error Handling	258
8.5	Active Object Priorities	260
8.6	Canceling Outstanding Requests	260
8.7	Removing an Active Object	262
8.8	Active Object Example	262
8.9	Active Object Issues	269
8.10	Using Active Objects for Background Tasks	271
9	Processes, Threads, and Synchronization	277
9.1	Processes	277
9.2	Using Threads on Symbian OS	286
9.3	Sharing Memory Between Processes	292
9.4	Memory Chunks	293
9.5	Thread Synchronization	297
10	Client–Server Framework	303
10.1	Client–Server Overview	304
10.2	A Look at the Client–Server Classes	305
10.3	Client–Server Example	306
11	Symbian OS TCP/IP Network Programming	323
11.1	Introduction to TCP/IP	324
11.2	Network Programming Using Sockets	327
11.3	Symbian OS Socket API	334
11.4	Example: Retrieving Weather Information	345
11.5	Making a Network Connection	356

12 GUI Application Programming	359
12.1 Symbian OS User Interfaces	360
12.2 Anatomy of a GUI Application	365
12.3 Application Classes	367
12.4 Resource Files	377
12.5 Dialogs	387
12.6 Symbian OS Controls	405
12.7 View Architecture	409
12.8 Application Icon and Caption	409
References	413
Index	415

Foreword

Jo Stichbury
Symbian Press

Symbian has recently reported combined cumulative licensee sales of over 145 million smartphones worldwide. And with over 120 smartphone models available, Symbian has by far the largest installed base of smartphones, at approximately 72% of the market. Calculations suggest that there are two Symbian smartphones shipping every second.

Symbian smartphone users can buy and install after-market applications, and evidence is that they are doing this in increasing numbers. Symbian Signed has recently reached a milestone, having signed over 20,000 application and content files, ranging from games and multimedia applications to enterprise and messaging utilities.

Symbian OS is a great platform for creating applications such as these, and others. New handsets are shipping with technologies such as Wi-Fi, GPS, DVB-H, HSDPA, IMS, multi-megapixel cameras, multi-GB storage, biometrics, industry-leading security, 3D hardware accelerated graphics, tilt-sensors, DNLA and uPnP (Universal Plug and Play), demand paging, VoIP and much more. . . there's scope to create some of the most imaginative smartphone applications ever.

What about the people creating these applications? Symbian has over 300 Platinum Partner companies and a community of over 55,000 developers. These are not just based in mainland Europe, but are worldwide: in Australia, Canada, Brazil, USA, India, People's Republic of China, Russian Federation, Singapore, Japan, and Thailand, to name just a few of the countries where Accredited Symbian Developers can be found.

Symbian OS offers a range of development environments for application development, from C and C++ to managed runtime environments such as Java, Python, and Ruby. The best access to the smartphone

hardware, and the best performance on Symbian OS, comes from native C++ and Software Development Kits for both major UI platforms – S60 and UIQ – are freely available. But, however available a development environment may be, it isn't necessarily easy to start work on a platform, particularly a mobile operating system that demands robust and efficient code. Newcomers to Symbian OS can be intimidated by the vocabulary and the range of information, libraries, and tools provided.

This book sets out to make it easier to get started. It is aimed primarily at C++ developers who are new to Symbian development. It makes no assumptions of knowledge about Symbian OS, or any other mobile platform, although it does assume a reasonable understanding of C++.

After introducing the Symbian platform and explaining how you can get the tools and kits you need, Steve goes through the fundamental topics needed to write an application to run on either UI platform built on Symbian OS v9 (UIQ 3 and S60 3rd Edition). This new edition of the book includes a chapter on platform security and application signing (Chapter 7), and a set of downloadable sample code for the major code examples in the book, for both UI platforms.

The original book *Developing Software for Symbian OS*, published in 2005 for versions of Symbian OS pre-dating v9, has been a very popular Symbian Press title. It was a ground-breaking book because it was our first aimed squarely at beginners. Readers appreciated Steve's practical approach to the subject, and they report having learnt far more than just the basics – it was a book packed with valuable and accessible information.

Based on the number of times we've seen the original book recommended on developer discussion forums, and our own belief in the quality of the title, we decided to ask Steve to create a new edition for Symbian OS v9. This book is the result, and I believe it is a valuable contribution to the Symbian Press series – our 23rd Symbian Press title. I anticipate it to be a useful addition to the bookshelves of many developers as they learn how to create more applications to help those millions of smartphone users to 'do more with Symbian'.

Foreword

Warren Day

Senior Technical Trainer, Symbian Software Ltd

Why read this book?

It has the best explanations I've read of the fundamental things we must understand about Symbian OS, in that it covers the 'what' and the 'how' of Symbian C++ software development.

And there are several parts to the how.

It is important to understand what facilities are provided (so we don't reinvent the wheel). Just as importantly we should know both how to *think* about these components and how to use them appropriately. A car isn't meant to be driven in just first gear; doing so would quickly empty the fuel tank and put great strain on the vehicle's internal workings.

As it is for a vehicle's transmission system, so it is also true for a software system. Misusing software components can result in everything from short battery life (so phones need frequent recharging) to inappropriate or poor functionality, and phones that are difficult to use. By having a clear insight into the fundamentals (such as memory management, string classes and asynchronous events), your software will be more efficient, because it utilizes the underlying system as intended.

We are living in exciting times. Society is evolving; being always-connected to others and the Internet is standard for many. Now smartphones have the ability to make phone calls, access the Internet, capture video, as well as perform personal organizer, word processor and spreadsheet functionality, all from a small handset. However, smartphones are very different from computers in one fundamental way, people have used phones before! The telephone is easy to use and works correctly about 100% of the time. Smartphones should be equally usable and reliable. This is the case with the other things a smartphone can do too, such as

take photographs, play video and music. People have equivalent experiences of reliability: the devices they are used to just work. Thus we now have raised standards for rich functionality, power efficiency and perhaps most importantly, reliability.

So, this book is a good 'start writing and running programs from scratch' book. It covers similar material to Symbian's OS Essentials training course and shows you how to take your first steps to writing good quality applications for Symbian smartphones as they should be written.

Biography

Steve Babin works at IBM developing enterprise software for smartphones based on Symbian OS and Microsoft Windows Mobile. He has a BSEE from Louisiana State University and over 20 years of software development and leadership experience on a variety of products – including medical devices, Java accelerators, avionics, Internet appliances, and system-on-chip silicon devices – using numerous operating systems. Steve is married to Sharon and has a daughter named Hillary. They live in Austin, TX. He is an Accredited Symbian Developer.

Author Acknowledgments

I would like to thank the people at Symbian Press for their interest in, and enthusiasm for, doing this update. It was a pleasure working with them again. I especially want to thank Jo Stichbury for her very thorough reviews and edits, as well as her excellent suggestions and advice. Special thanks also go to Satu McNabb and all the technical reviewers, including Antony Pranata, Steve Rawlings, Howard Sykes, Laura Sykes, Matthew O'Donnell, Jihad Al-Ansari, Chris Notton, Douglas Feather, Neil Hepworth, Bruce Carney, Ricky Junday, Roderick Burns, and Warren Day. Updating the book for Symbian OS v9 was quite a challenge, but I was glad to have the help of such talented people. Thanks also go to the people at John Wiley for their interest in this project, and their hard work – I can only imagine what it takes to bring a book like this all the way through the process and into the reader's hands. I also want to thank my wife Sharon and my daughter Hillary, for putting up with me on this – again.

Symbian Press Acknowledgments

Symbian Press would like to thank Steve Babin for resolutely revising his much-respected repository of knowledge. Thank you Steve, for being professional, reliable, and a pleasure to work with. We'd also like to thank each of the reviewers, and in particular Antony Pranata and Steve Rawlings, who lent us the weight of their experience on each of the UI platforms at very short notice. Thanks must also go to the Symbian Technical Training team for their feedback and support.

We couldn't have put this book together without the team at John Wiley, and it's thanks to Rosie Kemp that it happened so rapidly. Phil Northam came to the rescue in our darkest hour, and Jo has promised to buy him and Satu lunch, in recognition and with gratitude.

Symbian OS Code Conventions and Notations Used in the Book

For you to get the most out of this book, let's quickly run through the notation we use. The text is straightforward, and where we quote example code, resource files, or project definition files, they will be highlighted as follows:

```
This is example code;
```

C++ code for Symbian OS uses an established naming convention. We encourage you to follow it in order for your own code to be understood most easily by other Symbian OS developers, and because the conventions have been chosen carefully to reflect object cleanup and ownership, and make code more comprehensible. An additional benefit to using the conventions is that your code can then be tested with automatic code analysis tools, which can flag potential bugs or areas to review.

If they are unfamiliar, the best way to get used to the conventions is to look at code examples in this book, and those provided with your chosen SDK.

Capitalization

The first letter of class names is capitalized:

```
class TColor;
```

The words making up variable, class, or function names are adjoining, with the first letter of each word capitalized. Classes and functions have their initial letter capitalized while, in contrast, function parameters, local, global, and member variables have a lowercase first letter.

Apart from the first letter of each word, the rest of each word is given in lower case, including acronyms. For example:

```
void CalculateScore(TInt aCorrectAnswers, TInt aQuestionsAnswered);
class CActiveScheduler;
TInt localVariable;
CShape* iShape;
class CBbc; //Acronyms are not usually written in upper case
```

Prefixes

Member variables are prefixed with a lowercase 'i', which stands for 'instance'.

```
TInt iCount;
CBackground* iBitmap;
```

Parameters are prefixed with a lowercase 'a', which stands for 'argument'. We do not use 'an' for arguments that start with a vowel.

```
void ExampleFunction(TBool aExampleBool, const TDesC& aName);
```

(Note: TBool aExampleBool rather than TBool anExampleBool).

Local variables have no prefix:

```
TInt localVariable;
CMyClass* ptr = NULL;
```

Class names should be prefixed with the letter appropriate to their Symbian OS type (usually 'C', 'R', 'T', or 'M'):

```
class CActive;
class TParse;
class RFs;
class MCallback;
```

Constants are prefixed with 'K':

```
const TInt KMaxFilenameLength = 256;
#define KMaxFilenameLength 256
```

Enumerations are simple types, and so are prefixed with ‘T’. Enumeration members are prefixed with ‘E’:

```
enum TWeekdays {EMonday, ETuesday, ...};
```

Suffixes

A trailing ‘L’ on a function name indicates that the function may leave:

```
void AllocL();
```

A trailing ‘C’ on a function name indicates that the function returns a pointer that has been pushed onto the cleanup stack:

```
CCylon* NewLC();
```

A trailing ‘D’ on a function name means that it will result in the deletion of the object referred to by the function:

```
TInt ExecuteLD(TInt aResourceId);
```

Underscores

Underscores are avoided in names except in macros (`__ASSERT_DEBUG`) or resource files (`MENU_ITEM`).

Code Layout

You’ll notice that the curly bracket layout in Symbian OS code, used throughout this book, is to indent the bracket as well as the following statement:

```
void CNotifyChange::StartFilesystemMonitor()
{
    // Only allow one request to be submitted at a time
    // Caller must call Cancel() before submitting another
    if (IsActive())
    {
        _LIT(KAOExamplePanic, "CNotifyChange");
        User::Panic(KAOExamplePanic, KErrInUse);
    }
    iFs.NotifyChange(ENotifyAll, iStatus, *iPath);
    SetActive(); // Mark this object active
}
```


1

Smartphones and Symbian OS

Symbian OS is a full-featured, open, mobile operating system that powers many of today's smartphones. As these smartphones become more powerful and popular, the demand for smartphone software has grown. Symbian smartphones are shipped with a variety of useful pre-loaded and targeted applications, which are selected by each phone's manufacturer. Today, the average Symbian smartphone ships with over 30 pieces of third-party software pre-installed. However, the exciting aspect of Symbian smartphones is that they are 'open', meaning that users can further customize their phone experience by downloading, installing, and uninstalling applications written by third-party developers (or by the users themselves). Users can download applications from a PC to the smartphone through a link such as USB, or Bluetooth technology, or over-the-air via the Internet.

With the largest installed base of smartphones worldwide, Symbian OS offers a great opportunity for software developers to establish themselves in the mobile market by creating novel and exciting software for the growing mass of smartphone users around the world. There is a growing list of Symbian applications available as freeware or as paid downloads on numerous Internet sites (<http://www.handango.com> and <http://www.epocware.com> are good examples). They range from productivity, entertainment, navigation, 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 and inspire software developers to create good software for Symbian smartphones.

1.1 Notes on this New Edition

Developing Software for Symbian OS was first published in 2005, and in the two years since then smartphones have continued their phenomenal

growth rate. The number of Symbian OS smartphones shipped in 2006 alone was 51 million – a 52% increase from the year before. In the first half of 2007, 47.9 million smartphones were shipped (a 39% increase on the same period in 2006) and the total number of Symbian OS phones in circulation now surpasses 145 million. Smartphones now make up 9% of the total mobile market. Symbian continues to be the most widely shipped smartphone OS. According to Canalys, Symbian's share of the smartphone OS market was 72.4% in Q2 2007.¹ Many new Symbian smartphones have been introduced that run on the latest versions of Symbian OS, that is Symbian OS v9, which was a significant upgrade to previous versions of the operating system.

The main purpose of this edition is to update the original *Developing Software for Symbian OS* for Symbian OS v9. The basic programming concepts of Symbian OS have not changed, so much of the content of the core programming chapters remains. The main areas of change include covering the new Symbian OS v9 software development kits (SDKs) and development environment changes, as well as the significant addition of the platform security architecture to v9, which is used to protect the integrity of the smartphone. Since it affects various aspects of development, platform security is discussed in various places throughout this book as needed, and a new chapter has been added to the book to discuss this subject in depth.

In addition to updating the book for Symbian OS v9, we have made sure the book is updated in general for new developments that have occurred since the original book, and we have fixed a few errata reported against the original.

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 you as a programmer to exploit these features to their full potential. (For more information about the typical features and design of a smartphone, please consult *How Smartphones Work*, published by Symbian Press in 2006.)

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

1.2 Smartphone Concepts

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,

¹ <http://www.canalys.com>

microprocessors, semiconductor miniaturization, battery technology and, of course, the invention of telephone and radio, the modern cell 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 cell 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 and/or purse space! This is the basic idea of a smartphone – but a smartphone is more than just a PDA combined with a cell phone. Smartphones also contain features such as a digital camera, video and music players, and GPS, thus combining other portable devices as well.

1.3 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 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.

1.3.1 How Smartphones Communicate

Smartphones, like traditional cell phones, use radio to communicate with base towers, which in turn act as gateways into landline-based communication infrastructures. While traditional cell 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. Improving 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*.

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 cell phone technology.

Second Generation (2G)

Voice-centric digital systems with increased coverage and capacity. Introduces messaging.

Third Generation Transitional (2.5G)

Stepping stone to 3G. Introduces always-on network connections, bandwidths up to 170 Kbps, allowing better Internet browsing, email, and some audio video. GPRS has been the dominant technology here.

Third Generation (3G)/Fourth Generation (4G)

Supports 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.

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 USA, 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. Cell phones supporting GSM 900 and GSM 1800 will ensure coverage in Europe and many other areas outside the USA, while GSM 850 and GSM 1900 are used in the USA (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 either not locked to a specific carrier, or locked to the carrier that the SIM is associated with.

CDMA

CDMA, which stands for Code Division Multiple Access, is a cell phone standard that competes with GSM. CDMA currently dominates in the USA 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.

EV-DO is the high-speed, 3G version of CDMA. EV-DO supports rates up to 2.4 Mbps (with actual speeds averaging closer to 1 Mbps) and is adopted by many services including Verizon and Sprint in the United States.

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 dialup connection is started on a land-based telephone line using a PC modem. With CSD, you do not need any extra data plan like GPRS to send data. You can use up your existing voice minutes.

There are two major disadvantages to using CSD, however. First, it takes a long time to connect since this involves dialing a number and waiting for the server to answer the call. Second, it is 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.

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 (as CSD does), and the smartphone feels as if it is always connected.

Protocol-wise, GPRS runs on top of GSM. 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 in dialing a number and establishing a call.

GPRS is categorized as a 2.5G technology due to its speed. Although many networks now support higher-speed protocols, GPRS was an excellent stepping stone and preview for the now-available 3G technologies.

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, and these are the transfers that would benefit the most from constant bit rates.

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

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. Most modern GSM smartphones now support EDGE.

UMTS

UMTS, short for Universal Mobile Telecommunication Services, is a high-speed data transfer which supports bandwidths up to 2 Mbps. This

protocol is the basis of third generation mobile communications that make many media-rich services a possibility. 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. It's exciting that many service providers now have this high-speed service.

HSDPA

HSDPA, short for High-Speed Data Packet Access, is based on UMTS and supports even higher speeds than UMTS (up to 3.6 Mbps). It is known as a 3.5G technology. Many service providers have launched HSDPA support recently.

Wi-Fi

Wi-Fi is a popular communication protocol used to connect devices such as PCs, game consoles, PDAs, and mobile phones to a network wirelessly. Wi-Fi service is found not just at corporations for their employees to access their network, but also for Internet users at coffee shops, hotels, book stores, airports, and even parks.

Wi-Fi is a high-speed protocol when compared with cell tower-based protocols – it supports speeds up to 54 Mbps. Wi-Fi is a much shorter-range protocol when compared with cell-based radio protocols, with its range being about 600 ft.

In the last couple of years, numerous smartphones have been released with Wi-Fi capability. This allows for high-speed data transfer when in range of a Wi-Fi 'hotspot'.

A standard called WiMAX, which is in its beginning stages now, promises to extend the range of Wi-Fi to 2–3 miles for more global area coverage.

1.3.2 Smartphone Messaging

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

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 for many types of communication exchange, and is less intrusive than making a voice

call. SMS is a part of the GSM communication platform and used by cell 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.

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 even video along with your text message. MMS is a natural complement to smartphones due to its need for, and use of, audio and video capability and, in many cases, an attached video camera. 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.

Email

Having the ability to keep up with your email while on the road is a standard feature found in smartphones. With the 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.

Fax

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

1.3.3 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.

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 useable 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 websites to the WAP markup language before transferring to the mobile device. This is known as *transcoding*.

HTML

Although WAP was considered important for earlier mobile devices, the smartphones today have better memory, processing power, and displays. Because of this, it is feasible to include traditional HTML browsers that load websites directly in their native format, similar to a browser on a PC. Many smartphones have HTML browsers and those usually include WAP capability as well – sometimes combined in one browser.

1.3.4 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 link directly 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 mediums, along with some of their uses.

USB/Serial cable connection

Smartphones can be connected to a PC via either a USB or 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. Many products provide a cradle into which the smartphone can be plugged, for both PC connectivity and for charging the phone's battery.

Infrared (IR)

The smartphone provides the capability to communicate through an infrared port to a PC or other device such as a PDA or phone. 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.

Bluetooth technology

Bluetooth technology is a short-range radio technology that enables devices to find and connect to each other and communicate. While technologies like GSM replace long lengths of wire, Bluetooth technology replaces the rat's nest of short wires connecting various pieces of

equipment. Unlike infrared, Bluetooth technology does not require a line of sight and will even communicate through walls.

With Bluetooth technology, you can connect more conveniently to PCs and PDAs than you can with cable and IR to download applications and synchronize user data. 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 or connect to 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, cable, or IR to provide the PC with Internet connectivity.

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

1.3.5 Location Based Services

More smartphones are being equipped with features that allow a device to know its physical location. In some cases – for example, the Nokia N95 – Global Positioning Satellite (GPS) hardware is built into the phone. Alternatively, the phone may use the cell service's location information (e.g., the GSM cell id can tell which particular cell tower you are communicating with), which is being increasingly supported for emergency purposes.

Location Based Services (LBS) uses the device's location to provide a personalized experience to the user. The most obvious (and useful!) is the traditional map-based turn-by-turn navigation service. Other interesting applications are:

- Software that shows the hotels, stores, theaters, and other businesses in your immediate vicinity along with any special discounts or other deals offered. Weather and traffic can also be reported for your vicinity.
- Software that allows you to track your children who are carrying LBS-enabled smartphones.
- Software to help with health and fitness by tracking how far and fast you are walking or jogging.
- Games that use your location as input. For example, Blister Entertainment has a fishing game called Swordfish where you find schools of fish based on your phone's real location (launched in North America on Sprint Nextel, Bell Mobility, and Boost Mobile). Once you find them, you can catch them, arcade style. There are a few other LBS games around, and could be many more to come.

LBS is at a very early stage for smartphones at the time of this writing, but its future looks promising. Reference <http://www.lbszone.com> for more details on LBS.

1.3.6 Mobile TV

As smartphones get more powerful, have better displays and faster connections, mobile TV video is becoming a reality for these devices. Mobile TV content can consist of live television shows, news and sports, as well as movies on demand and music videos.

Currently, most mobile TV services use the existing cellular network for transferring video data using both 2G and 3G transport protocols. 3G is, of course, the better transport since a good TV picture requires a lot of bandwidth.

However, to realize its full benefits, mobile TV needs a dedicated video communication protocol, and the most promising protocol for this is *DVB-H*. DVB-H is an adaptation of the standard digital TV DVB-T protocol to make it viable for handheld devices. Mobile TV services on a smartphone would use a combination of DVB-H and the 2G/3G network. Although not common yet, some carriers have launched DVB-H mobile TV services (Vodafone in Italy is an example). The Nokia N92 and Nokia N77 are examples of smartphones that support DVB-H.

1.4 The Mobile OS

In the past, portable devices such as cell 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.

1.4.1 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.

Run on resource-limited hardware

Smartphones should be small, have long battery life, and cost as little as possible. To meet these requirements, smartphones, like other mobile devices, have limited memory and processing power compared with

desktop PCs and laptops. The operating system must be frugal in using hardware resources – especially memory. Not only must the OS itself not use much memory, but the architecture should be such that it provides limits and support to help OS applications also 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 like smartphones, which also allow third-party applications – that may be of questionable quality – to be downloaded.

Not only must the OS itself be designed to avoid crashing on its own, the OS 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 also be flexible, so that it can be customized for the varying form factors.

Library support for smartphone features

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 technology, cryptography, multimedia, UI, GSM/GPRS – the more smartphone feature support the better.

Support for application development

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.

1.5 Symbian OS – 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 of 1998, Symbian was formed as a joint venture owned by the major cell phone manufacturers of the day (Nokia, Ericsson, and Motorola) 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 Ericsson (15.6%), Nokia (47.9%), Panasonic (10.5%), Samsung (4.5%), Siemens (8.4%), and Sony Ericsson (13.1%), which, together, represent a major portion of the cell phone industry.

1.5.1 Symbian OS Overview

Symbian OS was designed from the ground up for mobile communications devices. While some competing operating systems (such as Microsoft's Windows Mobile for Smartphones OS) evolved from operating systems written for larger, more resource-laden systems, Symbian OS approached it from the other direction. Symbian's earlier versions (when 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. Furthermore, the hardware layers of the operating system are abstracted, so that phone manufacturers can port the OS to the specific requirements of their phone.

1.5.2 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 allows 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: (1) having too many different user interfaces inhibits code reuse among different devices and (2) too much work is required by the original equipment manufacturer (OEM) to create a GUI user interface 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 a 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 – S60, UIQ, and Series 80.

Symbian OS no longer supports the original user interface reference platforms and the smartphone programmer has no contact with these at all. Instead, the developer uses the software development kit (SDK) for the end platform supported by the phone. Also, there is no generic Symbian OS SDK for the developer – all core functionality is also included in the particular platform SDK.

Here are the major platforms for Symbian OS:

Nokia S60

The Nokia S60 (originally known as Series 60) user interface platform was originally designed for lower-end smartphones with small displays (176×208) and limited user input, such as a numeric-style keyboard used to enter text. This has changed in that S60 devices are getting more sophisticated, having larger displays, and even full keyboards. Nokia based S60 on the Symbian reference design known as Pearl, although Nokia did make significant modifications to it. S60 is a popular Symbian user interface. S60 is the most shipped platform for Nokia smartphones. At the time of writing, over 100 million S60 smartphones have been shipped.

The Nokia E61, Nokia E90 Communicator, Nokia N76, Nokia N93 and Nokia N95 are examples of phones that run S60 3rd Edition. Earlier S60 phones include the Nokia 6680, Nokia 7610 (which are S60 2nd Edition platforms) and Nokia 3650 (which is a S60 1st Edition platform). At the time of going to press, the latest S60 smartphone to be announced is the Nokia N81.

Nokia also licenses the S60 platform to other manufacturers such as Lenovo, LG, Panasonic, Samsung, and Siemens. At the time of writing,

the most recently announced S60 smartphone from a licensee is the Samsung SGH-i400.

Nokia Series 80

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

Nokia is discontinuing Series 80; it will not use it in any new phones going forward. Nokia now includes support for the communicator device form factor in the S60 3rd Edition platform. The Nokia E90 is the first communicator device to use this S60 platform.

UIQ

UIQ originated from a Symbian reference design known as Quartz. UIQ is owned, developed, maintained, and licensed by UIQ Technology AB. UIQ is designed for pen-based (i.e., touchscreen) smartphones with quarter VGA display and no keyboard. A virtual screen keyboard and handwriting recognition is provided for user input.

The Sony Ericsson P1i, W960i, W950i, P990i and M600i, and the Motorola MOTORIZR Z8, are examples of UIQ phones. These phones are based on UIQ version 3, the newest UIQ version at the time of this writing. Earlier UIQ version 2 phones include the Sony Ericsson P800/P900 and Motorola A920, A925, and A1000 smartphones.

Note that originally, S60 was designed for one-handed operation, and UIQ for two-handed operation. This distinction has been blurred, however, since higher-end S60 phones now have full keyboards, and some UIQ phones are now geared toward one-handed operation (e.g., the MOTORIZR Z8).

As mentioned, 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.

1.6 Symbian OS Smartphones

A large variety of Symbian smartphones are on the market today. These devices have various feature sets so that users have a choice of which smartphones to buy based on what features are important to them – and within the price range they are willing to pay.

For example, there are smartphones that target music lovers, emphasizing high-quality audio playback and large storage capability for songs in addition to other smartphone functionality. There are also smartphones that target video enthusiasts, which include high-resolution cameras and video capture. There are enterprise phones that contain large displays and keyboards that are well suited for the business person on the go. Another example is TV-oriented smartphones that support DVB-H capability for high-quality video service. And of course, there many other, more general phones that contain a combination of these and other features.

While Symbian OS powers many high- and medium-end smartphone models, another important – and big – market is the low-end, low-cost phone market. Symbian has been concentrating on getting more into this market in recent years. For example, in 2006 Symbian announced new scalable pricing options for phone manufacturers that lower the cost of licensing Symbian OS for high-volume smartphones.

This section introduces three Symbian OS-based smartphones: the Sony Ericsson P990i, Nokia N95, and Nokia 9300i Communicator. These phones are not necessarily representative of all Symbian OS smartphones, since they are more on the high-end, but they show a good sampling of some of the features we have discussed in this chapter, as well as representing the different UI platforms described in the last section. To see a complete list of Symbian OS smartphones, reference the Symbian phones section of Symbian's website (<http://www.symbian.com/phones>).

1.6.1 Sony Ericsson P990i

The Sony Ericsson P990i (shown in Figure 1.1) is a pen-based smartphone that uses the UIQ user interface. It has a 262K color, 240 × 320 pixel display with touchscreen, and a small keyboard that flips out. The phone has handwriting recognition, along with many pre-packaged organizer and game applications. The device plugs into a cradle that is connected to a PC via USB for downloading applications and syncing user data. IR and Bluetooth technology are also supported. The P990i has an integrated 2 megapixel camera, and will record video also. The phone contains a combination WAP/HTML browser, audio and video playback, email and SMS, as well as MMS. The device has 60 MB of internal memory for storage and supports an external memory card to expand this (a 64 MB memory card comes with the phone, but you can expand up to 4 GB).

The P990i supports UMTS, making it a 3G phone. But note that due to it using 2100 MHz for this, 3G for the P990i will not work in the USA since the UTM there is typically 850/1900 MHz. The P990i also supports Wi-Fi and GSM 900, 1800, 1900 MHz and GPRS.



Figure 1.1 The Sony Ericsson P990i

1.6.2 Nokia N95

The Nokia N95 (shown in Figure 1.2) is a S60-based phone with a 320×240 pixel, 16M color screen. Following the S60 model, this device has no touchscreen and two softkeys.

The Nokia N95 is part of Nokia's Nseries smartphone line, which are classified as multimedia devices. The phone has a high-resolution camera (5 megapixels), capable of taking both still pictures and video. The device has a high-quality audio player with stereo speakers capable of 3D sound effects.

The Nokia N95 also has GPS hardware built in, as well as maps. The device has maximum user storage of 160 MB and supports mini-SD memory cards. The built-in software includes a combination WAP/HTML browser and a media player, and it supports email, SMS and MMS, as well as voice-over-IP. Connectivity to other devices is supported via Wi-Fi, Bluetooth technology, and IR, as well as USB.

The Nokia N95 is a 3G phone that supports UMTS (2100 MHz) as well as GPRS and EDGE. It supports GSM 850, 900, 1800, and 1900 as well as Wi-Fi.



Figure 1.2 Nokia N95

An example of a lower-end Nokia Nseries phone is the Nokia N75, released in the United States for AT&T. The phone is a 3G flip-style phone with many of the same features as the Nokia N95, but with no GPS or Wi-Fi. Furthermore, its camera is a 2 megapixel camera instead of 5, and it has less memory. However, this phone is still powerful for the price.

1.6.3 Nokia 9300i Communicator

The Nokia 9300i (shown in Figure 1.3) is one of the latest in Nokia's high-end series of phones, known as communicators. Communicators look like traditional (although a bit heavier) cell phones, except that the case opens up into an easy-to-read landscape display and a QWERTY keyboard. This makes them good for enterprise use. Communicators use the S80 Symbian OS user interface. They have a 640×200 pixel screen with 4K colors (not a touchscreen). The devices include a WAP and HTML browser as well as email and SMS and MMS support. User input is via the hard keyboard and soft-labeled keys along the right side of the display.



Figure 1.3 Nokia 9300i

The Nokia 9300i Communicator supports Wi-Fi as well as GPRS, EDGE, and HSCSD. It supports GSM 900/1800/1900. The Nokia 9300 is a version of the 9300i that does not have Wi-Fi.

As mentioned earlier, Series 80 is discontinued, and the platform for communicators going forward is S60. The first communicator device to use S60 is the Nokia E90, which is shown in Figure 1.4.



Figure 1.4 Nokia E90

1.7 Other Smartphone Operating Systems

The smartphone market is competitive and so, not surprisingly, there are other choices of smartphone operating systems 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 phone 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 which make the best smartphones), but that's not the subject of this book.

This section gives a brief overview of four operating systems that compete with Symbian OS for the smartphone market: Windows Mobile for Smartphones, BlackBerry, Palm OS, and Linux.

1.7.1 Windows Mobile

There is little doubt that Microsoft 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 the desktop-based Windows operating system, and includes a subset of the windows user interface suitable for handheld devices. Microsoft released the Pocket PC as a PDA that ran the Windows CE-based OS called Pocket PC OS. As of 2003, the Windows CE and Pocket PC operating systems are merged into the Windows Mobile family.

Microsoft also aims to be a dominant player in the smartphone market, and has released another variant of Windows Mobile called Windows Mobile for Smartphones. An advantage of Windows Mobile is the availability of Pocket PC applications that can be run on Microsoft-based smartphones. In addition to this, Windows Mobile devices support miniature versions of many of its applications that are dominant in the desktop PC market – like 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 the mobile operating systems.

Windows Mobile 5.0 for Smartphones was released in 2005, and represents a significant improvement in their smartphone platform. Examples of Windows Mobile 5 smartphones are the Samsung Blackjack, Cingular 3125, and the Motorola Q.

At the time of this writing, Windows Mobile 6 is just being introduced, with its smartphone platform being released as the Windows Mobile 6 Standard Edition.

Microsoft has a relatively small portion of the smartphone market now (approximately 4–5% at the time of this writing), but its growth is very rapid, especially in the United States.

1.7.2 BlackBerry

Research In Motion (RIM) makes a line of devices known as BlackBerry devices. RIM uses a proprietary Java-based operating system for these devices. BlackBerry devices are extremely popular, especially in the United States. They are known for their great email service and functionality through a technology known as push email. The platform is open to third-party developers, but only for Java applications.

The BlackBerry Pearl is an example of a BlackBerry smartphone. The Pearl has a small form factor and includes a phone, browser, instant messaging, multimedia player, and a camera.

1.7.3 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's operating system, known as Palm OS, is like Symbian OS in that Palm designed it 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, which can also run on their smartphones. There is also a significant base of Palm OS application developers and documentation.

The Handspring Treo 650 is an example of a Palm OS-based smartphone. It supports both GSM and CDMA (via different models). The Treo 650 has all the standard smartphone features, like SMS, MMS, web browsing, and email, as well as the ability to connect to your PC via USB. It has a 320×320 pixel color display, a built-in thumb keyboard, and integrated digital camera.

Although Palm still plans to support Palm OS, they are now also shipping phones with the Microsoft Windows Mobile operating system. The Treo 700 w and 750 are examples of these.

1.7.4 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. Free cost and tapping 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. Motorola is a notable supporter of Linux and has released the A1200 smartphone based on this OS. Also, Motorola, NEC, Panasonic Mobile Communications, and Samsung Electronics, along with mobile operators NTT DoCoMo and Vodafone, have formed an alliance for creating a unified, Linux-based operating system for mobile devices.

Nokia has released a Linux-based tablet device called the Nokia N800. This device includes Wi-Fi and Bluetooth connectivity, a web browser, voice-over-IP support, a media player, email, and instant messaging. Since it is not based on Symbian OS, it does not use the S60 or Series 80 UI platform, but has a UI based on the open source platform called Maemo (<http://www.maemo.org>).

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 Vista (at least 1.5 GHz, 512MB RAM, and 2 GB free disk space is recommended).
- The C++ SDK for your smartphone model.
- A Windows development package (Win32 development tools with an Integrated Development Environment (IDE); Carbide.c++ is recommended).
- 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

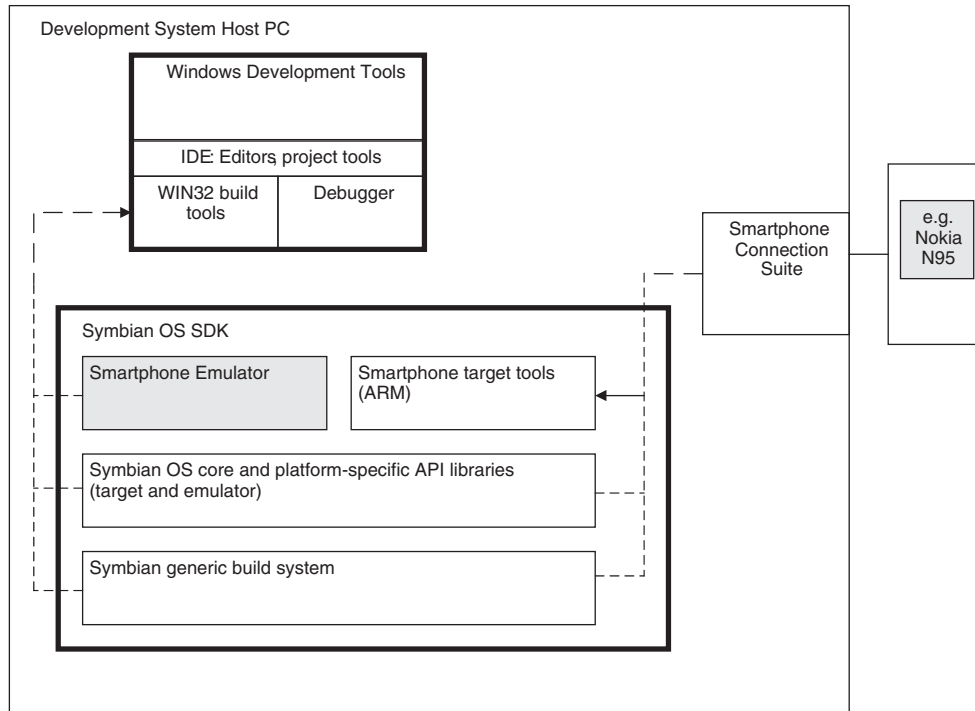


Figure 2.1 Development Tools

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 technology.

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 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 or Bluetooth message, or saving it to a removable memory card, using a PC with appropriate card writer, then putting the card into the phone and installing from there.

2.1.2 What is the Symbian OS Emulator?

The emulator is a Windows application that simulates a smartphone entirely on the PC – complete with functional buttons and small screen 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 upload your code to the smartphone for each code/compile/debug iteration.

- You can take advantage of the debugging support available from the Windows IDE, which includes single stepping and watch points.
- You can experiment with smartphone development on various platforms without having to buy or borrow the smartphones for those platforms.

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. You can even change the display resolution and orientation to match the various display modes of the target device so you can test your software in each of these modes.

Figure 2.2 shows a sample emulator screen for the S60 platform.



Figure 2.2 S60 Emulator

2.1.3 Getting the Symbian OS SDK

Your first priority should be locating the correct C++ 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 (<http://developer.symbian.com/main/tools/sdks/index.jsp>). 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 S60 3rd Edition SDK requires ActivePerl 518 and Java Runtime Environment 1.3.1 to be installed.

2.1.4 Some Example SDKs

This section describes some example Symbian OS SDKs. They represent the three main Symbian vendor software platforms mentioned in Chapter 1: S60, UIQ, and Series 80.

S60

- S60 Platform 1st Edition 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.
- S60 Platform 2nd Edition has a basic version that supports the Nokia 6600 smartphone, which is based on Symbian OS v7.0s.

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.
- The S60 3rd Edition SDK supports Symbian OS v9.1-based phones including the Nokia N77, E61i, and E65. A Feature Pack 1 also exists that supports Symbian OS v9.2 and phones including the Nokia N95.

All S60 SDKs are available for download from the Forum Nokia website (<http://www.forum.nokia.com>).

UIQ

- At the time of this writing, four UIQ SDKs exist:
 - UIQ 3.1

Supports Symbian OS v9.2 UIQ 3.1 devices including the Motorola MOTORIZR Z8 smartphone.

- UIQ 3.0
Supports Symbian OS v9.1 UIQ 3.0 devices including the Sony Ericsson W950, P990, P1, and M600 smartphones.
- UIQ 2.1
Supports Symbian OS v7.0 UIQ 2.1-based devices including the P900, P910, BenQ P30, and Arima U308 smartphones.
- UIQ 2.0
Supports Symbian OS v7.0 UIQ 2.0-based Sony Ericsson P800 smartphone.

The UIQ 3.0 and 3.1 SDKs are available for download from the UIQ developer website (<http://developer.uiq.com>). Earlier SDKs can be downloaded from Symbian or the handset manufacturer in question. See <http://developer.uiq.com/forum/entry.jspa?externalID=31> for links to the appropriate download locations.

Series 80

- 9200 Communicator SDK (Series 80) supports Nokia 9200 Communicator series smartphones, which are based on Symbian OS v6.0.
- Series 80 Platform 2.0 supports the Symbian OS v7.0s-based Nokia 9500 Communicator.

2.1.5 Getting a Windows Development Package

The Symbian OS SDK contains all that's needed for building application 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 package containing the tools needed to produce emulator executables. The recommended Windows development package at this time is Carbide.c++. At the time of writing, you can download the free Express version of Carbide.c++ version 1.2 from Forum Nokia (<http://www.forum.nokia.com/carbide>) and begin using it right away.

A Windows development package typically provides a source code editor, GUI-based building and debugging tools, and sometimes project management features or additional software tools. With an SDK alone, you will only be able to build for and load straight to your smartphone, and will have very limited debug support (normally via log files or printing messages to the screen), command line only tools, and no source code editor.

2.1.6 More on Carbide.c++

Nokia introduced Carbide.c++ as an IDE for Symbian OS development in 2005. Before that, the Symbian OS development package choices were Microsoft Visual Studio 6 and Microsoft Visual Studio 2003 .NET, Metrowerks CodeWarrior 3.1, and Borland C++ Builder X – and support for these varied based on the particular Symbian OS SDK you used. You can still use these development systems (again, with support depending on the SDK); however, Carbide.c++ provides the best support across the widest range of SDKs, as well as all the newer ones. Carbide.c++ contains an advanced set of features for Symbian OS development and is the one that will be supported and enhanced going forward.

Carbide.c++ is based on Eclipse, which is an open source project originated by IBM in 2001. The purpose of the Eclipse project is to provide a full-featured integrated development environment along with a framework for extending this environment. Eclipse provides extensibility via a plug-in framework so that the IDE can be customized to work for different development environments, or simply enhanced with features provided by third-party developers. This ability provides the end user with common Eclipse IDE functionality (editors, debuggers, build error handling, etc.), yet also with the various configurations and features specifically needed for a particular target environment.

Eclipse is written in Java; however, the development platforms created around Eclipse can and do support languages besides Java. Eclipse is becoming a *de facto* standard for integrated development environments.

Nokia selected Eclipse for its development platform due to its open and extensible nature. In this way, Carbide.c++ can leverage the open source community for the best in the common IDE features, yet still be able to create and maintain Symbian OS specific development functionality.

At the time of writing, Carbide.c++ only supports SDKs that have the WINSW Nokia x86 build target. If you need to use an SDK that does not have WINSW, then you will need to use one of the IDEs mentioned as supported in the SDK information (e.g., Microsoft Visual Studio 6 or Borland C++ Builder X).

At the time of writing, Carbide.c++ comes in the following different editions for Symbian OS C++ development. For more information on Carbide.c++, and the complete feature set of each version, reference the <http://www.forum.nokia.com/carbide> website:

Carbide.c++ Express

This is a free download for non-commercial developers (i.e., you cannot sell applications you develop using it). It is ideal for beginner Symbian programmers who later decide to upgrade to one of the commercial editions if needed. Carbide.c++ Express contains everything you need for building and debugging (on emulator) applications.

Carbide.c++ Developer

Carbide.c++ Developer has some significant additions from the Express version, including a graphical UI designer and on-device debugging. Furthermore, the license for this and the Professional edition allows you to develop commercial software with it. Unlike the Express version though, the Developer and Professional versions are not free.

Carbide.c++ Professional

Carbide.c++ Professional is for advanced developers at the device and system level. This edition contains added features such as advanced system-level performance profiling as well as system-level on-device debugging in addition to the Developer edition features.

Carbide.c++ OEM

This edition contains additional support for developers working within phone manufacturer teams, such as tools to create ROMs and support for JTAG debugging.

2.1.7 Carbide.vs

Carbide.vs is a plug-in to Visual Studio 2003 .NET, which provides that IDE with Symbian OS development capability including wizards for building Symbian OS applications. This could be a good option for you if you have Visual Studio 2003 .NET already and feel comfortable with this IDE. The Carbide.vs product is a free download, and unlike the Express edition, you can develop applications on it for commercial use. Note, however, that support going forward on Symbian OS development will likely be better with Carbide.c++.

Suggestions While Installing

Want something productive to do while waiting for your development tools to download and install? Here are a few suggestions.

- Bookmark the following websites for fast reference:
 - Symbian Developer Network (<http://developer.symbian.com/>). This site contains a wealth of information related to Symbian OS development. It includes articles and white papers, as well as forums and information on training classes, Symbian Academy and Symbian Press.
 - Forum Nokia (<http://forum.nokia.com/>) if you are doing S60 development. This site contains a lot of information on Symbian OS and Nokia S60, including articles and white papers, e-learning, example code, and developer tools.

- Sony Ericsson Developer World (http://developer.sonyericsson.com/site/global/home/p_home.jsp) if you are doing UIQ development. This site has similar information to the Forum Nokia site, except with emphasis on Sony Ericsson devices built on UIQ.
 - MotoDev (<http://developer.motorola.com/>) for additional information if you are creating applications for Motorola phones, such as the MOTORIZR Z8, based on UIQ 3.
 - UIQ Developer Community (<http://developer.uiq.com>) for more information about UIQ 3 development.
 - NewLC (<http://www.newlc.com>) – this is a good independent site that contains lots of information related to Symbian OS development.
- Visit the Symbian Signed website.

If you are developing Symbian OS software for distribution, then there is a good chance you will want (or even need) access to the Symbian Signed website to submit your software to be Symbian Signed. The Symbian Signed process is covered in Chapter 7. For now, while you are waiting for your installation to complete, go to <http://www.symbiansigned.com> for more information.

- Read the Symbian OS Essentials Booklets.

These booklets are found on the Symbian Developer Network at <http://developer.symbian.com/main/learning/press/essential/>. They are an excellent source for beginner Symbian OS programmers on some selected key subjects, and a good supplement to this book. New booklets are printed regularly, and if you have a topic you'd like to see covered by the Essentials Series, please send your suggestion to editor@symbian.com.

2.1.8 Is Windows the Only Development Operating System Supported?

The rest of this book will assume that you are using a Windows development tool since, at the time of writing, the only official support for Symbian OS development is for a PC running Microsoft Windows. However, there are efforts to change this, and GNUPoc is a good example.

The site <http://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.

If you're a Mac OS X user and don't have a PC, you may not need to buy one to develop software on Symbian OS. Take a look at the various options to run Windows on your Mac, depending on your hardware, such as Apple's Boot Camp or Parallels Desktop for Mac.

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.

Figure 2.3 shows an example of switching from the Series 60 3rd Edition FP1 SDK to the UIQ 3 SDK.

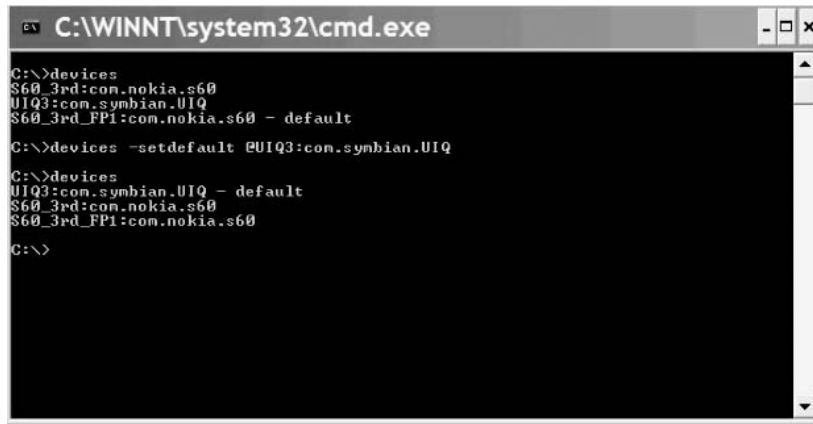
Switching to the correct SDK via the `devices` command is crucial for command line builds, although if you are using an IDE, this switching is automatic.

2.2.1 Quick Test of the Emulator

Type `epoc` from a command prompt. This should bring up the Symbian OS emulator for the supported smartphone type. The emulator displays a smartphone desktop where you can select and run various built-in

programs and setup utilities. If the emulator 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 S60 SDKs, which had a problem when you installed the SDK in a location other than the default. The default installation path was hardcoded 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`, located in the installation directory of the SDK, in the `\epoc32\data` subdirectory.



```

C:\>devices
$60_3rd:com.nokia.s60
UIQ3:com.symbian.UIQ
$60_3rd_FP1:com.nokia.s60 - default

C:\>devices -setdefault @UIQ3:com.symbian.UIQ

C:\>devices
UIQ3:com.symbian.UIQ - default
$60_3rd:com.nokia.s60
$60_3rd_FP1:com.nokia.s60

C:\>

```

Figure 2.3 Using the `devices-setdefault` Command

2.2.2 Quick Test Drive of Carbide.c++

The simplest way of testing out Carbide.c++, and to get a feel for its operation, is to generate a simple hello world application using its project wizard, and then build it and see the application run on the emulator. The steps to do this are as follows:

1. Once Carbide.c++ is running, choose **File->New->Project**.
2. You will be prompted for a wizard. Choose either S60 or UIQ under Symbian OS depending on which SDK you have loaded. Select Next.
3. Enter a project name and select Next, as shown in Figure 2.4.
4. Select a template also depending on which SDK you have, as shown in Figure 2.5.
For S60, you can select S60 3.x Application which is a hello world template for S60 3rd Edition.

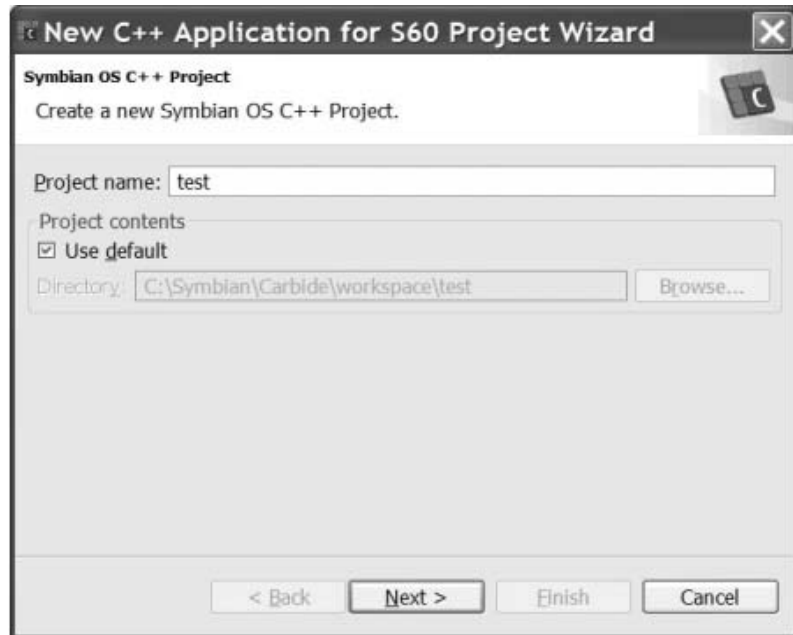


Figure 2.4 New Project Creation

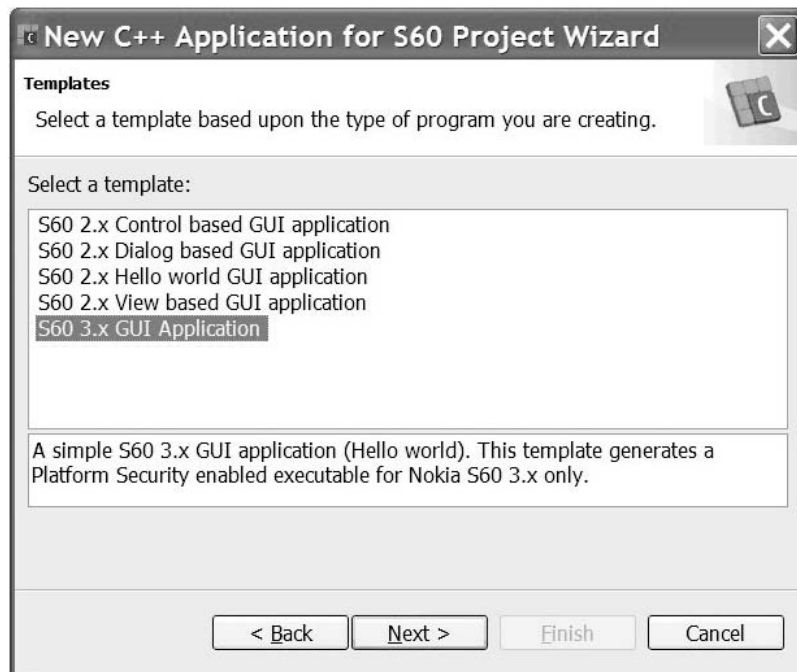


Figure 2.5 Application Template Selection

5. Select Next. The Symbian OS SDK applicable to your previous choice is displayed, as shown in Figure 2.6.
If you do not have the proper SDK, you will be notified here. Select Finish to take you to defaults for the rest. This will generate the project.
6. Select **Run->Run** to build the project and start the emulator so you can run it.
7. When the emulator comes up, go to the main screen or a subfolder such as in the S60 3rd Edition, where the installation folder the application is put into is 'Installation'. Click on this icon to run the hello world program within the emulated environment.

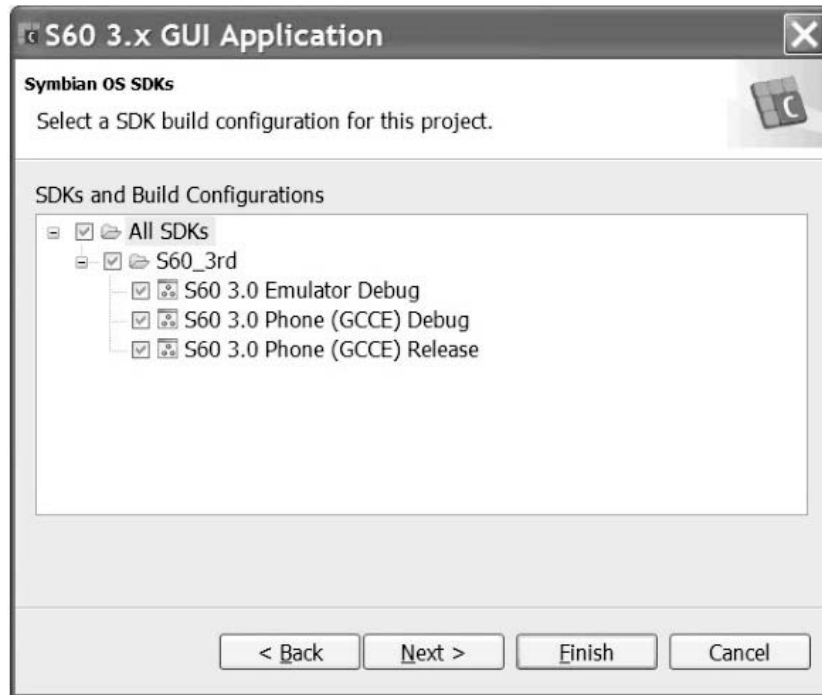


Figure 2.6 SDK Selection

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 for the emulator, platform by platform from the command line.

Although you likely will be using Carbide.c++ to build Symbian OS code, it's also a good idea to get familiar with building on the command line. I'll then provide some steps for building the examples within Carbide.c++.

Building a S60 example

Go to the S60 example directory from a command prompt. For S60 3rd Edition SDKs, it's located at `Symbian_Base\S60Ex`, where *Symbian_Base* is your SDK installation directory for S60. It's at `Symbian_Base\Series60Ex` for S60 versions before 3rd Edition. I'll assume you have installed the SDK in the default location. Table 2.1 shows the default directories for S60 SDKs.

Thus, for S60 3rd Edition FP1, type:

```
C:>cd \Symbian\9.2\S60_3rd\S60Ex
```

or for 2nd Edition:

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

Type `dir` and you will see a list of folders containing examples. Change directory to `HelloWorldBasic\group` to build the hello world program (it's called `HelloWorld` in S60 v1.0):

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

Type the following at the command prompt:

```
C:\...\HelloWorld\group>bldmake bldfiles
C:\...\HelloWorld\group>abld build winscw udeb
```

Table 2.1 Default SDK Directories for S60

SDK	Default Installation Directory
S60 3rd Edition FP1	c:\Symbian\9.2\S60_3rd_FP1
S60 3rd Edition	c:\Symbian\9.2\S60_3rd
S60 2nd Edition	c:\Symbian\7.0s\Series60_v2.0
S60 1st Edition	c:\Symbian\6.1\Series60

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 Microsoft tools then you need to specify a target of `winsb` or `wins` instead of `winscw` when running the `abl` command from the command line. `wins`, `winsb`, and `winscw` indicate emulator builds for Microsoft, Borland, and Carbide.c++ development tools, respectively. `winscw` is also used for CodeWarrior. `udeb` indicates that it is a debug as opposed to a release build (`urel`), which means you will have symbols in the code so you can use the IDE debugger. With newer SDKs, the release builds are no longer supported on the emulator.

It's worth pointing out that Borland C++ Builder X can use `wins`, if configured in Microsoft binary mode. The Borland tool 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 S60 emulator as described in section 2.2.1. Find your HelloWorld icon (in the folder labeled Installed or Other, or the main desktop depending on the SDK) and select it.

Building a UIQ example

Go to the UIQ example directory from a command prompt. This directory is located at `Symbian_Base\Examples\UIQ` for UIQ version 3. For UIQ2, it is at `Symbian_Base\UIQExamples`. `Symbian_Base` is your SDK installation directory (e.g., `c:\Symbian\UIQ3SDK` for UIQ 3, and `c:\Symbian\UIQ_21` for UIQ 2.1).

To build the hello world example in UIQ 3:

```
C:>cd \Symbian\UIQ3SDK\Examples\UIQ
```

Type `dir` and you will see a list of folders containing examples. Change directory to `QHelloWorld` to build the hello world program:

```
C:\Symbian\UIQ3SDK\Examples\UIQ>cd QHelloWorld
C:\Symbian\UIQ3SDK\Examples\QHelloWorld>
```

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

```
C:\Symbian\UIQ3SDK\Examples\QHelloWorld> bldmake bldfiles
C:\Symbian\UIQ3SDK\Examples\QHelloWorld> abl build winscw udeb
```

After the compilation completes, type `epoc` to start the emulator and run the application by selecting the HelloWorld icon from the desktop.

In the preceding sections, I mentioned 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 examples using Carbide.c++

The previous sections described how to build the examples from the command line. You can also build and run the examples using Carbide.c++ by importing them into the IDE. Here is how:

1. Start Carbide.c++.
2. Select **File->Import**. Select Symbian `bld.inf` as the import source and then click on Next, as shown in Figure 2.7.
3. The next screen will prompt for the location of the `bld.inf` file. Browse to the project directory where the `bld.inf` file is located and select it (e.g., `c:\Symbian\UIQ3SDK\Examples\UIQ\QHelloWorld\Group\bld.inf`), and then click on Next.
4. The `bld.inf` will then be parsed and the status should be 'Parsed successfully'. Click on Next.
5. Select the SDK and build configurations and check the 'All SDKs' box so you can build for emulator and devices. Click on Finish when done.
6. You should now have the example project in Carbide.c++. Select **Run->Run** to compile and run in the emulator.

Library freezing

Sometimes, you may get errors when 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. For now, if you get that error, type `abld freeze winscw` (or substitute `winscw` with the build target for whatever platform you are using), then reissue the `abld`

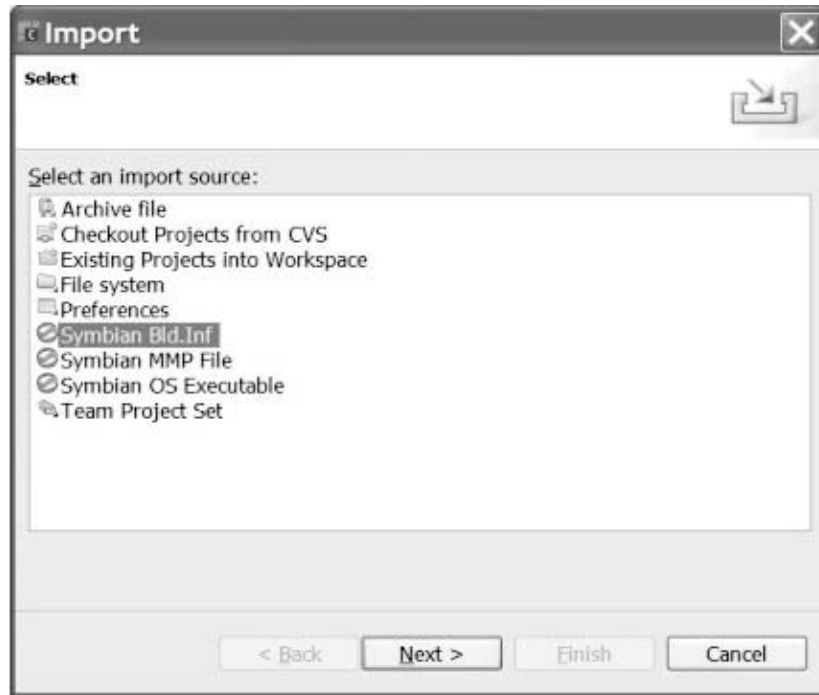


Figure 2.7 Import from Symbian bld.inf File

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.

2.3 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 both S60 3rd Edition and UIQ 3 smartphones.

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, and building the example for both would be ideal. Furthermore, note that it would be simpler to generate this application for both platforms using the Carbide.c++ project wizards, and again I think it is instructive to enter the code manually, as discussed, for a better learning experience.

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

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 example for both platforms contains the following files organized in the directories as shown:

```
\include
    SimpleEx.h
    SimpleEx.hrh
    SimpleExUid.h
\src
    SimpleEx.cpp
    SimpleEx_app.cpp
    SimpleEx_doc.cpp
    SimpleEx_ui.cpp
    SimpleEx_view.cpp
\group
    bld.inf
    SimpleEx.rss
    SimpleEx_reg.rss
    SimpleEx.mmp
    SimpleEx.pkg
```


To create the example you will perform the following steps:

1. Create the application header files.
2. Create the resource (RSS) and resource command header (HRH) files.
3. Create the registration resource file.
4. Create the source code for the application classes.
5. Create the project definition file (SimpleEx.mmp) and build file (bld.inf).
6. Build and run the example on the PC-based emulator.
7. Create the target package definition (PKG) file.
8. Generate the installation package (SIS file) and install it on a smart-phone.
9. Execute the SimpleEx application on the smartphone.

2.3.3 Header Files

This section describes the main header file for our example (SimpleEx.h), defined for S60 and UIQ platforms and the header file it includes (SimpleExUId.h). Put the header file for your particular platform in a file named SimpleEx.h and place it in the include directory (e.g., \SimpleEx\S60\include for the S60 headers).

The following is this header for S60 3rd Edition:

```
#ifndef __SIMP_H
#define __SIMP_H
/*=====
   S60 SimpleEx Header File
   =====*/
#include <eikenv.h>
#include <eikon.hrh>
#include "SimpleEx.hrh"

#include <akndoc.h>
#include <aknapp.h>
#include <aknappui.h>
#include "SimpleExUId.h"
const TUid KUidSimpleExApp = {SIMPLEEXUID};

// 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:
    CEikAppUi* CreateAppUiL();
    };
#endif

```

Here is the same version of the code for SimpleEx.h for UIQ 3:

```

/*=====
    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>
#include <QikViewBase.h>
#include "SimpleExUid.h"
const TUid KUidSimpleExApp = {SIMPLEEXUID};
/**
Identifies the hello world application's view.
Each view need an unique UID in the application.
*/
const TUid KUidSimpleExView = {0x00000001};
/*-----
    The Application Class
-----*/
class CSimpleExApplication : public CQikApplication
{
private:

```

```

    CApaDocument* CreateDocumentL();
    TUid AppDllUid() const;
};

/*-----
The Application View Class
-----*/
class CSimpleExAppView : public CQikViewBase
{
public:
    static CSimpleExAppView* NewL(CQikAppUi& aAppUi);
    static CSimpleExAppView* CSimpleExAppView::NewLC(CQikAppUi& aAppUi);
    void ConstructL();
    // from CQikViewBase
    TVwsViewId ViewId() const;
    void ViewConstructL();
    void HandleCommandL(CQikCommand& aCommand);
private:
    void Draw(const TRect&) const;
    CSimpleExAppView(CQikAppUi& aAppUi);
};

/*-----
The UI Class
-----*/
class CSimpleExAppUi : public CQikAppUi
{
public:
    void ConstructL();
private:
    CSimpleExAppView* iAppView;
};

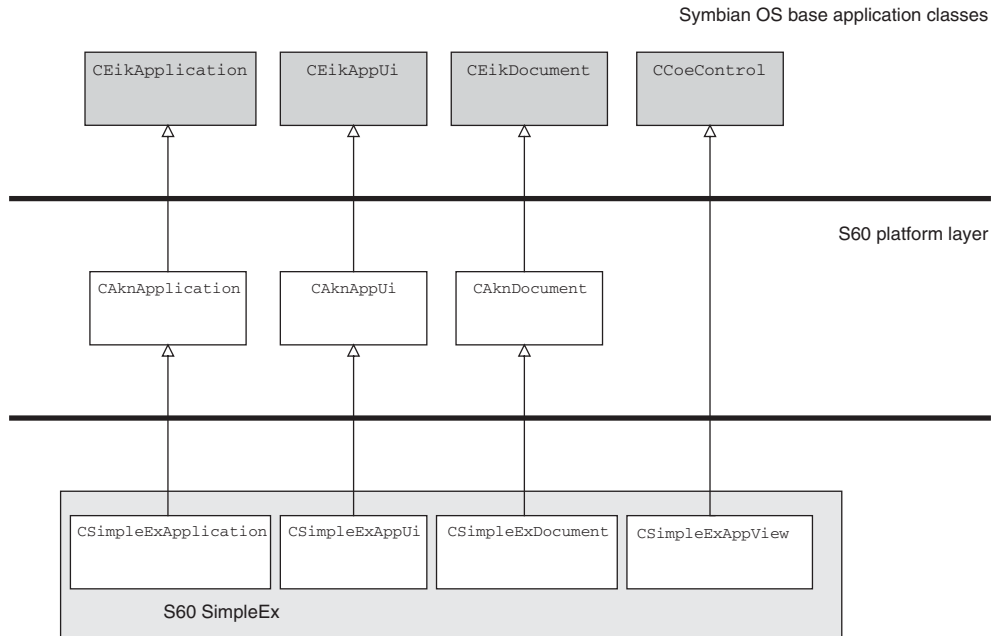
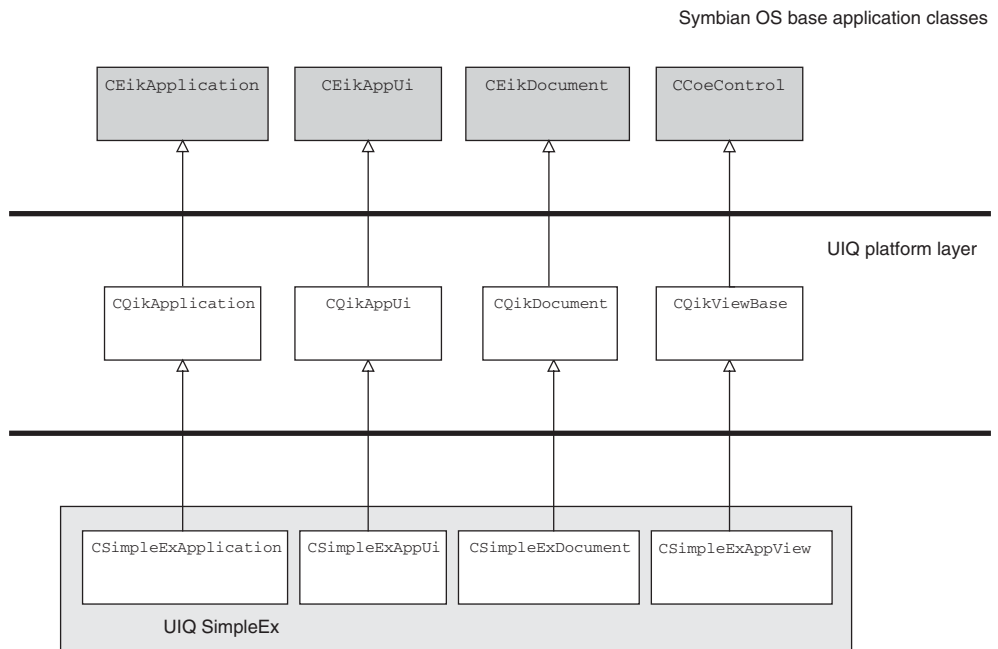
/*-----
The Application Document Class
-----*/
class CSimpleExDocument : public CQikDocument
{
public:
    CSimpleExDocument(CEikApplication& aApp) : CQikDocument(aApp) { };
private:
    CEikAppUi* CreateAppUiL();
};
#endif

```

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` for S60, and `CQikAppUi` for UIQ.

Figures 2.8 and 2.9 show the class hierarchy for SimpleEx, for S60 and UIQ, respectively.

Notice that the platform classes for S60 and UIQ derive from the same core Symbian OS base application classes. Fortunately, the application platform classes for the different platforms are similar from a development perspective due to their abstracted interface. Note also that while in UIQ I inherit the view class from the UIQ view class `CQikViewBase` since all applications in UIQ should do that, I inherit the S60 application class

**Figure 2.8** SimpleEx Class Hierarchy for S60**Figure 2.9** SimpleEx Class Hierarchy for UIQ

directly from `CCoeControl`, which is fine for applications with a single view such as this one.

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.

Note also that `SimpleEx.h` includes `SimpleExUid.h`. This file is the same for both S60 and UIQ platforms, and contains just one line:

```
#define SIMPLEEXUID 0xE000027F
```

This include simply assigns a macro variable to the software's unique User Identifier (UID) to be used in the C++ code as well as the registration resource file. UIDs will be discussed in more detail in Chapter 5. Suffice to say now that each application needs a unique UID. Normally UIDs are reserved and obtained through Symbian, but the range `0xE0000000–0xFFFFFFFF` can be randomly selected by the programmer for development-level code.

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

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.

The following shows the resource files for S60 and UIQ – enter the one corresponding to your platform into a file called `SimpleEx.rss`, and place it in the `group` directory.

The resource file for S60 3rd Edition is as follows:

```
/*=====
   S60 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";
        }
    };
}

```

Below is the resource file for UIQ 3:

```

/*=====
   UIQ SimpleEx Resource File
=====*/
NAME SIMP
#include <eikon.rh>
#include <qikon.rh>
#include <QikCommand.rh>
#include "SimpleEx.hrh"
RESOURCE RSS_SIGNATURE
{
}
RESOURCE TBUF r_default_document_name
{
    buf=" ";
}
RESOURCE EIK_APP_INFO
{
}
RESOURCE QIK_VIEW_CONFIGURATIONS r_simpleex_configurations
{
    configurations =
    {
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            command_list = r_simpleex_commands;
            view = r_simpleex_layout;
        },
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikSoftkeyStyleSmallPortrait;
            command_list = r_simpleex_commands;
            view = r_simpleex_layout;
        }
    };
}

```

```

    }
RESOURCE QIK_COMMAND_LIST r_simpleex_commands
{
    items =
    {
        QIK_COMMAND
        {
            id = ESimpleExCommand;
            type = EQikCommandTypeScreen;
            text = "Start";
        },
// This command is only visible in debug mode for finding memory leaks.
// Exit is not in production UIQ code.
        QIK_COMMAND
        {
            id = EEikCmdExit;
            type = ESimpleExCommand;
            // Indicate that this command will only be visible in debug
            stateFlags = EQikCmdFlagDebugOnly;
            text = "Exit";
        }
    };
}
RESOURCE QIK_VIEW r_simpleex_layout
{
    pages = r_simpleex_layout_pages;
}

// Defines the pages of a view.
// Only one page for this example.

RESOURCE QIK_VIEW_PAGES r_simpleex_layout_pages
{
    pages =
    {
        QIK_VIEW_PAGE
        {
            page_id = ESimpleExViewPage;
        }
    };
}

```

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. Historically in Symbian OS, the resource file had to be created using a text editor; however, there is a Carbide.c++ tool (UI designer) which can be used as a graphical resource editor. The UI designer is available in the Developer and Professional editions of Carbide.c++. Unfortunately, the free Express edition of Carbide.c++ does not support UI designer. Furthermore, UIQ 3 is not supported by the UI designer. There are other tools appearing that have Symbian UI visual design capabilities. One example is Wirelaxsoft's visual IDE tools that include graphical resource

editing support for UIQ 3. Let's skim through the highlights of the resource file to help understand the SimpleEx example. You will see that there are some significant differences between the S60 and UIQ resources. I will point these out as I go.

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. For S60, I define two things in the EIK_APP_INFO resource for SimpleEx: the default menu and the default softkey definitions. For UIQ, however, the EIK_APP_INFO is blank.

For S60, 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 UIQ, the user-level commands are more abstracted. In other words, the device supplies more of the intelligence of how the user options appear based on the different display modes of the phone. The developer defines a list of display modes the application will support in a QIK_VIEW_CONFIGURATIONS resource, and associates each of these modes with a list of user commands defined in a QIK_COMMAND_LIST resource. The QIK_VIEW_CONFIGURATION attribute ui_config_mode specifies the display mode – I specify KQikPenStyleTouchPortrait and KQikSoftkeyStyleTouchPortrait, which are two different portrait-style display modes. The attribute command_list for each of these points to our single QIK_COMMAND_LIST resource. Chapter 12 will discuss UIQ display modes in more detail.

The QIK_COMMAND_LIST specifies selections the user can make in the application, along with the ID of the commands sent to your application when the user selects them. The Start command is there, which will send the ESimpleExCommand command to the command handler when the user selects it. I also put an Exit command, which will only appear on the screen on debug builds of the code (stateFlags = EQikCmdFlagDebugOnly specifies this). Normally, UIQ programs do not have exit options for the user to select and simply go to the background when the user wants to run something else. Having an exit is useful when debugging, though, since on exit the software will generate an

exception if you have not cleaned up and freed your resources at the time of exit.

In S60, the `cba` attribute in the `EIK_APP_INFO` resource defines the application's softkeys (as default, they can be changed dynamically). Symbian refers to a set of softkeys as a command button array (CBA). In S60, there are two softkeys at the bottom of the screen. In the S60 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.

In UIQ, you do not specify CBAs specifically, however, in some display modes, the softkeys are automatically used for presenting the commands in the `QIK_COMMAND_LIST` resource.

SimpleEx.hrh

The file `SimpleEx.hrh` contains the command values that the controls send (specified in the resource file) for the application code to handle. In the S60 case we have only one, used when Start is selected:

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

For UIQ, we define an additional value to be used in the resource file's `QIK_VIEW` resource since UIQ requires this. Don't worry if you do not understand this at this point. Here is the UIQ version of `SimpleEx.hrh`:

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

The `.hrh` file is an include file that is used by both the resource file and the C++ code that handles the events.

SimpleEx_reg.rss

In Symbian OS v9, all applications are required to have a registration resource containing a `APP_REGISTRATION_INFO` resource. Here is the one I use for `SimpleEx` (it is the same for S60 3rd Edition and UIQ 3):

```
// SimpleEx_reg.rss
#include <AppInfo.rh>
```

```
#include "SimpleExUid.h"
UID2 KUidAppRegistrationResourceFile
UID3 SIMPLEEXUID// application UID
RESOURCE APP_REGISTRATION_INFO
{
    app_file = "SimpleEx";
}
```

All registration resources must include AppInfo.rh. UID2 should always be KUidAppRegistrationResourceFile and UID3 should be the unique UID3 assigned to the application. A resource APP_REGISTRATION_INFO must also be defined that specifies, at a minimum, the name of the application executable (without the extension), using the app_file attribute (SimpleEx in our example).

SimpleEx_reg.rss should be put in the group directory in our example.

2.3.5 Source Files

This section shows the application source files for the example. Three of the source files are the same for both S60 and UIQ, the other two vary. Type the code as shown into their respective files and place them in the src directory.

SimpleEx.cpp

This file is the same for both S60 and UIQ. It contains the executable entry point of the application (E32Main()). 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. SimpleEx.cpp is as follows:

```
/*=====
File: simpleEx.cpp
=====*/
#include <eikstart.h>
#include "SimpleEx.h"
// Create application object, return a pointer to it
EXPORT_C CApaApplication* NewApplication()
{
    return (static_cast<CApaApplication*>(new CSimpleExApplication));
}
GLDEF_C TInt E32Main()
{
    return EikStart::RunApplication( NewApplication );
}
```

SimpleEx_App.cpp

This file is the same for S60 and UIQ. It 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. Following is `SimpleEx_App.cpp`:

```
/*=====
File: simpleEx_app.cpp
This file contains the application class for SimpleEx.
=====*/
#include "SimpleEx.h"
CApaDocument* CSimpleExApplication::CreateDocumentL()
{
    return new(ELeave) CSimpleExDocument(*this);
}
TUid CSimpleExApplication::AppDllUid() const
{
    return KUidSimpleExApp;
}
```

SimpleEx_Doc.cpp

This file is also common between S60 and UIQ. The file contains the application document object. Although this example has no actual persistent data, the example still overrides the `CreateAppUiL()` method of the document class since the framework calls this method to create and pass a pointer to the application UI class. Following is `SimpleEx_Doc.cpp`:

```
#include "SimpleEx.h"
CEikAppUi* CSimpleExDocument::CreateAppUiL()
{
    return new (ELeave) CSimpleExAppUi;
}
```

SimpleEx_UI.cpp

This file contains the example application's UI class. In the S60 example, this class contains the `HandleCommandL()` method where the main action of the application occurs. All user events (except alphanumeric keyboard input) 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 GUI method available to all platforms. This looks quite good on UIQ, although fairly plain on the other platforms. In practice, you should follow the UI guidelines for the software platform you are using, which are available for download from their developer websites.

This may involve using platform-specific classes for controls or message displays instead of core Uikon controls.

As an example, in S60 you can use the S60-specific UI control class `CAknInformationNote` to pop up a message. This would look better for the S60 UI than the generic Symbian OS application framework 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.

Note further that, in the `HandleCommandL()` method, command `EAKnSoftkeyExit` handles the S60 Exit softkey that is put up at application start, as specified by the `R_AVKON_SOFTKEYS_OPTIONS_EXIT` option in the S60 resource file. Following is `SimpleEx_UI.cpp` file for S60:

```
/*=====
File: simpleEx_ui.cpp
This file contains the application UI class for S60 SimpleEx.
=====*/
#include "SimpleEx.h"
void CSimpleExAppUi::ConstructL()
{
    BaseConstructL(CAknEnableSkin);
    iAppView = CSimpleExAppView::NewL(ClientRect());
}
CSimpleExAppUi::~CSimpleExAppUi()
{
    delete iAppView;
}
void CSimpleExAppUi::HandleCommandL(TInt aCommand)
{
    switch(aCommand)
    {
        case EEikCmdExit:
        case EAKnSoftkeyExit:
            Exit();
            break;
        case ESimpleExCommand:
        {
            _LIT(KMessage, "Start Selected!");
            iEikonEnv->AlertWin(KMessage);
            break;
        }
    }
}
```

The UIQ 3 version of `SimpleEx_appui.cpp` is:

```
/*=====
File: simpleEx_ui.cpp
This file contains the application UI class for UIQ SimpleEx.
```

```
#include "SimpleEx.h"
void CSimpleExAppUi::ConstructL()
{
    BaseConstructL();
    iAppView = CSimpleExAppView::NewLC(*this);
    AddViewL(*iAppView);
    CleanupStack::Pop(iAppView);
}
```

In the UIQ version of the UI class, I implemented only one method: the secondary constructor, `ConstructL()`. This function simply creates the application view and registers it via `AddViewL()`. As we will see shortly, in UIQ the `HandleCommandL()` function is handled in the view class instead of the UI class like in the S60 version. The view mechanism shows its full benefits when using multiple views, but we use it here only for a single view.

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.

Following is the view class for S60:

```
/*=====
    File: simpleEx_view.cpp
    This file contains the application view class for SimpleEx.
=====*/
#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();

    TRect drawRect = Rect();
    gc.Clear();
    const CFont* 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 following shows the view class for UIQ:

```

/*=====
    File: simpleEx_view.cpp
    This file contains the application view class for UIQ SimpleEx.
=====*/
#include "eikenv.h"
#include <coemain.h>
#include "SimpleEx.h"
#include "SimpleEx.rsg"
#include <QikCommand.h>
CSimpleExAppView* CSimpleExAppView::NewL(CQikAppUi& aAppUi)
{
    CSimpleExAppView* self = CSimpleExAppView::NewLC(aAppUi);
    CleanupStack::Pop(self);
    return self;
}
CSimpleExAppView* CSimpleExAppView::NewLC(CQikAppUi& aAppUi)
{
    CSimpleExAppView* self = new (ELeave) CSimpleExAppView(aAppUi);
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}
CSimpleExAppView::CSimpleExAppView(CQikAppUi& aAppUi)
    : CQikViewBase(aAppUi, KNullViewId)
{
}
void CSimpleExAppView::ConstructL()
{
    BaseConstructL();
}
void CSimpleExAppView::ViewConstructL()
{
    // Loads information about the UI configurations this view supports

```

```

        // together with definition of each view.
        ViewConstructFromResourceL(R_SIMPLEEX_CONFIGURATIONS);
    }
TVwsViewId CSimpleExAppView::ViewId()const
{
    return TVwsViewId(KUidSimpleExApp, KUidSimpleExView);
}
void CSimpleExAppView::Draw(const TRect& ) const
{
    CWindowGc& gc = SystemGc();
    TRect drawRect = Rect();
    gc.Clear();
    const CFont* 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();
}
void CSimpleExAppView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        case ESimpleExCommand:
        {
            _LIT(KMessage,"Start Selected!");
            iEikonEnv->AlertWin(KMessage);
            break;
        }
        default:
            CQikViewBase::HandleCommandL(aCommand);
    }
}
}

```

As you can see, the UIQ view class is more complex than the S60 one. The UIQ version of the view class has the method `ViewConstructL()`, which in turns calls the function `ViewConstructFromResourceL(R_SIMPLEEX_CONFIGURATIONS)`. Remember that `R_SIMPLEEX_CONFIGURATIONS` is the name of the `QIK_VIEW_CONFIGURATIONS` resource defined in the UIQ resource file, which points to the commands associated with the different display modes of the phone (in our case, the same command set). The UIQ view class has the `HandleCommandL()` function in it that handles these commands. `ViewId()` returns the UID of the application along with a view ID that uniquely identifies this view (via the line: `TVwsViewId(KUidSimpleExApp, KUidSimpleExView)`). In `SimpleEx.h`, this unique view ID (as specified in `KUidSimpleExView`) is 1.

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 S60 and UIQ (they are very similar except for the platform-specific library shown in bold in the listings). Use the file corresponding to your platform. Name the file *SimpleEx.mmp* and place it in the group directory.

The S60 project file is as follows:

```
TARGET             SimpleEx.exe
TARGETTYPE         exe
UID                0x100039CE 0xE000027F
SOURCEPATH         ..\src
SOURCE             SimpleEx.cpp
SOURCE             SimpleEx_app.cpp
SOURCE             SimpleEx_view.cpp
SOURCE             SimpleEx_ui.cpp
SOURCE             SimpleEx_doc.cpp
SOURCEPATH         ..\group
START_RESOURCE     SimpleEx_reg.rss
TARGETPATH         \private\10003a3f\apps
END
START_RESOURCE     SimpleEx.rss
HEADER
TARGETPATH         resource/apps
END
SYSTEMINCLUDE      \epoc32\include
USERINCLUDE        ..\include
LIBRARY            euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY            avkon.lib gdi.lib
```

The UIQ project file is as follows:

```
TARGET             SimpleEx.exe
TARGETTYPE         exe
UID                0x100039CE 0xE000027F
SOURCEPATH         ..\src
SOURCE             SimpleEx.cpp
SOURCE             SimpleEx_app.cpp
SOURCE             SimpleEx_view.cpp
SOURCE             SimpleEx_ui.cpp
SOURCE             SimpleEx_doc.cpp
SOURCEPATH         ..\group
START_RESOURCE     SimpleEx_reg.rss
TARGETPATH         \private\10003a3f\apps
END
START_RESOURCE     SimpleEx.rss
HEADER
TARGETPATH         resource/apps
END
SYSTEMINCLUDE      \epoc32\include
USERINCLUDE        ..\include
LIBRARY            euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY            qikcore.lib gdi.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 Carbide.c++ 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 – this is all done behind the scenes from an application developer’s viewpoint.

The Symbian build system will be discussed in Chapter 5. 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. The locations to search for project-defined and system include files, and source files and locations, are also defined.

`TARGET_TYPE` should always be `EXE` 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 particular platform. S60 uses `avkon.lib` and UIQ uses `qikctl.lib`.

Creating the `bld.inf` file

The file `bld.inf` points the build tools to the correct project definition (MMP) file. Type this into a file and name it `bld.inf` in the `group` directory:

```
PRJ_MMPFILES
SimpleEx.mmp
```

2.4 Building and Executing on the Emulator

To build the application, change directory to the `group` subdirectory and execute these commands:

```
c:\...\group> bldmake bldfiles
c:\...\group> abld build winscw udeb
```



Figure 2.10 S60 3rd Edition Example



Figure 2.11 UIQ 3 Example

Make sure you have the appropriate SDK selected when you enter these commands. This can be done with the `devices` command as described in section 2.2.

Run the emulator on the PC (for example, by typing `epoc` at the command prompt as described in section 2.2.1) and select the application labeled `SimpleEx` from the desktop on UIQ or 'Installed' folder for S60.

Figures 2.10 and 2.11 show how the example looks on the different software platforms.

2.5 A Carbide.c++ Project

You can import the `SimpleEx` program into Carbide.c++ as described at the end of section 2.2.3. Once you have the project in Carbide.c++, you can place breakpoints and step through the code.

Figure 2.12 shows the S60 `SimpleEx` project in Carbide.c++. I set a breakpoint at the beginning of the command handler in the application UI class by double-clicking in the gray column at the left of the code line (the dot there indicates that a breakpoint is set). You can start running your program in the debugger by selecting **Run->Debug**. Then when you hit breakpoints, you can step through your code (the Run menu will give you the various options for stepping through the code).

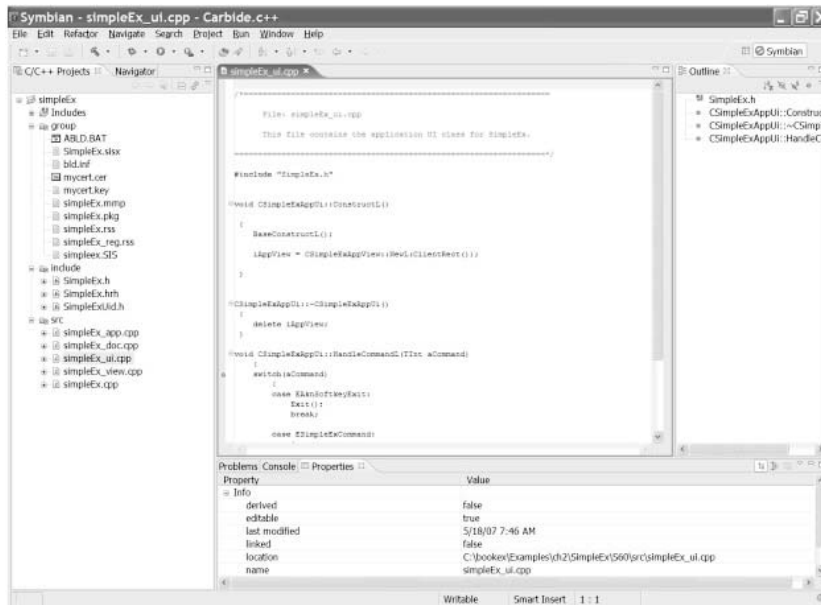


Figure 2.12 S60 SimpleEx in Carbide.c++

2.6 Building for the Smartphone

To build for a smartphone target (we are back on the command line now), you'll need to specify a different build target; ARM (*gcce*) in release (*urel*) mode:

```
c:\...\group>abld build gcce 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 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 build them into the SIS file) and where each of these files should be placed on the phone when the SIS is installed. Once we have a SIS file, we'll sign this SIS file with a self-created certificate in order to create what is called a SISX file. Although it sounds complicated, it isn't really and, for Symbian OS v9, smartphones typically require the applications to be signed (more on this in Chapter 7) so you need to familiarize yourself with this step early on.

Let's look at the package definition files (PKG files) for S60 3rd Edition and UIQ 3. 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 the location where you installed your SDK.

The following shows the S60 package file. The path in bold represents the default location of the S60 3rd Edition FP1 SDK and should be changed if you installed your SDK in a location different to the default directory (*c:\Symbian\9.2\S60_3rd_FP1*).

```
; SimpleEx.pkg for S60 3rd Edition
;
;Language - standard language definitions
&EN
; standard SIS file header
#{"SimpleEx"}, (0xE000027F), 1, 0, 0

;Localised Vendor name
%{"Vendor"}
;Supports Series 60 v 3.0
[0x101F7961], 0, 0, 0, {"Series60ProductID"}
;
;Files to install
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\release\gcce\urel\SimpleEx.exe"
- "!\:sys\bin\SimpleEx.exe"
```

```
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\data\z\resource\apps\SimpleEx.rsc"
- "!\:resource\apps\SimpleEx.rsc"
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\data\z\private\10003a3f\apps
\SimpleEx_reg.rsc" - "!\:private\10003a3f\import\apps\SimpleEx_reg.rsc"
```

The UIQ package file is as follows:

```
; SimpleEx.pkg for UIQ 3
;
;Language - standard language definitions
&EN
; standard SIS file header
#{ "SimpleEx" }, (0xE000027F), 1, 0, 0

; Localised Vendor name
%{ "Vendor-EN" }
; Unique vendor name mandatory for UIQ
: "Vendor"
; Supports UIQ 3
[0x101F6300], 3, 0, 0, { "UIQ30ProductID" }
;
; Files to install
"c:\Symbian\UIQ3SDK\epoc32\release\gcce\urel\SimpleEx.exe"
- "!\:sys\bin\SimpleEx.exe"
"c:\Symbian\UIQ3SDK\epoc32\data\z\resource\apps\SimpleEx.rsc"
- "!\:resource\apps\SimpleEx.rsc"
"c:\Symbian\UIQ3SDK\epoc32\data\z\private\10003a3f\apps\SimpleEx_reg.rsc"
- "!\:private\10003a3f\import\apps\SimpleEx_reg.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. Further discussion of the format of these files can be found in Chapter 5.

Next, run the following command at the command prompt:

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

This command creates the installation file SimpleEx.sis.

Although it's up to the manufacturer, you typically cannot install Symbian OS v9-based applications unless they are digitally signed by a certificate. For this example, we will self-sign the application with a certificate we generate ourselves.

Run the following command to create a certificate:

```
c:\...\group> makekeys -cert -len 2048 -dname "CN=MyName OU=MyCompany"
mycert.key mycert.cer
```

You will be prompted to ask if you want to define a password. Answer no. You will then be asked to press random keys or move the mouse to

generate some random data, after which you will have your certificate file (`mycert.cer`) and an associated private key (`mycert.key`).

To sign your SIS file, enter the following command:

```
c:\...\group\> signsis SimpleEx.sis SimpleEx.sisx mycert.cer mycert.key
```

This command will sign your `SimpleEx.sis` with your newly created certificate files. The signed copy is output as `SimpleEx.sisx`.

Install `SimpleEx.sisx` 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 users to install applications to their smartphones, and the installation process should be well documented in the smartphone's user manual. The `.sisx` suffix indicates a signed SIS file.

Because this application is not signed by a certificate that is certified and signed by a trusted CA, you will get a message indicating that the application is from a non-trusted source – ignore this and continue installing the application.

Once you load the program on the smartphone you can run it as you did from the emulator by locating it by name in the main menu and opening it. Congratulations on creating and running your first Symbian OS C++ program on a smartphone!

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 will 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:¹

- **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.
- **Base libraries:** user library, file system, database manager

¹ For more information about the components that comprise Symbian OS, I recommend you consult the system model diagram available from the Symbian Developer Network (http://developer.symbian.com/main/oslibrary/sys_models).

The user library (`euser.dll`) contains APIs that provide functionality such as string manipulation, linked lists, containers, math functions, error handling, and timers. The user library also provides ‘user-side’ 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.

As you would expect, the file system library (`efsrv.lib`) provides the functionality to manipulate files and directories, parse filenames, and perform file I/O. The database manager (`edbms.lib`) provides a set of interfaces for relation database access.

- **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 HTTP and HTTPS.
- **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 voice telephony and TCP/IP over cellular radio as well as local communication protocols such as Bluetooth technology, IR, and USB. The messaging framework for support is also included, with 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 components which provide services for multimedia, security, and the POSIX-compliant libraries for standard C.

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 desktop computers are able to), you can switch between running applications as needed. Also, as with operating systems such as Linux and Windows, 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 9 covers threads and processes and inter-thread communication in more detail.

3.2.1 Threads

Threads are streams of code execution 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. A single Symbian OS program can have multiple threads, but as you will see later, 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 executing within it. The code for a process is contained in a file whose extension is `.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.

3.3 Shared Code: Libraries, DLLs, and Frameworks

Shared code consists of packages of executable code that, as the name suggests, can be shared by other executables; providing functions to all running programs for reuse. These packages are usually known as dynamic link libraries (DLLs). This is more efficient than the traditional statically linked 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. 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.

Prior to Symbian OS v9, GUI applications were actually DLLs themselves; however, from Symbian OS v9 onwards, applications are implemented as fully independent executable processes.

On Symbian OS, the most common types of shared binary are *static interface libraries* and *polymorphic interface libraries*. They may also simply be referred to as DLLs when the context of the type of shared binary is established.

3.3.1 Static Interface Libraries

Static interface libraries are the traditional style of libraries, containing a collection of classes and functions that are made available to calling programs through a series of exported functions, as Chapter 5 will discuss. The base operating system library (`euser.dll`), which, for example, provides string manipulation functions, is an example of a static interface library. Static interface libraries typically end in `.dll`.

3.3.2 Polymorphic DLLs

Polymorphic interface libraries (most commonly known as ‘polymorphic DLLs’) are used as plug-ins to a framework, as opposed to simply providing classes and functions as in static interface DLLs. They provide a concrete implementation for some abstracted interface. A good example is a device driver, which is a polymorphic DLL that can be loaded or unloaded, kernel-side at runtime.

Polymorphic DLLs, in a similar fashion to static interface DLLs, are loaded when needed and linked to at runtime. Unlike a static interface library, which supplies the implementation of an API that is fixed at compile time and typically loaded by the system, a polymorphic DLL acts as a plug-in, and must be loaded by a framework or other client wishing to use it.

A polymorphic DLL implements an abstract interface as a concrete C++ 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., `.prt` for a networking protocol module).

Polymorphism is built into C++, and is arguably 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.

Polymorphic DLLs use C++ polymorphism across the DLL file boundary. To give you an idea of how this works, here is some code to load a polymorphic DLL and call a method:

```
RLibrary library;
_LIT(KPluginName, "printerX.apr");
library.Load(); // load a plugin
TLibraryFunction entry=library.Lookup(1)
GenericPrinter* printer= static_cast<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 printer'). 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. Lastly, `printer->PrintDocument(MyDoc)` is called, which will print the document using `PrinterX.apr`'s implementation of that function specific to the printer the plug-in represents.

3.3.3 ECOM Framework

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.

More information about ECOM can be found in *Symbian OS for Mobile Phones Volume 3* and *Symbian OS Explained*.

3.3.4 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 non-constant variables externally in functions (automatic variables are fine since they are on the stack), or have non-constant 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 int myVar;
    ...
}
```

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

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

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 other programs, which become *clients* to the server. Servers can 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 usually 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., `RFs`, `RFile`) are client-side classes that send commands to the file server (transparently to the API user), which then executes the functionality. The client-side class typically resides in a DLL.

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 (this can be synchronously or asynchronously).
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 API calls.

Figure 3.1 shows several callers communicating with two servers (for example, the file system server and the window server). Note that a caller may use more than one server and that some servers communicate with each other. As mentioned, servers are always in separate threads from their clients (although multiple servers can exist in a single thread).

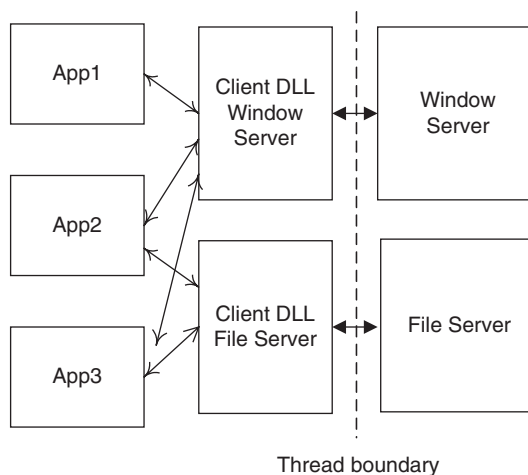


Figure 3.1 Client/Server Interaction

Data is transferred between the client and the server using inter-thread communication functionality within Symbian OS. In the case of the window and file servers used in the example shown in Figure 3.1, the servers are contained in their own processes, and reside in EXE files.

Symbian OS provides a client–server framework that handles the details of the communications between the client and server. Chapter 10 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.

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. Further, 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. For added efficiency, code in ROM is executed in place – that is, it is not loaded into RAM before executing, although this is not true for all types of ROM. 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 Sony Ericsson P990 and the Nokia N95 have 128 MB of internal flash space. 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 (or even more).
- **Hard Drive**
Hard drives have started appearing in smartphones. For example, the Nokia N91 has an 8 GB hard drive.

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-based 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.

Note that to save costs, even modern smartphone designs could include CPUs without an MMU, although this is not common. Symbian OS can support processors with and without MMUs.

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 9). 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):

Static Data and Stack Chunk

This chunk is where the static data and stack resides for the process and all its threads.

Heap Chunk(s)

Where the heap is stored. There can be a heap chunk for each thread, but a heap chunk can also be shared between all threads.

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 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 3.2.²

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 thread is scheduled to execute, the data chunks associated with its process are moved (remapped via the MMU) from the home area of virtual memory to the area known as the run area, and thread 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 – that is, 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

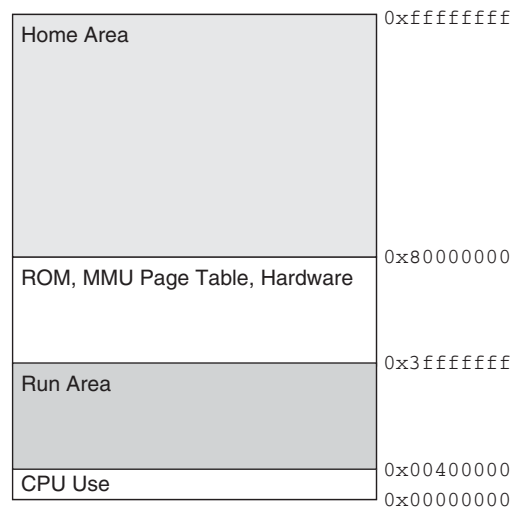


Figure 3.2 Virtual Memory Map

² Note that different memory organizations are best suited to different CPU architectures. Symbian OS describes these organizations as memory models and allows the device creator to select the most appropriate model for their chosen hardware. Currently the most common organization is the moving memory model that is appropriate for ARM v5 CPU architectures, the moving memory model is considered in this book and what is described in this section. The multiple memory model is likely to be used on an ARMv6 CPU architecture, and is not covered in this book.

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. There are also other reasons for having movable chunks like this, including moving in the memory map in order to give it space to grow.

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.

Remapping the data of a running process to a common virtual address area is not unique to Symbian OS. Many other multitasking 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. As we are running two instances of Program A, the ‘_nn’ 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 (may be relocated if chunks have grown in size).
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.
9. The kernel switches context from ProgramA_01 and calls Program B for the first time.
10. The kernel loads the code for Program B from flash to the home area and creates chunks for the ProgramB_01 data in the home area.
11. To execute Program B, the kernel remaps the ProgramB_01 data from the home area to the run area and the code image of Program B executes.

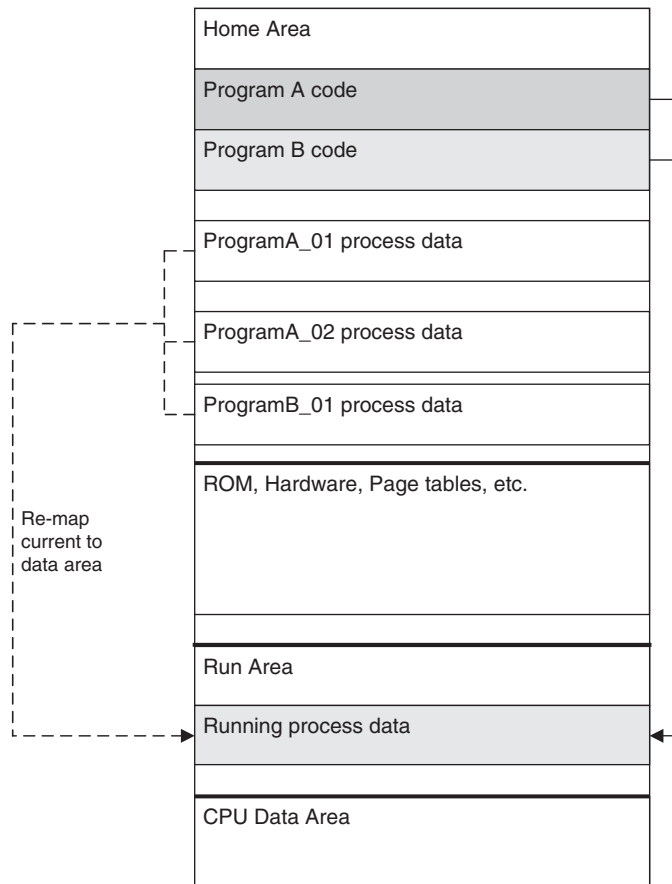


Figure 3.3 Structure of the Virtual Machine

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 (plus the code chunks); 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. Furthermore, 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

The time it takes for an operating system to switch between processes is greater than the time needed to switch between two threads within the same process. One reason is due to the need to relocate the process data in the memory map, although using an MMU to remap virtual to physical memory is much faster than copying the actual data from one area of physical memory to another. Even in the case of using an MMU, however, there is a performance hit. The time penalty is not so much the MMU remapping operation itself, but has to do with the processor cache. If the processor cache stores information referenced by virtual addresses as opposed to physical addresses, then a cache flush must be done every time the MMU remaps the addresses. This is because virtual addresses have now changed and the cache does not have the correct data to reflect this remap, since it stores the data by virtual addresses. This clearing and refilling of the cache causes a significant performance hit. Switching between threads in the same process, however, is faster, since no memory areas need to be remapped.

Note that with the ARMv6 architecture, the cache corresponds to physical memory instead of virtual, and the architecture contains optimized process switching hardware that Symbian OS v9 can take advantage of. In this case, process switching time is significantly reduced.

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. An example is the file system process. To get around this, Symbian OS has the concept of a *fixed process*, where the process data always stays in the home area, even when the process is 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 (of sufficient size) must be reserved and fixed in the system.

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 mediating inter-process data transfers. In addition, the kernel manages all the system memory, and acts as a gateway that provides access to device hardware.

This section covers some of the key highlights of the Symbian OS kernel. If you want to understand the kernel architecture in detail, *Symbian OS Internals* is an excellent source.

3.6.1 Symbian OS Kernels: EKA2 and EKA1

Symbian introduced EKA2 (short for EPOC Kernel Architecture 2) in Symbian OS starting at Symbian OS v8.1b; it is found in all Symbian OS v9-based phones (S60 3rd Edition and UIQ 3). EKA2 replaces the original Symbian OS kernel, retrospectively known as EKA1.

The motivation for creating EKA2 was to add features determined to be important through years of experience in implementing EKA1-based Symbian OS smartphones. A significant improvement in EKA2 is the ability for Symbian OS to run on single processor smartphones, which is important for keeping down costs. This was achieved in EKA2 by implementing real-time processing capability so that both embedded radio software and applications can run on the same processor. I will discuss this more, later in this section. Other additions include easing hardware porting and device driver implementation, changes to allow the software platform to be secured, along with numerous other changes.

3.6.2 Kernel Architecture

Figure 3.4 shows the kernel architecture. The kernel, device drivers, kernel extensions, ASSP (short for Application Specific Standard Product), and variant layers run in privileged mode and therefore have access to all memory and hardware resources.

Abstracting the hardware

Kernel functionality depends on the underlying hardware on a device. For example, timers are needed for task scheduling and timer services.

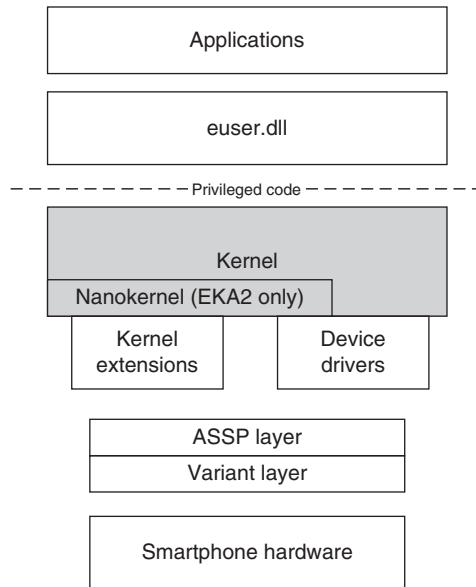


Figure 3.4 Kernel

Another example is the control of the MMU and flushing the cache. The methods of controlling these features vary with the specifics of 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 specifics of the hardware do not matter). At the bottom of Figure 3.4 (just above the hardware itself) are the ASSP and variant layers. These layers abstract the underlying hardware from a software viewpoint.

The ASSP abstracts hardware for a specific family of closely related devices, and the variant layer handles variations of particular devices within those families. As part of developing Symbian OS smartphones, manufacturers implement the classes that make up these layers. Since the software interface is common, the kernel is able to run unchanged on this hardware once the lower-layer classes are ported to that hardware.

Although not shown in the diagram, there is also a CPU layer that abstracts the capabilities of the particular CPU of the smartphone.

Device drivers are used primarily to 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 optional components that needed to be started early (e.g., DMA controller, power manager, etc.).

User library

Above the kernel in Figure 3.4 is `euser.dll`, which runs in user mode. This is a DLL that provides user-mode access to kernel functionality for applications. Since the kernel runs in privileged mode, the functions in `euser.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 component that actually allows user software like `euser.dll` to invoke the kernel is called the kernel *executive*. The executive consists of a set of software interrupt handlers, which, when invoked, will switch the CPU from user mode to privileged mode. When a kernel function is called from an application via `euser.dll`, a software interrupt (the SWI instruction on ARM) occurs that invokes the appropriate kernel function. In EKA1, some kernel calls had to be routed to a kernel server instead of being handled completely in the executive. The reason is that the executive in EKA1 had some restrictions on accessing kernel objects, thus requiring the commands to be done in the kernel server. However, in EKA2, this is not needed since it now can access kernel resources directly in the executive (as user threads now have a kernel-side stack).

Note that the user library contains more than just interrupts to invoke the executive – it also contains basic functions to do things such as string and array manipulation.

Nanokernel

As shown in Figure 3.4, EKA2 has a nanokernel as part of its architecture. The nanokernel, which makes up just a small percentage of the overall kernel code, supplies basic services, such as simple supervisor mode threads, and the rest of the kernel expands on these services. The nanokernel also handles all the system interrupts initially and contains other mechanisms to ensure a predictable response time for software running on it, thus enabling real-time behavior in Symbian OS. Real-time processing and the role of the nanokernel in EKA2 is discussed in the next section.

3.6.3 Real-Time Processing in EKA2

In a computer system it's a given that in order for a piece of software to work right, all computations must be correct. Real-time systems add another component to this – time. So in real-time systems not only must

all computations be correct, they must also be performed within time constraints required by the software. Deviations in this expected timing are considered failures. Real-time systems don't necessarily have to respond fast, they just have to respond within a known amount of time (although slower responding systems may not be suitable for certain applications).

Real-time processing was introduced in EKA2. Why is real-time support desired in Symbian OS? The primary reason is so that a smartphone's low-level telephony radio software (known as baseband functionality) can be run on the same processor as the Symbian OS applications in order to save costs. I'll discuss more in the next few sections. The GSM signaling stack is an example of baseband functionality.

Two-processor smartphone model

Typically, a smartphone contains two processors: a *baseband processor* and an *application processor* (sometimes known as the radio and PDA processors). Figure 3.5 shows a typical high-level view of a two-processor system.

The baseband processor runs the low-level radio telephony protocols, like the GSM signaling stack mentioned before. Baseband software is time-critical and requires a real-time operating system (RTOS). The RTOS can be proprietary, or a commercial one such as Nucleus, OSE, or VRTX. The application processor runs Symbian OS and its applications. An IPC (inter-processor communication) mechanism is supplied to communicate between the two processors when needed.

Why the need for two processors? The main reason is because Symbian OS (EKA1) does not provide the real-time response needed for the baseband software, thus requiring a dedicated processor running an RTOS. With EKA2, however, this is no longer a restriction, as shown in the next section.

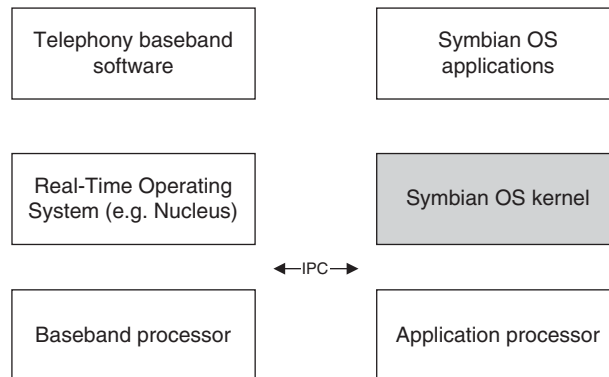


Figure 3.5 Software on Two Processors

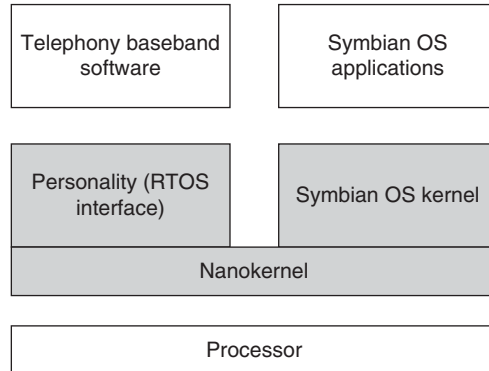


Figure 3.6 Software on One Processor Using EKA2

One processor using the real-time nanokernel

Figure 3.6 shows all the software running on a single processor in the smartphone using EKA2.

Since the nanokernel of EKA2 has real-time processing capability, it's used as the base for both the baseband and application software, thus allowing all software to run on a single processor.

Since it would be a major investment to rewrite baseband software to a different RTOS, Symbian supports a layer on top of the nanokernel, known as a *personality*. A personality is used to abstract the nanokernel interface to make it match the API of another RTOS. In this way, the nanokernel appears like that RTOS to applications that run on it. Personalities can be written for the different RTOSs that baseband software runs on, so that existing baseband software does not have to be rewritten to the nanokernel API. For example, if a smartphone manufacturer wants to reuse existing baseband software written for OSE RTOS to run on the nanokernel, it would use an OSE RTOS personality, and thus not have to modify the existing baseband software, which could be a time-consuming and error-prone task. New personalities can be written as needed, including ones to mimic proprietary operating systems.

Note that baseband functionality and EKA2's real-time functionality is transparent to the application programmer, so you will not need to be concerned at all with it when writing typical applications – unless of course you are implementing or integrating baseband functionality for a manufacturer (for example).

3.7 Active Objects and Asynchronous Functions

Although Symbian OS has support for multiple threads within a process, multithreading is not often used by application programmers. In

fact, it is discouraged. One reason is performance – lots of threads can bog down a system due to context switches. Another reason is that when a system object is created, it can usually be directly accessed only by the thread that created it, making managing multiple threads more difficult. The correct way to handle concurrency 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 of 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.

Active objects (i.e., a class derived from `CActive`) will invoke *asynchronous functions* that, unlike traditional functions, return immediately after being called and run in parallel with the calling thread. When an asynchronous function completes, a signal is sent to the active scheduler, which in turn calls the command handler of the active object that invoked the asynchronous function. Since the asynchronous function does not block the thread, other active objects can be called to invoke more asynchronous functions while waiting for the others to complete. In this way, a single program thread can have multiple asynchronous functions in progress at one time. So although 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 GUI Architecture

Symbian OS has a powerful and flexible GUI architecture. This section presents a high-level overview of its features. The details of the architecture as it pertains to application development are discussed in Chapter 12.

3.8.1 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 and S60.

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 (although it's possible that some could see this as an advantage since they would not have to refamiliarize themselves with new interfaces if they switched phones).

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.

3.8.2 Introducing the GUI Framework

The main components of the GUI architecture and their relation to each other are shown in Figure 3.7.

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 act as calling clients to this server by linking to its client-side library `ws32.dll`.

The window server handles the details of drawing window and control objects to the screen, as well as keeping track of which windows belong

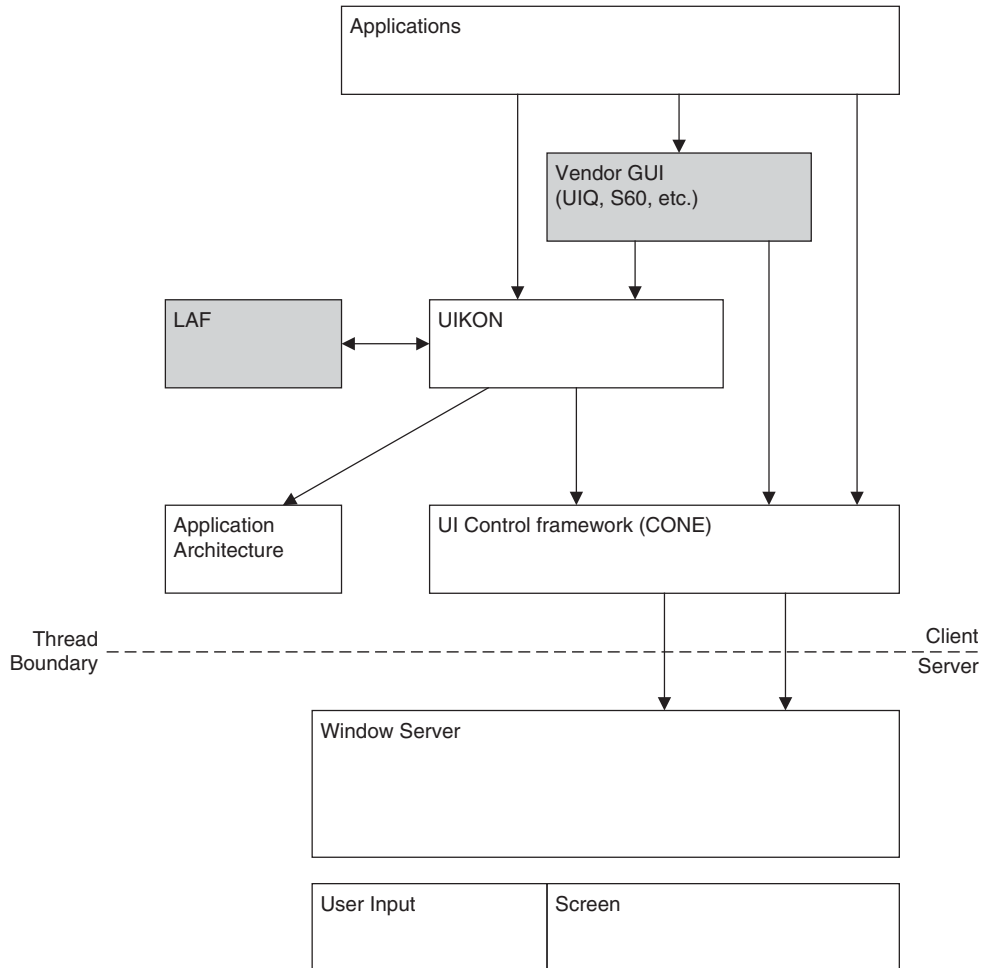


Figure 3.7 GUI Architecture

to which applications. The window server also ensures 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 (e.g., edit boxes, buttons, and text fields) 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. It can also customize application architecture-oriented behavior.

The vendor can also supply their own custom controls to supplement the existing *UIKON* controls. For example, *UIQ* and *S60* provide a version of a date and time editor control, which does not exist as a *UIKON* control.

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.

3.9 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. One type of application that falls into this category is any game with frequent screen updates (such as an action game). Symbian OS provides several mechanisms for providing high-performance graphics, including animation plug-ins and APIs for direct screen access.

Animation plug-ins are polymorphic DLLs 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).

3.10 The Communication Architecture

Communication 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 11 will discuss communications in more detail, as you learn to develop communication applications. This section (and Chapter 11) assumes some basic knowledge of network communications. For more information, see *Symbian OS Communications Programming, Second Edition*.

Figure 3.8 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 11.

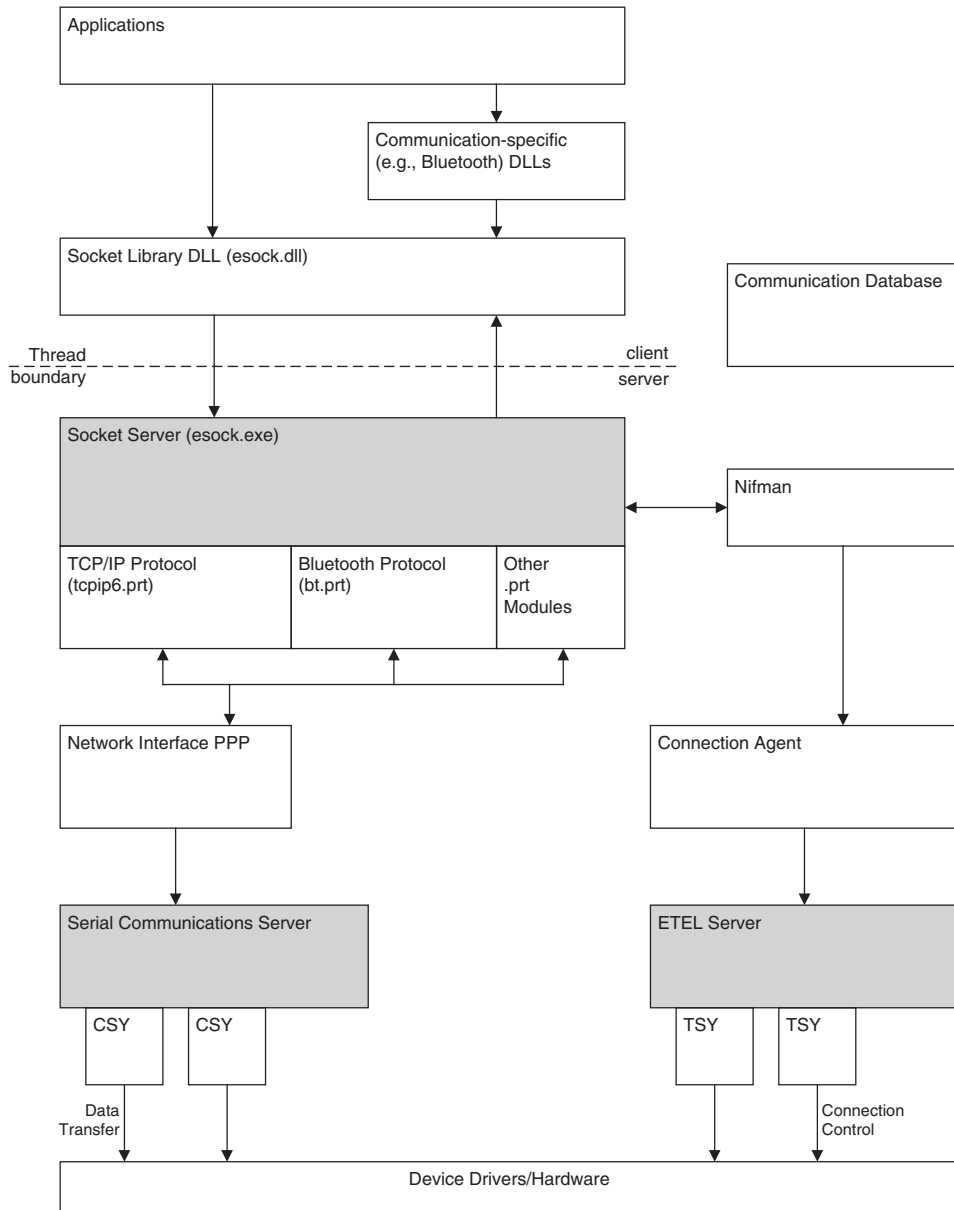


Figure 3.8 Communication Architecture

- **Communication-Specific Functionality**
In addition to the socket API, there are also APIs specific to certain types of communication, such as Bluetooth communication. 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 uses *Nifman* 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 *TSY*s for devices such as GPRS and GSM (these 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 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 multihoming 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 opened up many possibilities for devices that support multiple ways of accessing the Internet and has become increasingly important for smartphones built on Symbian OS v9.

3.11 Application Engines and Services

Symbian OS provides *application engines* to access and manipulate data used by key applications, such as Calendar and Contacts. This is useful in creating companion applications that work in conjunction with them. 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. Examples are the alarm and log servers, which handle setting and initiating alarms and logging various types of system information, respectively.

3.12 Platform Security

Platform security is a significant addition to Symbian OS v9. The primary role of platform security is to ensure the integrity of the phone by restricting access to sensitive functions and data only to software approved to access them.

At a high level, platform security divides the system into three areas: the Trusted Computing Base (TCB), the Trusted Computing Environment (TCE), and applications, which may be trusted or untrusted, as shown in Figure 3.9.

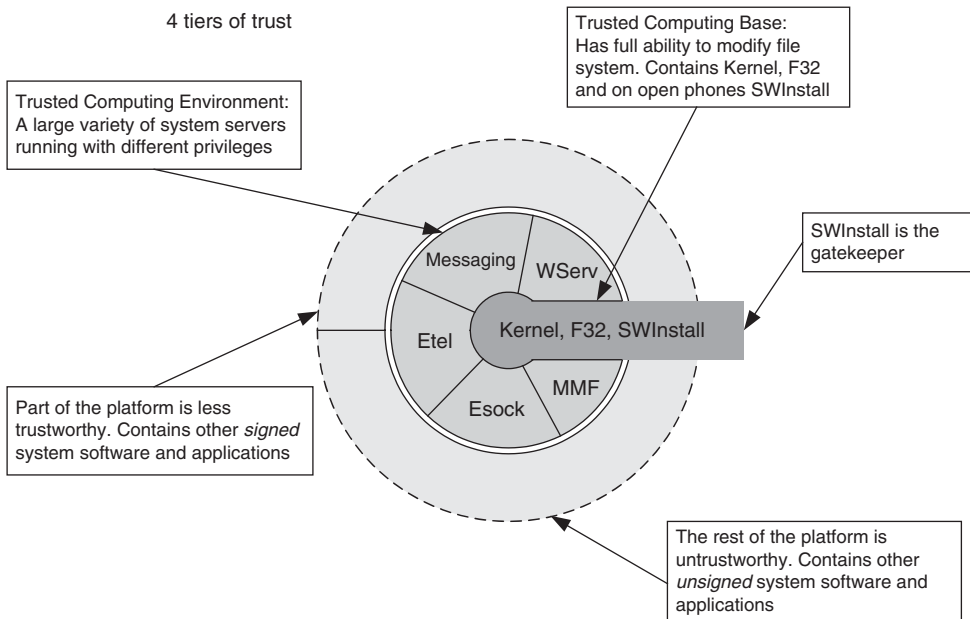


Figure 3.9 Platform Security Tiers of Trust

The TCB is the core software that has access to everything in the system. The kernel and the file server are examples of software that runs in the TCB. The rest of the software on the device trusts that the TCB software will behave correctly, and any code that runs in the TCB is reviewed very carefully.

The TCE contains system components that access critical phone resources, however, they are not granted blanket access, but only the particular access needed. Functionality that runs in the TCE is usually implemented in servers. Software at the application level communicates with the TCE to access sensitive system functionality.

Application-level software is typically less trustworthy and thus does not often have capabilities that allow access to sensitive functionality of the phone. However, applications do need certain user-level capabilities for performing things like accessing the Internet, making a phone call or Bluetooth connection, or accessing private user data.

In addition to protecting access to critical functionality on the phone, platform security also provides a mechanism called *data caging*, which allows an application to store data in a folder reserved for it that cannot be accessed by other applications (except for very high-privileged applications).

Chapter 7 discusses platform security in detail and includes what the programmer needs to know to develop software for this architecture. Chapter 7 also covers the procedures necessary to get your application *Symbian Signed*, that is, getting your application signed with a digital signature trusted by the device to indicate that your application is approved to access all the functions it uses.

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 S60 or UIQ in this chapter. The information presented here is generic for all platforms based on Symbian OS.

I begin the chapter with an overview of the use of C++ in Symbian OS, followed by a look at the basic data types, the key types of classes you'll use and create, and the Symbian OS naming conventions. 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 summarize the 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++. The first version of the operating system (known then as EPOC32) was created before C++ itself was formally standardized, that is, prior to 1998. For this reason, and the fact that C++ was not designed to be optimal on mobile devices, some of the later additions to C++, such as exceptions and namespaces, were not originally adopted by Symbian OS, although v9 has introduced them.

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 Non-standard C++ Characteristics

Although Symbian OS uses many of the object-oriented features of C++, some of its functionality is implemented in non-standard ways. This can require an adjustment, even for experienced C++ programmers. For example, instead of ISO standard C++ exceptions, Symbian OS has its own mechanism for exception handling in the event of conditions such as low memory or a dropped Internet connection. Also, Symbian OS does not provide a 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, which include making the OS 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, TInt:** 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, TUInt64:** A 64-bit integer, which is typedef'd to `long` and `unsigned long` from Symbian OS v9, and uses the available native 64-bit support. Before Symbian OS v9, `TInt64` was implemented as a class.

- **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 and compiler independent.

The example below shows some sample declarations for the basic data types.

```
TInt Foo(TInt aParm1, TText aParm2, TAny *aPtr)
    // returns an int
    // takes an int, a character, and a pointer
{
    TInt var1;
    TChar dummyChar;
    dummyChar = 'A';
    for (TInt i=0;i<10;i++)
        { /* some stuff */ }
    dummyChar.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 ('mixins').

4.4.1 Data Type Classes: T classes

Data type classes ('T classes') encapsulate one or more types, usually built-in types (like TInt, TText) or other T classes. T classes have optional methods for manipulating, comparing, and otherwise controlling the type(s) they contain. The TChar class described in the previous section is a good example of a T class: each instantiation of TChar holds a character value. The TChar class methods can be used to perform operations on that value.

Note that it's important that a T class not contain any data that requires destruction, so do not own heap-based objects or resource handles. T classes do not have destructors.

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: C 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.

Classes of this type should never be allocated on the stack. The zero initialization is done by CBase as a result of the `new` operator, and since `new` is not performed for stack instantiation, the member variables will contain undefined data when instantiated on the stack (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: R 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 that just hold a handle to the actual resource, which is controlled by the server. Furthermore, 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 – and the resource must be released explicitly by calling `Close()`.

Another example is *RThread*, as illustrated below. (The class *RThread* will be covered in more detail in Chapter 9.)

```
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: M 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 CMyClass: public CBase, MInterface1, MInterface2
{
    virtual void DoThis();    // override method from MInterface1
    virtual void Callback(); // override method from MInterface2
};
void CMyClass::DoThis()
{
    //implementation here
}
void CMyClass::Callback()
{
    // implementation here
}
```

This example shows two interface classes, `MInterface1` and `MInterface2`, each consisting of a single abstract function. Class `CMyClass` uses multiple inheritance to inherit from `CBase`, and from the two interface classes. `CMyClass` 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). Furthermore, 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 is assumed to be a valid pointer here
    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 encouraged. 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, `CClass2` implements two interface classes, `MProtocol1` and `MProtocol2`. `CClass1` implements `MProtocol1` only. Both classes, as is typical, also inherit from a normal, concrete class in addition to the interfaces they implement.

`Function1(MProtocol1 aProt1)` will accept an argument of type `CClass1` or `CClass2` since both of these classes are derived from `MProtocol1`. The argument is cast down to an `MProtocol1` type and `Function1()` will then manipulate the object as needed through the `MProtocol1` interface methods.

`Function2(MProtocol2 aProt2)` will accept objects of type `CClass2` since `CClass2` inherits from `MProtocol2`. Passing `CClass1` to this function will generate a compiler error since `CClass1` 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 mixin class concept as being a C++ implementation of the Java `interface` keyword.

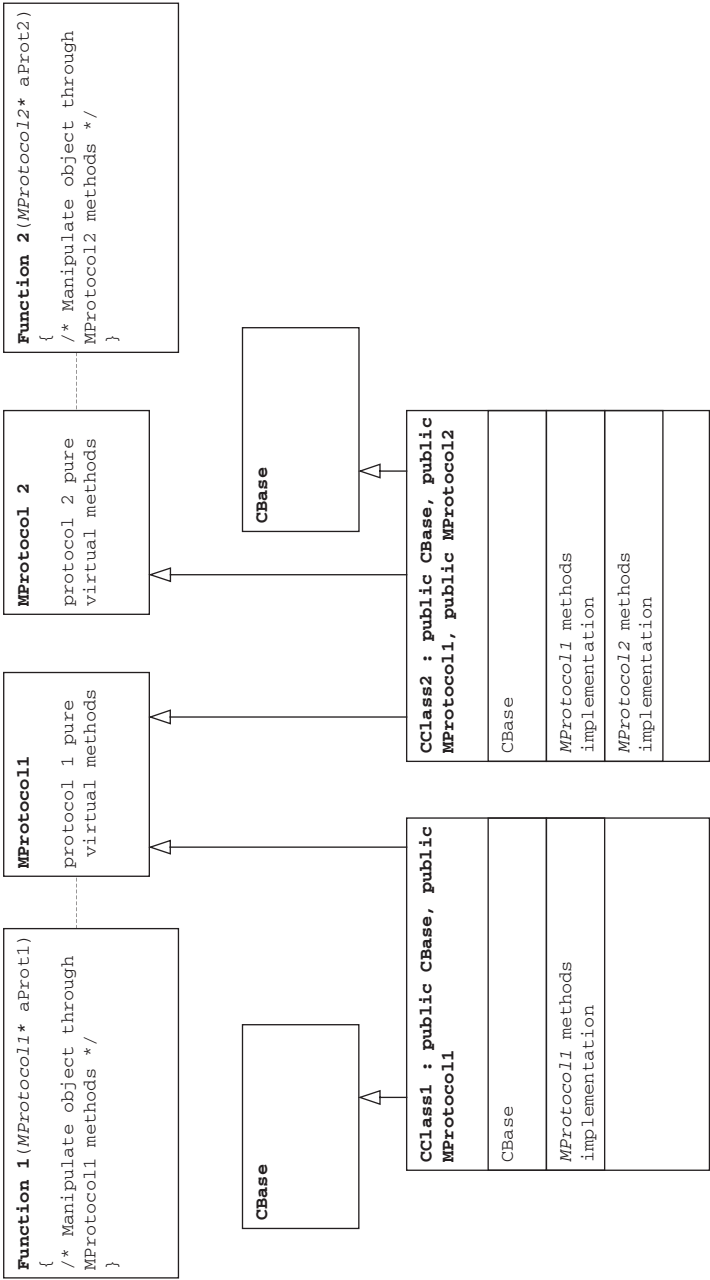


Figure 4.1 Interface Classes Example

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 application design and development effort. This section describes it in more detail.

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` statement, after the function call, can test for and handle an error.

But providing a return status 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 Leaves and TRAP Harnesses

To solve the problems just mentioned, Symbian OS provides an exception-based error handling and recovery mechanism based on *leaves* and *trap harnesses*. 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 harness exists, will also exit at that point. This process continues up the calling chain until a trap harness is encountered, at which time the error is handled. Figure 4.2 illustrates this exiting process up an example nested calling chain to a harness known, on Symbian OS, as a TRAP.

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 `Func2L()`, `Func3L()`, and `Func4L()` are in Figure 4.2), it will act as if a return occurred at that point: the stack will unwind and all automatic variables will go out of scope and thus will be deallocated from the stack. However, any cleanup that requires explicit destructor code will be skipped by the

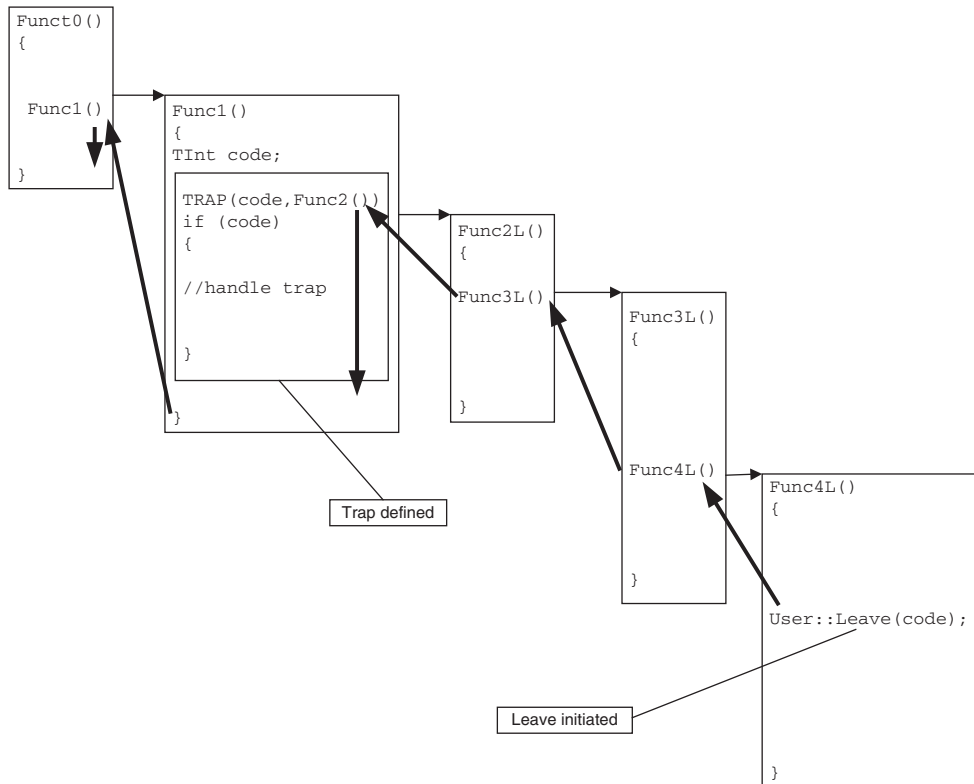


Figure 4.2 Leave/TRAP

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 the leave and TRAP is similar to C++'s built-in try, throw, catch exception handling, you may wonder why Symbian invented its own exception handling. The main reason is that exception handling was not a formal part of C++ at the time Symbian OS was written, and even after it was introduced in C++, it was considered too inefficient for Symbian OS. As a result, the compilers supplied with the Symbian OS SDK would flag an error if you tried to use the try, throw, or catch mechanism in your code. However, with Symbian OS v9 this is no longer the case – standard exception handling is now allowed. In fact, in Symbian OS v9, leaves and TRAPs are built on top of standard C++ exception handling. However, Symbian recommends you only use the standard exception handling if you are porting legacy code that already uses it over to Symbian OS; otherwise, you should continue to use leaves and TRAPs. Also, you should not mix the two exception handling methods in a single program. This chapter will cover the Symbian OS leaves and TRAPs only.

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

```
void FooBarL()
{
    TInt rc = SomeFunction();
    if (rc!=KErrNone)
    {
        User::Leave(rc);    // leave invoked
    }
    // The code here is not executed if a leave occurred above.
}

void MyFunctionL()
{
    ....
    FooBarL();
    // The code from here on will not be executed
    // if a leave occurred in FooBarL()
    ...
}

void StartHere()
{
    TInt leaveCode;
    // invoke MyFunctionL(), with a trap to catch leaves
    TRAP(leaveCode,MyFunctionL());
    if (leaveCode!=KErrNone)
    {
        // MyFunctionL() left, handle leaveCode here
    }
    // Code always executes here - regardless of leaves in MyFunctionL()
```

In the example, execution starts at `StartHere()`. `StartHere()` invokes `MyFunctionL()` through the TRAP macro and `MyFunctionL()` invokes `FooBarL()` (without a TRAP). If `rc` is an error 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 is written to the first argument of the TRAP macro (`leaveCode`) and execution continues normally. If a leave did not occur, `leaveCode` is set to `KErrNone` (`= 0`).

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

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()** leaves (either directly or if a leave occurs further down the calling chain), then **leaveCode** is set to the value with which the leave was seeded. If a leave did not occur (the normal case) and **FunctionL()** executes to completion, **leaveCode** is set to **KErrNone (0)**. 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 flag it as an undeclared variable.

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(leaveCode, MyFunctionL());
```

is equivalent to:

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

If **TRAPD** is used in place of **TRAP** (for example, in the **StartHere()** function shown in section 4.5.2), 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 name in the first argument. To avoid multiple declaration errors in that case, you could use **TRAPD** first and **TRAP** thereafter.

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, leave using
                           // error as the reason.
User::LeaveIfNull(TAny *aPtr); // if ptr is NULL, leave with 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 Does the ‘L’ Suffix 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 clean up any allocated resources properly if a leave occurs.

The following example shows some code with a cleanup problem – an accident waiting to happen.

```
MyFunction()
{
    CThisObject *obj;
    obj = new CThisObject;
    ...
    FuncL();
    ...
    delete obj;
}
```

If a leave occurs in `FuncL()`, the `delete obj` line would never be called and you would be left with allocated memory on the heap. Since `obj` 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 deletion in the class destructor. 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 be handled further up in the call stack. Clever, but still awkward to do for every call of an L function in your program, and expensive in terms of runtime speed, because setting up a TRAP requires a call to the system. 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. The following example shows the cleanup stack in use.

```
FuncL()
{
    CMyObject *myObj = new (ELeave) CMyObject;
    CleanupStack::PushL(myObj);
    CMyOtherObject *myOtherObj = new (ELeave) CMyOtherObject;
    CleanupStack::PushL(myOtherObj);
    DoSomethingL();
    CleanupStack::Pop(2, myObj); // Pop last two items off cleanup stack
    delete myOtherObj;
    delete myObj;
}
```

Both `myObj` and `myOtherObj` 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 `myOtherObj` 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, myObj)`.

The argument '2' means to remove the last two items pushed, and passing `myObj` is optional, but allows the cleanup stack to check in debug builds that the last object you think you are popping off is the correct one, thus helping you eliminate programming errors.

Note that when each object is created in the example, there is an `(ELeave)` after `new`. This causes a leave if the memory allocation fails. See section 4.5.11 for more details on how this works.

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 the previous example, `CleanupStack::PopAndDestroy(2, myObj)` could be used to replace `CleanupStack::Pop(2, myObj)` and the two `delete` statements.

Note that you should not push class member variables onto the cleanup stack. Any pointers allocated and assigned as member variables should be deleted in your object's destructor. If you push them onto the cleanup stack, you risk the possibility that deletion will be attempted twice, resulting in a crash. Only pointers allocated within a function as automatic variables should be pushed to the cleanup stack since once the function exits, there is no chance to free those pointers.

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 two of the most commonly used `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.

The reason `PopAndDestroy()` does not call `delete` for non-CBase objects is that it does not know your class type and thus cannot get to your destructor. CBase, on the other hand, has the destructor declared as virtual so `PopAndDestroy()` can call `delete` on it knowing that if you have an explicit destructor defined in your CBase-derived class, it would be called.

Calling `PopAndDestroy()` on non-CBase objects is fine for objects without explicit destructors. If your object does have an explicit destructor, however, make sure it is derived from CBase so that `PopAndDestroy()` will call your destructor, otherwise your object will only be partially destroyed.

Note that `PushL()` can itself leave if there is insufficient memory. 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 aUserCleanup)
```

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:

```
void MyCleanupFunc(TAny *aParam)
{
    // Will execute on leave or PopAndDestroy. Do special cleanup here.
}

void Foo()
{
    TInt data=1;
    CleanupStack::PushL(TCleanupItem(MyCleanupFunc, &data));
    // ...
    Func1L();
}
```

```
// ...
CleanupStack::PopAndDestroy();
}
```

If `Func1L()` leaves, `MyCleanupFunc()` will execute with the argument set to `&data` (in this case `data` is an integer variable containing 1 for an example argument). Note that in the normal case where `Func1L()` does not leave, `MyCleanupFunc()` will execute when `CleanupStack::PopAndDestroy()` is called.

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 clean up. The following example shows this function in action:

```
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 `CleanupClosePushL()` (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 clean up 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.

You may ask why CleanupDeletePushL() is ever needed since you can derive any classes you have destructors for from CBase and use CleanupStack::PushL(CBase*). This is a good question, and in fact Symbian recommends you do just that. However, a good use for CleanupDeletePushL() is when you are referencing CBase-derived objects through M class pointers. CleanupStack::PushL() will not recognize an object referenced by an M class pointer as being derived from CBase even though the object typically is through multiple inheritance. Calling CleanupDeletePushL() in this case will cause the object's destructor to be called.

4.5.10 LC Functions

Functions that end in LC provide an added convenience – upon successful completion the function returns with the object allocated and on the cleanup stack. This can be illustrated as follows:

```
void Func1L()
{
    TInt *buffPtr = User::AllocLC(1000); // system static API which
                                         //allocates memory
    ...
    FooL();
    CleanupStack::PopAndDestroy(buffPtr);
}
```

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 an 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 the following code?

```
Func1L()
{
    CMyClass *obj1 = new(ELeave) CMyClass;
    CMyClass *obj2 = new(ELeave) CMyClass;
    CleanupStack::PushL(obj1);
    CleanupStack::PushL(obj2);
    Call1L();
    Call12L();
    CleanupStack::PopAndDestroy(2, obj1);
}
```

The problem is that if a leave occurs when constructing the obj2 CMyClass object, then obj1 will not be destroyed. You must push obj1 before creating the second CMyClass instance (obj2). The corrected code is as follows:

```
Func1L()
{
    CMyClass *obj1 = new(ELeave) CMyClass;
    CleanupStack::PushL(obj1);
    CMyClass *obj2 = new(ELeave) CMyClass;
    CleanupStack::PushL(obj2);
    Call1L();
    Call12L();
    CleanupStack::PopAndDestroy(2, obj1);
}
```


The original code has an additional problem, in that if the `PushL()` of `obj1` leaves, then `obj2` will not be deleted. However, `obj1` will be cleaned up correctly in the event that `PushL(obj1)` leaves, because the cleanup stack guarantees not to leave until the pointer pushed to it is safely stored.

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 OS commonly uses the idiom known as *two-phase construction*.

4.5.13 Two-Phase Construction

The two-phase construction 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.

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 the following code.

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

Or, if you want to provide a `NewLC()` function, you could implement both, as shown below.

```
CMYObj* MYObj::NewL()
{
    CMYObj* self = NewLC();
    CleanupStack::Pop(self);
    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 pop-up with information regarding the error (the Symbian OS Library 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 at a specific offset via the `Write()` API function of class `RFile`, but your offset passed to the function is negative, 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 S60, when a panic occurs a box that simply says ‘Program Closed’ is displayed or the program just closes without any message at all. To cause the full panic information to appear, you need to create a dummy file (which can be empty) named `ErrRd` in the directory `\resource\` for S60 3rd Edition, and `\system\bootdata\` for previous

S60 versions, on the target system's C drive. This works for both the emulator and the smartphone. If no file manager is available on the S60 smartphone, then the file can be added via a SIS file (a signed one in the case of S60 3rd Edition). The following code shows an example of calling panic.

```
_LIT(KFooProgram, "Foo program"); // Defines a string and assigns to
                                   // KFooProgram
void Foo(TInt aX, TAny *aBuff)
{
    if (!aBuff)
    {
        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 used to confirm programming logic, as shown in the following code. The macro implements a simple `if` statement, so you could replace:

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

with one of the following lines:

```
__ASSERT_ALWAYS(aBuff, User::Panic(KFooProgram, 3));
```

or

```
__ASSERT_DEBUG(aBuff, User::Panic(KFooProgram, 3));
```

`__ASSERT_DEBUG` will only raise a panic in debug builds. Assert macros should not be used to check data that could legitimately fail (for example, you should not assert that there is an active Internet connection because it is possible that a connection has dropped and an application should handle this gracefully rather than simply panic and terminate).

4.6 Libraries

As Chapter 3 discussed, the two main types of library in Symbian OS are static interface shared libraries and polymorphic interface shared libraries (most commonly known as polymorphic DLLs). Static interface shared libraries are linked at compile time. Polymorphic DLLs, on the other hand, are loaded 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 types of shared library 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 into 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 Library

Chapter 5 discusses the various build issues encountered with library code (including freezing exports and the `def` file inner workings). This section outlines a few programming points and presents the basic code structure.

These rules must be followed when writing a library:

- 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 library users.
- In the CPP files, add `EXPORT_C` before the implementation of each function you want available to library users. There should be exactly the same number of `EXPORT_C` statements in the implementing CPP file as `IMPORT_C` statements in the corresponding header file.

These statements are required in order for a function to be available for use by clients of the library. `IMPORT_C` and `EXPORT_C` are mapped to compiler-specific keywords. Some development tools may be more lenient (such as the Microsoft compiler, which lets you get away with just adding the `IMPORT_C`'s in the header) but, for portability, you should use both `IMPORT_C` and `EXPORT_C` as specified.

The following is an example of a skeleton shared interface library.

```
//MyLibrary.h
class CLibraryClass
{
public:
    IMPORT_C CLibraryClass();
    IMPORT_C void Func1();
    IMPORT_C void Func2();
    void Func3();
    virtual void Func4();
};

//MyLibrary.cpp
EXPORT_C CLibraryClass::CLibraryClass()
{
    // do stuff
}

EXPORT_C void CLibraryClass::CLibraryFunc1()
{
    // do stuff
}

EXPORT_C void CLibraryClass::Func2()
{
    // do stuff
}

void CLibraryClass::Func3()
{
    // do stuff
}

void CLibraryClass::Func4()
{
    // do stuff
}
```

In this example, the constructor, `Func1()`, and `Func2()` are exported and available for use by other programs (indicated by `IMPORT_C`/`EXPORT_C` declarations). `Func3()` is not exported and thus is not available – a program will get an unresolved external link error if it calls 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 implicitly (although it does not hurt to add `IMPORT_C`/`EXPORT_C`).

The MMP file is shown below. `TARGETTYPE d11` determines that the binary type is that of a shared library.

```
TARGET      mydll.dll
TARGETTYPE  d11
UID         0x1000008d 0xE0004264
SOURCEPATH  ..\src
SOURCE      mydll.cpp
USERINCLUDE .
USERINCLUDE ..\include
SYSTEMINCLUDE \Eproc32\include
EXPORTUNFROZEN
```

4.6.2 Import Libraries

When you build a DLL, you actually build two files: a DLL and a static library (which can be a LIB or DSO file depending on your build target, see section 5.4.2). The static library is 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.

4.6.3 RLibrary API

As an alternative to static linkage to a library, you can use the `RLibrary` API to dynamically load it into memory and invoke its exported functions. The functions are invoked by ordinal number, and you have to know what functions correspond to what ordinal. Exporting library functions by ordinal numbers and `RLibrary` are described in more detail in Chapter 5.

4.6.4 Polymorphic DLLs

A polymorphic DLL is a normal shared library, but with only one exported function – a factory function that creates an instance of an object and returns it, as shown below.

```
//H file
IMPORT_C MPolyInterface *NewMyPolyL();
class CMyDerivedPolyPlugin : public CBase, public MPolyInterface
{
    // ...
};
// CPP file
EXPORT_C MPolyInterface *NewMyPolyL()
{
    return new(Eleave) CMyDerivedPolyPlugin;
}
// Implement rest of class member functions here
```

As covered in Chapter 3, polymorphic DLLs implement virtual functions, typically those defined in M class interfaces. You can create multiple

polymorphic DLLs that all implement this `NewMyPoly()` function, but return different `MPolyInterface`-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., `MPolyInterface`). This common code does not care which plug-in DLL has been loaded.

4.6.5 Static Data in DLLs

Symbian OS versions before v9 do not allow you to have writable static data in a DLL, although Symbian OS v9 and above do allow it. As I mentioned in Chapter 3, even in Symbian OS v9, writable static data is strongly discouraged in library code. This is a design preference for efficient use of memory. Tacking on a data memory region for DLLs is costly considering the large number of DLLs available. It can be a challenge sometimes to not use any static data in DLLs, especially true when porting existing code. Note that you can have global variables – they just should be read-only (i.e., of type `const`).

For Symbian OS versions before v9, be aware that the emulator will allow you to put static data in your DLLs, however, 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:

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

4.7 Executable Files

The EXE file is the basic executable image file for Symbian OS and all processes reside in one. For example, in Symbian OS v9 and above, GUI applications are EXE files. In Symbian OS releases before v9, a GUI application was actually a DLL, and was launched transparently from an EXE file called `apprun.exe`.

You may need to implement your own EXE files for non-GUI processes as well, such as if you implement a server. The following example shows 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++; // Used to illustrate that you can use
                     // writable globals in exe files
    }
}

```

This is the corresponding MMP file for building the source of this EXE:

```

// exe mmp file
TARGET      myexe.exe
TARGETTYPE  exe
SOURCEPATH  ..\src
SOURCE      myexe.cpp
USERINCLUDE .
USERINCLUDE ..\include
SYSTEMINCLUDE \Epoc32\include
LIBRARY     euser.lib

```

The executable in the example above loops for ever 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 even in pre-Symbian OS v9 versions.

An application can start an EXE file by using the `RProcess` class, which is discussed in Chapter 9.

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 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 this 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
{
public:
    ...
    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 TInt KMyConstant=3;
```

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 – that is, 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 stack as needed to prevent memory leaks.
- Any function that has a possibility of leaving should end in `L` (e.g., `FooL()`).
- Consider using the overload of operator `new` for instantiating objects without having to check that the pointer returned is valid (e.g., `CClass = new(ELeave) CClass`). However, remember that the code could leave if instantiation does not succeed.
- 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 on the stack.
- 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) before you exit the last TRAP harness created before that function call.
- Only use `CleanupStack::PushL()` for `CBase`-derived pointers, or for simple heap-based objects that have no destructors.
- Use the correct 'cleanup stack' push functions, to ensure safe cleanup of objects other than those derived from `CBase`; for example, `R`-objects pushed using `CleanupClosePushL()`. For non-`CBase` deleted heap objects, use `CleanupDeletePushL()` (and others, `CleanupReleasePushL()`, etc.).
- When writing a shared library DLL, place `IMPORT_C` in the `h` file before the declaration of each function that you want accessible to the DLL user and place `EXPORT_C` in the `CPP` file before each function implementation.
- Do not use global writable data in your DLLs, if at all possible. Read-only global variables (type `const`) is fine.
- Follow the naming standards for functions, 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`).

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 build and install your program successfully. I'll also cover 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

Section 2.1.3 explains how to get the correct SDK to work with your chosen smartphone model. You will typically download it as a zip file, which is then installed. By default, the SDK installs to a directory called `symbian` in the root of the installation drive you have chosen (e.g., `c:\symbian`). UIQ SDKs are placed in subdirectories of `\symbian`, which specify the product name. For example, the UIQ Version 3 SDK is installed at `\symbian\UIQ3SDK`. The Nokia S60 SDKs are placed in `\symbian`, using subdirectories that specify the Symbian OS version number and the product name. For example, the S60 3rd Edition FP1 SDK is installed by default to `\symbian\9.2\S60_3rd_FP1`.

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

5.1.1 The `epoc32` Directory

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

epoc32\include

This directory contains the system include files (.h and .inl files) needed for your software, stored either in this directory or in one of its subdirectories, such as \epoc32\include\ICL for the multimedia image conversion library header files. The files e32std.h and e32cmn.h are good ones to skim through because they contain a number of the common system API class declarations (although others are distributed through a number of include files in this directory). If you're not sure which file the header file for the class you want to use is located in, you can either use your favorite search technique or look in the Symbian OS Library documentation within the SDK. The class reference documentation specifies the header file, and library to link against, at the top of the page associated with the class.

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 command line build performed for a project located in 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\release

The \epoc32\release directory contains the executables for all supported target platforms. This is where the final executables (e.g., EXE and DLL files) are placed when you build your software. Directory \epoc32\release has subdirectories for each platform supported by the SDK (e.g., WINSCW, GCCE, and ARMV5). 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 used for production code.

On emulator build targets (WINSW), under both UDEB and UREL, there is a subdirectory 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., GCCE), 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. This directory also contains the emulator configuration file, `epoc.ini`, which is discussed further in section 5.6.1.

epoc32\winsw

This is the default location for the writable emulator memory drives for SDKs that support Carbide and Code Warrior. Pre-Symbian OS v9 versions of the SDK that supported Microsoft tools created these drives under a different directory: `epoc32\wins`.

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 browse to find them for your particular SDK. For example, the S60 3rd Edition SDK examples are in a directory named `S60Ex`. UIQ Version 3 SDKs have a directory called `UIQ` under the `Examples` directory that contains UIQ-specific examples.

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. The organization of the documentation files varies between SDKs. On UIQ 3, the documentation is stored in `\UIQ3SDK\Documentation`, while for S60 3rd Edition, the documentation directory is `S60Doc`.

SDK documentation typically exists as compiled HTML help files (`.chm`), but can also be presented in PDF and HTML.

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 `mvccsym2` for the Metrowerks compiler used to build emulator binaries and `gcc` for builds to run on a smartphone. There are also 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.

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.

5.3 Basic Build Flow

To build a Symbian OS program, you need two build files:

- Component description file (always named `bld.inf`).
- 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.

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 winscw udeb
```

to perform a winscw 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.

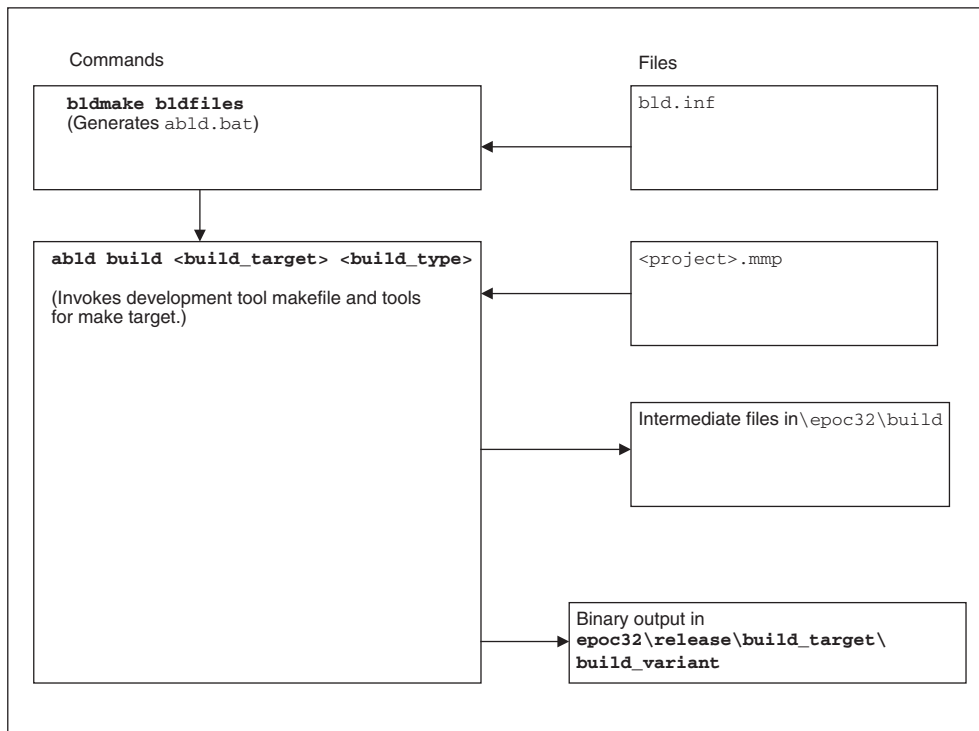


Figure 5.1 Build Flow

5.3.1 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 gcce urel**). There is also a **abld reallyclean**, which will remove additional files such as the generated makefiles and can be used to help resolve build issues.

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.

build_target specifies a build target name that indicates the toolset options and determines if you are building for the emulator or the phone. Section 5.4 describes build targets in more detail and provides a list of them.

build_type should be **udeb** for builds suitable for source-level debugging. **build_type** of **urel** indicates a release build with no symbols.

5.3.2 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.

The following shows the project definition file for an example program built with the S60 3rd Edition SDK.

```
TARGET           SimpleEx.exe
TARGETTYPE       exe
UID              0x100039CE 0xE000027F
SOURCEPATH       ..\src
SOURCE           SimpleEx.cpp
SOURCE           SimpleEx_app.cpp
SOURCE           SimpleEx_view.cpp
SOURCE           SimpleEx_ui.cpp
SOURCE           SimpleEx_doc.cpp
SOURCEPATH       ..\group
```

```

START RESOURCE    SimpleEx_reg.rss
TARGETPATH        \private\10003a3f\apps
END
START RESOURCE    SimpleEx.rss
HEADER
TARGETPATH        \resource\apps
END
SYSTEMINCLUDE     \epoc32\include
USERINCLUDE       ..\include
LIBRARY           euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY           avkon.lib gdi.lib
CAPABILITY        None
SECUREID          0xE000027F

```

This section describes some of the main MMP file statements. You should consult the Tools and Utilities guide in Symbian OS Library that comes with the SDK documentation for a 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
- **TARGETTYPE** *type* specifies the type of executable that is to be created. The two main target types are:
 - exe – Process executable, includes GUI applications (in pre-Symbian v9, GUI applications should be of type app).
 - dll – Dynamic Link Library (DLL), that is, a shared library executable.
- **UID** *uid2 uid3* specifies the second and third UIDs for your component. Refer to section 5.5 for information on UIDs in Symbian OS, for example:
 UID 0x100039CE 0xE000027F
 indicates a GUI application (0x100039CE) with a unique UID of 0xE000027F.
- **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.

- **START RESOURCE** `resource_file` indicates the start of a section that describes building resource file `resource_file`. The resource section is ended with the keyword **END**. The location of the resource file is determined by the last **SOURCEPATH** statement before the resource section. The **TARGETPATH** `target_directory` line within a resource section indicates where in the device the compiled resource file (`.rsc` file) should be placed on the device.

TARGETPATH `resource\apps` should be used for the application resources.

TARGETPATH `\private\10003a3f\apps` should be used for the application registration resource files. The build will compile these resources once for every language that appears in the **LANG** statement. The **HEADER** keyword within a resource block is optional and tells the build tools to generate an include file so that the application code can reference the resources in the file. Resource files, including resource registration files and language translations, will be discussed further in Chapter 12.
- **SYSTEMINCLUDE** `include_path_1 include_path_2...` contains a list of paths that will be searched for system include files (e.g., `#include <stdlib.h>`).
- **USERINCLUDE** `include_path_1 include_path_2...` contains a list of paths that will be searched for non-system 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.
```

- **CAPABILITY** *cap_1 cap_2* ... specifies the platform security capabilities (i.e., units of trust) needed by the application. These capabilities are discussed in more detail in Chapter 7. **CAPABILITY** *None* means that the application requires no capabilities, and this is the default if the **CAPABILITY** line is not put in the MMP file.
- **SECUREID** *sid* indicates the application's Secure Identifier (SID), which is used to construct the name of the secure data-caged directory assigned to the application as `\private\<sid>`. This directory is private to the application and cannot be accessed by others unless they have sufficient permission to do so – for example, by possessing highly privileged capabilities (very few applications have this permission). If the **SECUREID** line is not in the MMP file, then SID defaults to your application's UID3 (this default is recommended). SIDs are part of the platform security architecture, which is covered in more detail in Chapter 7.
- **LIBRARY** *lib_1 lib_2* specifies what libraries to link to. These are the statically linked import libraries your application uses to call shared library (DLL) code at runtime. You always specify the library files as `.lib` files here, even though the actual library files may end in `.dso`, depending on your build target (see the next section for more information).
- **EPOCSTACKSIZE** *stack_size* changes the application's stack size from its default of 0x2000 for Symbian OS v9.1 (0x5000 in OS versions before v9.1) to the value specified in *stack_size*. This is an optional keyword and not used in the examples of this book because the stack size should be kept to a minimum and the default is sufficient for our use.

5.4 Build Targets

As mentioned in section 5.3.1, build targets specify the toolset and mode for a build and if the build is for the emulator or the device. Specifying the build target in the `abld` command causes the application to be built using it. Note that you may also need to put that build target in your `bld.inf` file under the **PRJ_PLATFORMS** keyword if that build target is not one of the defaults supported by the SDK (**WINSCW** and **GCCE** are typically the default supported platforms).

The build targets supported by Symbian OS v9 SDKs are listed in the table below:

Build Target	Type	Compiler
WINSCW	Emulator	Nokia x86 compiler
GCCE	Native	GCC
ARMV5	Native	RVCT
ARMV5_ABIV2	Native	RVCT (ABI v2 mode)
ARMV6	Native	RVCT

When you specify a build target 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.

5.4.1 Emulator Build Targets

On Symbian OS v9 SDKs, WINSCW is the usual build target used to compile your Symbian OS application for the Windows emulator. As mentioned before, WINSCW uses the Nokia x86 compiler (previously known as the Metrowerks compiler), which is the compiler used by Carbide.c++ as well as Carbide.vs.

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. 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.

The emulator build target support varies with the SDK.

5.4.2 Native Build Targets

Native build targets specify builds for the smartphone device, which means building for the ARM processor. For Symbian OS v9, GCCE is the build target you will most likely use. GCCE uses the GCC CSL ARM tool chain that is provided free of charge along with the Symbian OS SDK (starting at Symbian OS v9).

You can also build for the smartphone using the ARMV5, ARMV5_ABIV2, or ARMV6 build targets. These build targets use the ARM RealView compilation tools (RVCT) development tool chain. The ARM RVCT compiler produces code that is more optimized and thus runs faster than GCC. These build targets, however, are intended for device manufacturer and system-level (e.g., device drivers) development. The main reason is that, unlike GCC, the ARM RVCT compiler is not included in the Symbian OS SDKs, and must be purchased separately – for a fairly hefty price.

Except for some mentions in the next couple of sections, this book will not discuss the ARMV5, ARMV5_ABIV2, or ARMV6 build targets and we'll use GCCE exclusively for native builds.

Application binary interface (ABI)

ARM tools support a standard known as an application binary interface (ABI) to allow different ARM tools to interoperate together in a consistent manner (for smartphones, they are also known as EABI, for embedded ABI). At the time of writing, ARM supports two ABI versions: v1 and v2. Both of these ABIs are supported by Symbian OS, with the one used depending on the build target you use. The table below shows the native build target and the EABI supported:

Build Target	EABI Supported
GCCE	ABI v2
ARMV5	ABI v1
ARMV5_ABIV2	ABI v2
ARMV6	ABI v1

For the most part, the ABI differences are made transparent by the Symbian OS build tools, and thus I will not go into a lot of detail on them. However, it can be useful to understand ABIs when dealing with Symbian OS import library files, in order to understand the difference between .lib and .dso library files and when they are used. This is discussed in the next section.

Import libraries for native builds

While emulator builds put their import library files under their respective build target directories (e.g., epoc32\release\winscw\udeb for a WINSCW debug build), native builds store all import libraries under epoc32\release\armv5\lib regardless of the build target. Why is

this? The reason is that due to the ABI, any tool following the ABI that a library is formatted in can use that library file. Therefore, instead of possibly duplicating library files in multiple build target directories (for example, the same ABI v2 library file in both GCCE and ARMV5_ABIV2 build target directories), the SDK centralizes all native libraries in one place for build targets to use as needed. The file suffix of the library file identifies if the library is an ABI v1 or v2 library, as described next.

LIB and DSO files

If you look in `epoc32\armv5\lib` you will notice files with both `.lib` and `.dso` suffixes. `.lib` files are libraries in ABI v1 format, `.dso` files are libraries in ABI v2 format. For example, you will see an `euser.lib` and an `euser.dso` file in this directory, which are the ABI v1 and v2 versions of the system user import library, respectively.

When you link to a library (e.g., the library is specified in the `LIBRARY` line of your MMP file), the library file is picked up from `epoc32\armv5\lib` as a `.lib` or `.dso` file based on the build target you build with. Further, when you build your own library, it is stored as a `.lib` or `.dso` file in `epoc32\armv5\lib` depending on the build target's ABI.

As an example, when you build using GCCE, the build tools will use `.dso` library files, since GCCE uses ABI v2 (as shown in the table earlier). So if you build an executable that links to one or more libraries, the `.dso` library file(s) are linked in from `epoc32\release\armv5\lib` and the executable output is written to `\epoc32\release\gcce\urel` (assuming a release build). As another example, when you use the ARMV5 build target, which invokes the ARM RVCT compiler in ABI v1 mode, then `.lib` library files are used in `epoc32\armv5\lib` and the executable output is written to `\epoc32\release\armv5\urel` (again assuming a release build).

Note that in the MMP file, you specify the libraries as `.lib` in the `LIBRARY` line, no matter what build target you use. The Symbian OS build tools will know whether to use the `.lib` or the `.dso` library file automatically, based on what build target you use. For example, when you have the line `LIBRARY euser.lib` in your MMP file, the build knows to use `euser.dso` in the case of ABI v2 build targets like GCCE. However, with Carbide.c++, on the build properties sheet, you will need to make sure to specify the libraries with a `.dso` suffix if you are using an ABI v2 build target or a build error will result.

I would like to stress that the libraries under `epoc32\armv5\lib` are the import libraries that are statically linked to executables. Shared libraries (i.e., DLLs) are considered executables, and are put under the build target directory along with the EXE executable files.

ARM and THUMB instruction set

ARM supports both a 32-bit instruction set, known simply as ARM, and a 16-bit instruction set, known as THUMB. The ARM instruction set is for maximum performance, but with the sacrifice of some added code size, while the THUMB instruction set produces the most compact code, but at some sacrifice to performance.

In Symbian SDKs before Symbian OS v9, there were separate build targets to specify the ARM and THUMB instruction sets. However, with Symbian OS v9, this is no longer the case. Now, the policy is to compile all **user-side** code using the THUMB instruction set to maximize storage space. All **kernel-side** code is compiled with the ARM instruction set.

You can override the THUMB instruction set default for your applications by either placing the qualifier `BUILD_AS_ARM` after the MMP file name in your `bld.inf` file. Alternatively, you can add the `ALWAYS_BUILD_AS_ARM` keyword to your MMP file.

5.4.3 Pre-version 9 SDK Build Targets

WINSCW is also supported in pre-Symbian OS v9 SDKs; however, the native build targets listed in the table above are not. The native build targets for pre-version 9 Symbian OS are: ARMI, THUMB, and ARM4 (with ARMI and THUMB the one most likely used – ARM4 is meant for system-level code).

Pre-Symbian OS v9 SDKs have the WINS build target, which supports building for the emulator using the Microsoft build tools. Note that Carbide.vs requires the WINS build target for SDKs earlier than Symbian OS v9.

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 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, UID2 is `KUidApp` (0x100039CE) to indicate that the software is a GUI application. UID2 is also 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.

5.5.1 Getting a UID

How do you obtain a unique UID3 for your program? You can reserve a UID from the Symbian Signed site (<http://www.symbiansigned.com>). You need to log into the site (registering if needed), at which time you will have access to a UID request page. You can request multiple UIDs at a time (up to a limit specified on the page, which at the time of writing is 20 per day) and receive your assigned UIDs immediately.

From the Symbian Signed website, you can reserve Symbian OS v9 UIDs in either the protected or the unprotected range. UIDs in the protected range are for applications that will be Symbian Signed (Symbian signing is explained in Chapter 7), and your software can only be Symbian Signed if its UID is in the protected range. UIDs in the unprotected range are for applications that are released, but not Symbian Signed. You’ll get an error if you try to install a non-Symbian Signed application that has a UID in the protected range.

Why separate the ranges like this? The reason is that this ensures Symbian Signed applications always have a unique UID, since the value can be verified as belonging to the developer as part of the Symbian Signed process. Non-Symbian Signed applications should also have unique UIDs, but since they are not verified by a third party, nothing physically prevents someone from using UIDs not reserved to them.

During development, you can use UIDs in the test range (0xE0000000 to 0xEFFFFFFF) without needing to reserve them. You can 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 development UID.

You can also reserve a UID for code for Symbian OS versions before version 9 from the Symbian Signed website. In this case, you need to reserve a UID from the protected range.

Below is a table describing the UID ranges in more detail:

Protected range	0x00000000–0x0FFFFFFF	Development use
	0x10000000–0x1FFFFFFF	Legacy allocations
	0x20000000–0x2FFFFFFF	Symbian v9 protected allocation
	0x30000000–0x6FFFFFFF	Reserved
	0x70000000–0x7FFFFFFF	Vendor IDs

Unprotected range	0x80000000–0x9FFFFFFF	Reserved
	0xA0000000–0xAFFFFFFF	Symbian v9 unprotected allocation
	0xB0000000–0xDFFFFFFF	Reserved
	0xE0000000–0xEFFFFFFF	Development use
	0xF0000000–0xFFFFFFFF	Legacy UID compatibility range

5.5.2 Vendor ID

A vendor ID is a special kind of UID used to identify the creator of the software. They are reserved within the UID range of 0x70000000–7FFFFFFF and specified in the keyword `VENDORID` in the MMP file.

The vendor ID defaults to 0 (i.e., no vendor ID) if the `VENDORID` keyword is not specified in the MMP file. A vendor ID is not required and in fact typically is not used in application development, and you need not bother with them at all. They are intended for phone manufacturers and network operators.

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. You are also able to perform advanced debugging (e.g., single stepping, break points, variable examination) of your Symbian OS applications using your Windows development IDE on the emulator (which you can also do with a device using on-target debugging).

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 smartphones. This includes supporting different device screen sizes and all possible resolutions, 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 winscw 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 5.6.2. First, let's look at how the emulator is configured.

5.6.1 Emulator Configuration

The emulator is configured through a file called `epoc.ini`. This file is located in the `<SDK installation location>\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\winscw\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:\resource` for a WINSCW UDEB build target will contain the combined files of the `epoc32\release\winscw\udeb\z\resource` and `epoc32\data\z\resource` directories.

By default, the simulated C drive of the smartphone is mapped to the SDK's `epoc32\winscw\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 by adding 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. The S60 3rd Edition SDK, for example, sets this to 32MB, 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 hotkeys 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, particularly when testing code that requires platform security capabilities, as discussed in section 7.4.5.

Here is an example `epoc.ini`:

```
# S60 in emulator title bar.
WindowTitle S60
# 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 Symbian OS Library Tools and Utilities guide in the SDK documentation for more details of how to set these configurations.

5.6.2 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 x86 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.
Note that this differs from Windows Mobile, where the emulator actually emulates the ARM instruction set and therefore the same binary can be run on both emulator and device.
- **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.
- **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.
- **Writable static variables in shared libraries**
Writable static variables are allowed in the emulator, and also on the real device with Symbian OS v9 (although not on the device before v9). However, there is a limitation on using writable static data in shared libraries on the emulator. This limitation is that only one Symbian OS process can access a shared library DLL that has non-constant static data in it; if multiple processes use that same shared library, an error will occur. This is because of the single Windows process model described in the previous bullet, and a single Windows process cannot contain two copies of the same writable data.
- **On the emulator the stack size is not restricted, thus some executables that run fine on the emulator may cause panics on a device.**

5.6.3 Debug Logging

You can print debug output from your program to a file in the emulator using `RDebug::Print()`. You need to include `e32debug.h` to access this. For example:

```
#include <e32debug.h>
...
TInt ret = Somefunc();
_LIT(KDebugmsg, "Somefunc() called, returning %d");
RDebug::Print(KDebugmsg, ret);
```

The output is written to the file `epocwind.out` in a Windows temp directory, typically `c:\Documents and Settings\<user name>\Local Settings\temp`. To enable RDebug logging to this file, you need the following line in `epoc.ini`.

```
LogToFile 1
```

You can also have your RDebug output show up on your IDE console. To do this, place the following in `epoc.ini`:

```
LogToDebugger 1
```

Note that in release builds, RDebug is not automatically disabled. For maximum performance it's best not to call RDebug in release builds, so you should either surround the calls by an `#ifdef _DEBUG` statement, or create a debug print macro that is defined as RDebug when `_DEBUG` is defined and is empty otherwise.

5.7 Building Shared Libraries

In this section, I show how to build shared library DLLs and describe some of 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 Shared Library

To build a shared library, set `targettype` to `dll` in your DLL's MMP file. Then, 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. The following shows a sample shared library MMP file from section 4.6.1:

```
//MMP
TARGET          mydll.dll
TARGETTYPE      dll
UID             0x1000008d 0x10004264
SOURCEPATH      ..\src
SOURCE          mydll.cpp
USERINCLUDE     .
USERINCLUDE     ..\include
SYSTEMINCLUDE   \Epo32\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 shown in the example above) 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 or DSO file, depending on the build target, see

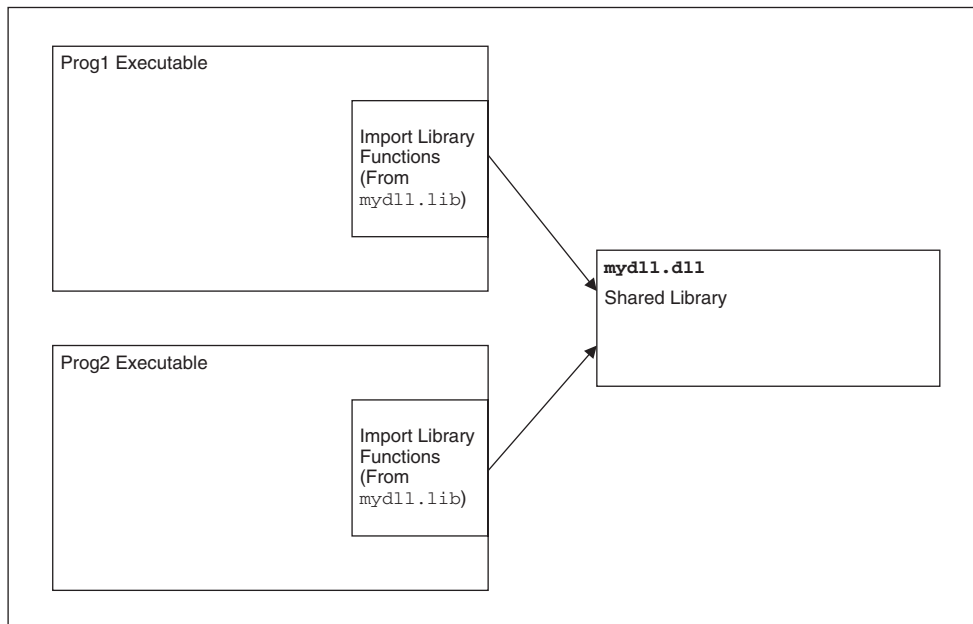


Figure 5.2 Import and DLL Libraries

previously). Both outputs are placed in the build platform's release directory (e.g., `epoc32\release\gccce\urel`). The emulator can load and use the DLL directly from that directory. On the target phone, DLLs should be placed in the `\sys\bin` 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 shared library.

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, 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 although I specified the import library with a suffix of `.lib` in the figure, the library could have a `.dso` depending on the build target, as discussed in section 5.4.2.

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 wrong function call could cause a software crash or other error that could be very hard to debug. 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 dynamically 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.

DLL freezing is only applicable if you are developing a shared library DLL, so if you do not plan on doing this you can skip this section. Also, as with the rest of this chapter, this section focuses on using the command line tools. However, I want to point out that with Carbide.c++, interface freezing is performed by simply selecting the 'Freeze exports' option in the project menu. Even if you are developing your DLL with Carbide.c++ though, this section is useful to understand what freezing is and why it's needed, although you could skip the command line explanation.

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 `EXPORT-UNFROZEN` to your DLL's MMP file to do this. Building in this mode is straightforward – you run `blmake 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 generated directly 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 `EABI` (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. The keyword `nostrictdef` can also be used in 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 winscw
abld freeze winscw
abld build winscw
```

Doesn't it seem strange to run `abld build winscw` 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 winscw` in place of the last `abld build winscw` command. This command builds the import library from the DEF file just created by the `abld freeze winscw` command. The `abld build` command 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

To illustrate the concept of library freezing a bit further, let's look at a quick example of a DLL and the .DEF file it generates when the library is frozen. I used the GCCCE native build target for this example. The following shows the source code for a sample DLL:

```
// mydll.h
#include <e32base.h>
class CMyClass : public CBase
{
public:
    static CMyClass *NewL();
    IMPORT_C void FuncA();
};
```

```

    IMPORT_C void FuncB();
    IMPORT_C void FuncC();
    IMPORT_C void FuncD();
    IMPORT_C void FuncE();
protected:
    IMPORT_C CMyClass(void);
};

// mydll.cpp
#include "mydll.h"

EXPORT_C CMyClass *CMyClass::NewL()
{
    return new(ELeave) CMyClass;
}

CMyClass::CMyClass()
{ // constructor code here }

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 }

```

The listing below shows a sample DEF file generated (by `abld freeze gcce`) for this sample DLL:

```

EXPORTS
    _ZN8CMyClass4NewLEv @ 1 NONAME
    _ZN8CMyClass5FuncAEv @ 2 NONAME
    _ZN8CMyClass5FuncBEv @ 3 NONAME
    _ZN8CMyClass5FuncCEv @ 4 NONAME
    _ZN8CMyClass5FuncDEv @ 5 NONAME
    _ZN8CMyClass5FuncEEv @ 6 NONAME

```

The static `CMyClass::NewL()` method is assigned ordinal 1, and functions `FuncA()` through `FuncE()` are assigned ordinals 2 through 6 (you can ignore the characters surrounding the function name).

Inserting a new function

In the example just discussed, suppose you insert a new class method, say `FuncC1()`, after `FuncC()`. If freezing were disabled, the ordinals could change in the next build (thus breaking compatibility for applications

using the older library); however, since the DLL is frozen, you can be assured that the ordinals for the existing functions will remain the same and `FuncC1()` will receive the next-higher ordinal (7). 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 removed one of the methods (`FuncA()`, for example). The next time you built the DLL, you'd see an error such as the following (again, using GCCSE, the error may differ with other build targets):

```
elf2e32 : Error: E1036: Symbol _ZN8CMyClass7FuncAEv Missing from ELF File
```

The reason is that backward compatibility would be broken, since existing applications may depend on the function you deleted. To delete this method you would need to unfreeze the DLL, and rebuild – and then freezing back again to create a new import library to correspond with the new version of the DLL.

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` or `.sisx` extension. 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.
- 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, WAP, or via email onto the smartphone itself and install them.
- Beam the SIS file to the phone using infrared or Bluetooth technology.

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`. The `makesis` tool takes a package definition file (known as a PKG file.) as input. This is simply a text file that specifies what files are included in the package (and their location on the PC) and where these files go on the smartphone. 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.

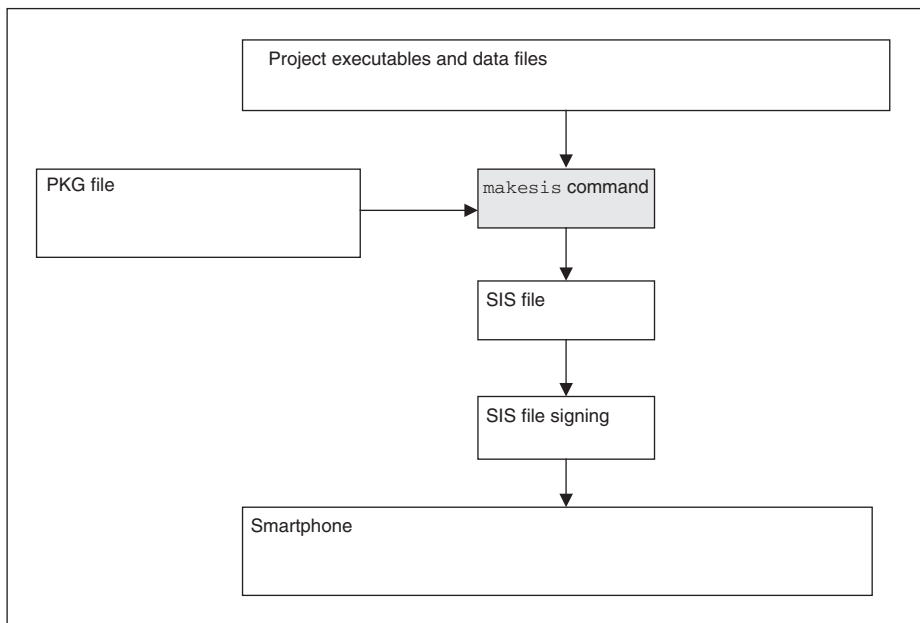


Figure 5.3 `makesis` Flow

- `\sys\bin`
This is where all executables on the phone reside (i.e., all EXE files and shared libraries). An executable cannot be started if it is not in this directory.
Note that before Symbian OS v9, each application had its own application subdirectory; this is no longer the case. Therefore, you need to be sure your executable names are unique across all applications. For example, if you name an executable `application.exe` you may have a conflict with some other application that has an executable with that name. One way to ensure uniqueness is to append the UID3 of your executable to its name. Carbide.c++ does this automatically when creating applications using its wizard.
- `\resource\apps`
This is where all compiled application resource files are put.
- `\private`
The `\private` directory contains reserved, secure subdirectories for storing data for applications. The subdirectory name is the `SECUREID` specified for the application (which defaults to the application UID), as described earlier in section 5.3.2. See also section 7.4.8 for a full description of this.
- `\private\10003a3f\import\apps`
This is a special directory under `private`, where application registration resources reside. We'll discuss it later, in Chapter 12.

5.9.2 Format of the PKG File

To create a SIS file you need to define a PKG file for your project, which is used to specify the contents of your SIS file. First I will discuss the basic statements in the package 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 smartphone when the SIS file is installed. The following shows the PKG file from Chapter 2:

```
; SimpleEx.pkg - S60
;
;Language - standard language definitions
&EN
; standard SIS file header
#{ "SimpleEx", (0xE000027F), 1, 0, 0
```



```
;Localised Vendor name
%{"Vendor"}
;Supports S60 v 3.0
[0x101F7961], 0, 0, 0, {"S60ProductID"}
;
;Files to install
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\release\gcce\urel\SimpleEx.exe"
-"\sys\bin\SimpleEx.exe"
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\data\z\resource\apps\SimpleEx.rsc"
-"\resource\apps\SimpleEx.rsc"
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\data\z\private\10003a3f\apps
\SimpleEx_reg.rsc" -"\private\10003a3f\import\apps\SimpleEx_reg.rsc"
```

Package file comments

Lines that begin with a semicolon 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 the previous example, the package header is:

```
#{"SimpleEx"},(0xE000027F),1,0,0
```

This indicates that the program's name is `SimpleEx`, the UID is `0xE000027F`, 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.

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, and this default is likely what you will typically use.

Reference the SDK documentation for the complete list of package types (they differ between Symbian OS pre-v9 and v9).

Product target

The line after the package header indicates the target platform for the installation. Consider the product target line from the example above:

```
[0x101F7961], 0, 0, 0, {"S60ProductID"}
```

The first number is the platform's UID, the last string is the name of the platform. This string will work on all S60 platforms.

Here are some platform UIDs, along with product names:

S60 2nd Edition	0x101F7960	S60ProductID
S60 3rd Edition	0x101F7961	S60ProductID
UIQ v2.0	0x101F617B	UIQ20ProductID
UIQ v2.1	0x101F61CE	UIQ21ProductID
UIQ 3	0x101F6300	UIQ30ProductID

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 on the smartphone. For example, in the previous example, the line:

```
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\release\gcce\urel\SimpleEx.exe"
-"!:\sys\bin\SimpleEx.exe"
```

copies, into the SIS file, the file named `simpleEx.exe` from the `c:\Symbian\9.2\S60_3rd_FP1\epoc32\release\gcce\urel` directory on the PC. Then, on installation, this file is copied to the smartphone, to `\sys\bin`. In this example, the destination file has the same name as the source file, 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.exe` is placed in `c:\sys\bin\`. 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. However, this is not recommended because it means that the user will not be given a choice of drive to use.

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\9.2\S60_3rd_FP1
```

a file specified as `'\epoc32\release\gcce\urel\simpleEx.exe'` will be interpreted as being relative to the S60 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. However, the runtime files will be deleted on installation if you place them in the application's private directory, which is in `\private\<application SID>` (described in section 5.3.2). This is because the application's private directory, along with its contents, is deleted upon uninstalling the application.

In Symbian OS versions before v9, you could specify that you want such files to be removed by using the FILENULL specification as follows:

```
"-"C\system\data\my_runtime_generated_file",FILENULL
```

However, FILENULL has been deprecated in Symbian OS v9, although it continues to be supported to ease PKG file migrations from previous Symbian OS versions. In the case of Symbian OS v9, FILENULL is not needed if you always store your data in the application's secure directory under `\private\<SID>`.

Note that runtime-created data files are only removed 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

Depending on how the device manufacturer has configured the phone's installation policy, 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:

```
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\release\gcce\urel\myprogram.exe"-  
"!:\sys\bin\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,{ "MyD11" }
```

indicates that a component named `MyD11`, 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:

```
[0x101F7961], 0, 0, 0, { "S60ProductID" }
```

is a requisite statement that the ‘component’ named `S60ProductID`, with a UID of `0x101F7961`, and version number 0.0.0 or higher, must exist in order for the installation to continue. Note that in the case of hardware dependency, like the case just mentioned, brackets are used to enclose the UID instead of parentheses.

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. Below is a list of some common languages. Reference the Symbian OS library in the SDK documentation for the complete 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 our sample PKG file, which 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\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.en",
 "c:\Symbian\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.fr",
 " c:\Symbian\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.fi",
}
- "!resource\apps\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 `\resource\apps\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.

The following shows the PKG file shown previously with modification to support multiple languages.

```
; SimpleEx.pkg - S60
;
&EN,FR,FI
; standard SIS file header
#{"SimpleEx_EN", "SimpleEx_FR", "SimpleEx_FI"}, {0x10005B94}, 1, 0, 0
; Supports S60
```



```
[0x101F7961], 0, 0, 0, {"S60ProductID", "S60ProductID", "S60ProductID"}
"c:\Symbian\9.2\S60_3rd_FP1\epoc32\release\gcce\urel\SimpleEx.exe"
- "!:\sys\bin\SimpleEx.exe"
"c:\Symbian\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.rsc"
- "!:\resource\apps\SimpleEx.rsc"
{"c:\Symbian\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.en",
"c:\Symbian\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.fr",
" c:\Symbian\9.2\S60_3rd_FP1 \epoc32\data\z\resource\apps\SimpleEx.fi",
} - "!:\resource\apps\SimpleEx.rsc"
```

5.9.5 Signing Your SIS File

As mentioned in Chapter 2, in Symbian OS v9 you must have your SIS file digitally signed before it can be installed on the phone. You can sign the application yourself using the `signsis` tool if either your application has no security capabilities (i.e., no `CAPABILITY` line or `CAPABILITY None` in your MMP file or Carbide project), if you have a developer certificate or if your application contains only user-level capabilities. Application signing is discussed in detail in Chapter 7.

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 `euser.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 and run the examples in this chapter on the emulator 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. For simplicity, the example does not TRAP the call to the `Console::NewL()` leaving function, but a later example shows how this should be done correctly, to ensure that `E32Main()` doesn’t contain functions that can leave.

```
#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.

The following example shows an expanded version of a console program that provides a general framework for running experiments, including the examples in this chapter. It also correctly creates a cleanup stack and handles error leaves correctly.

```
#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");
    console=Console::NewL(KName, TSize(KConsFullScreen, KConsFullScreen));
    CleanupStack::PushL(console);
```

```

RunExampleL();
_LIT(KAnyKey, "[Press any key]\n");
console->Printf(KAnyKey);
console->Getch();
CleanupStack::PopAndDestroy(console);
}
TInt E32Main()
{
    __UHEAP_MARK;
    CTrapCleanup* cleanupStack = CTrapCleanup::New();
    TRAPD(error, RunConsoleL());
    __ASSERT_ALWAYS((KErrNone!=error), User::Panic(_L("Example"), error));
    delete cleanupStack;
    __UHEAP_MARKEND;
    return(0);
}

```

The code in `E32Main()` creates a cleanup stack for your test code to use if needed. It also traps leaves that occur in your test code. You must always use the cleanup stack from within a TRAP, which is why the separate function, `RunConsoleL()`, has been defined. If you don't have a TRAP and attempt to use the cleanup stack, your code will panic with `E32USER-CBASE 69`.

`__UHEAP_MARK` and `__UHEAP_MARKEND` are useful macros that detect memory leaks on the heap. When `__UHEAP_MARK` is called, the heap level (i.e., the amount of heap used at the time) 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), 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 for catching memory leaks.

The MMP file for the console project is shown below.

```

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 winscw udeb

```

Note that the MMP file defined here is only suitable to run the console program on the emulator, which is all that's needed for the examples in this chapter. You can, however, also run a console program on the device, but you would need to define a `_reg.rss` resource file and specify it in a resource section of your MMP (resource sections are discussed in section 5.3.2, registration resources are discussed in Chapter 12).

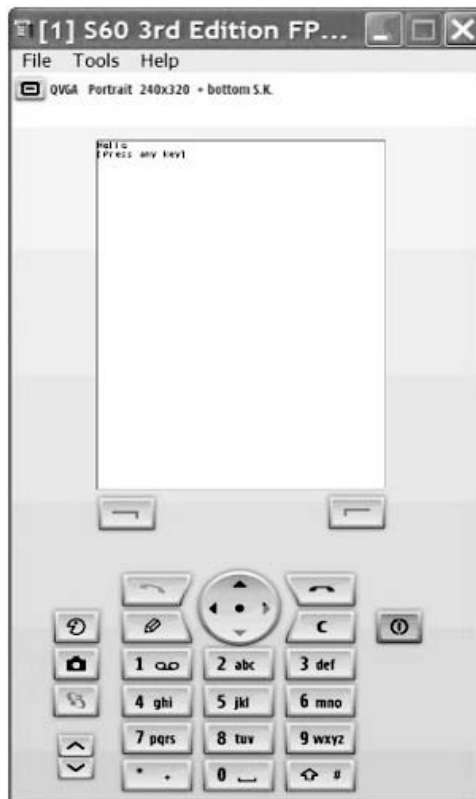


Figure 6.1 Console Output on the S60 3rd Edition FP1 Emulator

The `abld build` command shown builds a windows executable called `tests.exe` in the `<SDK Installation Directory>\epoc32\release\wincw\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` built with the code and MMP file shown above. 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 with other operating systems, they can be a source of some initial 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. Descriptor classes also 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 prevents data from being written outside of the allocated buffer. When an access is attempted beyond the buffer limit, the descriptor raises a panic 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 panics in your production code.

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. Note that the goal of this example is just to introduce a few C string and descriptor differences and not to show the best way to implement the actual functionality of the example.

First, let's look at the example, written using C strings:

```
char *namePrefix="Name:";
void MakeName(char *aName)
{
    char str[80];
    strcpy(str,namePrefix);
    strcat(str,aName);
    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 an array of `char`'s 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`.

Notice also 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.

Note that it would have been easier in the C example to create the string in `MakeName()` by using `sprintf(str, "%s %s", namePrefix, name)` instead of doing a string copy on the prefix and concatenating the name (the goal was just to show those string functions). In descriptors there is an equivalent function to `sprintf()` called `Format()`, and in `MakeName()` the line would be: `str.Format(_L("%S %S"), KNamePrefix, aName)`. The `Format()` descriptor method is covered in more detail in section 6.4.2.

6.3 The Descriptor Classes

In this section, we look closely at all of the descriptor classes.

There are 12 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` or `RBuf`.

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.

Furthermore, 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 <SDK Installation Directory>\epoc\include directory.

- TBuf8<n>: modifiable, 8-bit buffer, *n* is the buffer size.
- TBuf16<n>: modifiable, 16-bit buffer, *n* is the buffer size.
- TBufC8<n>: non-modifiable, 8-bit buffer, *n* is the buffer size.
- TBufC16<n>: non-modifiable, 16-bit buffer, *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.
- RBuf8: modifiable, 8-bit heap descriptor.
- RBuf16: 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 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.

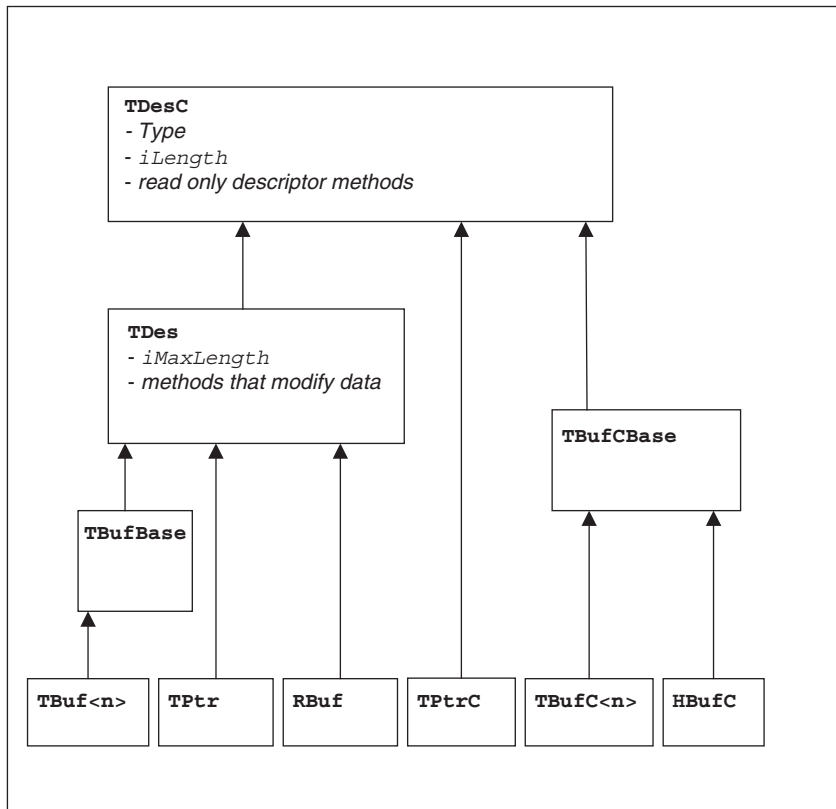


Figure 6.2 Descriptor Class Diagram

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 mentioned previously, while `TDesC` can represent all descriptors, `TDes` can only represent modifiable descriptors (e.g., `TBuf`). Furthermore, `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& aString)
{
    aString.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 on to describe 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 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.

Figure 6.3 below compares how literals created with `_LIT` and `_L` are stored in memory.

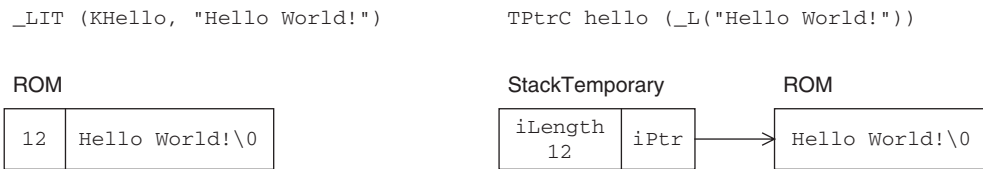


Figure 6.3 Memory Layout for Literal Descriptors

6.3.3 Buffer Descriptors

TBuf and TBufC are buffer descriptors, that is, 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. Reference section 6.7 for more details on templates and how Symbian OS uses them.

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> buf` 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) descriptor methods available to it. TBufC, however, inherits only from TDesC, and therefore has only the read-only TDesC methods available to it.

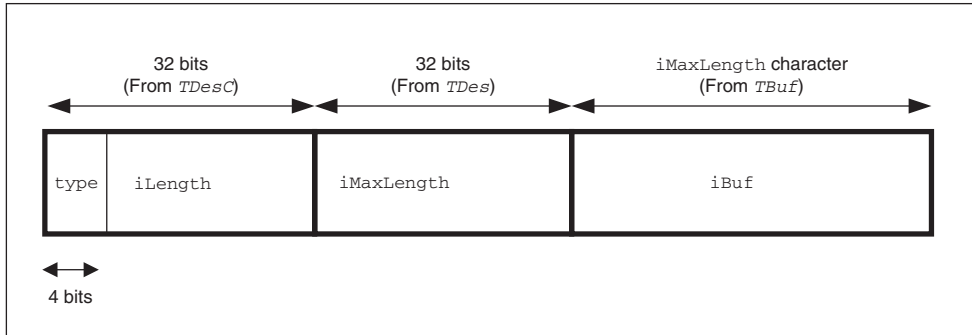


Figure 6.4 TBuf Memory Layout

Figure 6.4 shows how a TBuf descriptor appears in memory.

The fields labeled `type` and `iLength` 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.

The length of the data currently in the data buffer is indicated by a 28-bit value, shown in Figure 6.4 as `iLength`. This is the value returned by the `Length()` method (in units of data width) and `Size()` method (in bytes). The actual size of the allocated buffer is stored in `iMaxLength`, which is a 32-bit value that comes from class `TDes`. It is used to prevent the buffer from being accessed beyond the buffer's boundary. The allocated data buffer array is shown by `iBuf` – 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`. The descriptor methods in `TDes` and `TDesc` use the `Ptr()` method, which contains a `switch` statement on the `type` value to perform the locate to the descriptor data area 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 6.5.

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

allocated buffer size of 10. The buffer data is shown as a row of Xs, which indicate uninitialized memory.

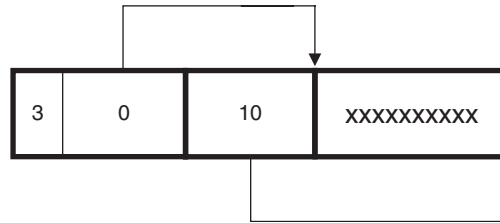


Figure 6.5 Initial State of TBuf<10>

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 6.6.

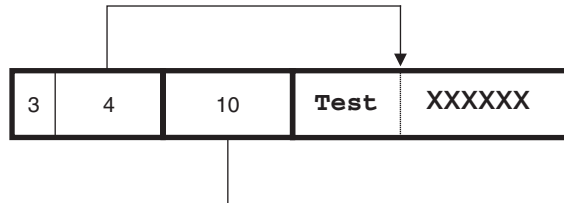


Figure 6.6 Copying Data to TBuf

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 6.7.

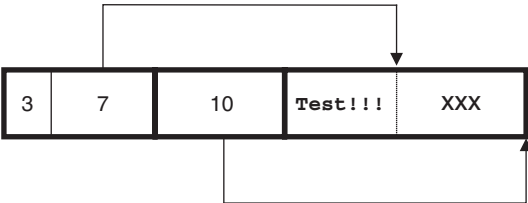


Figure 6.7 Appending to TBuf

If you then add the following:

```
_LIT(KString2, "1234");
buf.Append(KString2);
```

what happens? A panic 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 assignment through the = operator.

Figure 6.8 shows how TBufC is stored in memory.

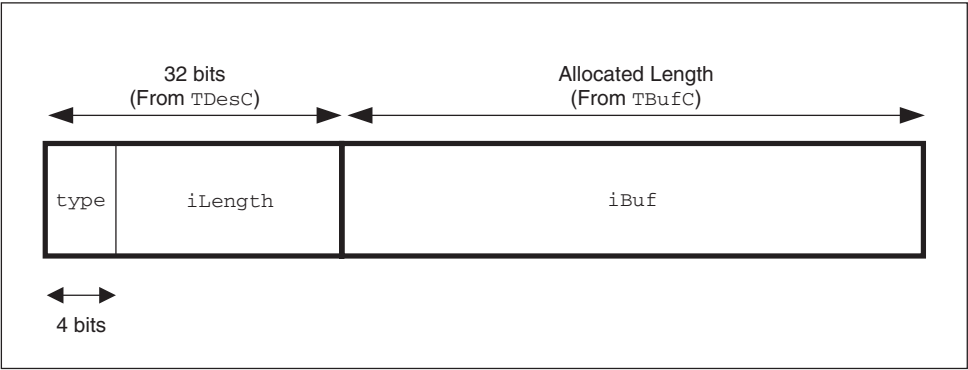


Figure 6.8 TBufC Memory Layout

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 6.9.

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;
```

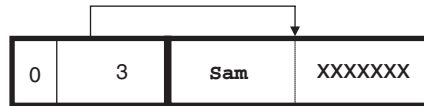


Figure 6.9 TBufC containing "Sam"

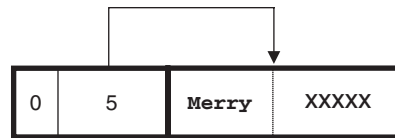


Figure 6.10 TBufC containing "Merry"

The descriptor memory will then appear as in Figure 6.10.

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 cause a panic, 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 6.11.

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 6.12 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). Like

TBufC, the buffer data also cannot be modified via this descriptor except by direct replacement of the data.

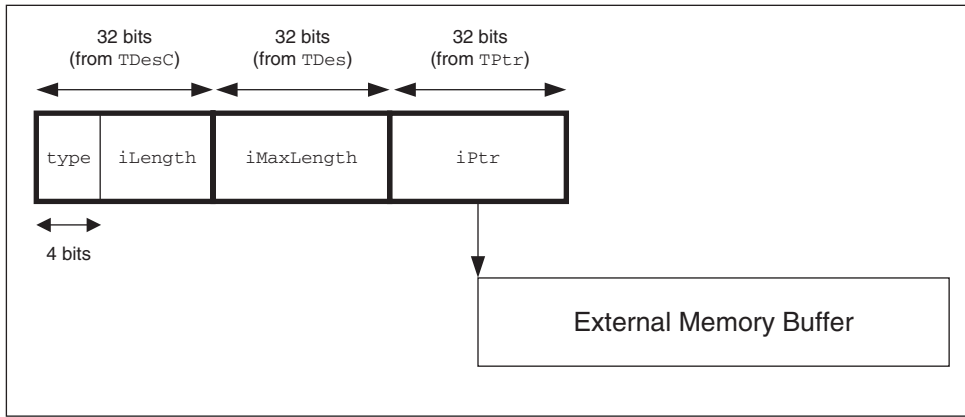


Figure 6.11 TPtr Memory Layout

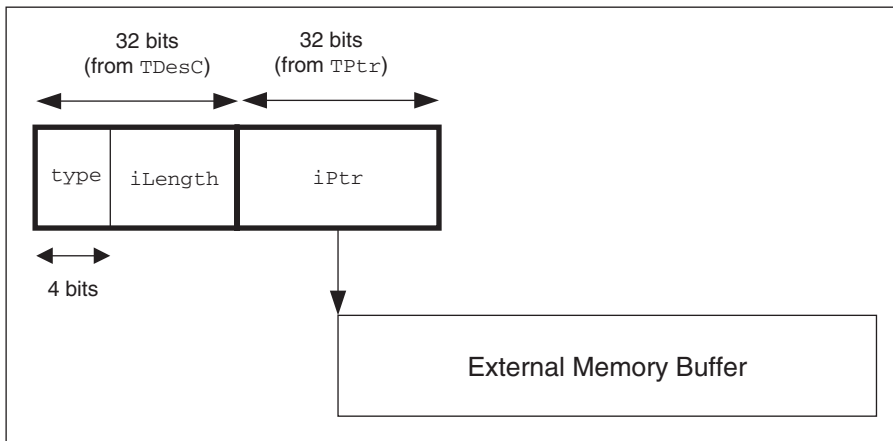


Figure 6.12 TPtrC Memory Layout

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 descriptor is constructed. For example:

```
_LIT(KString1, "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 6.13.

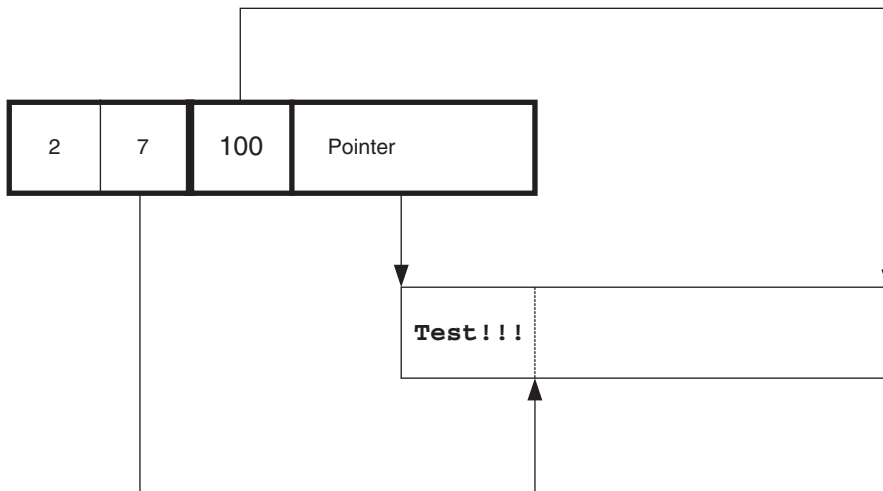


Figure 6.13 `TPtr` After Append

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 within the `TPtr` constructor. For example:

```
TInt buff = new malloc(100*sizeof(TInt));
```

```
for (Tint i=0;i<20;i++) *buff+=i; // write some data in
                        // 20 bytes of allocated buffer
TPtr myDes(buff,20,100);
...
```

This creates a `TPtr` descriptor that points to the allocated buffer pointed to by `buff`, sets the descriptor length to 20 and the maximum length of the descriptor to 100.

Or you can construct the `TPtr/TPtrC`, and then afterwards set the buffer with `Set(buff,length,max_length)` (only in the case of `TPtr`) or `Set(buff,max_length)` (where `length` would default to zero).

There are other ways of pointing `TPtr` and `TPtrC` to memory regions. You can find more in the SDK documentation about the various overloaded constructors and `Set()` methods.

6.3.5 Heap Descriptors

This section describes the two heap descriptors, `HBufC` and `RBuf`. Both of these descriptors allocate their data buffers on the heap, and they both allow their buffers to be resized when needed (although this resizing must be done explicitly by you, the programmer; it's not automatic). Unlike the other types of descriptors, you must also take responsibility for freeing the memory used by heap descriptors when you are finished using them. For `HBufC` this entails deleting the pointer you assigned the `HBufC` object to (preferably by pushing the pointer on the cleanup stack and calling `CleanupStack::PopAndDestroy()`). In the case of `RBuf`, you call the `Close()` method to free its memory (or preferably, use the `RBuf CleanupClosePushL()` method and then `CleanupStack::PopAndDestroy()` when finished).

Note that `RBuf` was introduced in Symbian OS v8, so it is relatively new. Before then, `HBufC` was used exclusively for heap-based buffers.

In general, you should use `RBuf` for modifiable data and `HBufC` for non-modifiable data. However, you will see that there is also a way to modify `HBufC` through a pointer descriptor. Let's start our discussion of heap descriptors with `HBufC`.

HBufC descriptor

`HBufC` is a non-modifiable descriptor class that is allocated entirely on the heap and referenced as a pointer. `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 characters (200 bytes).

`NewL()` and `NewLC()` methods are also available so you don't have to check the validity of the returned pointer because they leave on error. `NewLC()` pushes a created pointer onto the cleanup stack.

In memory, an `HBuFC` looks just like a `TBuFC` (see Figure 6.14).

The `type` field for `HBuFC` is set to 0 as for `TBuFC`, since it appears like `TBuFC` in memory.

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 aMaxLength)` creates a new descriptor on the heap, of size `aMaxLength`, copies the descriptor data to it and deletes the old one. The `ReAlloc()` function returns a new pointer to the heap descriptor to reflect the new allocation. `ReAlloc()` returns `NULL` if an error has occurred. A `ReAllocL()` function also exists, which is the same as `ReAlloc()` except the function leaves on errors.

Modifying `HBuFC` data

You can modify the data in a `HBuFC` buffer by using the `Des()` method. `Des()` returns a `TPtr` whose buffer pointer is initialized to point to the 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.

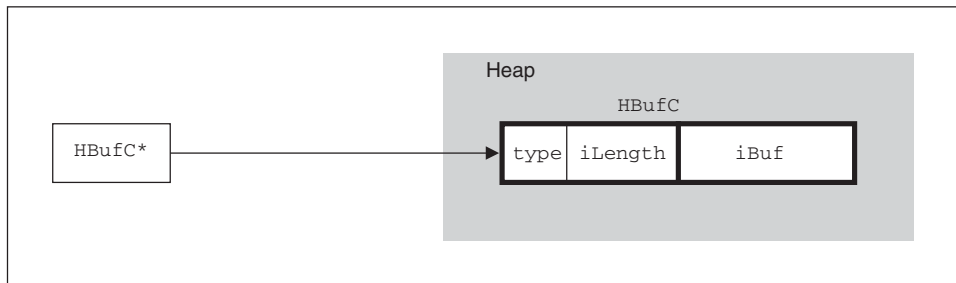


Figure 6.14 `HBuFC` Memory Layout

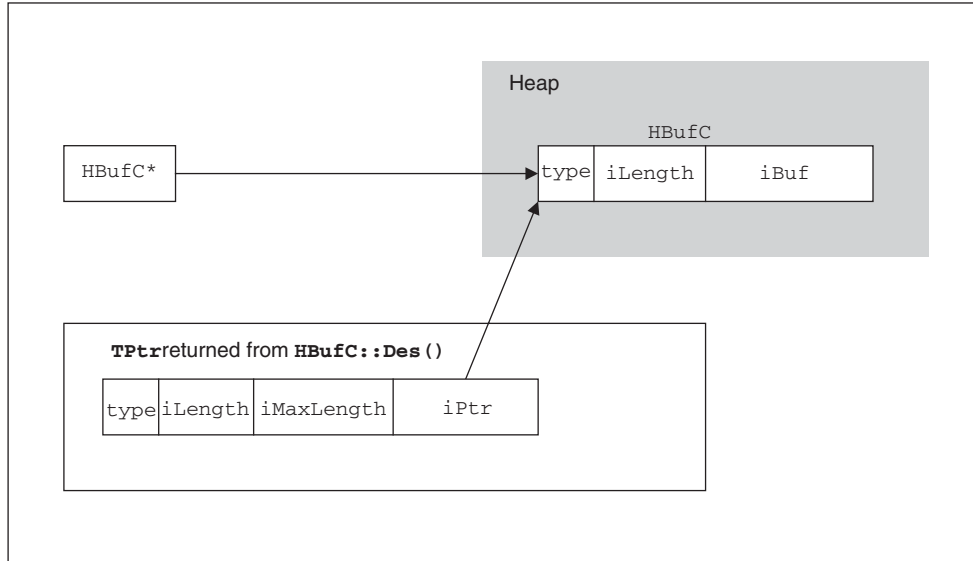


Figure 6.15 HBufC's Des() Method

See Figure 6.15 for how the TPtr returned by Des() is related to HBufC. The maximum 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 below.

```
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 = myHeapDes->ReAllocL(KString2().Length());
    //if you do not do this, the next line executed 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 through 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 from the example 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 a `HBufC` with a buffer big enough to fit the string "Test". It is then assigned that string. The buffer is then reallocated using the `ReAllocL()` method and set to a bigger string. The return value of `ReAllocL()`, which is the pointer to the newly allocated descriptor, is written to `myHeapDes`. 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 `HBufC` 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();
```

Given sufficient available memory, 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.

RBuf descriptor

Like `HBufC`, an `RBuf`'s data buffer resides on the heap; however, instead of calling a 'new' method to create the descriptor, you instantiate `RBuf` on the stack as an automatic variable. Internally, `RBuf` will allocate buffer memory on the heap (using `RBuf Create()`), or it may take ownership of preallocated heap memory, including a `HBufC` or another `RBuf` (using the `RBuf Assign()` method).

Like other R classes, `RBuf` has a `Close()` method. `RBuf`'s `Close()` method will free the data buffer on the heap owned by the descriptor. Always remember to call `Close()` when you are finished with the `RBuf` descriptor data, or a memory leak will occur.

Below is a simple example of using `RBuf`:

```
RBuf myBuff;
myBuff.CreateL(100) ;
myBuff.Copy(_L("My RBuf data"));
...
myBuff.Close();
```

The `CreateL()` method allocates a buffer on the heap (a buffer of length 100 in this case) for your data. A leave will occur if there is a problem creating the data buffer. Alternatively, you can call the `Create()` method, where an error would be indicated in the function return code instead of as a leave.

`RBuf` is derived from `TDes`, and thus acts like any other descriptor, allowing you to pass it to functions that take `TDes` descriptors as input, and for you to access all read and write descriptor methods (i.e., `TDesC` and `TDes` methods). In this example, I copy the string "My RBuf data" into the descriptor using the `Copy()` method.

Using RBuf::CleanupClosePushL()

The `myBuff.Close()` at the end of the previous example frees the allocated buffer belonging to the `myBuff` descriptor. However, what if a leave occurs after the buffer is allocated and before the `Close()` is called? In that case the memory is not freed, which is not desirable. A better and safer way to invoke `RBuf::Close()` is by using `RBuf`'s `CleanupClosePushL()`, which takes advantage of the cleanup stack to provide leave-safe code, like below:

```
RBuf myBuff;
myBuff.CleanupClosePushL();
// more code here, can leave on errors
// finished using myBuff
CleanupStack::PopAndDestroy(myBuff);
```

`CleanupClosePushL()` pushes the `RBuf` object as a cleanup item on the cleanup stack. After calling this function, a `leave`, or a call to `CleanupStack::PopAndDestroy()` – whichever occurs first – will invoke `Close()` automatically.

Creating RBuf from another descriptor

You can have your `RBuf` descriptor allocate memory and initialize it with a copy of the contents of another descriptor using variations of the `CreateL()`/`Create()` methods. Below is an example:

```
TBuf<100> myTbuf(_L("TBuf stuff"));
RBuf myRbuf1;
myRbuf1.CreateL(myTbuf);
myRbuf1.CleanupClosePushL();
RBuf myRbuf2;
myRbuf2.CreateL(myTbuf,500); //copies myTbuff, but allocates buffer of
                             //length 500 instead of myTbuf length
myRbuf2.CleanupClosePushL();
// more code here, can leave on errors
CleanupStack::PopAndDestroy(2,myRbuf1);
```

The prototype of the `CreateL()` functions used above are: `CreateL(TDesC& aDestoCopy)` and `CreateL(TDesC& aDesToCopy, TInt aMaxLength)`.

Taking ownership of preallocated buffers

Instead of creating its own buffer, an `RBuf` descriptor can take ownership of a buffer already allocated on the heap. `RBuf`'s `Assign()` method accomplishes this. What exactly does taking ownership of a buffer mean? It means that the `RBuf` object uses the assigned preallocated buffer as its own and is now responsible for freeing it when it is finished (i.e., when its `Close()` method is called).

Here is an example:

```
TInt bufferSize=100;
TUint16 buff=
static_cast<TUint16*>(User::AllocL(bufferSize*sizeof(TUint16)));
RBuf myRbuf;
myRbuf.CleanupClosePushL();
myRbuf.Assign(buff,bufferSize);
// ... more code here, can leave on errors
CleanupStack::PopAndDestroy(myRbuf);
```

In this example, the `RBuf` object `myRbuf` takes ownership of the raw allocated buffer, `buff`. `CleanupStack::PopAndDestroy()` (or a

leave if one occurs) invokes the `myRbuff Close()` method to delete the buffer. You do not need to call `delete` on the `buff` pointer.

You can also take ownership of a `HBufC` descriptor or another `RBuf` descriptor using the `RBuf::Assign(HBufC16 *aHbuf)` and `RBuf::Assign(const RBuf16 aRBuf)`, respectively. In addition, you can take ownership of a `HBufC` call in the `RBuf` constructor like the following:

```
HBufC* myHBufC = HBufC::NewL(100);
RBuf myRBuf(myHBufC);
```

Multiple RBuf assignments

You can assign buffers multiple times in an `RBuf`, but you need to make sure you free the previous one first via `Close()` before making a new assignment or you will have an orphaned allocation (and thus memory leak). For example:

```
TBuf<100> myTbuf(_L("TBuf stuff"));
RBuf myRbuf1;
myRbuf1.CreateL(myTbuf);
myRbuff.CleanupClosePushL();
// ... more code here, can leave on errors
HBufC* HBufC = HBufC::NewL(100);
CleanupStack::PopAndDestroy(myRbuf1); // free previous buffer
myRbuf1.Assign(HBufC);
myRbuf1.CleanupClosePushL();
// more code here, can leave on errors
CleanupStack::PopAndDestroy(myRbuf1); // free the HBufC
```

Resizing buffers

Like `HBufC`, `RBuf` provides a `ReAlloc()` method to change its buffer size (e.g., expanding the buffer when you are running out of room, or shrinking it to just the space needed). Unlike `HBufC`, however, where you would need to update the descriptor's pointer on `ReAlloc()` since the heap address of `HBufC` could have changed, the `RBuf ReAlloc()` works transparently since the heap pointer is stored internal to the class.

Like `HBufC`, a `ReAllocL()` version of the `ReAlloc()` also exists for `RBuf` that performs a leave if an error occurs.

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.

The following shows an example of using the compare functions:

```
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 from the code 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 substrings within a descriptor

To locate a substring within a descriptor, you can use `Find()`. `Find()` looks for the first occurrence of a substring within a descriptor and returns its start position, if it is found.

For example:

```
des1.Find(KSomeSubString);
```

returns `KErrNotFound (-1)` if the substring `KSomeSubString` is not found in `des1`, or the starting position of the substring in `des1` if the substring is found.

`FindF()` and `FindC()` are the same as `Find()` except that they use the tolerant fold and collation comparisons (respectively) to search for the string.

Here is an example of using the `Find()` functions:

```

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 the code given 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 substrings within descriptors, you can use `Match()` instead of `Find()`. `Match()` behaves like `Find()` except that you can supply wildcard characters when searching for a string match. `'*'` represents a sequence of any characters; `'?'` represents an occurrence of any single character. For 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 the previous code is as follows:

```

Match: string 2 and string 1 res = 0
Match: string 3 and string 1 res = 5
Match: string 4 and string 1 res = 0
Match: string 5 and string 1 res = -1

```

Extracting substrings 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 substring within the descriptor the methods are invoked on.

`Left()` defines a substring starting at the beginning of the descriptor and of a specified length. `Right()` defines a substring that starts a specified length before the end of the descriptor. `Mid()` specifies a substring that starts from a specified position and is of a specified length.

The following shows an example of using all three substring extraction functions:

```

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\" ");
    console->Printf(KFormat3, &SubStr);
}

```

The output 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 that 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).

Here is an example of the copy functions:

```
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 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. For 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 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 uppercase letters being appended for hexadecimal numbers.

The following code shows the appending functions in action:

```
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 from the code 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.

The following example shows the format functions in action:

```
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 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.

Here is an example of the case-switching functions:

```
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 the previous example 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. The deletion functions are illustrated as follows:

```
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 the previous code 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 it's 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()`. For example:

```

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 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. Here is an example using a binary buffer (of course, strings can also be used):

```
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 the code 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 raise a panic 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 6.16 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 a panic 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

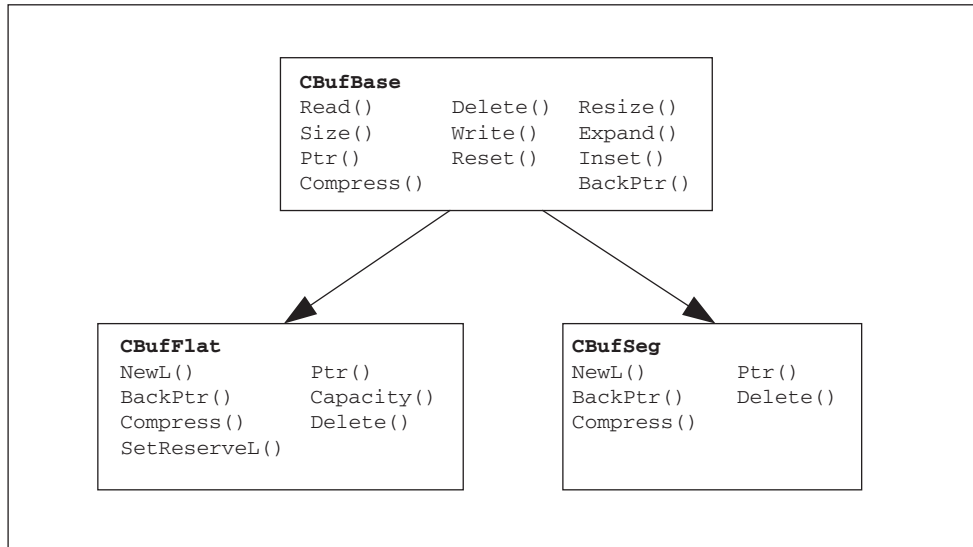


Figure 6.16 Dynamic Buffer Class Diagram

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(aGranularity)`, 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(aGranularity)`, 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 Symbian OS Library in the SDK documentation for more details of how these APIs are used.

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. For example:

```
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

```

The code above 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 if the line:

```
dynBuf->Write(90,dataAry1,30);
```

was replaced in the previous example 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` that 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 Templates in Symbian OS

Before I cover arrays, now is a good time to look at how templates are used in Symbian OS. Let's begin with a quick review of C++ templates in general.

Normally in C++, specifying a variable's data type is done at compile time and thus remains fixed while a program is running. This can be limiting in many cases. For example, if you want to create a list class where you do not care what type of objects are on the list, yet want to implement a common set of operations on them, then you would either need to deal with your data as `void*` (`TAny*` on Symbian OS), which requires the class user to cast the data thus losing your compiler type checking, or you would need to create one list class for each list item data type and duplicate the list operation code in each class. Neither of these is desirable. Templates solve this type of situation by providing a way for data types to be abstracted in the code and passed as parameters at runtime. For example:

```
template<typename T>
class TMyClass
{
public:
    // constructor
    TMyClass() { };
    AddItem(T aItem);
private:
    T iItems[100];
};
```

`template<typedef T>` specified that `T` represents a data type, which is passed in at runtime. So if someone instantiated the class as:

```
TMyClass<TInt> myclass;
```

then `TMyClass` would act as if it were defined as:

```
class TMyClass
{
public:
    TMyClass() { };
    AddItem(TInt aItem);
private:
    TInt items[100];
};
```

In other words, it would substitute the template parameter `T` with the data type `TInt`.

You can also pass values that are not data types to templates. Below is the same example, but also specifying the size of the allocated items array using a template value:

```
template<typename T ,TInt aMaxsize >
class TMyClass
```

```
{
public:
    TMyClass() { }
    AddItem(T aItem);
private:
    T iItems[aMaxsize];
};
```

An example of instantiating this template class is:

```
TMyClass<char, 500> myclass;
```

which will cause `T` to be substituted with `char` in `TMyClass`, and `500` to be the size of the `iItems` array.

Symbian OS uses an idiom for templates known as *thin templates*. This consists of having a private base class that implements core functionality using the `TAny*` data type to provide type-independent functionality. The thin template class inherits from this base class and acts as a thin layer above it (hence the name). The data types (and/or any values) are passed to the thin template class at runtime using the C++ template syntax when the class is instantiated. The thin template methods accept the data type specified to the template, cast the data down to `TAny*`, and call the corresponding base class functions. So even though the base class uses the `TAny*` data type, the thin template class, which the user uses, provides the type checking since it uses templates. I will not go into further detail about how thin templates are implemented in Symbian OS – since this is hidden from you and this knowledge is not needed – but having a basic understanding of templates helps understand the syntax when using template-based classes in Symbian OS.

The array classes described in the next section use thin templates.

6.8 Arrays

Symbian provides a large assortment of API classes for implementing data arrays. The Symbian OS Library in the SDK documentation provides a complete list of these classes, as well as their method descriptions. I will cover the key array classes here and provide some examples.

6.8.1 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]`.

The following is an example of using fixed arrays:

```
/* Initialize a simple fixed array of integers */
TFixedArray<TInt,10> array;
for (j=0;j<10;j++)
    {array[j] = j;}
array[10] = 10; // Causes a panic, 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.

In addition to providing range checking, `TFixedArray` also provides some useful methods such as `Reset()` to clear the entire array with zeros, `DeleteAll()` to invoke `delete` on each array element, `Count()` to return the number of elements in the array, and `Length()` to return the size of an array element in bytes. `Begin()` and `End()` methods are also provided for navigating the array.

6.8.2 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. For example:

```
_LIT(KString1, "My String");
_LIT(KString2, "Test One Two");
TBufC<20> MmDes(KString1);
TBufC<20> MmDes1(KString2);
CDesCArrayFlat myArray = new (ELeave) CDesCArrayFlat(5);
CleanupStack::PushL(myArray); // in case the appends leave
myArray->AppendL(myDes);
myArray->AppendL(myDes1);
/* ... */
CleanupStack::PopAndDestroy(myArray);
```

Figure 6.17 shows how `CDesCArray` is stored, when `MyDes` and `MyDes1` are appended.

To use the descriptor arrays, you need to include file `badesc.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 the array. Of course, the disadvantage of using this type of array is that

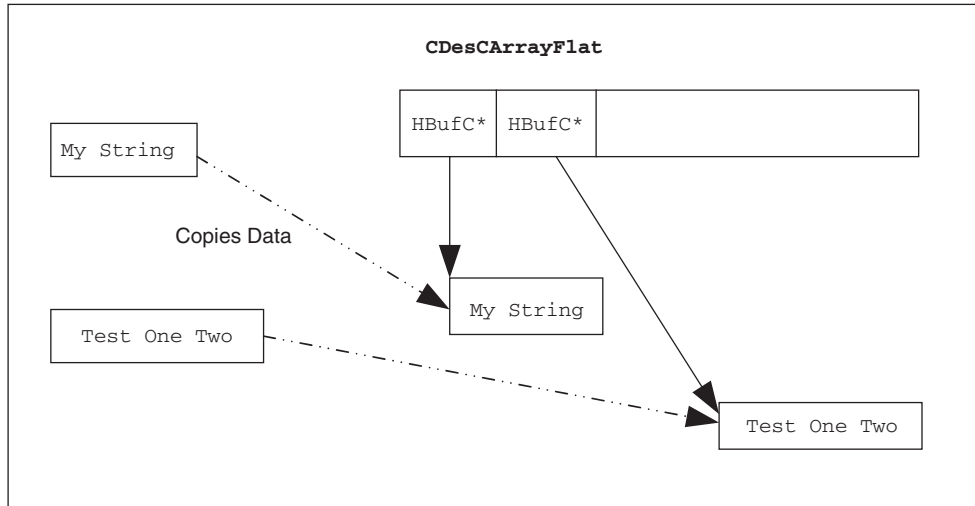


Figure 6.17 CDesCArrayFlat

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 the previous example, it would be stored as shown in Figure 6.18.

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.

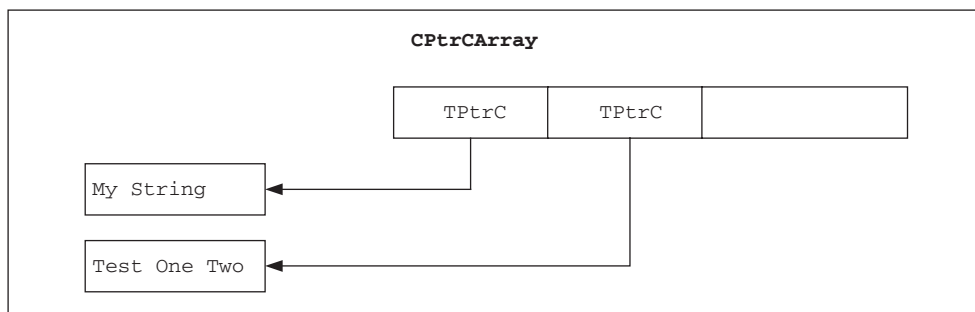


Figure 6.18 CPtrCArray

6.8.3 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.
- `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 holds pointers to type `T` objects, which can be of variable size, and located on the heap.

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<TExampleClass> myArray;
```

This will define a dynamic array `myArray` that contains objects of type `TExampleClass`. Note that this is similar to `TFixedArray` except an array size is not specified with the template. This is because the size is dynamic and not preallocated as it is for `TFixedArray`.

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 some 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, rather like the `RBuf` discussed in section 6.3.5.

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 aGranularity)
```

This specifies how the array is to be expanded in the background. `RArray<TExampleClass> myArray(10)`, for example, will cause the memory for `myArray` to be expanded in units of ten items. The default is eight.

```
RArray(TInt aGranularity, TInt aKeyOffset)
```

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.

The following example creates an array of `TAccount` elements using `TAccount::iAccountNumber` as the array's key:

```
class TAccount
{
public:
    TAccount();
    TInt iAccountNumber;
};
RArray<TAccount> myAccounts(10, _FOFF(TAccount, iAccountNumber));
```

Inserting and appending array data

`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.

Versions of these functions that leave on error also exist.

Finding data in an array

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).

Versions of these functions that leave on error also exist.

Sorting an array

`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

The following is an example of how to use class `RArray`:

```
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 output is as follows:

```
Name=Bob, Unit=132
Name=Sue, Unit=520
Name=Pippin, Unit=755
Name=Sally, Unit=1004
Who's at apartment 520? Sue!
```

The example 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 themselves. 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 a panic when the elements are accessed.

RPointerArray is declared as follows:

```
RPointerArray<CExampleClass> myArray;
```

This is like *RArray*, except that in this case an array of pointers to type *CExampleClass* 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 aPos)` 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. Versions of these functions that leave on error are also available.

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 Symbian OS Library documentation in the SDK for more information.

6.9 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.

6.9.1 Linked Lists

Class `TDbLQue<class T>` can be used to create a double-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 */
TdbLQue<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 single-linked list is available via the `TSglQue<class T>`, `TSglQueLink` and `TSglQueIter<class T>` classes.

6.9.2 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

Platform Security and Symbian Signed

Open platforms like Symbian OS enable a large selection of software to be developed, and made available to a user from many different sources. However, having too much ‘openness’ can be a risk to both the functionality of the device, and to private data. A malicious software developer, for example, could wipe out important data on a device, or call functions that could disable features on the phone. Granted, smartphones are more secure than PCs in some ways since the operating system itself is built into Read Only Memory (ROM), and therefore recoverable by performing a hard-reset operation, which is usually something a user can perform without needing to send the phone back to the manufacturer. However, an attack could still be disruptive since hard-resetting will cause all the user’s data to be lost. The phone could also be vulnerable to an application that claims to perform one function, but instead makes phone calls, sends text messages, or otherwise uses the data network by stealth, leading to unexpected and potentially large service bills. And it’s not just malicious software a user needs to worry about – a phone’s functionality can also be compromised unintentionally by poorly written software.

Customers expect smartphones to be highly reliable, and for their private information to remain private. To address this, Symbian introduced a major feature into Symbian OS v9 known as ‘platform security’, whereby large changes were made to the core of the OS to support vital concepts such as data protection and to restrict certain APIs to code that can be deemed ‘trusted’.

7.1 What is Platform Security?

Platform security has a simple goal: to keep out the kinds of intentionally rogue software, or ‘malware’, which plague the PC world. The

mechanisms on which platform security is built, and the software testing steps which form part of the signing process, also guarantee a level of application robustness as well as generally raising the code quality bar.

Platform security restricts unauthorized access to sensitive APIs that can cause bad things to happen in the device, maliciously or unintentionally. It also provides features that prevent an application from accessing the data of other applications, and preventing access to areas of the file system that store installed executable binaries and configuration files, thus stopping tampering. A secondary goal of platform security is to maintain the platform's openness as much as possible while keeping it secure. Platform security consists of both an architecture and a process.

The platform security architecture is the technical infrastructure by which Symbian OS secures itself. This includes specifying and classifying the various levels of trust of the software and strictly enforcing access restrictions based on how much trust is granted.

The platform security process – known as the Symbian Signed process – provides a way for a developer to get a level of trust granted to an application. It consists of a procedure for submitting application software for testing, specifying a requested level of trust, and evidence that the application can be trusted to behave appropriately when using the restricted functionality. Upon approval by Symbian (and in some cases, the phone manufacturer), the software is digitally signed with a trusted Symbian certificate so that it is then allowed to be installed and used on devices with trusted access to the system as requested. Having this signature is known as being Symbian Signed. If the software is not Symbian Signed, but is using API(s) that require this signing, then it will not be installable on the device.

Note that many applications will still not require access to restricted APIs and thus can be developed without being Symbian Signed. These applications can be built and installed right away on smartphones as with previous Symbian OS versions. However, you still need to understand platform security to know the limitations of the application, such as which system functions it can access, and how to go through the application signing process, should it later be required.

This chapter will describe the Symbian Signed process in detail in section 7.5.2, but first discusses the main concepts of Symbian OS platform security.

7.2 What Platform Security is Not

Since the term 'security' is a broad one, it's best to emphasize at this time what platform security does not cover. Platform security does not detect and remove viruses (although it does make it harder for a virus to exist since software must be Symbian Signed to access sensitive areas), nor does

it include network protection (e.g., a firewall, or VPN). Platform security does not include any cryptographic functionality like encrypting disk files (although as mentioned, cryptographic signing is used for identifying applications approved by the Symbian Signed process). Platform security basically provides protection for sensitive device functions that an application can control – including ones that can access private data – and provides a system of assigning trust to every executable binary file.

7.3 What this Means to a Developer

There's no doubt that platform security has put extra work on developers that was not there for previous versions of Symbian OS. Inevitably, public discussions have focused on the negatives. Platform security introduces significant system-wide changes, and signing imposes new requirements on developers. The upshot is more work, more complexity, more pain.

The good news is that since the publication of Craig Heath's book *Symbian OS Platform Security*, developers have access to an authoritative introduction to platform security on Symbian OS, complete with How To's covering all the major use cases – writing apps, writing servers, writing plug-ins, sharing data. It may not make the evolution to platform security trivial, but it makes it doable.

But above all, Heath makes the *positive* arguments about platform security persuasively. Market building is an essential dimension of platform security and Symbian Signed; an important part of their intent is to create willingness amongst users to download software, and that after all is what an open operating system is all about.

There are still many applications that can be developed that do not touch on the sensitive functions identified by platform security, and thus do not need to go through this process, although some may still choose to, as I'll describe later in section 7.5.1.

Let's begin our discussion on using platform security with the basic 'atoms' of trust defined by Symbian OS – capabilities.

7.4 Capabilities for API Security

Platform security divides sensitive operating system functions into 20 named sets known as capabilities. You can think of each capability as a box of functions with a combination lock on it. An application can be given the combination for as many of these boxes as it is approved to use. In this way, capabilities provide for fine-grained control over API security.

For an application to obtain a particular capability and use protected functionality (i.e., for it to get the 'combination') it must be authorized.

The Symbian Signed program is the authority which gives authorization, although phone manufacturers may also be involved where access to particularly sensitive functionality is requested.

Capabilities are assigned at the process level (i.e., EXE files) and are specified using the `CAPABILITY` keyword in an MMP build file, as I'll show in more detail in section 7.4.7. If you call an API function in your process that requires a capability not specified in the MMP file, it will return `KErrPermissionDenied (-46)`.

Not all API functions require a capability to use them. In fact, about 60% of the APIs are not associated with a capability at all. If your program consists of calls entirely in this set, then you do not need to worry about platform security, and you can run your software without going through the Symbian Signed process. Capabilities represent the APIs that are sensitive enough to be managed by platform security, such as those that directly access system or user data, or the phone hardware.

Capabilities fall into the following categories, based on just how sensitive they are:

- basic capabilities;
- extended capabilities;
- phone manufacturer capabilities.

For the last two categories, in addition to needing these capabilities in your MMP file, you must also have your application Symbian Signed to prove you are allowed to use these capabilities. Otherwise, the software cannot be installed. Software requiring just basic capabilities, or none at all, does not necessarily need to be signed. To explain this further, let's look at the three capability categories in more detail.

7.4.1 Basic Capabilities

Basic capabilities, also known as user grantable capabilities, represent device functions that are easy enough to comprehend so that a user can reasonably choose whether to accept or reject an application's access to them. These functions are not dangerous as far as causing damage to the system or its data, but they can cause unexpected operation and compromise user data if abused. Examples of functionality controlled by basic capabilities are accessing the phone network, sending an SMS, or initiating a Bluetooth connection. Basic capabilities also protect access to potentially private user data such as contacts and calendar information.

Symbian does not require programs with only basic capabilities to be Symbian Signed; however, there are advantages to getting them signed anyway, as will be discussed shortly.

Table 7.1 contains the complete list of basic capabilities.

Table 7.1 The Basic Capabilities

LocalServices
Location
NetworkServices
ReadUserData
WriteUserData
UserEnvironment

Let's look at each of the six basic capabilities listed in Table 7.1 in more detail.

LocalServices

`LocalServices` protects access to the cost-free, local connections like Bluetooth, Wi-Fi, and infrared that can be made, depending on the device in question. `CBluetoothSocket` is an example API class that requires you to have this capability, since it sets up a communication path via Bluetooth technology.

Location

`Location` protects functions on the phone that could get information about the device's location. `CTelephony::GetCurrentNetworkInfo()` is an example API that requires this capability, since it returns information about the device such as the device's current cell ID and location area code.

NetworkServices

`NetworkServices` protects access to network connections that may involve cost. Examples of this would be initiating phone calls, sending SMS, and starting an Internet connection. For example, the `RConnection::Start()` API function requires this capability.

ReadUserData

`ReadUserData` protects access to reading of potentially private user data such as text messages, calendar, to-dos, or contact entries. For example,

the API `CContactDatabase::OpenL()` function, used to open the device contact entries for reading, requires this capability.

WriteUserData

`WriteUserData` protects access to private user data described above. Again, using a contacts database function as an example, `CContactDatabase::DeleteContactL()` would require this capability since it modifies a user's contact database.

UserEnvironment

This capability protects access to the user's immediate environment. This includes access to the camera (video and still) and the voice recorder. The API class `CCamera` is an example of a class with API functions that require the `UserEnvironment` capability.

User granting basic capabilities

If an application is not Symbian Signed, the user will be notified if the application needs access to a basic capability, and given the choice to reject this access. This can happen in one of two ways: blanket grant or single-shot grant.

A blanket grant is when the user is notified at install time that the application requires them to grant permission to certain features. The user can choose to continue the installation, if the access seems appropriate for the application and the user trusts it, or cancel it.

In a single-shot grant, the user is notified and given the chance to reject access as previously, but the notification occurs each time the access is actually about to be performed, instead of only once at installation time. It is at the discretion of the phone manufacturer as to which type of user grant occurs for each basic capability, and it may even differ across different device ranges for a single manufacturer.

As an example, say you have developed an application that uses the `ReadUserData` and `NetworkServices` capabilities as specified in the following line in your MMP file:

```
CAPABILITY ReadUserData NetworkServices
```

On the Nokia E61 device (S60 3rd Edition), installing the application will show the blanket grant prompt for both capabilities (as can be seen in Figure 7.1).

As Figure 7.1 shows, the user now knows what the application needs to access. As a user, you can understand the implications of that access and choose to either continue or cancel the installation. For example, if



Figure 7.1 Blanket Grant Prompt on E61

you are installing a graphics drawing program and see this prompt, you may be suspicious that it needs access to the network or your contacts data, so this may cause you to cancel the installation.

Note that some manufacturers may decide to enforce a capability through a single-shot grant instead of a blanket grant at installation. In that case, the user is prompted each time the application tries to access a function requiring the capability, instead of just once during installation. For example, for `NetworkServices`, the user would be prompted to reject or continue a network access every time a network access is attempted. As an example, the blanket grant prompt for `NetworkServices` is shown here for the Nokia E61 (S60 3rd Edition) but may not be displayed for phones from other manufacturers, or even some other S60 3rd Edition devices.

Symbian Signing and basic capabilities

Although not required, it can still be preferable to get your application Symbian Signed if it uses basic capabilities. When an application is Symbian Signed, it is considered trusted and thus does not prompt the user for permission to access any functions (blanket or single-shot). This can improve usability, especially if excessive prompting would occur otherwise.

7.4.2 Extended Capabilities

Extended capabilities protect APIs that can affect the integrity of the system itself. For extended capabilities, it is not appropriate to prompt the user for permission since: (1) the user would probably not understand what the prompt is telling them anyway, and (2) in those situations, the user probably will not really know what the correct decision to make is

since they have no way of knowing the quality or intent of the code, or even what the effect would be if the functions were abused. Applications using APIs that require extended capabilities must be Symbian Signed to be installed and run.

It is entirely possible to write full-featured applications that do not use any of these extended capabilities. These capabilities are required for advanced programming to tie an application into the device closely (like a program that can globally capture key presses for example, or needs access to sensitive system settings). The APIs that these capabilities protect are, for the most part, not covered in this book.

Table 7.2 lists the extended capabilities.

Table 7.2 The Extended Capabilities

PowerMgmt
ProtServ
ReadDeviceData
WriteDeviceData
SurroundingsDD
SwEvent
TrustedUI

PowerMgmt

This capability is for power management and controls access to functions such as powering off the device and terminating processes. Example API functions protected by this capability are: `Power::PowerDown()` and `RProcess::Terminate()`.

ProtServ

`ProtServ` gives you the ability to protect a server by adding an '!' to the front of its name. We will not cover this concept in this book. Suffice to say that its purpose is to prevent a third-party server from pretending to be a system-protected server by spoofing its name.

ReadDeviceData

This capability is required to read critical system device data. For example, you need it to get the phone's serial number (a.k.a. IMEI) using `CTelephony::GetPhoneId()`. Reading the phone's lock code and list of installed applications are other examples.

WriteDeviceData

This capability protects the writing of device data. This protects writing of some parameters outside of the ones `ReadDeviceData` protects. For example, the system time can be read without `ReadDeviceData`, but can only be written if you have `WriteDeviceData`.

Other device parameters protected include lock settings, alarms, and time zone.

SurroundingsDD

The `DD` in the name refers to device drivers. This capability protects access to device drivers that allow you to know about the phone's surroundings, such as GPS hardware.

SwEvent

This capability protects the capturing of key presses and the ability to produce virtual key events. These functions are needed by things like screen capture programs that rely on a hotkey being selected to capture the screen. Another example would be a key capture virus that records all your keystrokes to send off to your computer on the network (but that use case is unlikely to be approved by Symbian Signed!).

TrustedUI

This capability provides the ability to create a trusted dialog in a way that cannot be reproduced by an application without this capability. The purpose is to assure a user that, if they see it, they can enter sensitive information, such as their banking PIN, and it won't be stolen by another application that is pretending to be the trusted one. This is rare, and in fact not supported in Symbian OS v9.1.

7.4.3 Phone Manufacturer Capabilities

Phone manufacturer capabilities protect access to the most critical APIs and data on the phone. As with extended capabilities, applications with phone manufacturer capabilities must be signed before the application can be installed. The capabilities are reserved for advanced system-level software and are not easy to get authorized. In addition to the Symbian Signed process described in section 7.5.2, there is an extra step involved to get approval from the manufacturer of the handsets for which the software is intended (e.g., Nokia or Sony Ericsson). Table 7.3 lists the capabilities in question.

Table 7.3 The Phone Manufacturer Capabilities

AllFiles
CommDD
DiskAdmin
MultimediaDD
NetworkControl
Drm
Tcb

Although the functions protected by phone manufacturer capabilities are for the most part outside of the scope of this book, let's discuss each capability in Table 7.3 in a little more detail.

AllFiles

AllFiles gives you read access to the entire file system. You can read most of the file system without it, but this capability additionally gives you read access to the \sys directory where the executables are stored (but not write access). Most critically though, AllFiles gives you read and write access to all the subdirectories of the \private directory where applications store their private data. I'll discuss the concept of file system data caging shortly in section 7.4.8.

CFileman::Copy(), for example, requires this capability if the source or destination of the copy is a private directory.

AllFiles is rarely approved by a phone manufacturer since this compromises all private application data on the phone. The most common users of this capability are the manufacturers themselves, for internal development and test tools.

CommDD

This capability is needed for direct access to communication drivers such as those for Wi-Fi and USB connectivity.

DiskAdmin

DiskAdmin controls access to disk administrative functions like formatting a drive, mounting and unmounting a file system, and other disk-handling functions. RFs::CheckDisk() is an example API that requires this capability.

MultiMediaDD

MultiMediaDD is needed if you want direct access to multimedia device drivers like audio and camera or priority access to multimedia APIs.

NetworkControl

NetworkControl is required for controlling network connections and protocols. An example is stopping a network connection with `RConnection::Stop()`.

Drm

This capability provides access to music protected by Digital Rights Management and encrypted on the device. As you can probably guess, it is hardly ever granted by phone manufacturers except to their trusted partner companies.

Tcb

Allows complete read and write access to the entire file system (All-Files gives read/write access to the `\private` directory but not to `\sys` or `\resource`). The Trusted Computing Base (TCB) forms a critical part of platform security and the Tcb capability is reserved for a very few components that form the core of Symbian OS. For more about the TCB, you should consult [Heath].

7.4.4 How Do You Know Which Capabilities You Need?

To find out what capabilities you need, your first stop should be the SDK documentation. The API reference section should tell you what capabilities are needed to access each API. However, there are cases where the SDK documentation does not mention the capability needed, or is unclear.

The best test sometimes, unfortunately, is trial and error performed by plugging capabilities into the MMP file until you find the one(s) you need (remember, an API will usually return `KErrPermissionDenied` if the caller does not have the capability needed). You can do this on the device for basic capabilities, however, this would not work when trying to find extended or manufacturer capabilities since you would not be able to install this to your device with these capabilities without being Symbian Signed.

The best way to experiment in order to determine the platform security capabilities required by your application is to use the Symbian OS emulator. The emulator will enforce the access specified in your MMP

file for capabilities (in the same manner as a device), but without requiring your application to be Symbian Signed.

The next section discusses some emulator configuration options related to platform security.

7.4.5 The Emulator and Platform Security

The emulator has configuration options related to platform security that can be useful for development. You can set these configuration options in the emulator configuration file `\epoc32\data\epoc.ini`.

PlatSecEnforcement

`PlatSecEnforcement` enables or disables platform security enforcement. When enabled, if a platform security check fails, the appropriate action is taken (which is typically to generate a `KErrPermission-Denied` error or leave). If platform security enforcement is disabled, however, the system continues as if the check passed. To enable, enter the following in `epoc.ini`:

```
PlatSecEnforcement On
```

To disable:

```
PlatSecEnforcement Off
```

The default for the emulator is for security enforcement to be enabled. You may want to disable security enforcement, for example, in the early stages of your project when you are concentrating more on program logic than security.

PlatSecDisableCaps

`PlatSecDisableCaps` can be used to selectively disable specific capabilities. For example, if you have:

```
PlatSecEnforcement On  
PlatSecDisableCaps Location NetworkControl
```

in the `epoc.ini` file, then all capabilities will be enforced except `Location` and `NetworkControl`. The emulator will bypass any checks for this capability in the system APIs.

PlatSecDiagnostics

PlatSecDiagnostics enables and disables logging of platform security-related failures to `epocwind.out`. From this log you can tell what APIs failed due to permission errors.

To enable, enter the following in `epoc.ini`:

```
PlatSecDiagnostics On
```

To disable:

```
PlatSecDiagnostics Off
```

7.4.6 Defining Capabilities For Your Project

As previously mentioned, you specify the capabilities your process needs in the MMP file using the `CAPABILITY` keyword. In the Carbide.c++ development tool, you can specify it in the project properties dialog, under C/C++ build. Select the tool settings tab and the system options under that.

If you do not have a `CAPABILITY` line in your MMP file, then the default is that you have no capabilities. `CAPABILITY NONE` specifies the same thing.

The basic way of specifying capabilities is to enter each capability needed after the `CAPABILITY` keyword, separated by spaces.

`CAPABILITY ALL` means to use all capabilities. The `'-'` sign can be used to subtract capabilities from `CAPABILITY ALL`.

For example, the line:

```
CAPABILITY ALL -Tcb -AllFiles -Drm -CommDD -Multimedia -NetworkControl  
-DiskAdmin
```

means all capabilities except the phone manufacturer capabilities (`Tcb`, `AllFiles`, `Drm`, `CommDD`, `Multimedia`, `NetworkControl`, and `DiskAdmin`).

7.4.7 Capabilities in DLLs

As mentioned previously in section 7.4, capabilities are defined at the process level. Since a DLL function executes in the same context as the calling process (i.e., it runs as part of that process), then the DLL function always executes with the calling process's capabilities.

You must still specify capabilities in your DLL's MMP file (or Car-bide.c++ project properties), but the meaning is different than it is for a process. Specifying capabilities in a DLL means that the DLL is trusted to run within a process that has those capabilities, not necessarily that it actually uses those capabilities.

Capabilities in DLLs can be summarized by the following two rules:

1. A DLL must have at least the capabilities of its calling process.
2. When executing, a DLL function always inherits the capabilities of the calling process (and no more).

For the first rule, if a process tries to load a DLL that does not have all of the capabilities of the process, it will fail. It's fine, however, if the DLL has capabilities that the calling process does not have. Another way of saying this is that the calling process's capabilities must be a subset of the DLL's capabilities. Following is an example to illustrate this.

If you have the following capabilities in your process (as specified in your MMP):

```
CAPABILITY ReadUserData ReadDeviceData
```

and in your DLL, you have:

```
CAPABILITY ReadUserData WriteUserData NetworkServices
```

then the process will not be able to load and use your DLL since it does not have `ReadDeviceData`. The DLL would have to have both `ReadUserData` and `WriteDeviceData` as a part of its declared capabilities for that process to use the DLL.

The second rule is just a recap of what we discussed before about the DLL functions always running with the same capabilities as the calling process.

Following is an example to illustrate the second rule.

If your process has these capabilities:

```
CAPABILITY ReadUserData WriteUserData NetworkServices
```

and the DLL has:

```
CAPABILITY ReadUserData WriteUserData NetworkServices NetworkControl
```

then your process will be able to load and use the DLL since it does contain at least the capabilities of the calling process. However, the DLL code will just have `ReadUserData`, `WriteUserData`, and `NetworkServices` while executing from a call from that process, since it inherits only the capabilities assigned to the calling process.

You'll want to be safe and ensure that your DLL contains all the capabilities a calling process may have so that the process can use the DLL (as per rule 1). However, like a process, your software will need to be Symbian Signed if the DLL specifies extended or phone manufacturer capabilities in its MMP – even if the DLL does not use any of the APIs protected by those capabilities. In that case, it's fairly easy to justify having them if you can guarantee that the DLL can be trusted in a process that does possess those capabilities. However, note that the intent is not to include the complete set of capabilities in your DLL (i.e., `CAPABILITY ALL`) just so that every process is guaranteed to be able to use the DLL – it's unlikely that would get approval for Symbian Signing.

7.4.8 Data Caging

In addition to providing protection for sensitive APIs, platform security also has a feature for protecting data in the file system. This feature is known as data caging. Data caging is not a general file protection scheme like Unix and Linux, where individual files can be given read and write access on an individual basis. Data caging in Symbian OS consists of setting aside specific directories in the file system and preventing access to them except to processes with a high level of trust or owners of a particular private area. These directories become cages of files so to speak, where the cages protect the contents of the area.

Let's look at the directories in the file system that are protected by data caging.

\sys directory

All system executables reside in the `\sys\bin` directory. In fact, they cannot be run from anywhere else. To prevent tampering with executables, platform security restricts access to this directory. Only processes with the `Tcb` capability can write to this directory. `Tcb` is a manufacturer capability and as stated previously in section 7.4.3, this capability is very hard to get approved. The software installer is an example of a process that has this capability and thus can install executables to `\sys\bin`.

\resource directory

The `\resource` directory contains the application resource files (e.g., bitmaps, fonts, and help files). All applications can read from it, but you need the `Tcb` capability to write to it.

`\private` directory

The `\private` directory contains the private data for all applications. There is a separate subdirectory reserved for each application. An application’s subdirectory is `\private\<SID>`, where `<SID>` is the application’s secure ID as identified in the `SECUREID` line of the application’s MMP file. If there is no `SECUREID` line in the MMP file, then `SID` is defaulted to the application’s `UID3` (which is the suggested course as mentioned in section 7.5.2). If there is no `UID` (but there always should be), the `SID` defaults to 0.

An application can access its own reserved subdirectory under the `\private` directory, but not any other application’s directory, unless it has the `AllFiles` capability. This ensures that an application’s data stays private.

There is no requirement that your application must write its data to its reserved private directory, but it’s a good idea to put any data there that you want to be kept private.

A DLL does not have its own private directory so it must use the private directory of the calling process. To discover the name of the calling process’s private directory, the DLL can call `RFs::PrivatePath()`.

Table 7.4 summarizes the capabilities required to read and write the protected areas of the file system:

Table 7.4 Locations of the Data Cages and Capabilities Required for Access

Directories and subdirectories for all drives (Z, C, etc.)	Capability required to read	Capability required to write
<code>\sys</code>	<code>AllFiles</code>	<code>Tcb</code>
<code>\resource</code>	None needed	<code>Tcb</code>
<code>\private\<ownSID></code>	None needed	None needed
<code>\private\<otherSID></code>	<code>AllFiles</code>	<code>AllFiles</code>

Directories other than the ones here can be read from and written to without requiring any capabilities.

7.5 Application Signing in Symbian

This section discusses application signing on Symbian OS. But before jumping in to Symbian OS specifics let’s review the concept of digital signing in general.

7.5.1 Digital Signing Overview

Digital signing is a way of marking a file such that a receiver of the file can be assured of the file's source and integrity. Digital signing acts like a traditional written signature with the added feature of ensuring that the file was not modified after the file was signed.

Digital signing is a part of the public key infrastructure (PKI), which protects data by assigning a person (or company) a key pair consisting of a public key and a private key. The public key is widely distributed, while the private key is privately held by the owner of the key. It is very important that the key is protected and never given to anyone else, or made vulnerable to theft! Data encrypted by a person's public key can only be decrypted by that person's private key. In this way, someone can send a secret message to anyone if they have their public key, and only the private key associated with that public key can decrypt the message.

Digital signing works in the reverse manner: data encrypted by a private key can only be decrypted by the corresponding public key. The signer encrypts a 'signature' on a file using their private key, which includes the file's checksum (i.e., hash). The person receiving this file gets the corresponding public key for the sender and can decrypt the signature. In this way, it's possible to verify that the signature was created by the holder of the private key, since that person is the only one with access to it. The verification process confirms that the checksum of the decrypted file is the same as the one that was signed, which guarantees that the file has not been tampered with since it was signed and distributed.

A certificate associates a person or company's identity with their public key and has both pieces of information physically in it. Its purpose is to provide assurance of who the public key belongs to. The certificate is itself signed by a certificate authority (CA), which is a body trusted to be able to verify a person's identity. For Symbian Signed identity verification, the CA used is a company called TC TrustCenter. A CA's certificate can in turn be signed by another CA (a more trusted party), and so on, all the way to a certificate root, creating what is known as a certificate chain. If you trust the certificate root, then you can trust the data in the certificate.¹

7.5.2 Application Signing

In Symbian, signing an application consists of digitally signing a Symbian OS installation file (SIS file), which contains it (SIS files, and the PKG files used to create them, are discussed in more detail in Chapter 5). Only the SIS file itself is signed, not its contents – even if it contains multiple executables and DLLs. This differs from Microsoft Windows Mobile, for

¹ For further information about PKI, please consult *Understanding PKI* (Adams and Lloyd, 1999) and Appendix B of *Symbian OS Platform Security* by Craig Heath.

example, which requires each executable and DLL, in addition to the installation (.cab) file, to be signed.

A developer can sign a SIS file themselves using the `signsis` tool – this is known as ‘self-signing’. Or the application can be signed externally, as is the case when an application is Symbian Signed.

The three types of application signing in Symbian are:

1. unsigned,
2. self-signed,
3. Symbian Signed.

Unsigned

Unsigned applications are not digitally signed at all. This is the state of the SIS file after it is created with the `makesis` tool (as discussed in Chapter 5).

In Symbian OS v9, generally an application must always be signed in some way (that is, it should be self-signed if Symbian Signing is not required), but the capabilities granted using self-signing are decided by the phone manufacturer. For example, S60 3rd Edition only allows signed applications to be installed on devices.

Self-signing

Self-signing is when an application is signed without going through Symbian Signed. To self-sign it is necessary to have a certificate and the private key of that certificate. Using the `signsis` tool it is possible to sign a SIS file built previously with `makesis`, or it is also possible to use the `createsis` tool to build and sign a SIS file in one go. Carbide.c++ uses this tool (as can be seen from its console output).

The most basic way to obtain a certificate for self-signing is to generate one using Symbian’s `makekeys` tool. The `makekeys` tool generates both the public certificate and the private key. Here is an example:

```
makekeys -cert -len 2048 -dname "CN=Bob Smith OR=Acme CO=US  
EM=bob.smith@acme.com" bobkey.key bobcert.cer
```

This generates a certificate that includes the identity of a developer named Bob Smith. The .cer file generated is the public certificate file and the .key file contains the private key for that certificate.

It is then possible to sign the install file with this certificate using `signsis`:

```
signsis MyApp.sis MyApp_signed.sis bobcert.cer bobcert.key password
```

where:

`MyApp.sis` is the file name of your unsigned application.

`MyApp_signed.sis` is the filename for the signed version of the application.

`bobcert.cer` is the certificate filename.

`bobcert.key` is the certificate's private key.

`password` is the password of the private key (can be omitted if there is no password).

Now you have a signed SIS file that you can install on a phone, unless:

- Your application uses extended or phone manufacturer capabilities.
- The phone manufacturer has prevented applications from being installed if they are not signed with a 'trusted' certificate (I'll discuss trusted and untrusted certificates in the next section).
- There is a setting in the phone's device settings to allow the user to disable installations from SIS files signed by untrusted certificates, and the user has selected it.

Trusted and untrusted certificates

If you create a certificate and key using `makekeys`, as described above, it is not considered trusted by the phone. This is because it's not signed by a CA who would have verified that your certificate actually identifies you. You can see why it's not trusted since it was easy for me to create a certificate for Bob Smith although that is not who I am. Symbian OS will still allow an application signed by an untrusted certificate to be installed; however, it will display a message saying that the application is from an untrusted source, and asks the user if they want to continue the install. An example of this dialog is shown in Figure 7.2.



Figure 7.2 Warning with Untrusted Signature

To prevent the display of this warning dialog, it is necessary to have a certificate issued by a Symbian Certificate Authority. This is achieved by applying to the Certificate Authority for a Publisher Id. Following a vetting process that confirms the identity of the applier the Publisher Id is issued. The Publisher Id contains the certificate and key file that is used to identify the source of the application.

When an application is signed with a trusted certificate, the user can use the application manager on the phone to view who the certificate belongs to and therefore the application's source. It also provides confidence for the user who is installing the application by providing information about the application's author.

Another type of certificate you can self-sign with is a developer certificate. Developer certificates are used to develop and test software that uses sensitive capabilities before the software is submitted for Symbian Signed testing. Developer certificates are downloaded from the Symbian Signed website and are described in more detail later, in section 7.7.

Symbian Signed

As mentioned previously, an application must be Symbian Signed if it uses capabilities identified as sensitive. It is not possible to Symbian Sign an application yourself – TC TrustCenter does this following a successful submission through the Symbian Signed process. The certificate TC TrustCenter uses for Symbian Signing chains to a Symbian Signed root certificate which resides on Symbian OS v9 devices. When a Symbian Signed application is installed, the installer sees that the application's digital signature chains to the Symbian Signed root stored on the device, and therefore recognizes the application as being Symbian Signed and grants it all the capabilities specified in its MMP file(s).



Figure 7.3 A Rejected Installation: Capabilities Are Not Authorized By A Valid Signature.

If the application's root certificate does not match the Symbian Signed root, and your application has extended or manufacturing capabilities specified in its MMP file(s), the installer will abort the install and display a message similar to the one in Figure 7.3.

The message in Figure 7.3 can vary between phone manufacturers.

Installation errors related to signing

I have pointed out a few examples of installation errors that can occur related to application signing. Other situations that can cause errors when installing are:

- If you are installing an unsigned application and the phone requires it to be signed (e.g., 'Certificate error. Contact the Application Supplier' is displayed on a Nokia S60 device).
- If the certificate you signed the application with has expired.
- If the application is signed with a developer certificate, but you have not specified the device for the certificate (this will be discussed more when we describe developer certificates). On S60 phones, the message will read 'Unable to Install. Constrained by Certificate'.
- If the application's UID is in the protected range, yet the application is self-signed. On S60 phones, you will see the message 'Unable to install a protected application from an untrusted supplier'.
- If the application installation tried to copy files to another application's private area (i.e., \private\<SID> where <SID> is the application's SID – see section 7.4.8 for more details). On S60, the message will simply read 'Unable to install'.
- If the UID of the application you are trying to install is used by another application already installed on the phone. On S60, the message will be 'Update error'.

These messages will vary between phone manufacturers.

UIDs, SIDs, and application signing

In Chapter 5 we discussed how all executables are uniquely identified by a UID3 and showed how to reserve these unique identifiers from Symbian. UIDs are divided into a protected range and an unprotected range. The protected range is from 0x00000000–0x7fffffff, the unprotected range is from 0x80000000–0xffffffff. Applications must have a UID in the protected range for them to be Symbian Signed, otherwise they will not be approved. If your application is not Symbian Signed, then it must use a UID in the unprotected range, otherwise the install fails.

An SID also specifies a unique ID for an executable, but the SID is used to form your application's private directory name. Refer to section 7.4.8 for details on how this is done. You can specify an SID by adding the `SECUREID` keyword to an MMP file. Symbian recommends that you use your UID3 as your SID, which is the default if the MMP does not have a `SECUREID` line.

7.6 Getting Your Application Symbian Signed

If your application uses extended or manufacturer capabilities, you must go through the Symbian Signed process, and get your application Symbian Signed before it can be installed and run by users. Or, if you don't use those capabilities, but you prefer that the user never has to be asked for authorization to perform operations, then the application should be Symbian Signed.

It is also worth noting that, depending on how you intend to distribute or sell your application, some channels may require an application to be Symbian Signed before they even consider it for distribution via their sales channels. Application robustness, quality, and identification of the origin of the application are the primary drivers of this demand.

An application that is Symbian Signed is also allowed to display the 'For Symbian OS' logo, as shown in Figure 7.4.

Symbian provides the website <http://www.symbiansigned.com> to guide you through the Symbian Signed process and to submit your application for approval and the actual signing. This site requires you to register. Once you've done so, you have access to a wealth of information including documentation and tools. The Symbian Developer Network also has a discussion forum for issues related to Symbian Signed, at <http://developer.symbian.com/forum/forum.jspa?forumID=2>.

The basic idea of the Symbian Signed process is that if your application passes a suite of tests to ensure that your application uses its capabilities responsibly, and if it is a well-behaved application in general, then it will be approved and signed. Symbian documents these tests in the Symbian Signed test criteria document, which is available on the Symbian Signed website.



Figure 7.4 The Official Logo for an Application Approved by Symbian Signed

7.6.1 Test Criteria

It is necessary to pass the Symbian Signed Test Criteria before your application can be Symbian Signed through the 'Tested Signed' process.

You'll need to understand the criteria carefully since they also drive functionality and you may need to add code to your application if the support needed to pass the tests is not already there.

There are a number of test tools available on the Symbian Signed website for use to test that your application complies with the criteria. By ensuring you pass these tests before you try and use the Symbian Signed process itself, you will minimize the risk of your application failing and having to be retested. However, Symbian also recognizes that there are some extenuating situations that can prevent some tests from being fully met, so it specifies a list of approved exceptions that can be taken for the tests and has a waiver process in place where the exceptions cannot be applied.

The application test criteria include all of the following:

- Programs must avoid DLL name clashes and must have reasonable startup time behavior, including progress indication; they must create files only in allowed locations and uninstall cleanly and completely, and reinstall successfully; it must be possible to see and terminate the application from the task list.
- UIDs must be legal and proper and owned by the submitting individual, the SIS file well formed, including correct and consistent versioning, and access to platform/manufacturer granted capabilities (more on this below) must be properly approved, as must any test waivers; any Publisher ID (where appropriate) must be valid.
- Programs must meet their function specifications (i.e., do what they say they do); must not interfere with system applications such as Phone, Clock, and Contacts. They must not interfere with system events, including alarms and high-load events like camera or phone device initialization or VoIP calls, nor with message or call transmission, including sending and receiving SMS and MMS messages, and notifying users of incoming calls.
- Programs must survive stress tests including handling exceptional events like OOM (out of memory) or power down while running, as well as rapid and repeated task switching.
- Programs must behave well with respect to privacy and billable events; must successfully backup and restore data (where appropriate); must demonstrate scalable UI compliance.

7.6.2 Symbian Signed Process Options

The outcome of the signing process is an application SIS file to which Symbian's 'root' digital signature has been appended in the form of a certificate, and which has in turn incorporated the digital signature of the developer, thus 'binding' the developer's identity to the application. (Or, in the case of the freeware and Publisher Certifier routes to signing, binding the publisher's identity to the application.)

At install time, the application certificate is verified against the Symbian root certificate. This validation and display of the certificate forms an essential part of the security architecture.

There are three principle signing routes to consider; any one of them will get you up and running with the Symbian Signed process; all result in software which has been signed with the Symbian root certificate; and between them, they should provide an acceptable route to market for everyone.

Freeware signing

Symbian provides a zero-cost route to signing; it's free, it's open to anyone; its two requirements are that (1) your application is sound, in other words that it meets the test criteria, and (2) your application really is free to its users.

There is nothing to stop you, now or later, creating alternative non-free versions of your software; however, any version of your software signed in this way really must be free.

If you want to give your software away, this route enables you to do so, and it will cost you nothing. It is an excellent way for small developers working in the 'hobbyist' tradition to bypass both the cost and most of the red-tape of the signing process, and get freeware out to their users. (That tradition is a long and noble one, by the way, as anyone who cares to trace the history of Symbian OS back to its origins in Psion days can attest.)

How does it work? Zip up your application SIS file, PKG file and plain text user manual, and submit. No Publisher ID is required; your application will be tested, and if it passes, it will be signed with the hosting test house signature and Symbian root.

At the time of writing (Q2 2007), the test/sign/release cycle for freeware is hosted through Cellmania. Details of how to find out more can be found in the References section.

Publisher certifier signing

Not free, but a lower-cost option for commercial hobbyists and the one-person ISV. The essential requirement is that your application meets the test criteria.

How does it work? The publisher/aggregator provides the test and signing service, typically for a fee, and provides the channel to market, typically for a commission on sales. No Publisher ID is required; your application will be signed with the publisher signature and Symbian root.

7.6.3 Test House Certification

The process to get your application signed via a test house is as follows:

1. Get a Publisher ID from TC TrustCenter (if you do not already have one).
2. Sign your application with the Publisher ID.
3. Submit your application through **<http://www.symbiansigned.com>**, selecting a third-party test house.
4. Handle any problems found by the test house.
5. Upon passing of the tests, they send you the signed application.

Let's look at the steps in more detail.

Getting a Publisher ID

Before you can submit your application to a test house for Symbian Signing, you will need a Publisher ID from TC TrustCenter.² This is a certificate that identifies your company or organization. You also get the private key for the certificate so you can self-sign your application with it. The Publisher ID costs 200 US dollars which is paid annually.

Signing your application with the Publisher ID is required by Symbian Signed to ensure that submitted applications come from an identifiable source and that they have not been tampered with along the way. This is always required before your application can be Symbian Signed via any route (although someone else could sign your application with their Publisher ID on your behalf, which is the case if you use a Publisher Certifier – more on that shortly).

To get the Publisher ID, you should go to the TC TrustCenter website (**<http://www.trustcenter.de/order/publisherid/dev>**). You can also navigate to the site from the Symbian Signed portal.

² Prior to May 2007, the Publisher ID required by Symbian Signed was supplied by Verisign, and if you have already acquired a Verisign Publisher ID, you can continue to use this until it expires. At that point, you should get a replacement from TC TrustCenter rather than renew with Verisign. The transition is seamless, and applications previously submitted to Symbian Signed using a Verisign Publisher ID will continue to work on your customers' phones.

Simply follow the instructions on the TC TrustCenter site to acquire your Publisher ID. Once your identity has been verified you will get the certificate and private key associated with the Publisher ID. Detailed instructions on how to use the files you download are available on the Symbian Signed website.

Sign your application with the Publisher ID

Once you have the .cer and .key files, you can use the Symbian `signsis` command line tool to sign your application, as shown earlier. For example, to sign `app.sis`, you would use the following command:

```
signsis app.sis app_publisherid.sis publish.cer private.key
```

where `publish.cer` and `private.key` are your Publisher ID files.

Submit your application to Symbian Signed

Submit your application to Symbian Signed via the ***<http://www.symbian-signed.com>*** portal. Follow the instructions given in the detailed documentation provided to guide you in making a submission. You will be directed to select a third-party test house to make sure your application passes the tests in the test criteria document (they will provide you with a quote for the costs).

You should make sure your application passes the tests in the test criteria document before you submit it to Symbian Signed. This will save time and money, since every time your application has to change, it needs to be resubmitted to the test house at a cost to you. It also prepares you better to handle issues and any problems reported by the third-party test house.

To submit to the Symbian Signed website you need the SIS file (signed with the Publisher ID), a `readme` containing description and usage information, and your application's package (i.e., PKG) file. When the test house receives your application it verifies the signature to make sure the submitted application came from you.

Support the test house as needed

Testing with the test house can be interactive depending on what your application does, how complex it is, and the issues and problems found. If you have any questions prior to submitting the Test Houses will provide

support to ensure your submission runs smoothly. The contact details for the Test Houses can be found on the Symbian Signed website.

Receiving your Symbian Signed application

On successful completion of the process, you will be notified and can then download your application which is now Symbian Signed.

Behind the scenes, when your application passes all its tests and is approved, the test house uploads your application to TC TrustCenter. TC TrustCenter then sign the file using the Symbian Signed root certificate, allowing the installer on the phone to check that the file has been Symbian Signed and can install. This means that the creator of the application can be identified by the user through the phone's application manager. Once Symbian Signed, the user will no longer see an 'untrusted source' warning message when they install it.

If you require phone manufacturer capabilities

If your application uses any of the phone manufacturer capabilities (Tcb, Drm, AllFiles, MultimediaDD, DiskAdmin, NetworkControl, or CommDD) you have extra steps to go through. You must get approval from the phone manufacturer to use these. To request this approval, you must complete and submit a capability request form that indicates which of the capabilities you need and the reason why you need them. The third-party test house will also need to run through some specific tests required by the phone manufacturer in question.

7.6.4 Self-Certification

An alternative to sending your application to a test house for approval is to become a self-certifier. This means that you run the Symbian Signed tests specified in the test criteria document yourself, without going to a third-party test house. Once you have completed all testing, you submit the test results, along with your application (and other information you normally send to a test house including waiver documents) via the Symbian Signed website. Once uploaded the SIS file is Symbian Signed and is available to download. Note that you must still have signed your application with your Publisher ID before submitting it.

If your application has phone manufacturer capabilities, you will also need a self-certification agreement with that manufacturer. Symbian will not sign a self-certified application with those sensitive capabilities until it has verified with the manufacturer that this agreement is in place.

7.7 Developer Certificates

You may ask how you can test your application on hardware before submitting it to Symbian Signed if it requires it to be Symbian Signed to run on devices. This is a good question! You can always test it on the emulator, which does not require Symbian Signing, but you do also need to test your application on a real phone. The answer to this, as hinted before, is to use a developer certificate (like Symbian Signing, you do not need a developer certificate if your application does not require capabilities from the extended or phone manufacturer sets).

7.7.1 What is a Developer Certificate?

Instead of signing with a certificate and key generated by the `makekeys` tool, a developer certificate is a certificate you download from Symbian for your day-to-day development work. You can sign your application yourself with this certificate, and then test the software on one or more phones.

There are, however, some limitations:

1. The signed application will only work on a specified phone or set of phones. The IMEI (i.e., serial numbers) of each phone is hardcoded into the certificate.
2. The developer certificate will expire after six months, after which time you will need to download a new certificate.

Developer certificates can be configured to contain a particular set of capabilities, and you must get approval from the phone manufacturer if you need the most sensitive capabilities in your developer certificate.

Developer certificate

A developer certificate allows you to install the signed software on a number of phones as long as they are registered in the certificate. A developer certificate can be used with any of the basic and extended capabilities.

Phone manufacturer certificate

If your application uses any of the phone manufacturer capabilities, you will need a phone manufacturer approved developer certificate. It is necessary to apply to the phone manufacturer for a developer certificate containing phone manufacturer capabilities. The Symbian Signed web site provides information on how to apply for a developer certificate containing these type of capabilities.

7.7.2 How Do I Get a Developer Certificate?

To get a developer certificate, you need to log into the Symbian Signed site (registering if you've not already done so) and perform the following steps:

1. Download the `DevCertRequest` tool.
2. Create the CSR request file using `DevCertRequest`.
3. Apply for the developer certificate, which includes uploading the .csr request file.
4. Download the certificate.

Let's look at these steps in more detail.

Download the `DevCertRequest` tool

The `DevCertRequest` tool can be downloaded from the Symbian Signed site. `DevCertRequest` runs on a PC and is used to create a CSR request file that is used to submit a developer certificate request.

Create the CSR request file using `DevCertRequest`

Run `DevCertRequest` and follow the instructions in the tool to generate a CSR request file. You will be prompted to enter the capabilities that you need and also the IMEI numbers of the phones you intend to use (a SIS file signed with the developer certificate generated from the request can only be installed on the devices specified at this prompt).

Upload the request file

Once the CSR file has been generated by the `DevCertRequest` tool, follow the instructions on the Symbian Signed site to submit the request by uploading the CSR file you created. If your development certificate requests manufacturer capabilities then follow the path for requesting a phone manufacturer approved development certificate. You will be prompted as to what phone manufacturer must be contacted (e.g., Nokia or Sony Ericsson). You will need to click through an agreement for that particular manufacturer before the request can be submitted.

Download the certificate

Once approved you can download the developer certificate from the Symbian Signed website.

7.7.3 Signing Using the Developer Certificate

To sign your application with a developer certificate, use `signsis` as described previously. For example:

```
signsis app.sis app_dev.sis devcert.cer devcert.key
```

where `devcert.cer` and `devcert.key` are the developer certificate's certificate and private keys, respectively.

8

Asynchronous Functions and Active Objects

Although Symbian OS allows you to create pre-emptively scheduled threads via `RThread`, as Chapter 9 will describe, you'll find that in most programs you write you rarely need to create threads yourself – even where you would normally do so 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 programmer on this platform. This chapter looks at asynchronous programming on Symbian OS and describes how to use and implement asynchronous functions and active objects. Let's start by looking at what asynchronous functions are, and then describe active objects, which is the framework used to take full advantage of them.

8.1 Asynchronous Functions

You can identify an asynchronous function on Symbian OS by its inclusion of an argument of type `TRequestStatus`. `TRequestStatus` is a type

class (which encapsulates 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 *aSomeArg);
```

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 *aSomeArg, 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 of processor cycles. A more efficient option is to use `User::WaitForRequest(TRequestStatus& 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:

```
TRequestStatus status;
MyFunc(someArg, status);
User::WaitForRequest(status);
// MyFunc() is complete, status now contains the function status
```

`User::WaitForRequest()` waits at a special semaphore owned by the calling thread, known as the *request semaphore* (section 9.5 discusses semaphores in Symbian OS). 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 that 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. In effect, it makes an asynchronous function synchronous (sometimes called 'pseudo-synchronous'). 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. If your code provides a user interface, it is very undesirable to block on an asynchronous function, since it isn't clear how long it will take to complete, and in the time it is blocked, your application will be unresponsive to user input.

The best way of using asynchronous functions (and what they were really designed for) is through active objects. Active objects allow you to take full advantage of asynchronous functions by allowing your application to run other code while waiting on the completion of one or more asynchronous functions.

8.2 Introducing Active Objects

Active objects are classes derived from `CActive`, which itself derives from `CBase`, as you'd expect from a C class on Symbian OS. 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()`.

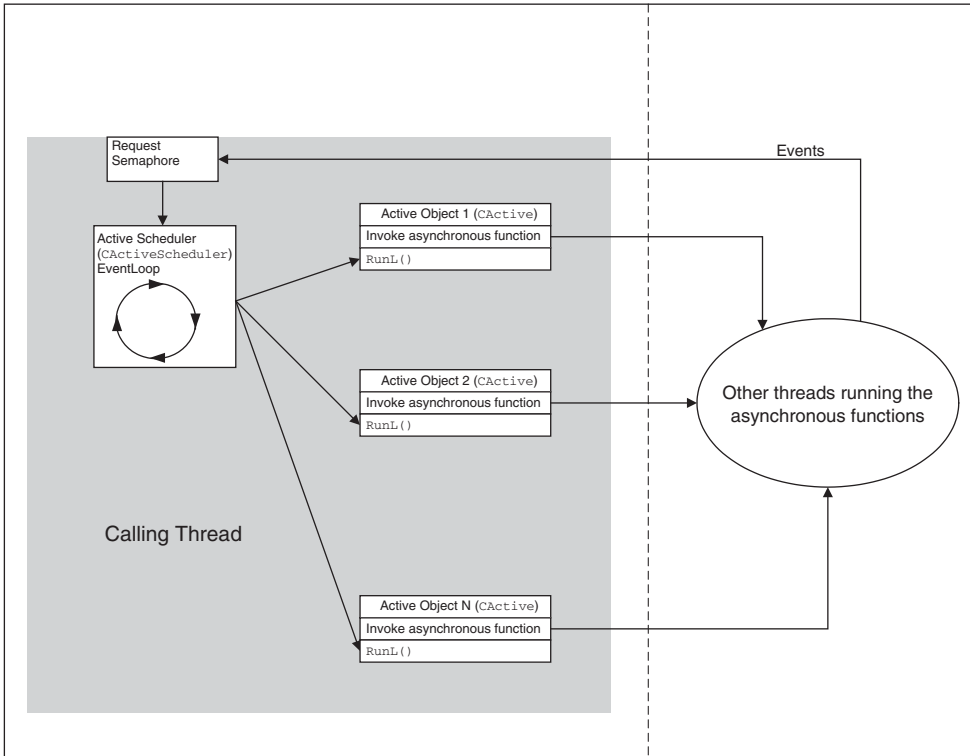


Figure 8.1 Active Object High-Level View

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.

Note that the `CActiveScheduler::Start()` is performed just one time at the beginning of starting a process. In many cases, such as in a GUI application, the active scheduler is started automatically by the application framework.

8.2.1 The Non-Pre-Emptive 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-pre-emptive 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 `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();
    void InvokeAsyncFunc(); // Some method to invoke the active object's
                           // associated asynchronous function.

protected:
    CMyActive();
    void ConstructL();
    // Overridden from CActive
    virtual void RunL(); // handles asynchronous function completion
    virtual void DoCancel(); // Cancels an outstanding
                           // asynchronous function call
    virtual TInt RunError(TInt aErr); // overridden
                                     // 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, `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 and passes the `iStatus` member variable of the `CActive` base class as the `TRequestStatus` parameter.
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. Override `CActive`'s `RunError()` method to handle leaves that occur in your `RunL()` event handler.
7. Create your active object's destructor, which should call the `CActive::Cancel()` method as part of cleanup.

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-phase constructor function, as discussed in section 4.5.13.

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 *aAo)`, where `aAo` 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 typically 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`, which is inherited from the `CActive` base class, 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 *aArg1, 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 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 completion result. 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 CMyCActive::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 outstanding asynchronous request. `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

If there is a possibility that the code in your `RunL()` function will leave, then you should always override the `RunError(TInt aErr)` method to handle the error. The `aErr` parameter will contain the leave code. You should return `KErrNone` once the error is handled to prevent the active scheduler from attempting to handle it. Reference section 8.4 for more details on active object error handling.

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 just add your own active objects and they will become part of the existing pool of active objects, without needing to call any of the `CActiveScheduler` API methods described in the next section.

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, which has no active scheduler installed by default if it does not use the application framework.

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, of type `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 a panic is raised (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).

So, before you start the scheduler and enter 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 for ever, 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 on the request semaphore, it unblocks and determines which active object the signal belongs to. It does this by looking for an active object that has both its `iActive` variable (the `TBool` member of the `CActive` base class 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 `CActiveScheduler::Stop()` function is invoked (in some active object's `RunL()` function), at which

time the event loop is exited and `CActiveScheduler::Start()` returns.

Pseudo-code for the active scheduler event loop (`CActiveScheduler::Start()`) is shown here:

```
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_panic();
}
} 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 panic occurs for the thread.

The pseudo-code shown in the previous example 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. A panic 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 a panic, 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 raise an `E32USER-CBase 47` panic.
- `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 section 8.3.2. 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 that 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 a panic, 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 a customizable method of `CActiveScheduler`, prototyped as:

```
void Error(TInt aErr)
```

where `aErr` is the value with which `RunL()` left or the value returned from an active object's `RunError()` method if it failed to handle a leave in `RunL()`.

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 panic. Since the default implementation of `CActive::RunError()` is to return the `RunL()` leave code, then a E32USER-CBase 47 panic is what you will get on a leave if you do not override any of the error-handling functions. It is very undesirable for your user to receive a panic like this, since it will close the application. You are certain not to pass Symbian Signed testing if it is observed as part of the testing process. So it makes sense to either ensure that an active object's `RunL()` method cannot leave, or implement an error handler in `RunError()` to ensure that the error is not propagated to the default active scheduler.

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, before the completion event). `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 8.2.2), your derived active object class must override `DoCancel()` to cancel any pending asynchronous

requests, and `DoCancel()` is never called directly, but is called by `CActive`'s `Cancel()` function. `CActive::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 call `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 for beginners, when canceling an asynchronous function, is to call `User::WaitForRequest()` 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();
}
```

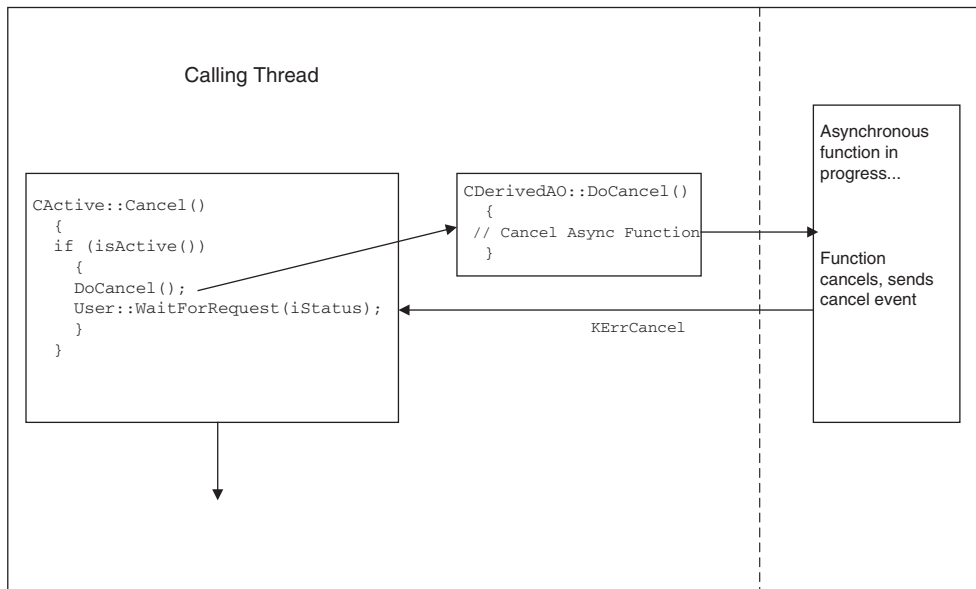


Figure 8.2 `Cancel()` and `DoCancel()` Operation

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` panic. Therefore the destructor in your derived 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. 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. A Stop item is also 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. The code below shows the declaration of our active object class, called `CCountdown`.

```
class CCountdown : public CActive
{
public:
    static CCountdown* NewL(CSimpleExAppView& aAppView);
    ~CCountdown();
    void StartCountdown();
    void Stop();
protected:
    CCountdown(CSimpleExApp& aAppView);
    void ConstructL();
    void RunL();
};
```

```

    void DoCancel();
private:
    TInt iCount;
    RTimer iTimer;
    TInt iInterval;
    CSimpleExAppView& iAppView;
};

```

The implementation of the CCountdown active object is shown as follows:

```

CCountdown* CCountdown::NewL(CSimpleExAppView& aAppView)
{
    CCountdown* self = new(ELeave) CCountdown(aAppView);
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop(self);
    return self;
}

CCountdown::CCountdown() // Construct high-priority active object
: CActive(CActive::EPriorityUserInput), iAppView(aAppView)
{
    iInterval = 1000000; // 1 second interval
    // Add to active scheduler
    CActiveScheduler::Add(this);
}

void CCountdown::ConstructL()
{
    User::LeaveIfError(iTimer.CreateLocal());
}

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()
{
    _LIT(KFormatString, "-%d-");
    TBuf<50> buff;
    buff.Format(KFormatString,iCount);
    iAppView.UpdateScreenText(buff);
    if (iCount)
    {
        iTimer.After(iStatus,iInterval);
        SetActive();
    }
}

```



```

        --iCount;
    } else
    {
        // Replace text with "Simple Example" (set in KSimpleExText).
        iAppView.UpdateScreenText(KSimpleExText);
        _LIT(KMessage, "Start Selected!");
        CEikonEnv::Static()->AlertWin(KMessage);
    }
}

void CCountdown::Stop()
{
    iCount=0;
    iAppView.UpdateScreenText(KSimpleExText);
    Cancel();
}

void CCountdown::DoCancel()
{
    iTimer.Cancel();
}

```

Let's step through the methods and discuss them. `CCountdown::NewL(CSimpleExAppView& aAppView)` is the static `NewL()` method that constructs the active object and calls the secondary constructor for it. A reference to `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. `StartCountdown()` 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. This is done since, as I discuss in section 8.9.3, you cannot have two asynchronous functions in progress at a time. While this example will ignore a start command if the countdown is in progress, another option would be to cancel the asynchronous function in progress first (by calling `Cancel()`), resetting `iCount` to 10, and then start the countdown again by reissuing the timer asynchronous function call.

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 `CCountdown::RunL()` method. `CCountdown::RunL()` calls a new method, implemented in the application view class, called `UpdateScreenText()` 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 `SetActive()`. This will cause `RunL()` to be re-entered 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. Furthermore, note that I did not override the active object `RunError()` function because it is not possible for the `RunL()` in this example to leave.

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 S60 program only. However, active objects are platform-independent, so the same modifications can be made to the UIQ version.

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.

Here is the modified resource file, which adds the Stop item to the menu:

```
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_ui.cpp` and `simpleEx_view.cpp`.

The following shows the modified `CSimpleExAppView` in `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();

    gc.Clear();
    const CFont* font = iEikonEnv->TitleFont();
    gc.UseFont(font);
    TRect drawRect = Rect();
    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 `UpdateScreenText()` 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.

The following shows the modified `CSimpleExAppUi` (in `simple-Ex_ui.cpp`):

```
void CSimpleExAppUi::ConstructL()
{
    BaseConstructL(CAknEnableSkin);
    iAppView = CSimpleExAppView::NewL(ClientRect());
    iCountdown = CCountdown::NewL(*iAppView);
    iAppView->UpdateScreenText(KSimpleExText);
}
CSimpleExAppUi::~CSimpleExAppUi()
{
    delete iAppView;
    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. In the header file, I also 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-pre-emptive) 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 call 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 short 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 (or if you must block, keep it to 10 ms or less). The GUI event handler, `HandleCommandL()`, is actually invoked from a `RunL()` of an active object that the application UI class inherits from. So if you delay for some reason when handling a GUI event, your GUI will become non-responsive 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 their use is generally discouraged, and beyond the scope of this book.

8.9.2 Avoid Stray-Signal Panics

Any Symbian programmer who has developed active objects will probably have encountered a `E32USER-CBase 46` panic. This is the stray-signal panic 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 panic 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 panic.
- **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 panic 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 a stray-signal panic.

8.10 Using Active Objects for Background Tasks

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(TTimeIntervalMicroSeconds32, 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.

Following is the header file.

```
#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);
protected:
    void RunL();
    CPrimaryTask() : CTimer(EPriorityStandard){ }
private:
    TInt iInterval;
    TInt iRepeatCnt;
};

class CBackground : public CTimer
{
public:
    static CBackground* NewLC();
    void StartBackground(TInt aInterval);
protected:
    void RunL();
    CBackground() : CTimer(EPriorityLow){ }
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. The following shows their implementation, and the main program.

```

#include <e32base.h>
#include <e32cons.h>
#include <e32debug.h>
#include "active.h"
/*-----
    Active Object: CPrimaryTask
-----*/
CPrimaryTask* CPrimaryTask::NewLC()
{
    CPrimaryTask* self=new (ELeave) CPrimaryTask;
    CleanupStack::PushL(self);
    self->ConstructL(); // Need to call CTimer::ConstructL()
    CActiveScheduler::Add(self);
    return self;
}

void CPrimaryTask::StartRunning(TInt aRepeat, TInt aInterval)
{
    iRepeatCnt=aRepeat;
    iInterval=aInterval;
    CTimer::After(iInterval);
}

void CPrimaryTask::RunL()
{
    {
        if (iRepeatCnt-- > 0)
        {
            _LIT(KFirstActiveWaiting, "CPrimaryTask: RunL");
            _LIT(KFormatWithNewLine, "%S\n");
            gConsole->Printf(KFormatWithNewLine, &KFirstActiveWaiting);

```

```

        RDebug::Print(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();// Need to call CTimer::ConstructL()
    CActiveScheduler::Add(self);
    return self;
}

void CBackground::RunL()
{
    _LIT(KBackgroundTaskMsg,"CBackground RunL");
    _LIT(KFormatWithNewLine, "%S\n");
    gConsole->Printf(KFormatWithNewLine,&KBackgroundTaskMsg);
    RDebug::Print(KBackgroundTaskMsg);
    ...// do task processing here
    After(iInterval);
}

/*-----
   E32Main() Entry point
-----*/
TInt E32Main()
{
    __UHEAP_MARK;
    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
    __UHEAP_MARKEND;
    return 0; // and return
}

/*-----
   Active Object Example main function
-----*/
void ActiveExampleL()

```

```

{
/*-----
   Create a full screen console
-----*/
_LIT(KConsoleTitle,"Console");
gConsole=Console::NewL(KConsoleTitle,TSize(KConsFullScreen,
KConsFullScreen));
CleanupStack::PushL(Console);
gConsole->Printf(_L("Active Object Example"));
gConsole->Printf(_L("Press Key to Begin"));
gConsole->Getch();
/*-----
   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); // Clean up backAct, //firstAct and
 mySched

_LIT(KActiveComplete,"Scheduler exited, cleanup");
_LIT(KFormatWithNewLine, "%S\n");
gConsole->Printf(KFormatWithNewLine,&KActiveComplete);
RDebug::Print(KActiveComplete);
_LIT(KPressKeyToExit,"[Press any key] to exit");
gConsole->Printf(KPressKeyToExit);
gConsole->Getch();
CleanupStack::PopAndDestroy(gConsole);
}

```

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 0xE00000298
SOURCEPATH  .
SOURCE      active.cpp
USERINCLUDE .
SYSTEMINCLUDE \Epoc32\include
LIBRARY     euser.lib efsrv.lib
```

This example both writes its output to the console and uses `RDebug` to write the output to the `epocwind.out` file and/or the IDE output console as described in section 5.6.4.

An example of the output from this example is shown below.

```
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CBackground RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CBackground RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CBackground RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CBackground RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CBackground RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
CBackground RunL
CPrimaryTask: RunL
CPrimaryTask: RunL
Scheduler exited, cleanup
```


9

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 APIs Symbian OS provides. I'll also describe how to synchronize, and communicate between threads, using shared memory regions, 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 may 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, on Symbian OS, implementing threads in your program is generally discouraged in favor of using active objects (see section 9.2).

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 objects and the client–server pattern, operate (these are covered in Chapters 8 and 10, respectively).

9.1 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.

9.1.1 An Example Process

The following example shows an example process that displays an information message to the emulator screen every two seconds, for 100 iterations, once it's started:

```
#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.

Here is the MMP file that builds the process in an executable called `MyProc.exe`:

```
// EXE MMP file
TARGET      MyProc.exe
UID         0 0xE000127F
TARGETTYPE  exe
SOURCEPATH  ..\src
SOURCE      MyProc.cpp
USERINCLUDE .
USERINCLUDE ..\include
SYSTEMINCLUDE \Epc32\include
LIBRARY     euser.lib
```

9.1.2 Launching a Process

The following code loads and runs an instance of `MyProc.exe`:

```
void LaunchProcessL()
{
    _LIT(KMyExeFile, "MyProc.exe");
    RProcess proc;
```

```

/* This will launch the MyProc.exe, passing the specified command line
   data to it (NULL in this case)*/
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. Note though that to terminate a process other than your own, you will need the PowerMgmt capability (see section 9.1.8). You will also not be able to change the priority of protected system processes.

The code above instantiates an RProcess object and then invokes its Create() method to load the EXE file specified as the first argument for the new process.¹ 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 does not exist) 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, which is overloaded to provide several creation options (refer to the SDK API documentation for details). The RProcess::Create() function used in the example is prototyped as follows:

```

TInt RProcess::Create(const TDesC& aExecutableFile,
const TDesC& aCommand, TOwnerType aType=EOwnerProcess)

```

aExecutableFile contains the name of the process executable file. The aCommand parameter is a descriptor containing a command line argument that specifies data to be passed to the process when it is launched. The aType parameter specifies handle ownership and has a default value of EOwnerProcess to indicate that this RProcess handle

¹ Note that the name of the process is specified with no path information because it is unnecessary. All binaries execute from the \sys\bin directory of the phone for security reasons, as described in Chapter 7.

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.

9.1.3 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, "MyProc.exe");
    _LIT(KMyExeFileCmd, "arg1 arg2");
    RProcess proc;
    User::LeaveIfError(proc.Create(KMyExeFile, KMyExeFileCmd));
    proc.Resume(); // start the process running
}
```

This will pass the argument string "arg1 arg2" to the process created. To retrieve the entire command string argument within the process itself, you can call `User::CommandLine()` as the following shows:

```
TBuf<200> cmdLine;
User::CommandLine(cmdLine);
```

Or you can use the class `CCommandLineArguments` to retrieve each argument individually from the process, as the following example shows:

```
CCommandLineArguments *args=CCommandLineArguments::NewL();
for (TInt i=0;i<args->Count();i++)
{
    TPtrC argPtr(args->Arg(i));
    _LIT(KArg, "arg %d = %S");
    console->Printf(KArg,i,argPtr);
}
```

Which would print:

```
arg 0 = c:\sys\bin\myproc.exe
arg 1 = arg1
arg 2 = arg2
```

Note that argument zero is always the process name.

You must include `bacline.h` and link to library `bafl.lib` to use the `CCommandLineArguments` class.

Another way to pass data to a process is by calling the `RProcess::SetParameter()` method. This is good for passing handles to a process, although you can also pass integers and descriptors. Section 9.4 shows an example of using `SetParameter()` to pass a handle to a process.

9.1.4 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 (of type `TProcessId`) or by passing its ASCII name (these are described shortly).

The following code opens a handle to a process by its process ID:

```
RProcess myProcess;
myProcess.Open(procId);
if (rc != KErrNone)
{
    /* open failed, handle error */
}
/* ... */
myProcess.Close(); // close handle when finished
```

The variable `procId` is the ID 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()`. As I mentioned before, closing the handle does not terminate the actual process it represents. We'll discuss how to do that shortly, in section 9.1.8.

`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`.

9.1.5 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 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.

9.1.6 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 next example shows how you can use `TFindProcess` to query a list of running processes on the phone.

The default match string of `TFindProcess` is '*', so this example simply traverses the list of all processes in the system.

```

TFindProcess fp;
TFullName procName;
while (fp.Next(procName) != KErrNone)
{
    _LIT(KProcessName, "process: %S\n");
    console->Printf(KProcessName, &procName);
}

```

Let's extend the example a bit and print some information about each process using the `RProcess GetMemoryInfo()` method:

```

void ListProcessesL()
{
    TFindProcess fp;
    TFullName procName;
    TProcessMemoryInfo memInfo;
    RProcess process;
    while (fp.Next(procName) != KErrNone)
    {
        User::LeaveIfError(process.Open(procName));
        process.GetMemoryInfo(memInfo);
        _LIT(KProcessFormatName, "Process: %S\n");
        console->Printf(KProcessFormatName, &procName);
        _LIT(KCodeBaseFormat, "    code base=%x\n");
        console->Printf(KCodeBaseFormat, memInfo.iCodeBase);
        _LIT(KCodeSizeFormat, "    code size =%x\n");
        console->Printf(KCodeSizeFormat, memInfo.iCodeSize);
        _LIT(KInitDataSizeFormat, "    initialized data size=%x\n");
        console->Printf(KInitDataSizeFormat,
            memInfo.iInitialisedDataSize);
        _LIT(KUninitSizeFormat, "    uninitialized data size=%x\n");
        console->Printf(KUninitSizeFormat,
            memInfo.iUninitialisedDataSize);
        process.Close();
    }
}

```

A sample portion of output from this example showing the first few processes follows:

```

Process: ekern.exe[100041af]0001.
code base=f8013000.
code size =3b844.
initialized data size=1bc.
uninitialized data size=14dc.
Process: efile.exe[100039e3]0001.
code base=f813e698.
code size =1aca4.
initialized data size=d0.
uninitialized data size=22c0.
Process: domainSrv.exe[1020e406]0001.
code base=f8172238.
code size =1fdc.
initialized data size=0.
uninitialized data size=0.

```

9.1.7 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 currproc;  
currproc.SetPriority(EPriorityHigh);
```

By default, the `RProcess` class constructor opens a handle to the currently running process. I could have skipped declaring `currproc` altogether and replaced the last two lines with the line:

```
RProcess().SetPriority(EPriorityHigh);
```

This calls both the constructor and the method (this is a common practice).

You can also set the process priority at build time, and not use `RProcess` at all. To do this, add the `epocprocesspriority` keyword to your EXE'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.

9.1.8 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()`. The method takes one integer argument – a code to indicate the reason the process was terminated through a return value. A process can kill itself, or any other process it has a handle to.

Your program will need the `PowerMgmt` extended capability and be Symbian Signed in order to perform a `Kill()` on a process if it is not the process in which the current thread is running. Otherwise the function will return `KErrPermissionDenied (-46)`. Please see Chapter 7 for more details about platform security.

`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 panic; `aReason` gives the specific error number.

9.1.9 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 reason. `ExitReason()` returns the value returned by the `E32Main()` function of the process for a normal exit, the reason code passed to the `Kill()` method, if `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.

9.1.10 Signaling when a Process Ends

The `RProcess::Logon()` method can be used to wait for a process to complete as follows:

```
void StartProcessWaitEndL()
{
    RProcess proc;
    User::LeaveIfError(proc.Create(MyExeFile));
    TRequestStatus stat;
    proc.Logon(stat);
    proc.Resume();
    // Thread is executing. Can add code here to run in parallel...
```

```
User::WaitForRequest(stat); // 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 simplicity here it is sufficient to say that they always return immediately, and send an event at a later time, when the function actually completes. On completion, the `TRequestStatus` variable contains the status of the completion.

To realize the full power of asynchronous functions, you should use an active object to set up a callback to be run when the function completes. But you can also simply wait for completion of the function via `User::WaitForRequest()`, as we did in the previous example, because 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.

9.1.11 Other Symbian OS Process Facts

The following are other facts concerning processes:

- As described in Chapter 3, switching between threads in different processes (and thus requiring a process switch) is expensive compared with 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.
- GUI applications are processes.
- The emulator does not actually run multiple Windows processes since it executes completely as a single process. However, the emulator will simulate processes using Windows threads (Symbian OS v9 does this transparently when you use the `RProcess` API).

9.2 Using Threads on Symbian OS

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, and others such as synchronization and re-entrancy.

However, you may find that creating your own threads is the best solution in some situations. Having an understanding of the way they work also 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. Like `RProcess`, `RThread` is also instantiated directly, and usually on the stack.

Below is an example that creates and starts a thread using `RThread`:

```
TInt ThreadFunc(TAny *)
{
    for (TInt i=0; i<10; i++)
    {
        _LIT(KThread, "Thread");
        User::InfoPrint(KThread);
        User::After(4000000);
    }
    return(0);
}

void StartThreadL()
{
    RThread thd;
    _LIT(KMyThread, "MyThread");
    User::LeaveIfError(thd.Create(KMyThread,
        ThreadFunc, KDefaultStackSize, NULL, NULL));
    thd.Resume();
}
```

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 to the emulator screen (the message is seen at the top right-hand corner, and displays for a couple of seconds then disappears) 10 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 causing a panic. For example, if you have a simple console application, you cannot pass the `CConsoleBase` pointer to the second thread when you start it, for it to use to write to the screen. As already mentioned, you also cannot draw to the screen from a thread other than the main thread of the GUI application.

9.2.1 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 overloads 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. This name will also appear in the 'Application Closed' dialogs if a panic 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. Note that the process it was running in will continue to execute.
- `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 versions 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.

9.2.2 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.

The following code will open the thread created in the previous example, 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.

```
RThread thd;
_LIT(KMyThread, "MyThread");
TInt rc=thd.Open(KMyThread);
if (rc != KErrNone)
{
    /* handle open error */
}
if (thd.ExitType() == EExitPending)
    thd.Suspend();
...
thd.Resume(); // continue thread execution
```

Unlike processes, the system does not append any numbers to the end of thread names so 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 section 9.1.6, 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 passing 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. As in the case of processes, you also need a way of supplying this ID at runtime, 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.

9.2.3 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 9.1 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 9.2 shows the absolute thread values that can be used, and how they relate to process priority values.

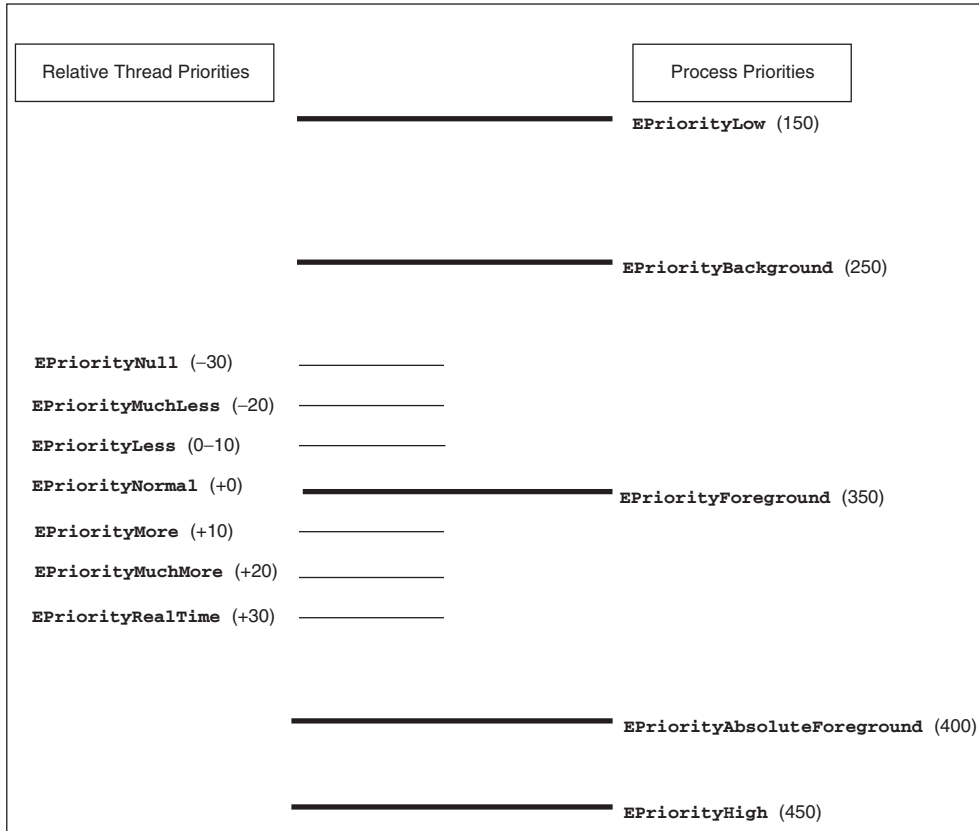


Figure 9.1 Relative Thread and Process Priorities

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.

9.2.4 Terminating a Thread

You can use `RThread::Kill(TInt aReason)` to terminate another thread in the current process, either remotely or from within the thread itself. As in `RProcess`, `RThread` also 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.
- `EExitPanic` means the thread ended due to a panic.
- `EExitPending` means that the thread is still running.

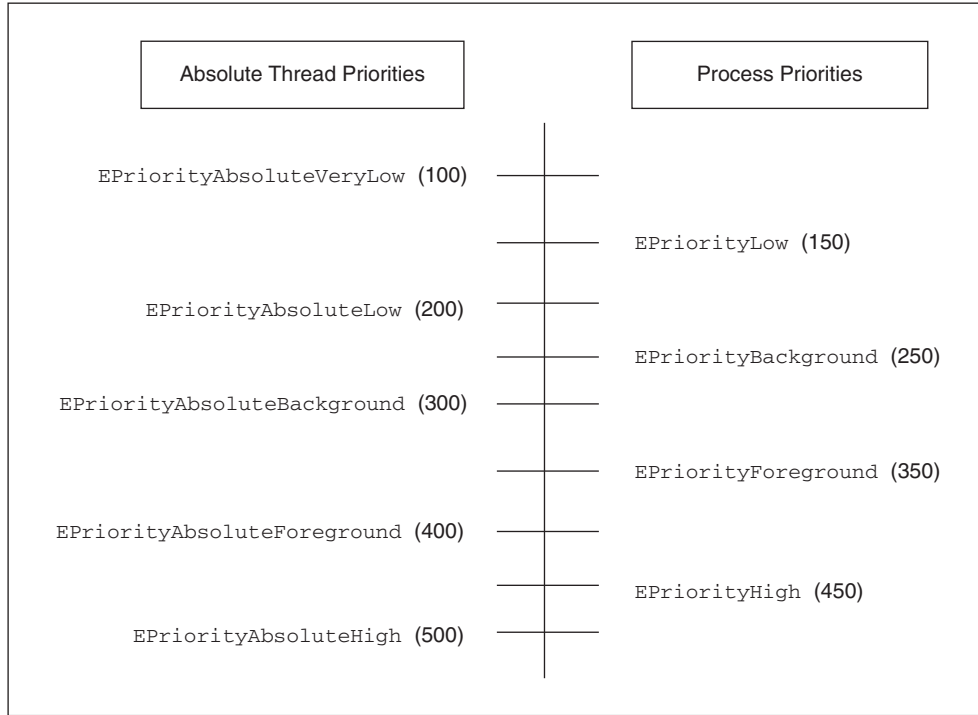


Figure 9.2 Absolute Thread and Process Priorities

`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.

9.2.5 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 a call to `User::WaitForRequest()` or use an active object such that the active object's `RunL()` method is called when the thread completes.

9.3 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 get a system panic. 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.

There are a number of ways by which processes can share data, such as through shared memory chunks, Publish and Subscribe, the Central Repository, message queues, and inter-process communication (IPC). IPC using the client–server framework will be discussed in Chapter 10, and the new memory chunks are discussed in the next section. The other data transfer techniques are beyond the scope of this book, but are documented in the SDK documentation, and in documentation available on the Symbian Developer Network.

9.4 Memory Chunks

Symbian OS provides support for shared memory regions that can be accessed directly across multiple processes. These shared memory regions are known as *memory chunks*. You can create your own memory chunks or access existing memory chunks by using the RChunk API class.

In order for your memory chunks to be shared between processes, they must be created as global (you can also have local memory chunks available just to a single process). 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 – that is, actual RAM that resides on the smartphone. Virtual memory is mapped to physical memory by the CPU's 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 9.3 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 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 aNewSize)` method, where `aNewSize` 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)
```

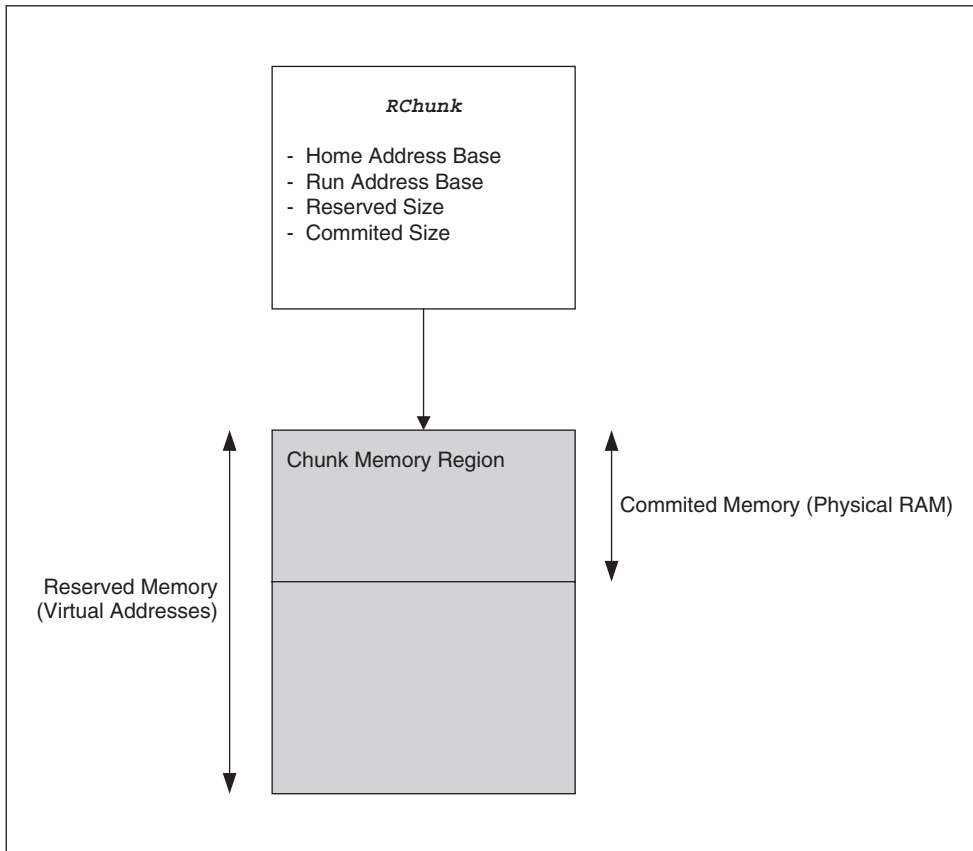


Figure 9.3 Layout of the Memory Chunk

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 the example in section 9.1.6 to output all the global chunk names in the system using `TFindChunk`).

When a process is finished with the chunk, `RChunk::Close()` must

be called. When the last reference to the global chunk is called, the global chunk itself is automatically deleted.

A more secure way of using global chunks between processes is to create the chunk without a name and reference it by its handle. This is more secure since the chunk cannot be found and used by another process via `TFindChunk` using this method. You can pass the handle to the chunk to another process either via client-server IPC or setting the handle as a parameter to the other process.

Here is an example of passing the handle as a parameter to a process:

```
const TInt KChunkIndex=0;
const TInt KChunkSize=0x1000;
RChunk chk;
chk.CreateGlobal(KNullDesC,KChunkSize,KChunkSize);
...
RProcess proc;
LIT(KProcName,"myproc.exe");
proc.Create(KProcName);
proc.SetParameter(KChunkIndex,chk);
proc.Resume();
```

This code creates an anonymous global chunk, and sets the handle for the chunk as a parameter to the `myproc.exe` process by calling the `RProcess::SetParameter()` method. The first argument of `SetParameter()` is the index of the parameter (also known as the slot). The second argument is the handle. After setting the parameter, the process is started with `RProcess::Resume()`.

Now let's see `myproc.exe` retrieve that handle and get access to the chunk:

```
// myproc.exe
RChunk chunk;
chunk.Open(KChunkIndex);
// chunk now points to the global chunk...
```

The version of `RChunk::Open()` used in this example takes a `TInt` as an argument, which represents the index of the parameter passed to the process where the `RChunk` handle was set. Error checking was omitted from these examples for simplicity.

9.4.1 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 aMaxSize)` 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 use of them internally.

9.5 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 four basic thread-synchronization functions: semaphores, mutexes, critical sections, and the RThread and RProcess Rendezvous() method.

9.5.1 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 that 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(KMySemName, "My Semaphore");
// Thread A:
/* ... */
RSemaphore sem;
 TInt rc=sem.CreateGlobal(KMySemName,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(KMySemName);
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, 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:

```

//ThreadA
RSemaphore sem;
TInt rc=sem.CreateGlobal(KMySemName,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(KMySemName);
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()`).

As with all Symbian OS R classes, when you are finished with an `RSemaphore` handle, you must call the `Close()` method for cleanup.

9.5.2 Creating and Opening Semaphores

The syntax for creating a global semaphore is:

```
TInt RSemaphore::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()`).

9.5.3 The Use of Semaphores in Symbian OS

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, described in the previous chapter, which is in turn used in the client–server framework. When an asynchronous function completes, 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 generated a completion event. 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 asynchronous functions in more detail and describes how the request semaphore is used by the active object framework.

9.5.4 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 has 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`.

9.5.5 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 re-entrant code section.
crit.Signal();
```

In this example, the non-re-entrant 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`.

9.5.6 Using Rendezvous ()

Both `RThread` and `RProcess` have a method called `Rendezvous()` that is used to indicate when a thread or process has reached a particular point in its execution. This method is commonly used when you start another process and want to block the original thread until the process it has kicked off is started up and initialized. Below is an example:

```
// main process
void StartProcessL()
{
```

```

RProcess proc;
User::LeaveIfError(proc.Create(KMyExeFile));
TRequestStatus stat;
proc.Rendezvous(stat);
proc.Resume();
// Thread is executing. Can add code here to run in parallel...
User::WaitForRequest(stat); // blocks until proc calls
                           // RProcess::Rendezvous()
                           // or process exits
User::LeaveIfError(stat.Int()); // leave if stat not KErrNone
}

// KMyExeFile Process
#include <e32base.h>
TInt E32Main()
{
    // Start up boilerplate code
    InitializeProcess(); // some function that initializes the process
    RProcess::Rendezvous(KErrNone); // signal that process is initialized
    // Clean up boilerplate code
    return(0);
}

```

There are two versions of `Rendezvous()`, one with the prototype of `void RProcess::Rendezvous(TRequestStatus& aStat)` and one with the prototype of `void RProcess::Rendezvous(TInt aRes)`. The process that starts `KMyExeFile` uses the first version of the function and, as you can see, it works in a similar manner as `Logon()` in that you start a process and then wait for an asynchronous event (which could also be done via an active object instead of `User::WaitForRequest()`, which I've used in the example above for simplicity). For `Rendezvous()` the asynchronous event will occur if:

- The process referenced by the process handle calls `RProcess::Rendezvous(TInt aRes)`, where `aRes` is a status. In this case, the asynchronous `TRequestStatus` variable will be set to `aRes`.
- The process referenced by the process handle exits.
- A panic occurs in the process referenced by the process handle.

In the example, `RProcess::Rendezvous(KErrNone);` is called by the `KMyExeFile` process after the process initializes, causing the asynchronous `Rendezvous()` function started in the parent process to complete with `stat` set to `KErrNone`.

The `RThread::Rendezvous()` method works in the same way as `RProcess::Rendezvous()`, but at the thread level.

`Rendezvous()` is commonly used when a client starts up a server process and wants to know when the server process is running and ready to accept commands. We will come to this shortly in Chapter 10.

10

Client–Server Framework

Symbian OS uses servers to manage resources centrally on behalf of one or more clients. A server does not normally have a graphical user interface (GUI), and, in most cases, runs in its own process to provide protection and modularity. (Servers and clients are always in different threads, although you can have multiple servers in the same process, or even in the same thread.) 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.

A client program uses a server through a client-side implementation class – the programmer invokes the functions of the server through methods in the client class. Each method sends the appropriate command to the server and receives the command results back to return to the caller. 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.

Symbian OS uses servers to implement much of its functionality. In fact, many of the Symbian OS API classes are client classes for servers. Many asynchronous functions are also 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 on smartphones with touchscreen input. GUI applications are clients of the window 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 system 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).

The fundamental classes that comprise the Symbian OS client–server framework are public so that you can implement your own servers, along with client-side classes. This chapter gives a very basic introduction to creating client–server software in Symbian OS. It describes the applicable classes in the client–server framework, and walks through a simple example. Even if you do not implement your own server, understanding this information will help when writing software that uses servers.

10.1 Client–Server Overview

A client command is submitted to the server, via the kernel, as a message. Servers process the messages one at a time – that is, 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 submit commands. 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 `CSession2`), and this allows context, such as state information, to be saved for that client while commands are being processed.

Note that some of the server-side classes end in a '2'. This is to indicate that it is version two of the client–server framework. Version 2 of this

framework was introduced in Symbian OS v8 and is therefore in Symbian OS v9 also. It replaced the older framework used in previous OS versions, and was introduced to make client–server communication more secure. This chapter covers version 2 of the client framework only and thus only applies to Symbian OS v8 and v9 (although the earlier framework is not that much different).

10.2 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-side implementation class.

Figure 10.1 shows the basic classes used in the client–server framework.

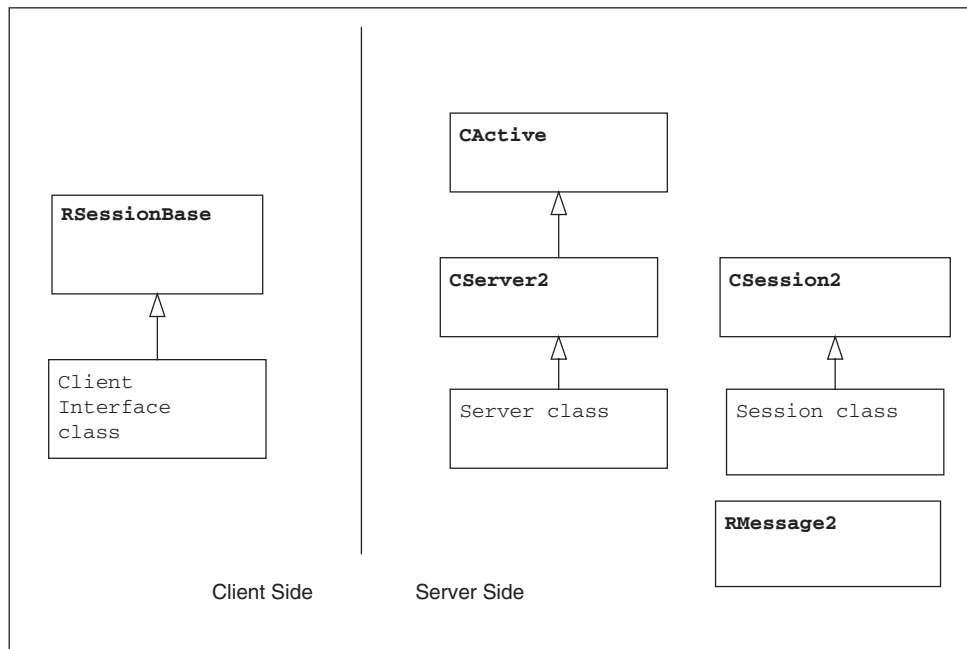


Figure 10.1 Key Client–Server Classes

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:

- `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 `CServer2`. This is the main server class that initially receives all server messages, establishes new sessions, and routes received commands to the appropriate session. `CServer2` is derived from `CActive`. There is just one instance of your `CServer2`-derived class for each server. The key methods of `CServer2` are:

- `NewSessionL()`
Your `CServer2`-derived class should implement this virtual function to create a new session (i.e., to instantiate and return an instance of your `CSession2`-derived class). `NewSessionL()` is called when a client requests a session with the server (via the client method `CreateSession()`).
- `StartL()`
This `CServer2` 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.

The class `CSession2` is the base class for a session object. An instance of a `CSession2`-derived class is created for each client session (by `CServer2`'s `NewSessionL()`). The session class handles the commands sent from the client. You override `CSession2`'s virtual `ServiceL()` method to handle the commands.

`RMessage2` is a server-side class that represents the message from the client, with methods to access it. It is a concrete class, and you do not derive from it yourself, but use it directly. The message consists of a command code and four 32-bit arguments. The messages are passed into the server by the kernel, which creates them from the commands sent from the client through `RSessionBase::SendReceive()` or `RSessionBase::Send()`. Although not shown in Figure 10.1, `RMessage2` is derived from a class called `RMessagePtr2`, which handled the details of the communication between the client and the server.

10.3 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). There are also performance issues to consider, since a context switch can occur for each server command. When servers reside in their own processes, as they typically do, this can be even more expensive since a process switch is relatively time-consuming, as discussed in section 3.5.7. If you do decide to use a client-server approach, it’s a good idea to implement fewer server commands passing more data rather than the reverse, in order to minimize context switches.

10.3.1 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();
    TInt BackSpace(TInt aNumChars);
    TVersion Version() const;
private:
    TInt StartServerProcess();
};
```

The client class is derived from `RSessionBase`, which inherits from `RHandleBase`.

Connecting to 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`.

The following shows the `Connect()` method for our example:

```
_LIT(KTextBuffServerProcess, "textbuffserver.exe");
TInt RTextBuff::Connect()
{
    TInt res;
    res=CreateSession(KTextBuffServerName,Version());
    if (res == KErrNotFound || res == KErrServerTerminated)
    {
        res = StartServerProcess();
        if (res == KErrNone || res == KErrAlreadyExists)
            res=CreateSession(KTextBuffServerName,Version());
    }
    return(res);
}

TInt RTextBuff::StartServerProcess()
{
    RProcess proc;
    TInt res = proc.Create(KTextBuffServerProcess,KNullDesC);
    if (res == KErrNone)
    {
        TRequestStatus stat;
        proc.Rendezvous(stat);
        proc.Resume(); // start the process running
        User::WaitForRequest(stat); // wait for server to be
                                   // ready for commands

        res = stat.Int();
        proc.Close(); // finished with the handle
    }
    return(res);
}
```

In `Connect()` we create our client session by calling `RSessionBase::CreateSession()`:

```
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() 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;
```

(The complete header is shown later in this section.)

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. Hence, 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 `CServer2` 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 start it. Likewise, the server is started if `CreateSession()` returns `KErrServerTerminated`, which indicates, as the constant implies, that the server had terminated unexpectedly. We start the server by calling `StartServerProcess()`, which uses `RProcess` to launch the server process. We then call `CreateSession()` again. `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 preallocated 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.

`StartServerProcess()` creates and starts the process via `RProcess`. In `StartServerProcess()`, after the process is created, I call `proc.Rendezvous(stat)`, start the process with `proc.Resume()`, and then wait for the `Rendezvous()` function to complete using `User::WaitForRequest(stat)`. As described in section 9.5.6, `Rendezvous()` completes when the process (in this case the server process) calls `RProcess::Rendezvous()` or if the process terminates. As you will see shortly, our server process calls `RProcess::Rendezvous(KErrNone)` when the server is up and ready to receive commands. When the server calls the `Rendezvous()` function, our `User::WaitForRequest(stat)` will complete and `StartServerProcess()` will return. If `StartServerProcess()` returns either `KErrNone` or `KErrAlreadyExists`, `CreateSession()` is attempted again (which should succeed this time). The reason the value `KErrAlreadyExists` is checked also is that it is possible that another client could be starting the server at the same time, and beats this one to the actual process creation.

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()`.

The following shows the code for the `AddText()`, `GetText()`, `BackSpace()`, and `Reset()` functions on the client side:

```
TInt RTextBuff::AddText(const TDesC& aText)
{
    TIpcArgs args(&aText);
    TInt rc = SendReceive(ETextBuffAddText, args);
    return rc;
}
void RTextBuff::GetText(TDes& aText)
{
    TIpcArgs args(&aText);
    SendReceive(ETextBuffGetText, args);
}
```

```

    }
void RTextBuff::Reset()
{
    SendReceive(ETextBuffReset);
}
TInt RTextBuff::BackSpace(TInt aNumChars)
{
    TIpcArgs args(aNumChars);
    TInt rc=SendReceive(ETextBuffBackSpace,args);
    return rc;
}

```

The methods here 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, const TIpcArgs& aArgs)
```

`aCommand` indicates the command that the server session is to process. You define your set of commands for the server in a header file as you will see shortly. The server side uses the command to determine what action it should perform.

`TIpcArgs` is a class that can package up to four 32-bit arguments that can be sent to the server along with the command. The arguments can be in the following form:

- 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).

In the example, we use `TIpcArgs` to package only one argument. You can specify more by simply adding (up to four) the arguments to the constructor (e.g., `TIpcArgs args(myArg1, myArg2, myArg3, myArg4)`).

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, const TIpcArgs& aArgs, 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 aArg, TRequestStatus& aStat)
{
    TIPCArgs args(aArg);
    SendReceive(ESomeFunction, args, 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 (which here is the name of the process, but this doesn't have to be the case¹), 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.
};
```

10.3.2 Server Implementation

Following is the server's include file (`tbuffserver.h`), which contains the server and session class definitions:

¹ Note also that system server names start with '!', which are considered protected names. A server needs the `ProtSrv` extended capability to have a name that starts with '!'. This protection is in place to prevent third-party servers from impersonating system servers.

```

#include "textbuff.h"
enum TTextBuffPanic
{
    EInvalidCommand,
    EInvalidDescriptor,
    EServerInitError
};
const TInt KMaxTextBufferSize = 4096;
class CTextBuffServer : public CServer2
{
public:
    static CTextBuffServer* NewLC();
    virtual CSession2 *NewSessionL(const TVersion &aVersion,
                                   const RMessage2 &aMessage) const;
private:
    CTextBuffServer();
};
class CTextBuffSession : public CSession2
{
public:
    static CTextBuffSession* NewL(CTextBuffServer* aServer);
    CTextBuffSession();
private:
    virtual void ServiceL(const RMessage2& aMessage);
    virtual void ServiceError(const RMessage2& aMessage, TInt aError);
    CTextBuffSession() {};
    void ConstructL(CTextBuffServer* aServer);
    void DispatchMessageL(const RMessage2& aMessage);
    void AddTextL(TDesC& txt);
    void BackSpaceL(TInt aNumChars);
    TDesC& GetText();
    void Reset();
    void ClientPanic(const RMessage2& aMessage, TInt aPanicCode) const;
    RBuf 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 `CServer2`, and there is only one instance of this when the server is running. The session class is derived from `CSession2`, and an instance of this class is created for each session a client opens.

The server is created and started by calling the static `NewLC()` function of our `CTextBuffServ` class. `CTextBuffServ::NewLC()` and the `CTextBuffServ` constructor are shown below:

```

CTextBuffServer* CTextBuffServer::NewLC()
{
    CTextBuffServer* self=new (ELeave) CTextBuffServer;
    CleanupStack::PushL(self);
    self->StartL(KTextBuffServerName);
    return self;
}

```

```
CTextBuffServer::CTextBuffServer()
: CServer2(EPriorityStandard,ESharableSessions)
{
}
```

The constructor passes the server's priority to the CServer2 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::NewLC() calls the base class method, CServer2::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.

Following is the server startup code for the textbuffserv process:

```
static void StartServerL()
{
    // create and install an active scheduler
    CActiveScheduler *pA=new (ELeave) CActiveScheduler;
    CleanupStack::PushL(pA);
    CActiveScheduler::Install(pA);
    // create server and install
    CTextBuffServer *serv;
    serv = CTextBuffServer::NewLC();
    RProcess::Rendezvous(KErrNone); // ready to accept commands
    CActiveScheduler::Start();
    CleanupStack::PopAndDestroy(2,pA);
}

GLDEF_C TInt E32Main()
{
    CTrapCleanup* cleanup=CTrapCleanup::New();
    // create a cleanup stack
    TRAPD(res,StartServerL());
    if (res)
    {
        _LIT(KTxtBuffServerPanic,"TextBuffServerExample");
        User::Panic(KTxtBuffServerPanic,EServerInitError);
    }
    delete cleanup;
    return(res);
}
```

When the server process is started, control goes to E32Main(), which is the process entry point. Upon entry, the program creates a cleanup stack

and calls `StartServerL()`. `StartServerL()` creates and installs the thread's active scheduler and creates the server class by calling `CTextBuffServ::NewLC()`. Once this is done, the server is ready to accept commands, so it calls `RProcess::Rendezvous(KErrNone)` to indicate this to the client (see section 10.3.1). The active scheduler is then started with `CActiveScheduler::Start()`. The server is now waiting for a client to request a session with the server.

When the server receives a command to create a session (via a client calling `RSessionBase::CreateSession()`), `CServer2` invokes its virtual `NewSessionL()`. This function is implemented in the `CServer2`-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()`:

```
CSession2 *CTextBuffServer::NewSessionL(const TVersion &aVersion,
                                         const RMessage2 &aMessage /*not used here*/) 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 `CSession2`-based object. Thereafter all client messages, through the session that created this session class instance, will go to this instance.

`NewSessionL()` also 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*/)
{
    iTextBuff.CreateL(KMaxTextBufferSize);
}
```

```
CTextBuffSession::~CTextBuffSession()
{
    iTextBuff.Close();
}
```

`CTextBuffSession::NewL()` calls the second-phase construction method (`ConstructL()`) and uses the cleanup stack to safely handle any leave that may occur. For this example server, a text buffer is associated with each session, and the `ConstructL()` function creates the buffer as an `RBuf` descriptor.

Processing messages from the client

When a server receives a message from a client, the server creates an instance of a class called `RMessage2` to hold the message contents. Then the server invokes the `ServiceL()` method of the appropriate session object – supplying the `RMessage2` object as its argument.

The following shows the session command handler for the `textbuff-serv` example:

```
void CTextBuffSession::ServiceL(const RMessage2& aMessage)
{
    DispatchMessageL(aMessage);
    aMessage.Complete(KErrNone);
}

void CTextBuffSession::DispatchMessageL(const RMessage2& aMessage)
{
    // check for session-relative requests
    switch (aMessage.Function())
    {
        case ETextBuffAddText:
            TBuf<200> tmp;
            aMessage.ReadL(0,tmp);
            AddTextL(tmp);
            break;
        case ETextBuffGetText:
            TPtrC buff = GetText();
            aMessage.WriteL(0,buff);
            break;
        case ETextBuffReset:
            Reset();
            break;
        case ETextBuffBackSpace:
            BackSpaceL(aMessage.Int0());
            break;
        case ETextBuffCloseSession:
            CActiveScheduler::Stop();
            break;
        default:
            ClientPanic(aMessage,EInvalidCommand);
            break;
    }
}
```

```

// Handles leaves from CTextBuffSession::DispatchMessageL()
// A bad descriptor error implies a badly programmed client, so panic it
// Report other errors to the client by completing the outstanding request
//with the error
void CTextBuffSession::ServiceError(const RMessage2& aMessage, TInt
    aError)
{
    if (KErrBadDescriptor==aError)
        ClientPanic(aMessage,EInvalidDescriptor);
    else
        aMessage.Complete(aError);
}
void CTextBuffSession::Reset()
{
    iTextBuff.Zero();
}
void CTextBuffSession::AddTextL(TDesC& aText)
{
    if ( (aText.Length() + iTextBuff.Length()) > iTextBuff.MaxLength())
        User::Leave(KErrTooBig);
    else
        iTextBuff.Append(aText);
}
void CTextBuffSession::BackSpaceL(TInt aNumChars)
{
    if (aNumChars <= iTextBuff.Length())
    {
        TInt newLength = iTextBuff.Length() - aNumChars;
        iTextBuff.SetLength(newLength);
    } else
        User::Leave(KErrTooBig);
}
TDesC& CTextBuffSession::GetText()
{
    return iTextBuff;
}

```

ServiceL() invokes another method, DispatchMessageL(), to handle the message. Below are some of the key RMessage2 methods for accessing the command code and arguments of the message sent, as well as some other functionality:

- 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.
- Write() writes data to descriptors passed as arguments from the client via SendReceive() and Send(). This method will be discussed in more detail shortly. A leaving version of this function, WriteL(), also exists.

- `Read()` reads data from descriptors passed as arguments from the client via `SendReceive()` and `Send()`. This method will also be discussed in more detail shortly. A leaving version of this function, `ReadL()`, also exists.
- `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 aReason)` 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.
- `HasCapability()` checks to see if the client that sent the message has a specified capability. For example, `aMessage.HasCapability(ESCapabilityWriteDeviceData)` (assuming `aMessage` is an `RMessage2` from the client) would return `ETrue` if the client process has the `WriteDeviceData` capability (see Chapter 7 for more on capabilities). System servers will use this method in cases where the server only wants to perform certain operations on behalf of clients with specific capabilities. A leaving version of this function, `HasCapabilityL()`, also exists.

If `ServiceL()` leaves due to an error, the framework calls the virtual session method `ServiceError()`, passing it the current session message as well as the leave code. The example overrides this method to cause a client-side panic to occur if it detects that the leave was due to a bad client-side descriptor. Otherwise, `ServiceError()` completes the `RMessage2` message with the error code by calling `RMessage2::Complete()`.

`ClientPanic()` in our example is implemented as follows:

```
void CTextBuffSession::ClientPanic(const RMessage2& aMessage, TInt aPanic)
const
{
    _LIT(KTextBuffServSess, "CTextBuffSession");
    aMessage.Panic(KTextBuffServSess, aPanic);
}
```

Transferring 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.

When the client sends a buffer to the server, it is in the form of a pointer to a descriptor. The server cannot directly read and write using this descriptor if they are in different processes, which is typically the case for servers. Doing so would result in a panic, and the data would not be available in the place specified by the pointer anyway since the client process it belongs to is not currently active. This limitation is there to provide security since the operating system does not want one process to intentionally or unintentionally corrupt another. Section 3.5.6 discusses this in more detail. To access these buffers, the server uses the `RMessage2::Read()` and `RMessage2::Write()` methods, passing the index of the parameter in which the client placed the buffer in the first argument, and a descriptor to read the data from, or write the data to, in the second argument.

Returning to the example server, `DispatchMessageL()` calls `RMessage2::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 `RMessage2::Read()` as follows:

```
TBuf<200> tmp;  
res = Read(0, tmp);
```

This reads the data from the descriptor pointer supplied by the client in its first `TIpCArg` argument (index 0) sent via `SendReceive()` in the client method `RTextBuff::AddText()`. 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.

If the pointer in the message argument specified does not point to a valid descriptor, the `Read()` function will return an error. The error is handled by a utility function, which concludes by calling `RMessage2::Panic()` to panic the client thread.

Once the text has been read, `AddText()` (the server-side one) 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 `RMessage2::Write()` method.

`ETextBuffBackSpace` is an example of a case where the first argument is an integer rather than a pointer. This integer, read by the `RMessage2::Int0()` method, indicates the number of characters to backspace in the text buffer.

10.3.3 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
```

10.3.4 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* – that is, they only run while they are being used to save 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()
{
    SendReceive(ETextBuffCloseSession);
    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.

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.

11

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. Many phones also have Wi-Fi capability, allowing them to connect to a wireless Local Area Network (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 with the BSD socket API.

Note that to call the Symbian OS network API functions discussed in this chapter, your application will need the `NetworkServices` basic capability. Since this capability is in the basic user-grantable category, you can run the examples presented in this chapter without requiring them to be Symbian Signed (you can simply ‘self-sign’ them with a self-generated certificate without having to request a Symbian Signed developer certificate) to install and run on the phone. Reference Chapter 7 for more information on platform security and Symbian Signed.

We begin this chapter 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 11.3.

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.

11.1 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 and the 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 in programs (e.g., multiplayer 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 address¹ – 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 11.1 shows a simple diagram of TCP and UDP layered on top of IP.

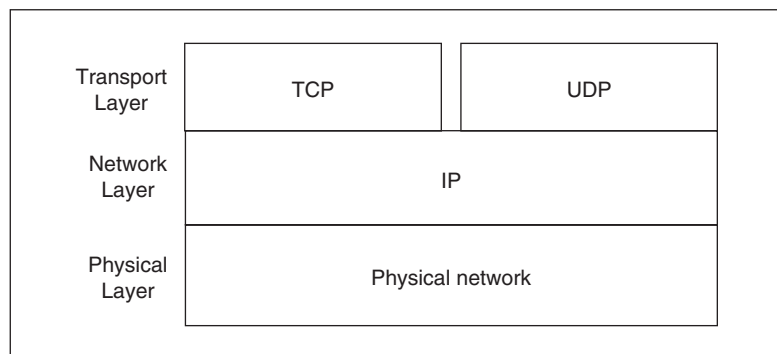


Figure 11.1 TCP, UDP, IP Layering

¹ In the new IPv6 protocol, IP addresses consist of eight 16-bit values (128 bits), which greatly expands the addresses that are available from IPv4's 32-bit addresses. At the time of writing, IPv4 is still the dominant protocol though and is what this chapter assumes.

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).

11.1.1 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 (a 16-bit unsigned integer) – 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.

11.1.2 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.

11.1.3 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/web	80
POP3 email	110

Service	Port
IMAP email	143
Streaming media	537
Doom	666
Landmark messaging	3969

Figure 11.2 illustrates an example of client–server network communications and shows the relation of IP and port addresses.

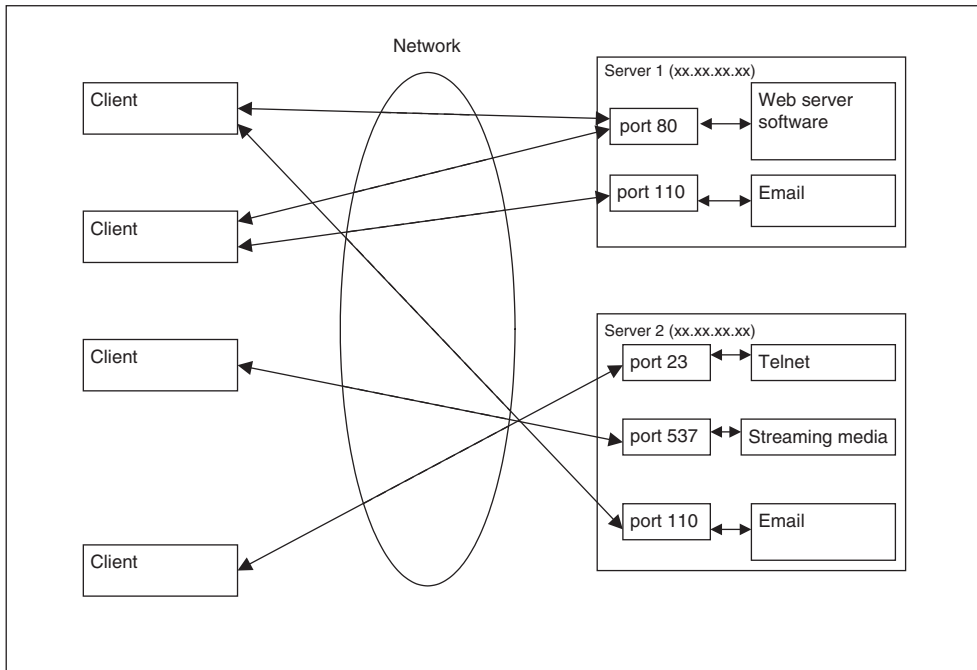


Figure 11.2 Client and Server Communications

11.1.4 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 generally need to be 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 (in the range 49,152–65,535 to avoid well-known port addresses).

11.2 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

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 (which is usually on another machine, but can also be on the same machine using the 127.0.0.1 loop-back address), and then transfers data over the socket using read and write commands.

11.2.1 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.

11.2.2 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 v9 also supports the BSD C-based socket API as part of the standard C library provided by the P.I.P.S. libraries.² 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.

11.2.3 BSD C API Socket Client Example

The following code 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.

```
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 = htons(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);
    }
}
```

² P.I.P.S. stands for **P.I.P.S. Is POSIX** on Symbian OS. The P.I.P.S. libraries provide `libc`, `libdl`, `libm`, and `libpthread` for Symbian OS v9, replacing the previous partial support for standard C on Symbian OS offered by `estlib`.

More information about P.I.P.S. can be found on the Symbian Developer Network wiki (<http://developer.symbian.com/wiki/display/oe/P.I.P.S.+Home>).

```
// 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_DGRAM` 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 = htonl(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 every day – 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.

11.3 Symbian OS Socket API

Symbian OS provides a C++ socket API that, 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 technology, 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.

11.3.1 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 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 sub-sessions to an established `RSocketServ` session and are derived from `RSubSessionBase`.

11.3.2 HTTP Example Using Symbian OS Socket API

Now let's look at the `OutputWebPage()` program from section 11.2.3, rewritten to use the Symbian OS socket API. 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 11.3.3).

Note that Symbian OS does provide an HTTP framework API, and using this framework makes the most sense when implementing code to retrieve web pages (I will not cover this framework in this book). However, I will use the general Symbian OS network APIs to implement the HTTP examples here for the purposes of demonstrating general network programming in Symbian OS.

```
#include <in_sock.h>
void HandleError(TDesC& aMsg, RSocketServ& aSockServ)
{
    aSockServ.Close();
    PrintError(aMsg);
}

TInt OutputWebPage(const TDesC& aServerName, const TDesC& aDoc)
{
    RSocketServ sockSrv;
    TInt res = sockSrv.Connect();
    if (res != KErrNone)
    {
```



```

    _LIT(KSockOpenFail, "Error connecting to socket server");
    HandleError(KSockOpenFail, sockSrv);
    return res;
}

RSocket sock;
res = sock.Open(sockSrv, KAfInet, KSockStream, KProtocolInetTcp);
if (res != KErrNone)
{
    _LIT(KSockOpenFail, "Socket open failed");
    HandleError(KSockOpenFail, SockServ);
    return res;
}

TNameEntry nameEntry;
RHostResolver resolver;
res = resolver.Open(sockSrv, KAfInet, KProtocolInetTcp);
if (res != KErrNone)
{
    _LIT(KResvOpenFail, "host resolver open failed");
    HandleError(KResvOpenFail, SockServ);
    return res;
}

TRequestStatus status;
resolver.GetByName(aServerName, nameEntry, status);
// wait for completion of asynchronous function. In real code,
// much better to use active objects for completions instead of these
// calls.
User::WaitForRequest(status);
resolver.Close();
if (status != KErrNone)
{
    _LIT(KDnsFail, "DNS lookup failed");
    HandleError(KDnsFail, SockServ);
    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");
    HandleError(KSocketConnectFail, SockServ);
    return res;
}

// Assemble HTTP GET command
TBuf8<300> getBuff;
_LIT8(KGetCommand, "GET");
getBuff.Copy(KGetCommand);
getBuff.Append(aDoc);
LIT(KCRLF, "\\xD\\xA");
getBuff.Append(KCRLF);
// 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);
}
```

Connecting to the socket server

Before using the socket API, you must first establish a session with the socket server. Note that when I use the term ‘socket server’ here, I mean the client–server process in Symbian OS that handles sockets, and not a destination server machine. To establish a session with the socket server, you instantiate an `RSocketServ` object and call its `Connect()` function, as shown below:

```
RSocketSrv sockSrv;
TInt res = sockSrv.Connect();
if (res != KErrNone)
{
    _LIT(KSockOpenFail,"Error connecting to socket server");
    HandleError(KSockOpenFail,sockSrv);
    return res;
}
```

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 3.10), and in a forthcoming book from Symbian Press.³

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);
```

³ *Symbian OS Communications Programming*, John Wiley & Sons, 2007, ISBN 0470512288.

`Open()` has the following form:

```
TInt Open(RSocketServ& aServ, TInt aAddrFamily, TInt aSocketType, TInt
        aProtocol);
```

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. `KAF_INET` specifies the TCP/IP v4 protocol suite. `socketType` is set to `KSOCK_STREAM` for TCP, or `KSOCK_DGRAM` 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, KAF_INET, KProtocolInetTcp);
if (res != KErrNone)
{
```

```

    _LIT(KResvOpenFail, "host resolver open failed");
    HandleError(KResvOpenFail, SockServ);
    return res;
}

TRequestStatus status;
resolver.GetByName(aServerName, nameEntry, status);
User::WaitForRequest(status);
resolver.Close();
if (status != KErrNone)
{
    _LIT(KDnsFail, "DNS lookup failed");
    HandleError(KDnsFail, SockServ);
    return res;
}

```

As mentioned previously (see section 11.3.1), `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");
    HandleError(KSocketConnectFail, SockServ);
    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. The following code shows a way of sending data via UDP using `SendTo()`. Note that I have omitted error checking for simplicity.

```
RSocket sock;
RSocketSrv sockSrv;
sockSrv.Connect();
sock.Open(sockSrv,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 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, TInt aFlags, 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).

There is another `RSocket` method to receive data, 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, TInt aFlags,
              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();
```

11.3.3 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, a better way to call these socket functions is within active objects, which were discussed in Chapter 8. The active object framework allows the system to handle events, such as those arising from user input, while the socket functions are active. When they complete, the associated active object's `RunL()` method handles events as appropriate. So the `Connect()` call, for example, could look something like:

```
void CMyActiveObject::DoNetworkStuff()
{
    iSock.Connect(destAddr,iStatus);
    SetActive();
}
void CMyActiveObject::RunL()
{
    // Invoked when the asynchronous function Connect() completes.
    // iStatus contains the completion status
}
```

Although you may want to refine how errors and completions are handled by the calling program (for example, having the calling program pass callbacks for its own handlers); this would vary depending on what your software does and how it would be used.

With active objects, you can have your program continue to process other, non-network, 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 OutputWebPage(const TDesC& aServ, const TDesC& aDoc);
    enum TLoadState
```

```

    {
        EResolvingName,
        EConnecting,
        ESending,
        EReceiving
    };
protected:
    void RunL();
    void DoCancel();
    CWebPage();
    void ConstructL();
private:
    TLoadState 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:

```

void CWebPage::OutputWebPage(const TDesC& aServerName, const TDesC& aDoc)
{
    TInt res = iSocketSrv.Connect();
    if (res != KErrNone)
    {
        _LIT(KSockOpenFail, "Socket server connect error");
        HandleError(KSockOpenFail);
        return;
    }
    iUrlDoc.Copy(aDoc);
    // Resolve name, rest handled by RunL()
    iState=EResolvingName;
    res =
    iSocket.Open(iSocketSrv, KAfInet, KSockStream, KProtocolInetTcp);
    if (res != KErrNone)
    {
        _LIT(KSockOpenFail, "Socket open failed");
        HandleError(KSockOpenFail);
        return;
    }
    res = iResolver.Open(iSocketSrv, KAfInet, KProtocolInetTcp);
    if (res != KErrNone)
    {
        _LIT(KResvOpenFail, "host resolver open failed");
        HandleError(KResvOpenFail);
        return;
    }
    iResolver.GetByName(aServerName, iNameEntry, iStatus);
    SetActive();
    // first asynchronous 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 below:

```
void CWebPage::RunL()
{
    if ( (iStatus != KErrNone) && (iStatus != KErrEof) )
    {
        // error, abort sequence, no further RunL()s will be invoked
        _LIT(KWebPageFail, "Error loading web page");
        HandleError(KWebPageFail);
    } 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;
                _LIT8(KGetCommand, "GET ");
                getBuff.Copy(KGetCommand);
                getBuff.Append(iUrlDoc);
                _LIT(KCRLF, "\\xD\\xA");
                getBuff.Append(KCRLF);
                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;
default:
    ASSERT(EFalse); // Should never get here
}
}
}

```

`iState` uses an enumeration to indicate 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.

`HandleError()` would close all the network objects and could display the error string, such as below:

```

void CWebPage::HandleError(TDesC& aErrMsg)
{
    iSocket.Close();
    iResolver.Close();
    iSocketServ.Close();
    PrintError(aErrMsg); // some generic message display routine
}

```

The error handling as well as the completion handling in this example is not ideal, since typically you would want the caller of `OutputWebPage()` to have control over error handling and know when the web page retrieval is complete. The next example will provide the ability to notify the caller of completion and errors so it can handle itself.

11.4 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

http://www.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 `GetTemperature(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.

11.4.1 wunderground.com

http://www.wunderground.com is a website that provides weather information. In addition to its HTTP website, 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 11.3 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 11.3, 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.

The following shows the active object class definition for the example:

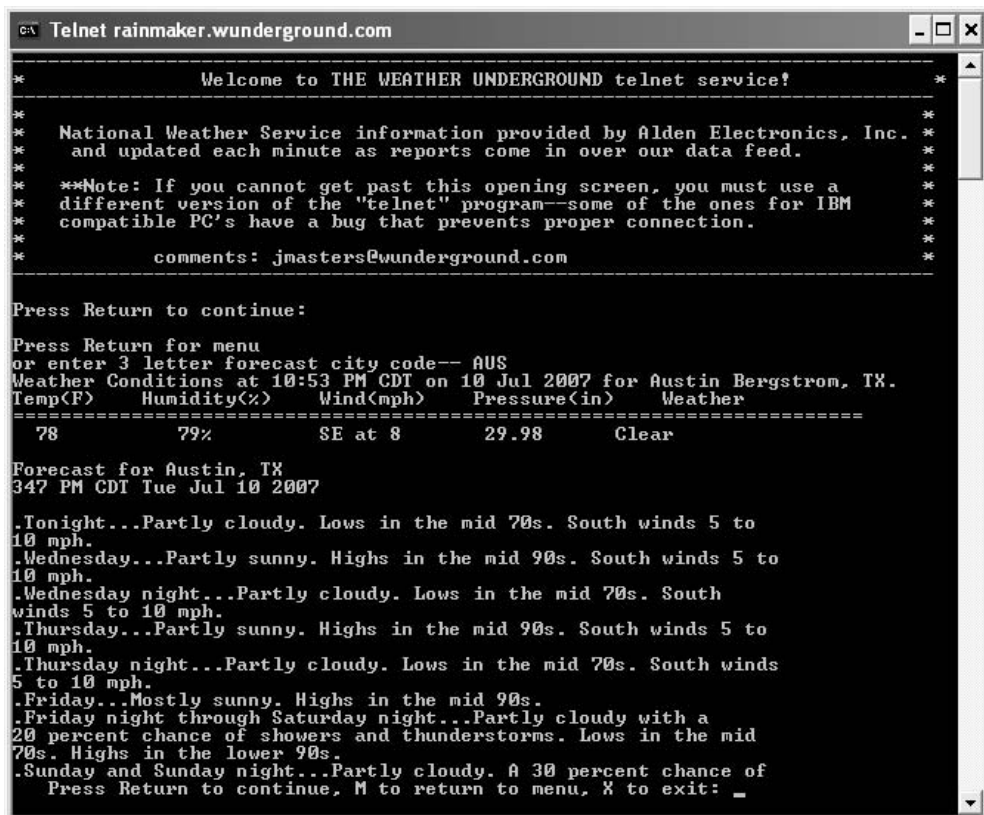
```
#include <e32base.h>
#include <in_sock.h>
class CWeatherInfo : public CActive
{
public:
    static CWeatherInfo* NewL(MWeatherObserver& aObserver);
    ~CWeatherInfo();
    void GetTemperature(const TDesC& aCity);
protected:
    CWeatherInfo(MWeatherObserver& aObserver);
    void RunL();
    void DoCancel();
    enum TLoadState
    {
        EInitializing,
        EResolvingName,
        EConnecting,
        ESending,
```

```

    EReceiving
    };
private:
    TLoadState iCommState;
    RSocketServ iSocketSrv;
    RSocket iSocket;
    TNameEntry iNameEntry;
    RHostResolver iResolver;
    TBuf8<20000> iNetBuff;
    TSockXfrLength iLen;
    TBuf<16> iCityCode;
    MWeatherObserver& iObserver;
    void Cleanup(TInt aError);
};

```

Following is the code for the `NewL()`, constructor, destructor, and `DoCancel()` functions:



```

Telnet rainmaker.wunderground.com

*-----*
* Welcome to THE WEATHER UNDERGROUND telnet service! *
*-----*
*
* National Weather Service information provided by Alden Electronics, Inc.
* and updated each minute as reports come in over our data feed.
*
* **Note: If you cannot get past this opening screen, you must use a
* different version of the "telnet" program--some of the ones for IBM
* compatible PC's have a bug that prevents proper connection.
*
* comments: jmasters@wunderground.com
*-----*

Press Return to continue:

Press Return for menu
or enter 3 letter forecast city code-- AUS
Weather Conditions at 10:53 PM CDT on 10 Jul 2007 for Austin Bergstrom, TX.
Temp(F) Humidity(%) Wind(mph) Pressure(in) Weather
=====
78 79% SE at 8 29.98 Clear
=====
Forecast for Austin, TX
347 PM CDT Tue Jul 10 2007

.Tonight...Partly cloudy. Lows in the mid 70s. South winds 5 to
10 mph.
.Wednesday...Partly sunny. Highs in the mid 90s. South winds 5 to
10 mph.
.Wednesday night...Partly cloudy. Lows in the mid 70s. South
winds 5 to 10 mph.
.Thursday...Partly sunny. Highs in the mid 90s. South winds 5 to
10 mph.
.Thursday night...Partly cloudy. Lows in the mid 70s. South winds
5 to 10 mph.
.Friday...Mostly sunny. Highs in the mid 90s.
.Friday night through Saturday night...Partly cloudy with a
20 percent chance of showers and thunderstorms. Lows in the mid
70s. Highs in the lower 90s.
.Sunday and Sunday night...Partly cloudy. A 30 percent chance of
Press Return to continue, M to return to menu, X to exit: _

```

Figure 11.3 Output of wunderground telnet session

```

CWeatherInfo* CWeatherInfo::NewL(MWeatherObserver& aObserver)
{
    CWeatherInfo* self = new(ELeave) CWeatherInfo(aObserver);
    CActiveScheduler::Add(self);
    return self;
}

CWeatherInfo::CWeatherInfo(MWeatherObserver& aObserver)
: CActive(CActive::EPriorityStandard), iObserver(aObserver)
{}

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. An object with an interface of type `MWeatherObserver` is passed to `NewL()`. I defined `MWeatherObserver` as follows (putting in the same header file as `CWeatherInfo`):

```

class MWeatherObserver
{
public:
    virtual void TemperatureReport(TDesC& aCity, TDesC& aTemperature)=0;
    virtual void TemperatureError(TDesC& aErrStr, TInt aErrCode)=0;
};

```

The parameter object should implement `TemperatureReport()` and `TemperatureError()`, which act as callbacks to handle the completion of the retrieval of the temperature started via `GetTemperature()`. `CWeatherInfo` will call `TemperatureReport()` upon successfully obtaining the temperature of the request city. If an error occurs, `TemperatureError()` is called, passing it an error string as well as a code.

Recall from Chapter 4 that M classes are for adding a particular interface to an object. Interfaces are usually added to objects via multiple inheritance. The class that derives from the interface class must implement the virtual functions in the interface (they are defined as pure virtual in the interface class). For example, when I modified the S60 `SimpleEx` to use `CWeatherInfo`, I inherited the `SimpleEx` UI class from `MWeatherObserver` and implemented `TemperatureReport()` and `TemperatureError()` in the UI class – we'll look at that code shortly (in UIQ I inherited the view class from `MWeatherObserver` instead of the UI class).

The `CWeatherInfo` constructor passes the active object priority to the base constructor and sets the passed `MWeatherObserver` class reference to the member variable `iObserver` for later use. The destructor calls `DoCancel()`, which cancels any asynchronous call in progress so that the active object can safely be destroyed.

The following code shows the implementation of `GetTemperature()`. You call this function to start the process of collecting the weather information from which the temperature will be extracted and displayed.

```
void CWeatherInfo::GetTemperature(const TDesC& aCity)
{
    // if we are already getting the temperature, then return.
    if (IsActive())
        return;
    iCommState=EInitializing;
    TInt res =iSocketSrv.Connect();
    if (res != KErrNone)
    {
        Cleanup(res);
        return;
    }
    iCityCode.Copy(aCity);
    res = iSocket.Open(iSocketSrv, KAfInet, KSockStream, KProtocolInetTcp);
    if (res != KErrNone)
    {
        Cleanup(res);
        return;
    }
    // Resolve name, rest handled by RunL()
    res = iResolver.Open(iSocketSrv, KAfInet, KProtocolInetTcp);
    if (res != KErrNone)
    {
        Cleanup(res);
        return;
    }
    iCommState=EResolvingName;
    _LIT(KWeatherServerName, "rainmaker.wunderground.com");
    iResolver.GetByName(KWeatherServerName, iNameEntry, iStatus);
    SetActive();
}
```

`GetTemperature()` 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 `GetTemperature()` was called while a temperature retrieval is already in progress. If the active object is not active, `GetTemperature()` copies the city code, which has been passed to it, to a member variable for sending to the server later, 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 an 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 upfront 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 lookup and making the program faster.

The call to `GetTemperature()` 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.

The active object's `RunL()` function is as follows:

```
void CWeatherInfo::RunL()
{
    if (iStatus != KErrNone)
    {
        Cleanup(iStatus.Int());
    }
    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:
            {
                _LIT(KCRLF, "\\xD\\xA");
                TBuf8<300> getBuff;
                getBuff.Copy(KCRLF);
                getBuff.Append(iCityCode);
                getBuff.Append(KCRLF);
                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.
-----*/
        _LIT8(KFileTok,"=\xA");
        TInt pos = iNetBuff.FindF(KFileTok);
        TBuf<100> temp;
        if (pos != KErrNotFound)
        {
            temp.Copy(iNetBuff.Mid(pos+2,10));
            temp.Trim();
            iObserver.TemperatureReport(iCityCode,temp);
            Cleanup(KErrNone);
        } 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 11.4.

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.

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

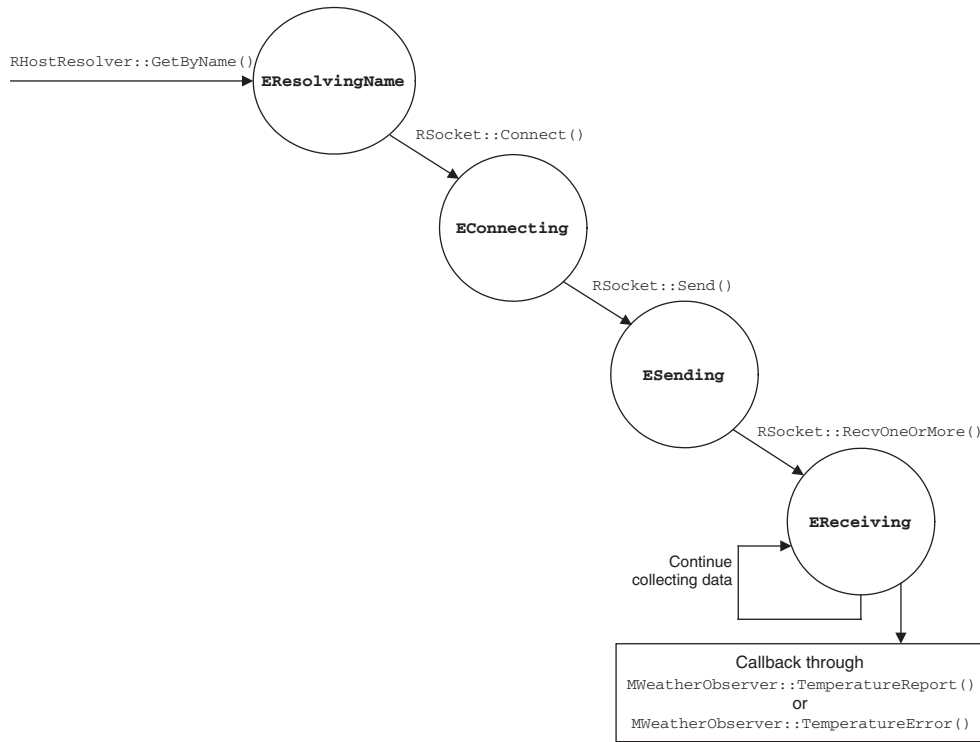


Figure 11.4 Weather Example State Machine

(see Figure 11.3), which immediately precede the temperature. Once the temperature is found, it calls the `TemperatureReport()` function of the `MWeatherObserver` class passed in on the `NewL()` (stored in `iObserver`), passing it the city code and the temperature. If an error occurs at any time, the method `CWeatherInfo::Cleanup()` is called, which is implemented as follows:

```

void CWeatherInfo::Cleanup(TInt aError)
{
    iSocket.Close();
    iSocketSrv.Close();
    iResolver.Close();
    TBuf<50> errStr;
    if (aError!=KErrNone)
    {
        switch (iCommState)
        {
            case EInitializing:
            {

```

```

        _LIT(KErrStr, "Error initializing communications");
        errStr.Copy(KErrStr);
        break;
    }
    case EResolvingName:
    {
        _LIT(KErrStr, "Error resolving name");
        errStr.Copy(KErrStr);
        break;
    }
    case EConnecting:
    {
        _LIT(KErrStr, "Error connecting to server");
        errStr.Copy(KErrStr);
        break;
    }
    case ESending:
    {
        _LIT(KErrStr, "Error sending request");
        errStr.Copy(KErrStr);
        break;
    }
    case EReceiving:
    {
        _LIT(KErrStr, "Error receiving data");
        errStr.Copy(KErrStr);
        break;
    }
    default:
    {
        _LIT(KErrStr, "Unknown error");
        errStr.Copy(KErrStr);
        break;
    }
}
iObserver.TemperatureError(errStr, aError);
}
}

```

`Cleanup()` closes all the network objects (it does not hurt to call `Close()` on a handle that is not yet open), and creates an error message based on what the state stored in `iCommState` is at the time of the error. It then invokes the observer's `TemperatureError()` callback function, passing it the error message along with the error code.

11.4.2 Adding this Code to `SimpleEx`

The full sample code for adding the functionality described above to the S60 and UIQ versions of `SimpleEx` is available from the website for this book, which can be found on the Symbian Developer Network wiki (<http://developer.symbian.com/wiki>). Here are the basic steps I

performed to integrate this functionality in SimpleEx:

1. I added a new menu item called 'Get temperature' to the resource file, changing the menu definition for S60 and the command resource for UIQ. I named the command `ESimpleExTemperatureCommand`.
2. I added the `ESimpleExTemperatureCommand` to the command enum in the program's `simpleEx.hrh` file.
3. For S60, I added the `MWeatherObserver` interface to `CSimpleExAppUi` in the class definition using multiple inheritance and added to the class the two virtual functions of `MWeatherObserver` that must be overridden:

```
class CSimpleExAppUi : public CAknAppUi, public MWeatherObserver
{
    ...
    // from MWeatherObserver
    virtual void TemperatureReport(TDesC& aCity, TDesC& aTemperature);
    virtual void TemperatureError(TDesC& aErrorStr, TInt aErrCode);
    ...
};
```

For UIQ, I added this interface to `CSimpleExAppView` instead of `CSimpleExAppUi` (I wanted it where the application UI handler resides).

4. I added a private member variable, `CWeatherInfo *iWeather`, to the `CSimpleExAppUi` class in the case of S60, and to the `CSimpleExAppView` class in the case of UIQ.
5. I included the previously listed example source (either in separate files or in the existing source and include files).
6. In the `CSimpleExAppUi::ConstructL()` (`CSimpleExAppView::ConstructL()` for UIQ) I added the statement:

```
iWeather = CWeatherInfo::NewL(*this);
```

to create the example's active object.

7. In the `CSimpleExAppUi` destructor I added:

```
delete iWeather;
```

8. In the `CSimpleExAppUi::HandleCommandL()` (`CSimpleExAppView::HandleCommandL()` for UIQ), I added a case for the command `ESimpleExTemperatureCommand`, which looks

as follows:

```
case ESimpleExTemperatureCommand:
{
    // Display the temperature in Austin, TX
    _LIT(KCityCode,"AUS");
    iWeather->GetTemperature(KCityCode);
}
```

9. I overrode the callback functions of the MWeatherObserver as follows:

```
void CSimpleExAppUi::TemperatureReport(TDesC& aCity,TDesC& aTemp)
{
    TBuf<50> str;
    _LIT(KTempTitle,"Report:");
    _LIT(KTempMessage,"Temperature in %S is %S");
    str.Format(KTempMessage,&aCity,&aTemp);
    TRAPD(res,iEikonEnv->InfoWinL(KTempTitle,str));
}

void CSimpleExAppUi::TemperatureError(TDesC& aErrStr,TInt aErrCode)
{
    TBuf<50> str;
    _LIT(KErrTitle,"Error:");
    _LIT(KErrMsg,"%S (%d)");
    str.Format(KErrMsg,&aErrStr,aErrCode);
    TRAPD(res, iEikonEnv->InfoWinL(KErrTitle, str));
}
```

Again, I did this in CSimpleExAppView class for the UIQ version.

10. In the MMP file, I added `insock.lib` and `esock.lib` to the `LIBRARY` line, to include the libraries which implement the socket calls. I also added the line `CAPABILITY NetworkServices`, since, as I mentioned previously, the application requires that capability to communicate on the network. If you do not have this capability, the network API calls will fail.

Build and run the updated SimpleEx program on the phone (Chapter 2 shows how to use `make sis` to create a SIS file to enable you to install and run SimpleEx on the phone). 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 (in this case it is hardcoded to AUS, but you could modify to have the user enter it) is displayed on the screen. Figure 11.5 shows the output of this example on an S60 device.

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

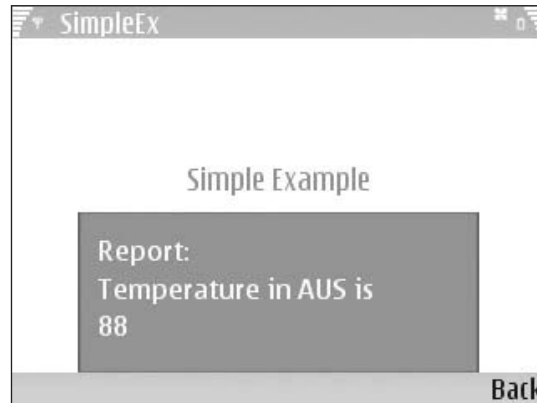


Figure 11.5 Weather Example Screen

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.

11.5 Making a Network Connection

Up to this point, we have not discussed how the network connections themselves are represented and brought up, and we just relied on the operating system to select the connection for us implicitly (while in many cases it will prompt the user for a connection) in our examples. This section will give a very brief overview of connections in Symbian OS. You should consult the SDK documentation for more details.

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 dialup 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** or **general.t-mobile.co.uk**). 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., via your SIM or through a service SMS message).

11.5.1 Establishing a Connection for a Program

The example Symbian OS socket code, presented in section 11.3, 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*. You start up a particular connection using the `RConnection` API (not covered in this book) and associate that connection with a socket. Symbian OS also supports having multiple connections active at once, for example one socket could use a GPRS connection and another a Wi-Fi one at the same time.

12

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 smartphone users do not get confused – yet it should also creatively provide good presentation and user control of the features unique to your application. UI style guides exist for both UIQ and S60 and are invaluable sources of information about what can and what cannot be done on each UI, so that your application has a consistent interface suited for that platform. The UI style guides for UIQ and S60 are available on the developer sites (<http://www.developer.uiq.com> for UIQ and <http://www.forum.nokia.com> for S60). Up-to-date links to these, and other links referenced in this book, will be kept in the Symbian Developer Network wiki (<http://developer.symbian.com/wiki/>).

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 for the two major UI platforms that use Symbian OS v9. First, there is a discussion of the UIQ 3 and S60 3rd Edition user interfaces. 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.

12.1 Symbian OS User Interfaces

First, let's take a quick tour of the UIQ 3 and S60 3rd Edition user interfaces from the user's point of view.

12.1.1 UIQ 3 Phones

UIQ 3 is the UI platform adopted by Sony Ericsson and others, including Motorola, for their Symbian OS smartphones. Examples of UIQ 3 phones include the Sony Ericsson W960i and P1i, and the Motorola MOTORIZR Z8. UIQ was originally designed for smartphones with quarter VGA resolution portrait screens, which had touchscreen input, but now with the introduction of UIQ 3 it can run in numerous user and display modes including portrait and landscape, with and without touchscreens.

The typical screen resolution is still QVGA portrait (240×320), or landscape, but UIQ 3 also supports a smaller, 240×256 screen format that can also be in portrait and landscape.

In addition to having the ability to display in portrait and landscape, in the two different screen sizes, UIQ also supports two interaction styles: *pen* and *softkey*. The pen style is a way of presenting the user interface to the user that is optimized for touchscreen devices, for example in the Sony Ericsson P990. The softkey style provides a good one-handed user experience without needing a touchscreen by using hardware keys and optional jog dial, and is found, for example, in the Motorola MOTORIZR Z8.

Figures 12.1 and 12.2 show a sample screen of an application using portrait pen style and softkey style, with smaller portrait screen, respectively.

There is also a mode that combines the softkey style with a touchscreen. On the screen, this looks like the softkey style in Figure 12.2, except that there is a back key in the title bar, and the touchscreen can be used to select the screen controls.

A single device can support multiple display and interaction modes. For example, on the Sony Ericsson P990i, when you close the flip, the phone goes in the small portrait softkey mode and the touchscreen is disabled. When the flip is open, the phone goes to the larger portrait pen mode. In fact, Figures 12.1 and 12.2 were taken on a P990i smartphone with the flip open and closed, respectively.

For the most part, you only need minor changes to your application to support these multiple display modes and the UIQ platform does the rest. Section 12.4.3 discusses the changes needed for your application to support the various display modes.

A UIQ 3 application screen is divided into the following areas:

- Title bar
The title bar shows the name of the application, with its icon. In pen mode, you select the application's name via the touchscreen to



Figure 12.1 UIQ 3 Application, Pen Style



Figure 12.2 UIQ 3 Application, Softkey Style

display the pull-down menu. In non-touch, softkey mode, the title bar cannot be selected in any way and is for informational purposes only. As mentioned previously, in softkey touch mode, a back button is also displayed in the title bar. The title bar can also contain a view context bar owned by an application view that can contain items like screen tabs (not shown in the figures).

- **Application area**
The large central area below the title bar is the application area. This is where the application-specific data and controls that make up an application view are displayed.
- **Button bar**
In pen mode, the application area has an optional button bar below the application area. 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. A back button resides on this bar. Buttons can also contain graphics instead of text. In softkey mode, there is no button bar but instead there is a softkey area.
- **Softkey area**
In softkey mode, as shown in Figure 12.2, the labels for softkeys are displayed at the bottom of the screen (these line up with device hardware keys). There are three softkeys. The rightmost softkey, labeled 'More', contains the application menu. In some cases, the middle button can also be used to display multiple options.
- **Status pane**
UIQ devices display a one-line status pane that can be at the top or the bottom of the screen depending on the display configuration (for example, in Figure 12.1, pen style, it's at the bottom). The status pane contains items such as signal strength, battery status, and connection status. In a touchscreen mode you can select the icons in this pane for more detailed information.

User input

There are multiple ways that a user can enter data on UIQ phones. If they have a touchscreen, you can use handwriting recognition or a virtual touchscreen-based keyboard. You can also enter data on the device keyboard, with some models supplying a full QWERTY hardware keyboard.

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

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 (or in the case of softkey style, softkeys are used instead of buttons). UIQ supports simple, single-page dialogs and multipage dialogs, in which each page can be selected via a row of tabs. Figure 12.3 shows an example of a multipage dialog (this is the browser's Preferences dialog with the pen interaction style).

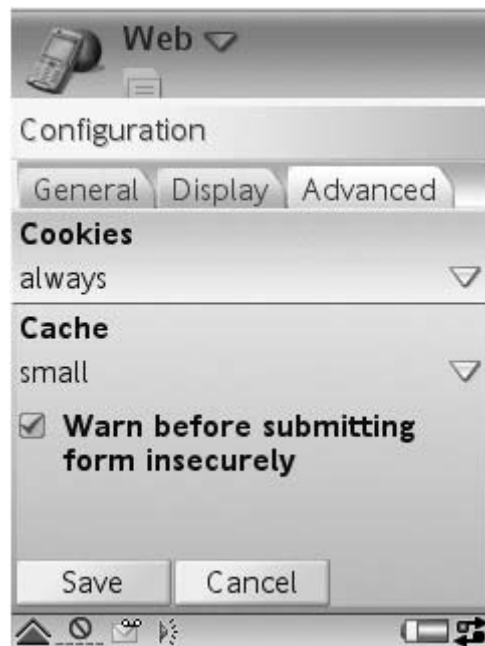


Figure 12.3 UIQ Dialog (Pen Style)

12.1.2 S60 Phones

The S60 user interface from Nokia was originally made for devices with smaller screens (basic resolution 176 × 208) and no touchscreen. S60 3rd

Edition supports more varied phone display sizes and resolutions, ranging from basic resolution to QVGA, double (352 × 416), and even HVGA (240 × 640) in both portrait and landscape orientations. At the time of writing, there are no S60 3rd Edition smartphones with touchscreens, so the S60 UI is optimized for one-handed operation. S60 is a very popular platform and is the platform used on the majority of Symbian OS smartphones. Smartphones running S60 3rd Edition include the Nokia 5500, E61i, N95, and E90 (which is the first communicator device for S60 – previous communicator devices used a separate UI platform, based on Symbian OS, called Series 80). Nokia also licenses the S60 platform to other smartphone manufacturers such as Samsung and LG. Examples of non-Nokia S60 smartphones for S60 3rd Edition include the SGH-i520 and SGH-i400 from Samsung.

Figure 12.4 shows an example S60 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 12.3. As with UIQ, the layout and system status content of this pane is determined by the phone manufacturer.
- **Main panel**
The main panel 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. The control pane displays an Options softkey that, when selected, presents the application's menu.

User input

A four-way controller key, rather like a joystick, is used to navigate between screen fields. Text input is accomplished through the phone's numeric pad, or, if the phone has one, through the QWERTY keyboard.

S60 dialogs

As in UIQ, S60 dialogs are pop-up 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



Figure 12.4 S60 3rd Edition Application Screen (E61)



Figure 12.5 S60 Dialog (E61)

selected by using the hardware controller key, as opposed to tapping on a touchscreen. Furthermore, S60 dialogs do not contain a row of GUI buttons, but use the two softkeys instead. Figure 12.5 shows an example of an S60 dialog.

12.2 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.
- A resource registration file (normally named `<application_name>_reg.rss`) required to register the application with the device so it is selectable by a user.
- 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). In S60 this can be a single scalable bitmap (in what is known as SVT-G format).

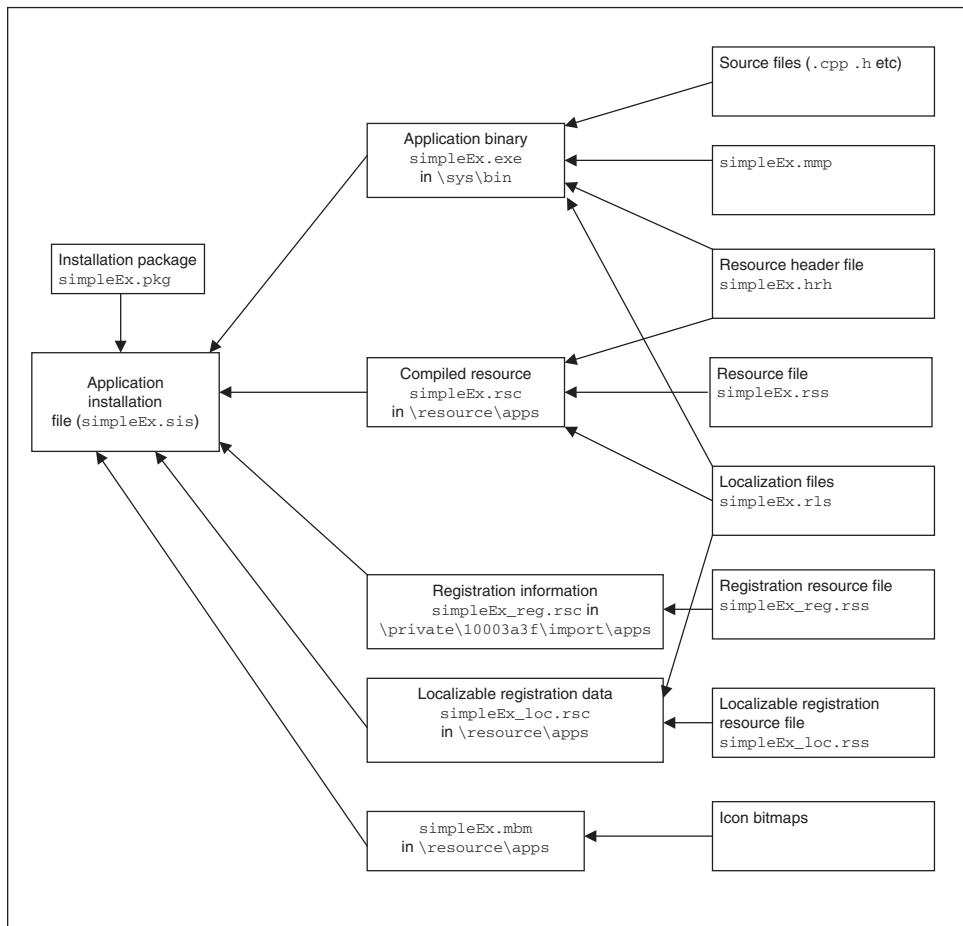


Figure 12.6 Application Project Elements and Build Flow

- A localizable application information resource file (normally named `<application_name>_loc.rss`) to define the application's caption, number of icons, and other information (this, together with the icon bitmaps, is used to generate the RSC file for download to the phone along with the application).

Figure 12.6 shows how all of these elements come together to build an application (using `SimpleEx` as an example application name).

12.3 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.

12.3.1 `E32Main()` Entrypoint and `NewApplication()`

A Symbian OS GUI application is a process executable (EXE file) and therefore must contain an `E32Main()` entrypoint function. For a GUI application this just calls `EikStart::RunApplication()`, passing it a pointer to a function that returns the application object to run. Below are these two functions for `SimpleEx`:

```
EXPORT_C CApaApplication* NewApplication()
{
    return (static_cast<CApaApplication*>(new CSimpleExApplication));
}
GLDEF_C TInt E32Main()
{
    return EikStart::RunApplication( NewApplication );
}
```

The next section describes the basic classes used to implement your application, including the application class.

12.3.2 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. UIQ and S60 have their own platform-specific versions of these classes, but they derive from a common set of application classes as shown in Figures 12.7 and 12.8. Note that in S60, I show the view class inherited from `CCoeControl`, but for applications with multiple views (which I do not cover in this book), you would inherit from `CAknView` (see section 12.7) for the view class and `CAknViewAppUi` for the UI class.

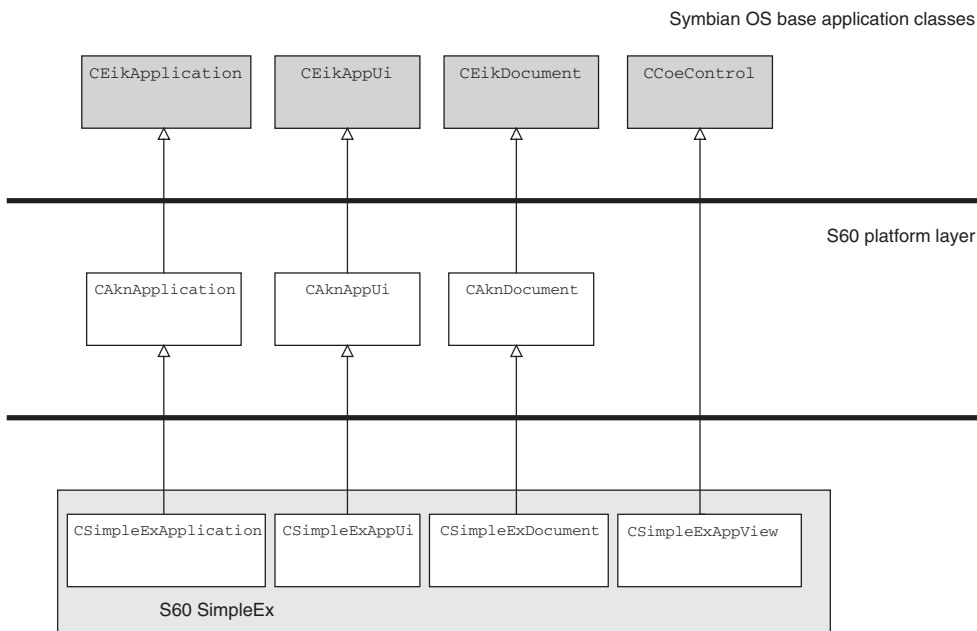


Figure 12.7 S60 Application Class Hierarchy

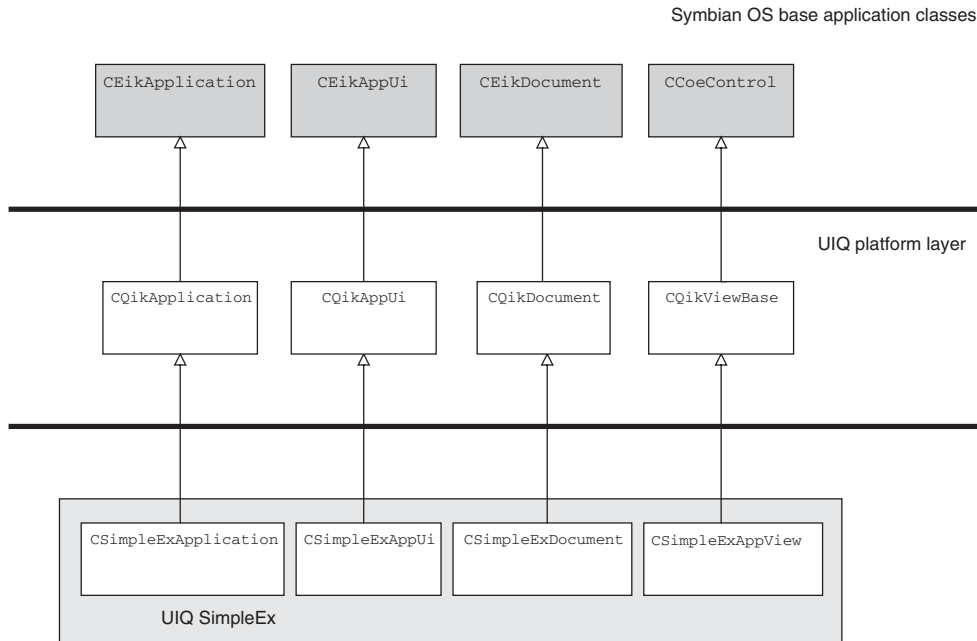


Figure 12.8 UIQ Application Class Hierarchy

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;
};
  
```

S60 declares it in the same way, except that the class is derived from CAknApplication. Here is the implementation of this class for SimpleEx:

```

#include "SimpleEx.h"
CApaDocument* CSimpleExApplication::CreateDocumentL()
{
    return new(ELeave) CSimpleExDocument(*this);
}
TUid CSimpleExApplication::AppDllUid() const
{
    return KUidSimpleExApp;
}
  
```

After the GUI application is started, and 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 you specify in the application's MMP file (`KUIdSimpleExApp` is defined to be the application's UID in a header file) – this can be a very difficult-to-debug problem if these do not match. 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` for UIQ 3 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 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 S60 applications fall into this category, the default behavior of the S60 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 is responsible, upon construction, for creating the application's default view. In S60, for basic one-view applications (i.e., the view is a simple control), the UI class contains the command handler that handles all the commands the users initiate via the GUI. If you are using multiple views, however, this command handler would be in the view class. In UIQ, this command handler is always in the view class.

The declaration for the UIQ version of SimpleEx's application UI class is as follows:

```
class CSimpleExAppUi : public CQikAppUi
{
public:
    void ConstructL();
private:
    CSimpleExAppView* iAppView;
};
```

Below is the implementation of the ConstructL() function in the UI class:

```
void CSimpleExAppUi::ConstructL()
{
    BaseConstructL();
    iAppView = CSimpleExAppView::NewLC(*this);
    AddViewL(*iAppView);
    CleanupStack::Pop(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; however, note that the framework actually owns the view instance, so an explicit delete need not be done in your application.

The S60 UI class is similar to UIQ in that it also creates the view in its ConstructL(). S60, however, also implements the HandleCommandL() function in the UI class, in the case of single control view

(like in the S60 SimpleEx example), to process the application's command events. Below is SimpleEx's S60 implementation of the UI class including the command handler:

```
void CSimpleExAppUi::ConstructL()
{
    BaseConstructL(CAknEnableSkin); // S60 requires CAknEnableSkin argument
    iAppView = CSimpleExAppView::NewL(ClientRect());
}
CSimpleExAppUi::~CSimpleExAppUi()
{
    delete iAppView;
}
void CSimpleExAppUi::HandleCommandL(TInt aCommand)
{
    switch(aCommand)
    {
        case EAknSoftkeyExit:
        case CEikCmdExit:
            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 S60 UI class should also override the `HandleResourceChangeL()` function. This will ensure that your application will run when you change the orientation of the screen from portrait to landscape and vice versa (for example, the Nokia E70 will automatically switch to landscape mode when you unfold its keyboard). You can also cause an orientation switch to occur on the S60 emulator by pressing the button at the top left of the emulator screen. For the examples in this book, the below function will suffice to ensure it works during orientation switches:

```
void CSimpleExAppUi::HandleResourceChangeL( TInt aType )
{
    CAknAppUi::HandleResourceChangeL( aType );
    if ( aType==KEikDynamicLayoutVariantSwitch )
```

```

    {
        iAppView->SetRect( ClientRect() );
    }
}

```

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.

Let's look at the SimpleEx application's view implementation and briefly discuss its key points starting with UIQ.

The declaration for SimpleEx's application view class for UIQ is as follows:

```

class CSimpleExAppView : public  CQikViewBase
{
public:
    static CSimpleExAppView* NewL(CQikAppUi& aAppUi);
    static CSimpleExAppView* CSimpleExAppView::NewLC(CQikAppUi& aAppUi);
    // from CQikViewBase
    TVwsViewId ViewId()const;
    void ViewConstructL();
    void HandleCommandL(CQikCommand& aCommand);
private:
    void Draw(const TRect&) const;
    CSimpleExAppView(CQikAppUi& aAppUi);
    void ConstructL();
};

```

The NewL() and NewLC() functions are implemented as follows:

```

CSimpleExAppView* CSimpleExAppView::NewL(CQikAppUi& aAppUi)
{
    CSimpleExAppView* self = CSimpleExAppView::NewLC(aAppUi);
    CleanupStack::Pop(self);
    return self;
}
CSimpleExAppView* CSimpleExAppView::NewLC(CQikAppUi& aAppUi)
{
    CSimpleExAppView* self = new (ELeave) CSimpleExAppView(aAppUi);
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

```

The view's static NewL() factory method both instantiates the application view class and calls its ConstructL() secondary constructor. The application UI calls this NewL() function to create the view class, passing a pointer to itself as an argument.

The constructor and the secondary constructor, `ConstructL()`, are shown below:

```
CSimpleExAppView::CSimpleExAppView(CQikAppUi& aAppUi)
: CQikViewBase(aAppUi, KNullViewId)
{
}
void CSimpleExAppView::ConstructL()
{
    BaseConstructL();
}
```

In the constructor we just pass a pointer to the UI class and a NULL view ID (`KNullViewId`), which is all that is needed for the example. In `ConstructL()` you must call the `BaseConstructL()` method.

Below is the `ViewConstructL()` method:

```
void CSimpleExAppView::ViewConstructL()
{
    // Loads information about the UI configurations this view supports
    // together with definition of each view.
    ViewConstructFromResourceL(R_SIMPLEEX_CONFIGURATIONS);
}
```

Here we load the `QIK_VIEW_CONFIGURATIONS` resource from the resource file. This is applicable for UIQ only. This resource defines the various UIQ display and interaction modes that the application can support. Section 12.4.2 will provide more detail on this.

`ViewId()` returns a unique identifier for the view, which consists of a combination of the application UID and the view UID:

```
TVwsViewId CSimpleExAppView::ViewId() const
{
    return TVwsViewId(KUidSimpleExApp, KUidSimpleExView);
}
```

The view UID must be unique for all views within an application. Since we only have one view, define `KUidSimpleExView` as 1.

Although not required, as mentioned, we overrode the `Draw()` method to draw text directly on the screen:

```
void CSimpleExAppView::Draw(const TRect&) const
{
    CWindowGc& gc = SystemGc();
    gc.Clear();
    const CFont* font = iBikonEnv->TitleFont();
    gc.UseFont(font);
    TRect drawRect = Rect();
    TInt baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;
    _LIT(KScreenText, "Simple Example");
    gc.DrawText(KScreenText,
```



```
drawRect,baselineOffset,CGraphicsContext::ECenter, 0);
gc.DiscardFont();
}
```

Although I do not discuss the details of how drawing is done, 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'.

In UIQ, the command handler is in the view class as opposed to the UI class we implemented for the S60 example. Here is the UIQ command handler in the view class:

```
void CSimpleExAppView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        case ESimpleExCommand:
        {
            _LIT(KMessage,"Start Selected!");
            iEikonEnv->AlertWin(KMessage);
            break;
        }
        default:
            CQikViewBase::HandleCommandL(aCommand);
    }
}
```

Now let's look at the view class for S60. Its declaration is below:

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

Since we just have a single view, I chose to inherit the S60 view class directly from `CCoeControl` – we do not need the added support of `CAknView`. Here is the implementation of the S60 view class:

```
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();
    gc.Clear();
    const CFont* font = iEikonEnv->TitleFont();
    gc.UseFont(font);
    TRect drawRect = Rect();
    TInt baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;
    _LIT(KScreenText, "Simple Example");
    gc.DrawText(KScreenText,
        drawRect, baselineOffset, CGraphicsContext::ECenter, 0);
    gc.DiscardFont();
}

```

The difference from UIQ here is that, since we have directly inherited from `CCoeControl`, in `ConstructL()` we have to create the view window, size it, and activate it. Furthermore, our command handler is not here in the view like in UIQ, and is instead in the UI class.

12.4 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.

12.4.1 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 uppercase characters, 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 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`. 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 `CHARACTER_SET_UTF8` to your resource file.
- **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 `STRUCT`'s, 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.

Attributes can also 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 STRUCT's 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"} };
}
```

12.4.2 SimpleEx's Resource File

Armed with the knowledge in the above section, let's look at the resources file again (originally presented in Chapter 2 (see section 2.3.4)), starting with UIQ and then S60.

UIQ SimpleEx resource

Following is the SimpleEx resource file for UIQ 3:

```
NAME SIMP
#include <eikon.rh>
#include <qikon.rh>
#include <QikCommand.rh>
#include "SimpleEx.hrh"
RESOURCE RSS_SIGNATURE
{
}
RESOURCE TBUF r_default_document_name
{
    buf="";
}
RESOURCE EIK_APP_INFO
{
}
RESOURCE QIK_VIEW_CONFIGURATIONS r_simpleex_configurations
{
    configurations =
    {
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            command_list = r_simpleex_commands;
            view = r_simpleex_layout;
        },
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikSoftkeyStyleSmallPortrait;
            command_list = r_simpleex_commands;
            view = r_simpleex_layout;
        }
    };
}
RESOURCE QIK_VIEW r_simpleex_layout
{
}
```

```

    pages = r_simpleex_layout_pages;
}
// Defines the pages of a view.
// Only one page for this example.
RESOURCE QIK_VIEW_PAGES r_simpleex_layout_pages
{
    pages =
    {
        QIK_VIEW_PAGE
        {
            page_id = ESimpleExViewPage;
        }
    };
}
RESOURCE QIK_COMMAND_LIST r_simpleex_commands
{
    items =
    {
        QIK_COMMAND
        {
            id = ESimpleExCommand;
            type = EQikCommandTypeScreen;
            text = "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 UIQ, `EIK_APP_INFO` should be defined empty – the defaults are appropriate.

The `QIK_VIEW_CONFIGURATIONS` resource specifies what display and user configurations are supported by the application. I introduced these modes in section 12.1.1. The resource attribute configurations is an array of `QIK_VIEW_CONFIGURATION` structures that define the

supported configurations along with the commands and view pages associated with that mode. `ui_config_mode` is set to the supported configuration. The table below lists the major configurations that exist (their support varies per smartphone):

<code>KQikPenStyleTouchPortrait</code>	Pen interaction style, portrait display
<code>KQikPenStyleTouchLandscape</code>	Pen interaction style, landscape display
<code>KQikSoftkeyStylePortrait</code>	Softkey interaction style, portrait display
<code>KQikSoftkeyStyleLandscape</code>	Softkey interaction style, landscape display
<code>KQikSoftkeyStyleSmallPortrait</code>	Softkey interaction style, portrait, small screen size
<code>KQikSoftkeyStyleSmallLandscape</code>	Softkey interaction style, landscape display, small screen size
<code>KQikSoftkeyStyleTouchPortrait</code>	Softkey interaction style with touchscreen, portrait display
<code>KQikSoftkeyStyleTouchLandscape</code>	Softkey interaction style with touchscreen, landscape display

The `command` attribute is set to point to the user commands that translate to softkeys and menu items (it depends on mode, how these are actually presented). As a side note, a command list can be referenced in other places besides configurations, like in views or view pages (although I will not cover these in this chapter).

In our case we define only two modes – `KQikPenStyleTouchPortrait` and `KQikSoftKeyStyleSmallPortrait`. Both of these point to the same `QIK_COMMAND_LIST` resource that defines our ‘Start’ command (`command_list = r_simpleex_commands`). Both also point to the same view (`view = r_simpleex_layout`).

S60 SimpleEx resource

Below is the S60 3rd Edition 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";
        }
    };
}

```

The first two resources, `RSS_SIGNATURE` and the `TBUF r_default_document_name`, are the same as `UIQ`.

Instead of having an empty `EIK_APP_INFO` resource, however, `S60` defines a control button array (CBA) resource and a menu resource:

```

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

```

The line `cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT` causes the left softkey on the application to read Options and display the menu, and the right softkey to send the exit command to the application. `menubar` points to the `MENU_BAR` resource that contains the menu pane with the start items on it. This menu will be presented to the user when the user presses the options softkey.

12.4.3 Application Registration Resource File

GUI applications are EXE files in the `sys\bin` directory (in Symbian OS v9) and in order for the device to recognize an executable as a GUI application, and to display them on the desktop for user selection, the application must be registered via a special resource file known as a registration resource file. The compiled registration resources are placed in the `\private\10003a3f\import\apps\` directory on the device. The source file names are typically `<application name>_reg.rss`. The user will not be able to run an application on the device without a registration resource specified for that application.

Here is the registration file for `SimpleEx`, which is generic and can be used for UIQ and S60 applications:

```
#include <AppInfo.rh>
#include "SimpleExUid.h" // defines the application UID
UID2 KUidAppRegistrationResourceFile
UID3 SIMPLEEXUID// application UID
RESOURCE APP_REGISTRATION_INFO
{
    app_file = "SimpleEx";
}
```

As you can see, there is not much to this. Besides including the `UID2 KUidAppRegistrationResourceFile`, which is standard, you specify your application's UID (via the `UID3` line) and the name of your EXE file (minus the `.exe` suffix) in the `app_file` attribute of the `APP_REGISTRATION_INFO` resource. This information is sufficient for the device to register your application to the device and have it appear on the desktop.

Although not used in the `SimpleEx` registration resource, `APP_REGISTRATION_INFO` also has a `localizable_resource_file` attribute. This attribute is optional and points to what is known as a localizable application information file that specifies the application's caption (in multiple languages if desired) as well as defines the application icons. If you have a localizable application information file, the file name must be set to the `localizable_resource_file` attribute of `APP_REGISTRATION_INFO` in order for it to be used. Localizable application information files will be discussed more in section 12.8.

To compile the registration file, you need to have the following in your MMP file, substituting `SimpleEx_reg.rss` with your application registration resource file name:

```
START RESOURCE    SimpleEx_reg.rss
TARGETPATH        \private\10003a3f\apps
END
```

Note that 10003a3f is the SID of the application architecture component, and the directory in TARGETPATH is its private directory.

12.4.4 Supporting Multiple Languages

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 your application to support different languages. Symbian recommends that you put all your strings into a RLS file, 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 done 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\UIQ3SDK\epoc32\data\z\resource\apps\simpleEx\SimpleEx.ren"
" c:\Symbian\UIQ3SDK\epoc32\data\z\resource\apps\simpleEx\SimpleEx.rfr"
" c:\Symbian\UIQ3SDK\epoc32\data\z\resource\apps\simpleEx\SimpleEx.rde"
}-" !:\resource\apps\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 (these are listed in section 5.9.4).

The package line in the above example will cause the correct resource file to be installed on the phone (as `\resource\apps\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.

12.4.5 Reading Resource Strings From Code

You should try to avoid using displayable strings directly in the code, since this makes localizing your applications very difficult. For example, I hardcoded a string in the `SimpleEx` example in the following lines:

```
_LIT(KMessage,"Start Selected!");
iEikonEnv->AlertWin(KMessage);
```

The recommended way of doing this is to define a TBUF resource. You could define it as follows:

```
RESOURCE TBUF r_start_selected {buf="Start Selected!";}
```

But a better way is to 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, you could 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.

Note that it is not good to declare large TBuf (or any large stack allocating declaration) as an automatic variable. The TBuf could be allocated as a member variable to a heap allocated class. Or you can use an RBuf or HBuf heap descriptor to store the string. For HBuf, there is a function called `CCoeEnv::AllocReadResourceAsDes16L()` to read a resource string to an HBuf. For example:

```
HBufC* message = iCoeEnv->AllocReadResourceAsDes16L(R_START_SELECTED);
```

Don't forget to delete the HBufC when you are finished.

12.5 Dialogs

Much of your GUI application programming will be concerned with creating and managing dialogs. This section discusses the basics of creating a dialog and presents a simple dialog you can add to `SimpleEx` to illustrate how to create and manage a dialog box and its associated controls.

Dialogs exist on all Symbian OS platforms, although some of the details of their usage vary from platform to platform or even with modes within the same platform (as with UIQ).

A dialog is a 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. For S60, this resource is called `DIALOG`. For UIQ, it is called `QIK_DIALOG`.
2. Create a dialog class 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()` specifying the resource identifier of your `DIALOG` or `QIK_DIALOG` resource.

To best explain the process, let's look at an example of a dialog box for both UIQ and S60. This dialogue is added to `SimpleEx`, and allows you to set the text that is displayed in the middle of the screen, as well as the colour of the text.

12.5.1 Creating a Basic UIQ Dialog

Figure 12.9 shows the UIQ version of the `SimpleEx` dialog described in the last section. 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. 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.

Before describing the UIQ dialog code, let me emphasize that this book covers UIQ version 3. If you have experience already on UIQ versions before that, you will note that UIQ 3 is very different from previous versions, including how dialogs are implemented. S60 3rd Edition GUI programming, on the other hand, is very similar to previous S60 versions.

First, let's look at the resources needed to create this UIQ dialog, which I added to the `SimpleEx` resource file. I will not cover all details of the resource, but will discuss the main points. To simplify the example, I still define the strings directly in the resource file. However, as discussed in the last section, the correct way is to define them in an RLS file using the `rls_string` keyword.

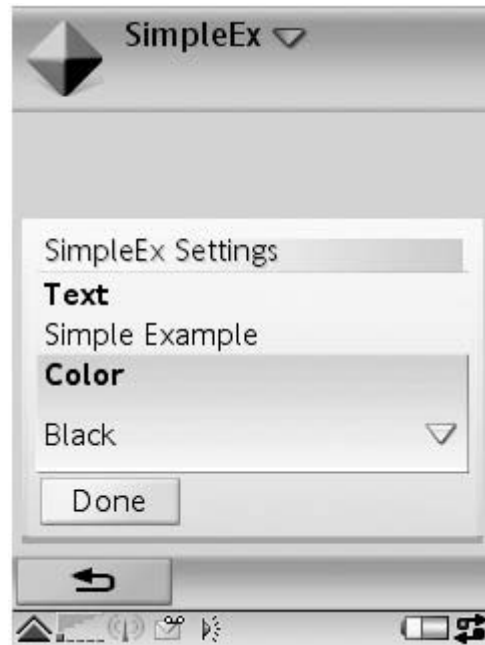


Figure 12.9 SimpleEx Dialog in UIQ

Below is the QIK_DIALOG resource:

```
RESOURCE QIK_DIALOG r_Simpleex_dialog
{
    title="SimpleEx Settings";
    configurations =
    {
        QIK_DIALOG_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            container = r_simpleex_dialog_container;
            command_list = r_simpleex_dialog_commands;
        }
    };
    controls = r_simpleex_dialog_controls;
}
```

Like UIQ views, dialogs can specify one or more display configurations that the dialog will support, along with a container specifying what is on the dialog and the commands associated with the dialog for each mode. In the example, we support only the `KQikPenStyleTouchPortrait` mode. The `controls` attribute points to a `QIK_CONTROL_COLLECTION` resource, which defines a group of controls. The controls in this collection will not necessarily all be displayed on the dialog (although they are for this example), but are meant to be a raw group of controls that can

be referenced by numerous dialogs if desired. As we'll see shortly, the container (`r_simplex_dialog_container` for our example) defines exactly how the dialog is displayed and what controls are used. Below is our `r_simplex_dialog_controls` `QIK_CONTROL_COLLECTION` resource, along with the control resources referenced:

```
RESOURCE QIK_CONTROL_COLLECTION r_simplex_dialog_controls
{
    items =
    {
        QIK_CONTROL
        {
            unique_handle =ESimpleExText;
            type=EEikCtEdwin;
            control = r_dialog_edit;
        },
        QIK_CONTROL
        {
            type=EEikCtChoiceList;
            unique_handle=ESimpleExColor;
            control=r_dialog_choice;
        }
    };
}
RESOURCE CHOICELIST r_dialog_choice
{
    array_id=r_color_list;
}
RESOURCE EDWIN r_dialog_edit
{
    flags = EEikEdwinResizable;
    maxlength = 256;
}
RESOURCE ARRAY r_color_list
{
    items=
    {
        {
            LBUF { txt="Black"; },
            LBUF { txt="Red"; },
            LBUF { txt="Green"; },
            LBUF { txt="Blue"; }
        };
    }
}
```

The `items` attribute of `QIK_CONTROL_COLLECTION` is an array of `QIK_CONTROL` structures. Each `QIK_CONTROL` item indicates a control. Here we have our text box and choice list. The `unique_handle` attribute specifies an ID for the control that your dialog class can reference to access the control. All control handles are stored in the application HRH file as an enum. Pointers to the control object are obtained by calling the `LocateControlByUniqueHandle<control class>(controlId)` method in the UIQ dialog class, as we will see shortly. The `type` attribute specifies the type of control. For example, the

first control is `EEikCtEdwin`, which is a text editor for our text field. A type of `EEikCtEdwin` means that the control resource is of type `EDWIN`.

`EDWIN` is defined as:

```
STRUCT EDWIN
{
    LONG flags=0;
    WORD width=0;
    WORD lines=1;
    WORD maxlength=0;
}
```

In the example, I define the flags so that the field is resizable and the maximum length of the entered string (`maxlength`) is 255.

See the SDK documentation for lists of all the predefined control resource structures, and descriptions of their attributes.

As second control 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 resource type is `CHOICELIST`.

`CHOICELIST` is defined 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
}
```

Here is the command resource for our dialog:

```
RESOURCE QIK_COMMAND_LIST r_simpleex_dialog_commands
{
    items=
    {
        QIK_COMMAND
        {
            id = EDialogDoneCmd;
            type = EQikCommandTypeDone;
            text = "Done";
        },
        QIK_COMMAND
        {
            id = EDialogCancelCmd;
            type = EQikCommandTypeCancel;
            cpfFlags = EQikCpfFlagHardwarekeyOnly;
        }
    };
}
```

These commands will be handled via the `HandleCommandL()` method of our UIQ dialog class. The type attribute defines what kind of command it is, and this information is used to help determine how the command will be presented to the user (menu, softkey, button, etc.). The line:

```
cpfFlags = EQikCpfFlagHardwarekeyOnly;
```

in the last `QIK_COMMAND` indicates that the cancel button will not be on the dialog itself, but instead associated with a hardware key (i.e., a softkey).

Below is our container resource, with two building blocks that define what will be on our dialog:

```
RESOURCE QIK_CONTAINER_SETTINGS r_simpleex_dialog_container
{
    controls =
    {
```

```

        QIK_CONTAINER_ITEM_CI_LI
        {
            type = EQikCtCaptionedOnlineBuildingBlock;
            control = r_dialogs_text_dialog_simpleex_building_block_text;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            type = EQikCtCaptionedOnlineBuildingBlock;
            control = r_dialogs_text_dialog_simpleex_building_block_choice;
        }
    };
}

RESOURCE QIK_SYSTEM_BUILDING_BLOCK
    r_dialogs_text_dialog_simpleex_building_block_text
{
    content =
    {
        QIK_SLOT_CONTENT
        {
            slot_id = EQikItemSlot1;
            caption = "Text";
        },
        QIK_SLOT_CONTENT
        {
            slot_id = EQikItemSlot2;
            unique_handle=ESimpleExText;
        }
    };
}

RESOURCE QIK_SYSTEM_BUILDING_BLOCK
    r_dialogs_text_dialog_simpleex_building_block_choice
{
    content =
    {
        QIK_SLOT_CONTENT
        {
            slot_id = EQikItemSlot1;
            caption = "Color";
        },
        QIK_SLOT_CONTENT
        {
            slot_id = EQikItemSlot2;
            unique_handle=ESimpleExColor;
        }
    };
}

```

The `QIK_CONTAINER_SETTINGS` resource defines a container that is basically a list of controls to be displayed on our dialog. The controls attribute of `QIK_CONTAINER_SETTINGS` defines an array of `QIK_CONTAINER_ITEM_CI_LI` structures, which specifies each control in the container. We would like two items on our dialog, one for the text edit box along with a caption, and one for the color choice list also with a caption. UIQ defines a special mechanism for creating compound controls via what are known as building block resources. We define two of these building blocks: `r_dialogs_text_dialog_simpleex_`

`building_block_text` to define a compound control consisting of the edit box with its caption, and `r_dialogs_text_dialog_simpleex_building_block_choice` for the choice list with its caption. Both are of type `EQikCtCaptionedOnlineBuildingBlock`, which indicates that the item consists of both a caption and a control. The control attribute of the `QIK_CONTAINER_ITEM_CI_LI` structure points to these `QIK_SYSTEM_BUILDING_BLOCK` resources, which defines one or more 'slots' that make up the dialog item. For a captioned control, the resource should consist of two slots, one for the caption and one for the control itself. The control is referenced in slot 2 via the `unique_handle` attribute. This matches up to the control with that handle in our control collection resource (`r_simpleex_dialog_controls`).

Note that although we use containers and building blocks to create a dialog, you can also use them to create views, which display the controls directly on your application screen without using a dialog. The container is referenced in the `page_content` attribute of the `QIK_VIEW_PAGE` structure of the `QIK_VIEW_PAGES` resource associated with your view. Reference the UIQ SDK documentation and examples for more details on this.

Now let's look at the dialog class declaration:

```
class CSimpleExDialog : public CQikSimpleDialog
{
public:
    static TInt RunDlgLD(TDes& aText, TRgb& aColor);
    ~CSimpleExDialog();
    void HandleCommandL(CQikCommand& aCommand);
    void PreLayoutDynInitL();
private:
    CSimpleExDialog(TDes& aText, TRgb& aColor);
    void ConstructL();
    CQikCommandManager& iCmdManager;
    TDes& iText;
    TRgb& iColor;
};
```

And here is the dialog class implementation:

```
TInt CSimpleExDialog::RunDlgLD(TDes& aText, TRgb& aColor)
{
    CSimpleExDialog* self = new (ELeave) CSimpleExDialog(aText, aColor);
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop();
    return self->ExecuteLD(R_SIMPLEEX_DIALOG);
}

CSimpleExDialog::CSimpleExDialog(TDes& aText, TRgb& aColor)
: iCmdManager(CQikCommandManager::Static(*iCoeEnv)), iText(aText),
  iColor(aColor)
{
}
```

```

CSimpleExDialog::~CSimpleExDialog()
{
}
void CSimpleExDialog::ConstructL()
{
}
const TInt KNumColors=4;
const TRgb colorList[KNumColors]={KRgbBlack, KRgbRed, KRgbGreen, KRgbBlue};
void CSimpleExDialog::PreLayoutDynInitL()
{
    CEikEdwin* edwin =
        LocateControlByUniqueHandle<CEikEdwin> (ESimpleExText);
    edwin->SetTextL(&iText);
    TInt currColorIndex=0;
    for (TInt i=0; i< KNumColors; i++)
    {
        if (iColor==colorList[i])
        {
            currColorIndex=i;
            break;
        }
    }
    CEikChoiceList* chlist =
        LocateControlByUniqueHandle<CEikChoiceList>(ESimpleExColor);
    chlist->SetCurrentItem(currColorIndex);
}
void CSimpleExDialog::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
    case EDialogDoneCmd:
    {
        const CEikEdwin* edwin = LocateControl
            ByUniqueHandle<const CEikEdwin> (ESimpleExText);
        edwin->GetText(iText);
        const CEikChoiceList* chlist =
            LocateControlByUniqueHandle<const CEikChoiceList>(ESimpleExColor);
        TInt cIndex = chlist->CurrentItem();
        iColor = colorList[cIndex];
        CloseDialog(aCommand.Id());
        break;
    }
    default:
    {
        CQikSimpleDialog::HandleCommandL(aCommand);
        break;
    }
    }
}

```

The `CSimpleExDialog::RunDlgLD()` method will create the dialog, run it, and destroy it. 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 like we do here. `HandleCommandL()` is called when the user invokes one of the commands we defined in the dialog. Here, we have a Done button: when depressed,

we read the data in the controls and return them in the variables referenced by `iText` and `iColor` (passed to `RunDlgLD()`) and the dialog is dismissed.

Launching the dialog

To launch the dialog, I added a menu item called `Settings` to the command resource in the resource file:

```
RESOURCE QIK_COMMAND_LIST r_simpleex_commands
{
    items =
    {
        QIK_COMMAND
        {
            id = ESimpleExCommand;
            type = EQikCommandTypeScreen;
            text = "Start";
        },
        QIK_COMMAND
        {
            id = ESimpleExDialog;
            type = EQikCommandTypeScreen;
            text = "Settings";
        }
    };
}
```

I added the following case to the command handler switch statement in the `SimpleEx` view class:

```
case ESimpleExDialog:
    CSimpleExDialog::RunDlgLD(iDisplayText, iTextColor);
    DrawNow();
    break;
```

I also updated the `SimpleExDraw()` function to use `iDisplayText` and `iTextColor` to display the window text:

```
void CSimpleExAppView::Draw(const TRect&) const
{
    CWindowGc& gc = SystemGc();
    gc.Clear();
    const CFont* font;
    font = iEikonEnv->TitleFont();
    gc.UseFont(font);
    TRect drawRect = Rect();
    TInt baselineOffset=(drawRect.Height() - font->HeightInPixels())/2;
    gc.SetPenColor(iTextColor);
    gc.DrawText(iDisplayText, drawRect,
                baselineOffset, CGraphicsContext::ECenter, 0);
    gc.DiscardFont();
}
```

12.5.2 Creating a Basic S60 Dialog

Now let's create a dialog for S60 that's equivalent to the UIQ one just presented. There are various ways to create a dialog for S60. For this example, I use an S60 dialog type known as a *form*. A form is a good general-purpose type of dialog to use as it guarantees that the dialog conforms to the S60 UI guidelines. Forms derive from `CAknForm`, which itself extends `CAknDialog`. `CAknDialog` is the dialog class that all S60 dialogs inherit from. Note that using `CAknDialog` directly in S60, however, can cause problems and should be avoided.

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()`.

An S60 form has two modes: in 'view' mode it acts as an application view that displays a list of data items, and in 'edit' mode it 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. You can also 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 12.10.

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

```



Figure 12.10 Form-Based SimpleEx Dialog for S60

```

        {
            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 an S60-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();
    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 TInt KNumColors=4;
const TRgb colorList[KNumColors] =
{
    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 < KNumColors; 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, 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 `EAKnCt-PopupFieldText` control type that is specified in the resource file.

In a form, you override `CAknForm`'s `SaveFormDataL()` function with the logic needed to have your dialog settings take effect. `SaveFormDataL()` is called when the user selects the `Save` option on the form menu. In this example, `SaveFormDataL()` 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. 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 dialog is launched as follows:

```

case ESimpleExDialog:
    CSimpleExForm* form = CSimpleExForm::NewL(iAppView->iDisplayText,
        iAppView->iTextColor);
    form->ExecuteLD(R_SIMPLEEX_DIALOG);
    break;

```

12.5.3 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.

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. In the `LIBRARY` section of your MMP file, you also need to add `eikcdlg.lib` to use dialogs that begin with `CEik`, and `qikdlh.lib` to use dialogs that begin with `CQik`.

S60 predefined dialogs

S60 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 `CAknQueryDialog`. 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 = "Are you sure you want to do this?";
            };
        }
    };
}
```

And the code to invoke it is:

```
CAknQueryDialog* dlg = CAknQueryDialog::NewL(aQueryTxt);
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 = "Enter Text:"; // prompt text
                control = EDWIN
                {
                    width = 5;
                    lines = 1;
                    maxlength = 15;
                };
            };
        }
    };
}
```

And the code to invoke it is:

```
TBuf<20> text; //where the user text will be placed
CAknTextQueryDialog* dlg =
    CAknTextQueryDialog::NewL(text);
// Sets the maximum length of the text editor, if you want to expand or
// shrink from resource value.
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:'. You can supply default data to the text edit box by initializing `text` to some default data before calling `CAknTextQueryDialog::NewL()`.

Other S60 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 S60, 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:

```

HBufC* msg =
    CCoeEnv::Static()->AllocReadResourceAsDes16L(R_MSG_DELETE_PENDING);
CAknConfirmationNote* note = new(ELeave) CAknConfirmationNote();
note->ExecuteLD(*msg);
delete 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.

12.6 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`, and S60-specific controls are in `avkon.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.

12.6.1 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.

Edit 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, you usually use the above classes directly. List boxes are rarely used in UIQ dialogs (choice lists are used instead), and list boxes are most often added directly to your application view.

In S60, you can use list box query dialogs as an alternative to including a list box control in a dialog. S60 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 S60, 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 check box). 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, but not in S60.

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.

12.7 View Architecture

Symbian OS 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. A view can have its own set of commands and its own command handler.

In theory, an application is not required to use the view architecture (and in fact our `SimpleEx` S60 example does not), however, with UIQ (UIQ version 3 specifically), its architecture depends so much on it that it is basically required (although our `SimpleEx` example contains just one view).

The view class for UIQ, as we have seen, is `CQikViewBase`. The view class for S60 is `CAknView`. Once you have set up your view classes properly, you can register them, set one as the default, and switch between them at will. For detailed information on creating and managing views, reference [Harrison], as well as the SDK documentation.

12.8 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 a special resource file known as a localized application information resource file (typically named `<application name>_loc.rss`). If this resource file does not exist, then the system uses a default icon and caption for the application.

This section provides only a brief introduction to using localizable application information files to supply your application with icons and captions. You should refer to the SDK documentation for more detail, as well as examples.

The general sequence of events needed to define your application caption and icons is as follows:

- Create your icon images using an image editor.
- Write the localizable application information resource file defining the caption, number of icons, and other information.

- Update the MMP to build this resource file (output will be a binary RSC file) and to write the bitmaps to a MBM (multi-bitmap) format file. If you use scalable bitmaps, a MIF file is produced instead, but not via the MMP file (this will be discussed a little more shortly).
- Update your PKG file so that the installation will copy the application information RSC and icon bitmap (MBM or MIF file) to the `\resource\apps` directory of the smartphone.

12.8.1 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.

There is also now a Scalable Vector Graphics Tiny (SVG-T) standard that allows you to create a scalable image and thus you need only one copy.

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.

All icon graphics get converted to MBM files, which is what Symbian uses. This can be done in the MMP file, or via separate tools. The details of creating and building bitmaps can be found in the SDK documentation.

12.8.2 Creating the Localized Application Information Resource

Here is a sample localized application information resource file:

```
#include <AppInfo.rh>
RESOURCE LOCALISABLE_APP_INFO
{
    caption_and_icon =
    {
        CAPTION_AND_ICON_INFO
        {
            caption = "My Application";
            number_of_icons = 3;
            icon_file = "\\Resource\\Apps\\MyAppIcon.mbm";
        }
    };
}
```

The `LOCALISABLE_APP_INFO` resource defines the application information. The attribute `caption` is your application's caption text.

Although I have it hardcoded here, you can use an RLS header file to define the strings in different languages and reference the `rls_string` identifier for the caption (see section 12.4.4). The attribute `icon_file` specifies the MBM file that contains your icons and `number_of_icons` specify how many icons are in the MBM file.

12.8.3 Building Information File and Bitmaps

To build the localizable information file, enter the following in the MMP:

```
START RESOURCE myapp_loc.rss
TARGETPATH \resource\apps
END
```

If you are using BMP files for your images, you can build them via the MMP file. As mentioned previously, if you use scalable bitmap files (SVG-T format), you do not use the MMP to produce the MIF file (scalable bitmap equivalent to the MBM file). Instead, you specify with a separate makefile that is also referenced in the `bld.inf` file. Reference the S60 example programs for how this is done. At the time of writing, UIQ does not support having the scalable images SVG-T files in your application. Using BMP files, however, works on both platforms.

An example of building the icon bitmaps in your MMP is as follows:

```
START BITMAP myapp.mbm
TARGETPATH \resource\apps
SOURCEPATH ..\images
SOURCE c24 myapp_small.bmp
SOURCE 8 myapp_small_mask.bmp
SOURCE c24 myapp_large.bmp
SOURCE 8 myapp_large_mask.bmp
END
```

This would generate a MBM file with two icons, one small and one large. The `c24` indicates a 24-bit color image. The bitmaps with the `_mask` in are monochrome bitmaps used to define the image mask. You then need to update your package file to copy the localizable information RSC file and the bitmap MBM file to `\resource\apps` on the smartphone.

References

- [Babinv1] Steve Babin (2005)
*Developing Software for Symbian OS:
An Introduction to Creating Smartphone Applications
in C++*
Publisher: John Wiley & Sons Ltd
ISBN: 0470018453
- [Campbell] Iain Campbell (2007)
*Symbian OS Communications Programming, Second
Edition*
Publisher: John Wiley & Sons Ltd
ISBN: 0470512289
- [Harrison] Richard Harrison and Mark Shackman (2007)
*Symbian OS C++ for Mobile Phones, Vol. 3:
Application Development for Symbian OS v9*
Publisher: John Wiley & Sons Ltd
ISBN: 0470066415
- [Heath] Craig Heath (2006)
*Symbian OS Platform Security:
Software Development Using the Symbian OS Security
Architecture*
Publisher: John Wiley & Sons Ltd
ISBN: 0470018828

- [Sales] Jane Sales (2005)
Symbian OS Internals: Real-Time Kernel Programming
Publisher: John Wiley & Sons Ltd
ISBN: 0470025247
- [Stichbury] Jo Stichbury and Mark Jacobs (2006)
The Accredited Symbian Developer Primer: Fundamentals of Symbian OS
Publisher: John Wiley & Sons Ltd
ISBN: 0470058277
- [Stichburyv1] Jo Stichbury (2004)
Symbian OS Explained: Effective C++ Programming for Smartphones
Publisher: John Wiley & Sons Ltd
ISBN: 0470021306

Online References

Symbian Developer Network: **<http://developer.symbian.com>**

Other online references and resources are available on the wiki page for this book, which can be found on the Symbian Developer Network (**<http://developer.symbian.com/wiki>**). Please leave a comment on the wiki page if you find a broken link, and we will do our best to find a replacement.

Index

- 1G networks 4
- 2G networks 4–6
- 2.5G networks 4, 6
- 3G networks 6, 16–19
- 4G networks 4

- a (function arguments) prefixes, concepts 120
- ABIs *see* application binary interfaces
- abld 36–8, 56, 127–31, 142–4, 145–9
- abld freeze 37–8, 128–31, 142–9
- abstract classes, C++ concepts 93–4, 98–100
- ActivateL 374–7
- active objects
 - see also* asynchronous functions
 - active scheduler 81–2, 249–75
 - blocks 269, 304–5
 - CActive 82, 249–75
 - cancellations 251–2, 253–4, 260–1, 263–8
 - class creation 251–4, 262–8
 - complexity problems 269–71
 - concepts 69, 249–75, 285–6, 287–92, 300–1, 335–57
 - construction 251–2, 262–8
 - creation 251–4, 262–8
 - CTimer 271–5, 321
 - definition 249
 - examples 262–8, 335–57
 - high-level view 249–50
 - issues 269–71
 - network programming 335–57
 - non-preemptive multitasking model 250–1, 269–71
 - outstanding requests 270–1
 - priorities 260
 - removal 262
 - requestor implementation 253, 262–8
 - servers 303–4, 311–12, 313–21
 - sockets 335–57
 - stray-signal exceptions 269–70
 - thread uses 271–5
 - tips 269–71
 - uses 250–1, 271–5
- active scheduler
 - concepts 81–2, 249–75, 313–21, 335–57
 - customization 257–60
 - definition 249–50, 254
 - detailed workings 255–6
 - error handling 258–60, 269–70
- event-loop pseudo-code 255–6
- examples 262–8, 313–21
- GUI applications 254–3, 269, 342–3, 355–6
- high-level view 249–51
- installation 254–7, 313–21
- leave/trap mechanism 258–9, 262–8, 269–70, 314–21
- starting 254–7, 313–21
- ActivePerl 518 25–6
- Add 214–15, 249–54, 257, 269–70
- AddField 397–401
- AddFirst 214
- AddItemL 397–401
- AddLast 214
- addresses
 - IP (Internet Protocol) 325–6, 331–4, 338–9, 349–53
 - memory 71–2, 293–7
 - MMU 71–2, 74–5, 294–7
 - physical/virtual memory addresses 71–7, 294–7
- AddText 310–21
- AddViewL 52
- Adjust 294–6
- After 119, 251, 269, 271–5, 287–8
- aknapp.h 40–4

- aknappui.h 40–4
- akndoc.h 40–4
- alarms 90
- AlertWin 50–4, 402
- AllFiles 226, 229, 232, 243
- Alloc 181–5
- AllocL 181–3
- AllocLC 110–11, 181–3
- animation plug-ins
 - see also* graphics
 - concepts 85–6
- APIs *see* application programming interfaces
- apparc.lib 128–31
- AppDllUid 56, 369–77
- Append 167–8, 171, 175–7, 179–80, 192–4, 210–13, 340–57
- AppendFill 193–4
- AppendFormat 194
- AppendJustify 193–4
- AppendNum 193–4
- application binary interfaces (ABIs) 133–5
- application engines, concepts 90–1
- application framework
 - see also* graphical user interface framework
 - concepts 64–5, 83–5, 359–411
- application processor, concepts 80–1
- application programming interfaces (APIs) 63–4, 77–9, 86–9, 104–14, 117, 143–4, 161–215, 359–411
 - base libraries 63–4
 - BSD socket API 86–9, 323–4, 328–34, 340–1
 - capabilities 129, 131, 160, 220–31
 - data organization classes 165–215
 - platform security 90–1, 217–46
 - sockets 86–9, 323–4, 328–57
 - TCP/IP 323–57
 - types 77–9, 90–1, 104–14
 - User 94–5, 104–14, 115
- application protocols, concepts 90–1
- application services/engines/protocols, concepts 64–5, 90–1
- application views 38–56, 360–5, 367–411
- application-specific cleanups, concepts 107–9
- applications
 - see also* executables; processes; software
 - availability 1, 12, 15–16
 - captions 409–11
 - classes 38–56, 365–411
 - communications architecture 86–9
 - components 38–56, 365–411
 - concepts 1–2, 12, 38–56, 83–5, 149–60, 365–411
 - downloads 1, 9–10, 149–50
 - example applications 38–56, 365–411
 - GUI programming 38–56, 359–411
 - header files 39–56, 115–18, 126–31, 141–4, 151–60, 365–411
 - icons 366–7, 409–11
 - installation 23–5, 39–56, 59–1, 149–60, 236–7, 365–411
 - package definition files 39–56, 59–61, 150–60, 365–411
 - platform security 90–1
 - project build files 54–6, 115–18, 119–49, 365–411
 - registration files 48–9, 366–7, 384–5
 - resource files 44–56, 128–31, 141–4, 365–7, 377–411
 - signed applications 30, 60–1, 91, 160, 218–46, 324
 - SimpleEx 38–56, 262–8, 353–6, 365–411
 - source files 49–54, 128–31, 141–4, 365–411
 - TCP/IP applications 323–57
 - third-party suppliers 1, 13–14
 - types 1, 3–4, 12
 - UIDs 38–56, 115–18, 119, 128–31, 135–7, 152–60, 237–8, 239, 367–411
 - view architecture 409
- APP_REGISTRATION_INFO 48–9, 384–5
- apprun.exe 118–19
- arguments, processes 279–81
- Arima U308 27
- ARM 24–6, 59–61, 71–2, 79, 118–19, 124–5, 132–5, 139–40
 - see also* CPUs
- ARMV5... 132–5
- ARMV6 132–5
- ARRAY resource 359–60, 390–6
- arrays
 - classes 205–13
 - concepts 166–8, 173–4, 198, 205–13, 378–80, 390–6
 - data-finding method 211
 - descriptors 166–8, 173–4, 198, 206–8
 - dynamic arrays 209–13
 - fixed arrays 205–6
 - inserting/appending data 210–11
 - methods 210–13
 - RArray 209–13
 - sorting method 211–12
 - templates 205–6, 209–13
 - thin templates 205–8
- ASCII 281–2, 332
- assert macros, concepts 114–15
- _ASSERT_ALWAYS 114–15
- _ASSERT_DEBUG 114–15
- Assign 185–6
- ASSP 77–9
- asynchronous functions
 - see also* active objects; Logon
 - cancellations 251–2, 253–4, 260–1, 263–8
 - concepts 81–2, 247–75, 285–6, 287–92, 300–1, 311–12, 335–57
 - definition 247
 - examples 262–8, 335–57

- high-level view 249–51
- request semaphores 248–75, 300–1
- servers 303–4, 311–12, 313–21, 335–57
- sockets 335–57
- TRequestStatus 247–61, 336–57
- AT&T 6
- audio 4, 8, 227
- automatic network connections, concepts 323–4
- AVKON_CONFIRMATION_QUERY 404
- avkon.lib 56, 128–31
- avkon.rh 44–56, 382, 405
- avkon.rsg 44–56
- BackPtr 200–3
- BackSpace 310–21
- badef.rh 391–2
- badesca.h 207–8
- baf1.lib 207–8
- bandwidth, concepts 3–11
- BARM 146
- Base 294–6
- base classes, TDes... 167–71
- base libraries
 - see also libraries
 - concepts 63–4
- baseband processor, concepts 80–1
- BaseConstructL 372–7
- basic data types, concepts 94–5, 119–20
- batteries 11, 362–5
- Begin 206
- BenQ P30 27
- Berkley Unix see BSD
- bin 151, 231–2, 366–7
- binary data
 - see also descriptors
 - concepts 165–8, 198
 - strings 165–6
- bind 329
- bitmaps 231, 304–5, 365–7, 404–5, 409–11
- BlackBerry 21
- blanket grant notifications, capabilities 222–3
- bld.inf 37–8, 39–56, 126–31, 365–411
 - see also Component Description File
- bldmake 35–8, 56–8, 127–31, 142–4, 145–9, 163–4
- blocks
 - active objects 269, 304–5
 - memory 71–2, 269
- Bluetooth connectivity 1, 9–10, 12, 16–22, 23–4, 64–5, 86–9, 150, 220–1, 334–5
- BMP files 410–11
- Borland C++ Builder 28, 36–7
 - see also Integrated Development Environments
- browsing 3–4, 8–9, 16–22, 323–57
 - see also Internet specifications 16–22
- BSD socket API 86–9, 323–4, 328–34, 340–1
 - see also sockets
 - concepts 323–4, 328–34, 340–1
 - examples 329–34
- bt.prt 86–9
- buffer descriptors
 - see also descriptors; TBuf...
 - concepts 168–70, 173–7, 292–3, 380–3
 - definition 158, 173
- build targets 34–8, 56–61, 124–5, 128–35, 142–9, 153–60, 163–4
 - emulator 132–5
 - native build targets 132–5
 - pre-version 9 SDK build targets 135
- build tools 23–61, 123–60
- buttons 359–411
- BWINS 146
- BYTE 378–80, 406
- C++ 27–30, 71–2, 84–5, 93–122, 166–8, 323–4, 329–57
- basic data types 94–5
- concepts 93–122, 166–8, 323–4, 329–30
- features 93–4
- nonstandard characteristics 94–5
- overload features 93–4, 108–9, 111–13, 279–80
- overview 93–4
- sockets 323–4, 329–57
- STL 94–5
- strings 166–8
- Symbian OS 93–122, 166–8, 323–4, 329–57
- templates 93–4, 109–10, 203–5
- c: drive 70–1, 138–41
 - see also internal flash disk
- C (heap-allocated) classes, concepts 96–100, 120
- CA see certificate authority
- caches 76–7
- CActive 82, 249–75, 305–21
 - see also active objects; client-server...
- CActiveScheduler 249–75, 314–21
 - see also active scheduler
 - concepts 249–51, 252–75
 - methods 249–54, 257, 269, 274–5, 314–21
- CActiveScheduler::Add 249–54
- CActiveScheduler::Start 250–1, 269, 274, 314–21
- CAknApplication 40–56, 368–77
- CAknAppUi 41–56, 265–8, 368–77
- CAknConfirmationNote 405
- CAknDialog 388–405
- CAknDocument 41–56, 368–77
- CAknErrorNote 405
- CAknForm 397–401
- CAknInformationNote 51–4, 405
- CAknListQueryDialog 405
- CAknNumberQueryDialog 405
- CAknProgressDialog 405

- CAknQueryDialog 403–5
- CAknSingleGraphicStyle-ListBox 407
- CAknSingleStyleListBox 407
- CAknTextQueryDialog 404–5
- CAknTimeQueryDialog 405
- CAknView 368–77, 409
- CAknViewAppUi 368–77
- CAknWaitDialog 405
- calendars 9, 90, 360–5
- cameras 3, 16–22, 222, 227
- Cancel 251–2, 260–2, 263–8, 348–57
- capabilities
 - assessment of needs 227–8
 - basic capabilities 220–3
 - blanket grant notifications 222–3
 - categories 220
 - concepts 129, 131, 160, 219–31
 - DLLs 229–31
 - extended capabilities 220, 223–5
 - phone manufacturer capabilities 220, 225–7, 243–4
 - platform security 90–1, 129, 131, 160, 219–32
 - single-shot grant notifications 222–3
 - specification syntax 229–31
 - TCB 90–1, 227, 229, 231–2, 243
- CAPABILITY 129, 131, 160, 220, 229–30, 355
- Capacity 200–3
- CAPTION_AND_ICON_INFO 410–11
- captions 409–11
- Carbide 23, 27–30, 32–8, 46–7, 58, 144, 151, 160, 229–30, 234–5
- CArrayFix... 209–13
- CArrayPtr... 209–13
- case conversions, descriptors 165–6, 191–2, 194–5
- CBA *see* control button array
- CBase 96–100, 107–10, 120, 269
- CBluetoothSocket 221–2
- CBufBase 199–203
- CBufFlat 199–203
- CBufSeq 199–203
- CBufStore 371–7
- CCamera 222
- CCirBuf 214–15
- CCoeControl 40–6, 265–8, 368–77
- CConsoleBase 161–5
- CConsoleBase::Getch 162–5
- CConsoleBase::Printf 162–5
- CContactDatabase 222
- CCountdown 262–8
- CDesC8Array... 206–8
- CDesC16Array... 206–8
- CDesCArray... 207–8
- CDirectFileStore 371–7
- CDMA network protocol 4–5
- CEikApplication 43–56, 368–77
- CEikAppUi 43–56, 368–77
- CEikChoiceList 408
- CEikColumnListBox 407
- CEikComboBox 408–9
- CEikDialog 121, 387–97
- CEikDocument 43–56, 368–77
- CEikEdwin 390–405, 406
- CEikHierarchicalListBox 407
- CEikonEnv 402
- CEikProgressInfo 407
- CEikSetPasswordDialog 403
- CEikTextListBox 407
- CEikTimeDialogSetCity 403
- CEikTimeDialogSetTime 402–3
- cell phones *see* mobile phones
- Cellmania 240–1
- CEmbeddedStore 371–7
- certificate authority (CA) 233, 236
- CFileman 226
- CFileStore 371–7
- char 94–5
- CHARACTER_SET keyword, resource files 378–80
- check boxes 408
- choice lists 406–7, 408
- CHOICELIST 390–6, 408
- chunks
 - see also* RChunk
 - concepts 72–7, 97–8, 293–302
 - creation 72–3, 293–6
 - detailed workings 293–7
 - types 72–3, 97–8, 293–7
- Cingular 3125 20
- circuit-switched voice
 - communication 4–6
 - see also* GSM...
- circular buffers, concepts 214–15
- classes
 - see also* C...; M...; R...; T...
 - active objects 251–4, 262–8
 - applications 38–56, 365–411
 - client–server model 303–21
 - concepts 95–100, 115, 119–20, 165–215, 367–77
 - controls 406
 - descriptors 167–86
 - dialogs 394–6
 - DLLs 115
 - libraries 115–18
 - naming conventions 95–6, 119–20
 - programming basics 95–100
 - Series 60 (S60) 40–56, 367–411
 - sockets 97–100, 304–5, 334–57
 - Symbian OS 95–100, 119–20, 165–215, 367–77
 - templates 204–5
 - types 95–100, 119–20, 165–215
 - UIQ 41–56, 367–411
 - variable names 119–20
- clean 128–31
- CleanClosePushL 180, 184–6
- Cleanup 352–3
- cleanup, exception handling 101–2, 105–14, 162–5
- cleanup stack
 - complexities 107–9

- concepts 105–14, 162–5, 180, 202–3
 - object types 107–10
- CleanupClosePushL 109–10
- CleanupDeletePushL 109–10
- CleanupReleasePushL 109–10
- CleanupStack 106–14, 119–20, 121, 180, 184–6, 202–3, 207–8, 262–8, 273–4, 314–21, 374–7
- CleanupStack::Pop 106–14, 262–8, 315–21, 374–7
- CleanupStack::PopAndDestroy 106–14, 180, 184–6, 202–3, 274, 314–16
- CleanupStack::PushL 106–14, 119–20, 121, 207–8, 262–8, 273–4, 314–21, 374–7
- Clear 396
- client classes, concepts 303–21
- client-side code, sockets 328–34
- client–server model
 - see also CActive; CServer; CSession; RSessionBase
- active objects 303–4, 311–12, 313–21
- classes 303–21
- concepts 68–70, 83–5, 292–3, 303–21, 326–57
- definition 303–4
- examples 306–21
- implementation 312–21
- message-processing example 316–18
- overview 304–6
- pointers 316–19
- service-invoking methods 310–12, 349–53
- shutdown issues 320–1, 334
- sockets 326–57
- starting 307–10, 313–16
- TCP/IP 326–57
- TextBuffServ example 306–21
- transient servers 320–1
- ClientRect 374–7
- Close 97–100, 109–10, 120, 180, 184–6, 210–13, 263–8, 279–80, 281–2, 294–6, 299, 307–12, 320–1, 335–57
- CnvUtfConverter 199
- code 27–30, 64–5, 70–5, 84–5, 93–122, 323–4
 - see also threads
- C++ 27–30, 71–2, 84–5, 93–122, 323–4
- chunks 72–3
- critical sections 301
- naming conventions 68, 95–6, 119–21
- shared code 65–8
- start-up code 70–1
- CodeWarrior 36
- collation method, descriptor
 - comparisons 187–90
- colorList 399–401
- combo boxes 408–9
- CommandLine 280–6
- CommDD 226–7, 229, 243
- committed memory 294–7
- communication database,
 - communications architecture 86–9
- communication methods,
 - smartphones 3–11
- communications architecture
 - see also local device
 - communication features
 - components 86–9
 - concepts 3–11, 16–22, 64–5, 86–90, 323–57
 - overview 86–90
- Communicators (Nokia) 14–20, 27
 - see also Nokia
- Compare 187–9
- comparisons, descriptors 187–9
- competitors, Symbian OS 13–14, 20–2
- Complete 318
- Component Description File 39–56, 126–31
 - see also bld.inf
- compound/simple controls,
 - contrasts 393–4
- Compress 200–3
- computers 3, 7–11
 - see also PCs
- CONE (control environment),
 - concepts 83–5, 377–8
- cone.lib 128–31
- configuration, emulator 31–2, 138–40
- Connect 97–100, 307–10, 335–57
- connection agents,
 - communications architecture 86–9
- connectivity features 1, 9–10, 16–22, 64–5, 86–90, 149–50
 - see also local device
 - communication features
 - concepts 9–10, 64–5, 86–90, 149–50
 - specifications 16–22
- const 118–19, 172
- constants, naming conventions 120–1
- ConstructL 51–4, 112–13, 251–2, 262–8, 312–21, 372–7, 400
- constructors 51–4, 112–13, 177–80, 251–4, 262–8, 312–21, 348–57, 372–7
- leaves 112–14
- two-phase constructors 112–14
- contact entries 9, 90
- context switches 76–7, 82
- control button array (CBA) 383
- controls
 - classes 406
 - compound/simple contrasts 393–4
 - concepts 83–5, 310–21, 377–8, 390–409
 - drawing controls 41, 52–4
 - GUI controls 405–9
 - header files 405–6
 - implementation options 406
 - libraries 406
 - requirements 406

- conversions, descriptors 165–6, 191–2, 194–7, 199
- Copy 167–8, 175–7, 179–80, 184, 191–2, 194, 198–9, 340–57
- copying data, descriptors 167–8, 175–7, 179–80, 191–2
- Count 206
- CPermanentFileStore 371–7
- CPP files 162–5, 265–8, 365–411
- CPtrC8Array 207–8
- CPtrC16Array 207–8
- CPtrCArray 207–8
- CPUs 24–6, 59–61, 71–2, 77–9, 139–40, 293–6
 - see also* ARM...; x86...
- CQikApplication 41–56, 368–77
- CQikAppUi 41–56, 368–77
- CQikDocument 42–56, 368–77
- CQikSimpleDialog 388–405
- CQikViewBase 42–56, 369–77, 409
- CQikZoomDialog 403
- crashes 12, 165–6, 278
- Create 184–6, 279–80, 288–91
- CreateAppUiL 50–2, 369–77
- CreateDocument 369–77
- CreateGlobal 293–302
- CreateLocal 296–7, 299–300
- CreateSession 305–21
- CreateWindowL 374–7
- critical sections, concepts 301
- cryptography 12, 219, 233
- Crystal 15
 - see also* Series 80
- CSD network protocol 5, 356–7
 - see also* HSCSD...
- CServer
 - concepts 305–21
 - methods 305–21
- CSession 305–21
- CSimpleExApplication 40–56, 262–8, 369–77
- CSimpleExAppUi 43–56, 265–8, 353–6, 372–7
- CSimpleExAppView 43–56, 262–8, 374–7
- CSimpleExDialog 394–6
- CSimpleExDocument 41–56, 369–77
- CSimpleExForm 399–401
- CSR request files, developer certificates 245–6
- CStreamDictionary 371–7
- CStreamStore 371–7
- CSY modules, serial communications server 86–9
- CTelephony 221, 224
- CTextBuffServ 313–21
- CTimer 271–5, 321
- CTrapCleanup::New 107–8
- Current 257
- CWeatherInfo 346–57
- d: drive 71
 - see also* removable memory cards
- D suffixes, concepts 121
- data buffers
 - see also* descriptors
 - concepts 165–6, 168–215, 292–3
- data chunks
 - see also* chunks
 - concepts 72–5
- data collection classes, concepts 213–15
- data input methods 13–22, 78–9, 83–5, 137–41, 303–4, 360–411
 - concepts 13–22, 78–9, 83–5, 137–41, 303–4, 360–411
 - platforms 13–22, 359–411
 - specifications 16–22
- data organization classes, concepts 165–215
- data transfers, concepts 3–11
- data types
 - classes 96–100, 120, 165–215
 - concepts 94–5, 120
- data-caging concepts, platform security 91, 226, 231–2
- database manager 64, 221–2
- debuggers 24–6, 114–15, 127, 137–8, 165–6
 - assert macros 114–15
 - log files 25–6, 141, 263–5
 - Windows development tools 24–6, 127, 137–8
- DEF files 145–9
 - see also* freezing
- DEFNAME 146
- Delete 195–6, 200–3
- DeleteField 397–401
- delete trap 107–8
- DeleteCurrentItem 397–401
- Deque 262
- Des 180–6
- descriptors
 - see also* HBuf...; RBuf...; TBuf...; TPtr...
 - 8/16 bit conversions 198–9
 - advantages 165–6
 - appending methods 167–8, 171, 175–7, 179–80, 192–4
 - arrays 166–8, 173–4, 198, 206–8
 - binary data 165–8
 - buffer descriptors 168–70, 173–7, 292–3, 380–3
 - case conversions 165–6, 191–2, 194–5
 - class types 167–86
 - comparisons 187–9
 - concepts 165–215, 307
 - conversions 165–6, 191–2, 194–7, 198–9
 - copying data 167–8, 175–7, 179–80, 191–2, 194
 - definitions 165–6, 168, 173, 177, 180–1
 - deletions 195–6
 - examples 166–8
 - exception handling 165–6
 - fill method 192–3
 - formatting data 194
 - heap descriptors 168–70, 180–6, 315–16
 - hierarchy 169–71
 - importance 165–6

- lengths 169–98
- memory layouts 174–86
- memory overruns 165–6, 176–7
- methods 186–98
- modifiable/non-modifiable
 - descriptors 169–77, 181–3, 187–98
- modifying methods 191–7
- non-modifying methods 187–98
- NULL-terminated string
 - conversions 196–7
- pointer descriptors 168–70, 177–80, 318–20
- size-setting method 186, 197, 202–3
- sub-strings 188–90
- types 168–215
- wildcard searches 189–90
- destructors 96–100, 107–9, 260–2, 346–57
- DevCertRequest 245–6
- developer certificates 236, 244–6
- Developing Software for Symbian OS (Babin), new edition 1–2
- development tools
 - see also* software development kits; Windows development tools
 - basic pieces 23–5
 - components 23–31, 123–60
 - concepts 2, 23–38, 123–60
 - examples 23–61
 - firing up 31–8
 - needs 23–5
 - overview 23–5, 125–6
 - problems 37–8
 - quick start guide 23–61
 - tools 12, 13–15, 23–61, 123–60
- device contrasts, emulator 139–40
- device drivers
 - concepts 70–1, 77–9, 86–9, 139–40
 - definition 89
 - emulators 139–40
 - ROM 70–1, 138–41
 - devices 31–8
 - dial-up connections,
 - drawbacks 5
 - DIALOG resource 387–405
 - dialogs 359–60, 363–5, 387–405
 - classes 394–6
 - creation 387–405
 - launching 396
 - list boxes 406–7
 - resource definition 388–405
 - Series 60 (S60) 364–5, 387–405
 - stock dialogs 401–5
 - UIQ 363, 387–405
 - digital signing
 - see also* signed applications
 - concepts 232–3
 - dir 36–7
 - direct screen access, APIs 85–6
 - directories 34–8, 59–61, 123–6, 150–60, 231–2, 303–4
 - DiscardFont 374–7, 396
 - DiskAdmin 226–7, 229, 243
 - DispatchMessageL 316–19
 - DLG_LINES 397–406
 - DLLs *see* dynamic link libraries
 - DMA controller 79
 - DNS *see* Domain Name System
 - DoCancel 248–9, 251–2, 253–4, 263–8, 346–57
 - document classes, applications 38–56, 367–77
 - documentation
 - OS requirements 12
 - SDK directories 125–6
 - Domain Name System (DNS) 332–4, 349–53
 - domain names, IP addresses 332–4, 338–9, 349–53
 - Doom network service 327
 - downloaded applications 1, 9–10, 16–22, 149–50
 - Draw 41, 52–4, 266–8, 374–7, 396
 - drawing controls
 - concepts 52–4
 - graphics context (GC) 52–4
 - DrawRect 396
 - DrawText 52–4, 374–7, 396
 - drive letters 70–1
 - DRM capability 226–7, 243
 - DSO files 134–5, 142–3
 - Duplicate 299–300
 - DVB-H 11, 16
 - dynamic arrays
 - see also* arrays
 - concepts 209–13
 - dynamic buffers
 - see also* CBuf...
 - area pointers 205–6
 - class diagram 199–200
 - concepts 199–203
 - inserting/deleting data 202–3
 - methods 201–3
 - reading/writing methods 201–3
 - size changes 202–3
 - types 199–200
 - uses 199
 - dynamic link libraries (DLLs)
 - 63–4, 65–8, 78–9, 86–9, 114–18, 141–9
 - capabilities 229–31
 - classes 115
 - concepts 65–8, 78–9, 86–9, 114–18, 141–9, 229–31
 - creation 115–18, 141–4
 - definition 65–6, 115
 - emulator 118–19, 140
 - executables 134–5
 - extension names 68
 - freezing mechanism 144–9
 - GUI applications 66, 118–19
 - MMP files 141–9
 - multiple DLLs 117–18
 - ordinals 143–9
 - programming basics 114–18
 - RLibrary 117–18, 143–4
 - rules 115–18
 - types 65–8, 78–9, 86–9, 114–18
 - DynInitMenuPanelL 397–401

- e: drive 71
 - see also* removable memory cards
- E (enumeration members) prefixes, concepts 121
- e32base.h 162–5
- e32cmn.h 124
- e32cons.h 162–5
- e32debug.h 141
- e32des8.h 169–70
- e32des16.h 169–70
- E32Main 119, 163–4, 273–5, 278–86, 314–21, 367–77
 - see also* processes
- e32std.h 101, 124, 169–70
- E32USER – CBase 42 349
- E32USER – CBase 46 269
- E32USER – CBase 47 259
- E32USER – CBase 69 163
- EAKnSoftkeyExit 51–4
- Echo network service 326–7
- Eclipse 28–30
 - see also* Carbide
- ECOM API, application protocols 67
- edbms.lib 64
- EDGE network protocol 4–5, 6, 17, 323–4
 - see also* GSM...
- Edit 397–401
- Edit Label 397–401
- EditCurrentLabel 397–401
- editor controls
 - see also* controls
 - concepts 310–21, 397–405, 406, 409–11
- editors, Windows development tools 24–6
- EDWIN 390–405, 406
- efsrv.lib 64
- EIK_APP_INFO 45–56, 380–3
- eikcdlg.lib 403
- eikcore.lib 128–31
- eikon.rh 44–56, 378, 382, 405
- EKA1, concepts 77, 80–1
- EKA2, concepts 77–81
- ELeave 111–13, 117–18, 147–9
- emails 3–4, 8, 12, 16–22, 64–5, 150, 323–4
- embedded SIS files, installation 156
- emulator
 - see also* epoc
 - build targets 132–5
 - concepts 24–6, 31–8, 56–8, 118–19, 124–5, 137–41, 164, 286
 - configuration 31–2, 138–40
 - device contrasts 139–40
 - DLLs 118–19, 140
 - EXE files 119
 - fonts 139–40
 - memory capacity 138–9
 - multiple processes 140, 286
 - pixels 139–40
 - platform security 228–9
 - quick test 31–8
 - running 138–9
 - SDK 24–6, 31–8, 124–5, 137–41, 164
 - Series 60 (S60) 24–5, 32–8, 56–8, 138–40
 - static data in DLLs 118–19, 140
 - UIQ 57–8
 - virtual drives 138–41
- encapsulation features, C++ 93–4
- End 206
- enum 121, 373
- ENUM keyword, resource files 378–80
- enumerations, naming conventions 121
- EPOC 12–13, 77
- epoc 31–8, 138–41
 - see also* emulator
- EPOC32 93
- epoc32 123–6, 127–31, 138–41, 164, 228
- epoc32/build 124–5, 127–31, 138–41
- epoc32/data/epoc.ini 228–9
- epoc32/data/z 124–5, 138–41, 152–60
- epoc32/include 124, 128–31, 138–41, 278
- epoc32/release 124–5, 127–31, 132–5, 138–41, 142–4, 152–60
- epoc32/tools 124–5
- epoc32/winscw 125–6
- EPOC_DRIVE_D 138–40
- epoc/include 169
- epoc.ini 31–2, 138–41, 228–9
- epocprocesspriority 284–5
- EPOCSTACKSIZE 131
- epocwind.out 275
- EPriority... 290–1
- Ericsson, Symbian ownership 13–14
- Error 258–60
- errors 12, 101–14, 165–6, 257–60, 269–70
 - see also* exception...
 - active scheduler 257–60, 269–70
 - assert macros 114–15
 - concepts 101–14, 258–9, 269–70
 - leave/trap mechanism 101–14, 121–2, 180–1, 212–13, 253–4, 258–60, 262–8, 314–21
 - panics 113–14, 163, 168, 285–6, 313–21
 - return codes 101
- ESimpleExCommand 50–4
- Esock 90
- esock.dll 86–9
- ETEL server, communications architecture 86–9, 90
- euser.dll 64, 66, 79
- euser.lib 128–31, 134–5, 278
- EV-DO 5
- event handlers 84–5, 249, 253–4
- events, active objects 82, 247–75, 285–6, 287–92, 300–1
- examples, quick start guide 23–61
- Excel 20–2

- exception handling 94–5, 101–14, 121–2, 165–6, 180–1, 212–13, 253–4, 258–60, 262–8
 - see also* errors
 - assert macros 114–15
 - cleanup 101–2, 105–14, 162–5
 - concepts 101–14, 165–6
 - descriptors 165–6
 - leave/trap mechanism 101–14, 180–1, 212–13, 253–4, 258–60, 262–8, 314–21
 - panics 113–14, 163, 168, 285–6, 313–21
 - programming basics 101–14
 - return codes 101
 - stray-signal exceptions 269–70
- EXE files 65, 68–70, 74–6, 118–19, 128–31, 277–86
 - see also* executables; processes
 - emulator 119
 - programming basics 118–19
 - structure 118–19
- executables 65, 68–70, 74–7, 118–19, 124–5, 149–60, 277–86
 - see also* applications; EXE files; processes
 - concepts 118–19, 124–5, 134–5, 277–86, 288
 - DLLs 134–5
 - epoc32/release 124–5, 127–31, 132–5, 138–41, 142–4, 152–60
 - programming basics 118–19
 - SIS files 23–5, 39–56, 59–61, 149–60, 233–5, 365–411
 - threads 288
- executed-in-place code, concepts 70–7
- ExecuteLD 121, 387–405
- ExitReason 285–6, 289, 291–2
- ExitType 285–6, 291–2
- Expand 200–3
- explicit network connections, concepts 357
- EXPORT_C 115–18, 143–4, 147–9
- exports, libraries 115–18, 141–9
- EXPORTUNFROZEN 116–17, 141–4, 145–9
- extensions, kernel 77–9
- ExternalizeL 371
- F32 90–1
- fax 8
- features, smartphones 1–9
- file server
 - see also* RF... servers
 - concepts 68–70, 303–4
- file system 13–14, 64, 68–70, 90–1, 226–7, 303–4
 - platform security 226–7, 231–2
 - structure 90–1
- FILENULL 155
- FILERUN (FR) 156
- FILETEXT 154–5, 159
- Fill 192–3
- FillZ 192
- Find 188–90, 211
- ‘fire and forget’ protocols 324–5
- firewalls 219
- First 214
- fixed arrays
 - see also* arrays
 - concepts 205–6
- fixed processes, concepts 76–7
- flash memory *see* internal flash disk
- flat dynamic buffers, concepts 199–203, 206–8
- flushing costs, caches 76–7
- _FOFF 210
- folding method, descriptor
 - comparisons 187–90
- font and bitmap server
 - see also* servers
 - concepts 304–5
- fonts 139–40, 231, 304–5, 374–7
- foreign languages, installation support 157–60, 385–7
- FORM 397–401
- Format 194
- format, resource files 253–4
- formatting data, descriptors 194
- forms, Series 60 (S60) 397–401
- Forum Nokia 28–9
- freeware 1, 240–1
- freeze 128–31, 142–9
- freezing
 - concepts 116–17, 141–9
 - DEF files 145–9
 - definition 144–5
 - disabling methods 116–17, 141–4, 145–6, 149
 - enabling methods 145–6
 - importance 144–5
 - libraries 37–8, 115–16, 128–31, 141–9
 - new-function inserts 148–9
 - violated interfaces 149
- FTP network service 326–7
- Function 316–19
- function arguments, naming conventions 120
- function names, conventions 119–21
- games 16–22, 323–4
- GC *see* graphics context
- gcce 59–61, 128, 131–5, 137–8, 143, 147–60
- gdi.lib 129–31
- generations, mobile
 - communications 4–11
- generic build system, SDK 24–6
- GetByAddr 335–57
- GetByName 335–57
- Getch 162–5
- gethostbyname 332, 335
- GetMemoryInfo 283–4
- GetTemperatureL 346–57
- GetText 310–21, 406
- global memory chunks
 - see also* chunks
 - concepts 293–302
- Global Positioning Satellite (GPS) 10–11, 17–18, 225
- global variables, restrictions 118–19, 120
- GNUPoc 30–1

- GPRS network protocol 4, 5–6, 12, 16–22, 88–9, 323–4, 356–7
see also GSM. . .
- GPS *see* Global Positioning Satellite
- graphical user interface framework (GUI)
see also application framework; Series. . .; UIQ. . .
- active scheduler 254–3, 269, 342–3, 355–6
- anatomy 38–56, 365–7
- application classes 38–56, 365–411
- application programming 38–56, 359–411
- concepts 13–22, 27–30, 38–56, 64–6, 83–5, 118–19, 125–6, 137–8, 161–2, 254–3, 269, 287, 303–4, 342–3, 355–6, 359–411
- controls 83–5, 310–21, 377–8, 397–409
- dialogs 359–60, 363–5, 387–405
- DLLs 66, 118–19
- examples 38–56, 359–411
- icons and captions 366–7, 409–11
- overview 38–56, 83–5, 359–65
- resource files 44–56, 377–411
- servers 303–4
- Symbian OS 13–14, 27–30, 38–56, 64–6, 83–5, 118–19, 125–6, 161–2, 359–411
- types 13–22, 359–65
- view architecture 409
- Windows development tools 27–30
- graphics
 animation plug-ins 85–6
 direct screen access 85–6
 high performance graphics 85–6
- graphics context (GC) 52–4
- GSM network protocol (Global System for Mobile Communication) 4–6, 9, 12, 16–22, 80, 88–9, 323–57
see also EDGE. . .; GPRS. . .
- GUI *see* graphical user interface framework
- handle classes 97–100, 279–86, 287–92, 297–8, 300–1, 318–20
see also RMutex; RProcess; RSemaphore; RThread
- HandleCommandL 50–4, 265–8, 354–6, 372–7, 392–6, 401
- HandleResourceChangeL 373–7
- handles, concepts 96–100
- Handspring Treo 650 21
see also Palm OS
- handwriting recognition 15–17
- Harrison, Richard 409
- HasCapability 318
- HBufC 168–215
see also heap descriptors
- concepts 168–215
- memory layout 181–6
- TBufC 181
- header files
 concepts 39–56, 115–18, 126–31, 141–4, 151–60, 365–411
- controls 405–6
- heap chunk, concepts 72–3
- heap classes, concepts 96–100
- heap descriptors
see also descriptors; HBufC. . .; RBuf. . .
- concepts 168–70, 180–6, 315–16
- definition 168, 180–1
- modifications 181–3
- usage of other descriptors 183
- Heath, Craig 219, 227
- help files 231
- hierarchy, descriptor classes 169–71
- high performance graphics 85–6
see also graphics
- home area, virtual memory map 73–7
- hot keys, emulator 138–40
- How Smartphones Work (Northam) 2
- HRH files 39–56, 365–411
- HSCSD network protocol 6, 19
see also CSD. . .
- HSDPA 7
- HTML 8–9, 16–22, 125–6, 323–4, 346–8
- HTTP 64–5, 326–7, 330–4, 339–41, 344–57
- <http://www.wunderground.com>** 346–57
- HVGA 364
- i (member variables) prefixes 119–20
- IAPs *see* Internet Access Points
- icons 366–7, 409–11
- IDEs *see* Integrated Development Environments
- image masks 411
- IMAP accounts 8, 327
see also emails
- IMEI serial numbers 224, 244
- implicit network connections, concepts 357
- import libraries
see also static libraries
- concepts 115–18, 133–5, 142–9
- native builds 133–5
- IMPORT_C 115–18, 143–4, 147–9
- include 385–7
- InfoPrint 119
- InfoWinL 402
- infrared connectivity (IR) 9–10, 16–22, 23–4, 64–5, 87–9, 150, 221–2, 334–5
- inheritance features
 C++ 93–4, 98–100
- interface classes 98–100
- Insert 200–3, 210–13

- Install 254-7, 314-21
- installation 23-5, 39-56, 59-1, 149-60, 164, 236-7, 254-7, 313-21, 365-411
 - see also* SIS files
 - active scheduler 254-7, 313-21
 - advanced pkg options 154-7
 - concepts 38-56, 149-60, 236-7, 365-411
 - directories 150-1
 - file-specification methods 153-4
 - language support 157-60, 385-7
 - PKG files 39-56, 151-60, 233-5, 240-1, 365-411
 - requisite lines 156-7
 - running executables 156
 - runtime-generated file removal 155
 - text notices 154-5
- install.exe 151
- instant messaging 8, 22, 323-4
- instantiated classes, concepts 94-6, 115, 119-20, 169, 287
- int 94-5
- Integrated Development Environments (IDEs) 23-61, 137-41, 365-7
 - see also* Borland...;
 - Metrowerks...; Microsoft...
- concepts 23-7, 56, 137-41, 365-7
- providers 27-30, 36-8, 56
- quick-start examples 23-7, 36-56, 365-7
- selection criteria 28
- inter-thread communications, concepts 77-9, 281-2, 292-7, 318-20
- interface classes
 - concepts 96, 98-100, 119-20
 - example 98-100
 - inheritance features 98-100
- interface freezing *see* freezing
- internal flash disk
 - see also* c: drive; memory...
 - concepts 70-1, 138-41, 149-60
- InternalizeL 371
- Internet 1, 3-4, 7-8, 88-9, 323-57
 - see also* browsing: TCP/IP...
- Internet Access Points (IAPs) 88-9, 356-7
- Internet Protocol Suite *see* TCP/IP
- IP (Internet Protocol) 324-57
 - see also* TCP...
 - addresses 325-6, 331-4, 338-9, 349-53
 - concepts 325-57
 - domain names 332-4
 - layering diagram 325
 - port addresses 325-57
- IR *see* infrared connectivity
- ISPs 5-6
- Java 21, 25-6, 99, 124-5
- Java Runtime Environment 25-6
- jog dial 360
- K (constants) prefixes, naming conventions 120-1
- KDynamicLibraryUid 135-6
- kernel 63-91, 284, 287-92
 - architectural overview 77-9
 - concepts 63-4, 72-9, 284, 287-92
 - definition 63-4, 77
 - EKA2 concepts 77-81
 - executive 78-9
 - extensions 77-9
 - MMU 71-2, 74-5
 - platform security 90-1
 - process priorities 284
 - roles 63-4, 72, 74-5, 77-9
 - server 77-9
 - threads 287-92
 - user library 79
- KErrAlreadyExists 310
- KErrCancel 261
- KErrEof 345
- KErrNoMemory 101, 104-14
- KErrNone 101, 103-4, 147-9, 213, 248-9, 253-4, 260-1, 265-8, 273-5, 281-2, 289, 293-6, 297-8, 302, 309-10, 335-57
- KErrNotFound 101, 188-90, 211, 212, 281-2, 309-10
- KErrNotSupported 309-10, 315-21
- KErrPermissionDenied 220, 227-8
- KExecutableImageUid 135-6
- keys 14-20, 84-5, 138-40, 303-4, 360-411
 - emulator 138-40
 - platforms 360-5
 - virtual keyboards 15-17, 138-40
- Kill 285-6, 291-2
- KQikPenStyle... 382
- KQikSoftkeyStyle... 382
- KRequestPending 248-9, 256, 260-1, 270
- KUIdApp 135-6
- _L 167-8, 171-3, 181-2
- L suffixes, concepts 105-14, 119-20, 121
- LAF *see* Look and Feel
- Landmark messaging 327
- LANG keyword, resource files 386-7
- LANGUAGE keyword, resource files 385-6
- language support
 - installation 157-60, 385-7
 - MMP files 385-7
 - resource files 385-7
- LANs *see* Local Area Networks
- laptops 3
- Last 214
- LBS *see* Location Based Services
- LBUF 391-6
- LC functions, concepts 110-13, 121
- Leave 101-14, 212-13, 279-80

- leave/trap mechanism
 - active scheduler 258–9, 262–8, 269–70, 314–21
 - concepts 101–14, 121, 162–5, 180–1, 212–13, 253–4, 258–60, 262–8, 314–21
 - constructors 112–14
 - object creation 110–13
- Left 190
- Length 170–1, 206
- Lenovo 14, 26
- LG 14–15, 364
- LIB files 128–31, 134–5, 138–41, 142–3
- libraries 12–13, 24–6, 63–4, 65–8, 78–9, 83–5, 114–18, 128–31, 141–9, 275, 278, 355–6, 403
 - see also* dynamic link...; middleware
- application protocols 90–1
- base libraries 63–4
- classes 115–18
- concepts 24–6, 63–5, 78–9, 83–5, 114–18, 141–9
- CONE 84–5
- controls 406
- freezing 37–8, 115–16, 128–31, 141–9
- OS requirements 12
- programming basics 114–18
- SDK 24–6, 141–9
- types 65–8, 79, 83–5, 114–18
- UIKON 83–5
- user library 79
- LIBRARY 128–31, 142–4, 275, 278, 355–6, 403
- linked lists
 - see also* TDb1Que
 - concepts 214
- Linux 21–2, 31, 64–5, 231, 247
- list boxes 359–65, 406–7
- LISTBOX 406–7
- _LIT 167–8, 171–80, 187–99, 201–3, 212, 272–5, 278–82, 293–6, 297–8, 308–12
- LLINK 379–80, 391–6, 406–7
- Load 144
- loading methods, polymorphic
 - DLLs 144
- Local Area Networks (LANs) 89, 323
- local device communication
 - features
 - see also* connectivity features
 - concepts 9–10, 64–5, 86–9
- local memory chunks
 - see also* chunks
 - concepts 296–7
- local semaphores
 - see also* semaphores
 - concepts 299–300
- LOCALISABLE_APP_INFO 410
- localization, resource files
 - 366–7, 385–7, 409–11
- LocalServices 221–3
- Location 221–3, 228
- Location Based Services (LBS)
 - 10–11
- log files 25–6, 141
- log servers 90
- Logon 285–6, 292
 - see also* asynchronous functions
- long 94–5, 378–80
- Look and Feel (LAF), concepts
 - 83–5
- Lookup 144
- LowerCase 165–6, 194–5
- LTEXT resource files 378–80
- M (mixin) classes, concepts 96, 98–100, 120
- MACRO 130–1
- macros
 - assert macros 114–15
 - naming conventions 121
 - string literals 166–8, 171–2
- Maemo 22
- make 125–6
- makefiles, build system overview
 - 125–6
- makekeys 234–5
- MakeName 166–8
- makesis 59–61, 150–60
- malicious software 217–19
- manufacturers 12–15
 - see also* individual manufacturers
- Match 189–90
- MBM files 410–11
- MegabytesOfFreeMemory
 - 138–40
- member variables, naming
 - conventions 119–20
- memory 11–12, 16–22, 63–4, 65, 70–7, 138–41, 149–60, 174–86, 293–7
 - see also* Random Access...; Read Only...
 - addresses 71–2, 293–7
 - blocks 71–2, 269
 - capacity specifications 16–22, 70–1, 138–9
 - chunks 72–7, 97–8, 293–7
 - committed memory 294–7
 - concepts 70–7, 138–41, 174–86, 293–7
 - descriptors 174–86
 - emulator configuration
 - 138–41
 - frugal requirements 11–12
 - organization 71–7
 - orphaned memory 105–6
 - out-of-memory situations 12, 239
 - overrun problems 165–6, 176–7
 - physical/virtual memory
 - addresses 71–7, 294–7
 - processes 72–7, 292–7
 - shared memory 65–8, 114–18, 292–7
 - types 70–3
- memory cards *see* removable memory cards
- memory leaks 105–6, 163–4, 184
- Memory Management Unit (MMU)
 - concepts 71–7, 286
 - page tables 73–7, 294–7
 - protection role 71–2, 76
- memory maps, concepts 71–7, 286, 294–7
- menu/softkey items, resource files
 - 44–56, 380–405

- MENU_BAR 45–56, 383
- MENU_ITEM 45–56, 383, 408
- menu_pane 45–56
- menus 44–56, 359–411
- MENU_TITLE 45–56, 383
- messages, client–server model 303–21
- messaging, smartphones 7–8, 16–22, 64–5, 323–4, 356
- Metrowerks 28, 36–7
 - see also* Integrated Development Environments
- Microsoft 13–14, 20–1, 28–30, 36–7, 125–7
 - see also* Integrated Development Environments; Windows
- Mobile Smartphone OS 13–14, 20–1, 233–4
- Smartphone OS 13–14, 20–1
- Visual Studio 28–30
- Mid 190
- middleware
 - see also* libraries
 - concepts 12, 64–5
- Mixin 98–100
- MMC storage media 16–22, 71
 - see also* removable memory cards; storage media
- MMF 90–1
- MMP files 38, 39–56, 116–18, 119, 126–31, 134–5, 141–9, 160, 163–4, 220, 227–31, 236–7, 275, 278–86, 355–6, 365–411
 - see also* project...
 - concepts 39–56, 126–31, 141–9, 160, 163–4, 220, 227–31, 236–7, 275, 278–86, 355–6, 365–411
 - DLLs 141–9
 - language definitions 385–7
 - processes 278–86, 365–411
- MMS *see* Multimedia Messaging Service
- MMU *see* Memory Management Unit
- mobile phones
 - see also* smartphones
 - concepts 1–22, 359–60
 - generations 4–11
 - hardware limitations 359–60
 - historical background 2–3, 11
 - network protocols 3–11, 12, 16–22, 88–9, 323–57
 - PDAs 3–4, 7, 9–10, 20–2
 - platform security 90–1
 - specifications 16–22
- mobile TV 11
- modem features 10
- modifying methods, descriptors 191–7
- MotoDev 30
- Motorola
 - see also* UIQ
 - A920/A925 6, 15
 - A1000 15
 - A1200 22
 - MOTORIZR Z8 15, 27, 360
 - Q 20
- multi-homing features 89
- Multimedia Messaging Service (MMS) 8, 12, 16–22, 64–5, 89, 239
- multimedia support, Symbian OS 13–14, 227, 323–4
- MultiMediaADD 226–7, 229, 243
- multipage dialogs, *see also* dialogs
- multiple DLLs, concepts 117–18
- multiple inheritance features, C++ 93–4, 98–100
- multiple processes 140, 277–86, 321
 - see also* processes
- multiple threads 64–5, 82, 140, 277–8, 286–92
- multitasking aspects, Symbian OS 13–14, 64–5, 140, 250–1
- music players 3
- mutexes
 - see also* synchronization
 - concepts 77–9, 97–8, 300–1
- mvccsym2 126
- NAME keyword, resource files 377–9
- naming conventions 68, 95–6, 119–21
- nanokernel, concepts 78, 79, 81
- native build targets 132–5
- NEC 22
- network connections, concepts 356–7
- network interface manager (NIFMAN), communications architecture 86–9
- network programming
 - see also* sockets; TCP...
 - active objects 335–57
 - concepts 323–57
- network protocols
 - concepts 3–11, 12, 16–22, 86–9, 227, 323–4, 356–7
 - specifications 16–22
- network services, well-known server-side port addresses 326–7
- NetworkControl 226–9, 230–1, 243
- NetworkServices 221–3, 324, 355–6
- New 96, 111–13, 117–18, 120, 180–1
- NewApplication 49–54, 367–77
- NewL 112–13, 120, 148–9, 200–3, 251–4, 263–8, 306–21, 346–57, 374–7
- NewLC 30, 112–13, 120, 181, 266–8, 272–5, 313–21, 374–7
- NewMyPoly 117–18
- NewSessionL 306–21
- Next 282–4
- NIF files 88–9
- NIFMAN *see* network interface manager
- nmake 125–6
- NOCOMPRESS (NC) 152–60
- Nokia 13–15, 26–7, 123–60
 - see also* S60...
 - 3230 26
 - 3600/3650 14, 26–7
 - 3620/3660 26–7
 - 5500 364
 - 6121 classic 14
 - 6260 26

- Nokia (*continued*)
 - 6600 6, 26
 - 6630 26
 - 6670 26
 - 6680/6681/6682 14, 26
 - 7610 14
 - 9210 15
 - 9290 6, 15
 - 9300i 15, 18–19
 - 9500 Communicator 15, 27
 - E61i 14, 26, 222–3, 364
 - E65 26
 - E90 Communicator 14–15, 19, 364
 - N-Gage 26–7
 - N70 26–7
 - N76 14–15
 - N77 11, 26
 - N90 26–7
 - N91 71
 - N92 11
 - N93 14–15
 - N95 14–15, 17–18, 24, 26, 70, 364
 - N800 22
 - SDK 26–7, 38–56, 123–60
 - Symbian ownership 13–14
- non-modifying methods, descriptors 187–98
- non-preemptive multitasking
 - model, active objects 250–1, 269–71
- nostrictdef 146
- Nucleus 80
- NULL 110–13, 156–7, 172, 191–2, 196–7, 289, 375–7
- object types, cleanup stack 107–14
- object-oriented operating systems 12–13, 94–5
- OEM hardware 13–14, 76, 78–9, 83–5
- OkToExitL 401
- OnStarting 257–8
- OnStopping 257–8
- Open 97–100, 113–14, 281–3, 286, 289–90, 299–300, 337–57
- ‘open’ aspects, Symbian OS phones 1
- OpenFileL 371–7
- OpenGlobal 299–300
- operating systems 1, 11–15, 30–1
 - see also* Symbian OS
 - BlackBerry 21
 - competitors 13–14, 20–2
 - historical background 11–15
 - Linux 21–2, 31, 64–5
 - Microsoft Mobile Smartphone OS 13–14, 20–1, 233–4
 - Palm OS 21, 64–5
 - requirements 11–12
 - resource-limitations 11–12
 - robustness needs 11–12
- option buttons 408
- Order 211
- ordinal function references 143–9
- orphaned memory, dangers 105–6
- OSE 80
- OSs *see* operating systems
- out-of-memory situations 12, 94–5, 101, 239
- OutputWebPage 330–57
- overload features, C++ 93–4, 108–9, 111–13, 279–80
- overrun problems, memory 165–6, 176–7
- owning manufacturers, Symbian OS 13–14
- package definition files 39–56, 59–61, 150–60, 365–411
 - see also* PKG files
- packets
 - see also* GPRS...; HSDPA...; UMTS...
 - concepts 5–7, 323–57
 - TCP/IP 323–57
- page tables, MMU 73–7, 294–7
- Palm OS 21, 64–5
- Panasonic 14, 22
 - Symbian ownership 13–14
 - X700 26–7
 - X800 26–7
- Panic 113–14, 163, 284–5, 313–21
- panics
 - concepts 113–14, 163, 168, 284–6, 313–21
 - examples 113–14, 163, 313–21
 - SDK list 113–14
- passwords 60–1, 403
- PCs 3, 7–11, 16–22, 24–6, 31–8, 56–8, 118–19, 124–5, 137–41
 - see also* Windows
 - emulator 24–6, 31–8, 56–8, 118–19, 124–5, 137–41, 286
 - installation 149–50
- PDA's 1, 3–4, 7, 9–10, 20–2
- PDF files 125–6
- PE files 118–19
- Pearl design 14
 - see also* Series 60
- pens 15–17, 382
 - see also* touch screens
- performance issues
 - context switches 76–7, 82
 - switched processes 76–7, 82
- Perl scripts 124–5
- permissions, platform security 90–1
- personality, concepts 81
- PETTRAN 118–19
- phone manufacturer capabilities 220, 225–7, 243–4
- physical memory addresses, concepts 71–7, 294–7
- pictures 3, 8, 11, 16–22
- ping 350
- P.I.P.S. 330
- pixels 139–40, 409–11
- PKG files 39–56, 59–61, 150–60, 233–5, 240–1, 365–411
 - see also* package definition files
 - advanced pkg options 154–7

- concepts 39–56, 59–61, 150–60, 233–5, 240–1, 365–411
- installation 39–56, 151–60, 233–5, 240–1, 365–411
- language support 157–60, 386–7
- PKI *see* Public Key Infrastructure
- platform security
 - see also* signed applications; Symbian Signed
 - capabilities 90–1, 129, 131, 160, 219–32
 - concepts 2, 71–2, 76, 90–1, 217–46
 - data-caging concepts 91, 226, 231–2
 - developer certificates 236, 244–6
 - emulator 228–9
 - exclusions 218–19
 - goals 217–18
 - malicious software 217–19
 - MMU 71–2, 76
 - permissions 90–1
 - SID 129, 131, 232, 237–8
 - trust principle 218
- platforms, Symbian OS 13–22, 153–60, 359–65
- PlatSecDiagnostics 229
- PlatSecDisableCaps 228–9
- PlatSecEnforcement 228–9
- plug-in DLLs
 - see also* dynamic link libraries
 - concepts 66–9, 85–6, 115–18, 135–6
- Pocket PC OS 20–1
- pointer descriptors
 - see also* descriptors; `TPtr...`
 - concepts 168–70, 177–80, 318–20
 - definition 168, 177
- polymorphic DLLs
 - see also* dynamic link libraries; plug-in...
 - concepts 66–8, 86–90, 115–18, 135–6
 - loading methods 144
 - virtual declarations 66–7
- Pop 106–14, 202–3, 262–8, 315–21, 374–7
- pop-up fields 364–5, 398–401
- POP3 accounts 8, 326–7
 - see also* emails
- PopAndDestroy 106–14, 180, 184–6, 202–3, 274, 314–16
- port addresses
 - concepts 325–57
 - IP (Internet Protocol) 325–57
 - well-known server-side addresses 326–7
- POSIX 64, 330
- PowerMgmt 224–5
- PPP module 88–9
- pre-emptive multithreading,
 - concepts 65, 247, 250–1
- pre-version 9 SDK build targets 135
- prefixes, naming conventions 95–6, 119–20
- PreLayoutDynInitL 394–401
- printf 161–5, 188, 194
- priorities
 - active objects 260
 - processes 284
 - threads 290–1
- Priority 284–5
- private data, data-caging concepts 91, 226, 231–2
- private directory 151–60, 227, 231–2
- private keys 233–5, 246
- PRJ_PLATFORMS 131–2
- processes
 - see also* applications
 - arguments 279–81
 - chunks 293–7
 - code chunks 73–4
 - concepts 64–5, 72–9, 277–86, 292–7
 - critical sections 301
 - definition 65, 277–8
 - E32Main 119, 163–4, 278–86, 367–77
 - end-signaling method 285–6
 - examples 278
 - fixed processes 76–7
 - inter-process communications 77–9, 281–2, 292–7
 - launching method 278–80, 289
 - memory 72–7, 292–7
 - MMP files 278–86, 365–411
 - multiple processes 140, 277–86, 321
 - names 281–3
 - performance issues 76–7
 - platform security 2, 71–2, 76, 90–1
 - priorities 284
 - processes-running queries 282–3
 - protection 71–2, 76
 - Rendezvous 301–2, 310
 - RProcess 97–100, 119, 279–86, 292–7, 301–2, 309–10
 - running 279–80, 289
 - shared memory 292–7
 - status checks 285–6
 - switched processes 73–7, 82, 286
 - terminations 284–5
 - virtual memory map 73–7, 286
 - wildcard searches 282
- programming basics 93–122, 247–75, 323–57, 359–411
- asynchronous functions 247–75
- basic data types 94–5
- C++ in Symbian OS 93–4, 102–3
- descriptors 165–215, 292–3, 307
- DLLs 114–18
- exception handling 101–14
- executables 118–19
- GUI applications 38–56, 359–411
- libraries 114–18
- naming conventions 68, 95–6, 119–21
- Symbian OS classes 95–100
- TCP/IP applications 323–57
- progress bars 407
- PROGRESSINFO 407

- project build files, concepts
 - 54–6, 115–18, 126–60
- project definitions
 - 54–6,
 - 115–18, 119, 126–31,
 - 141–9, 163–4, 278–86,
 - 385–7
 - see also* MMP files
 - concepts 126–31, 141–9,
 - 278–86
 - definition 126–7
 - DLLs 141–9
- project management tools,
 - Windows development tools
 24–6
- protection
 - MMU role 71–2, 76
 - processes 71–2, 76
 - semaphores 298–9
- protocol 330–1
- protocol modules
 - see also* Bluetooth...;
 - infrared...; TCP/IP
 - communications architecture
 86–9, 323–57
- protocols, interface classes
 - 98–100
- ProtServ 224–5
- proxy servers, WAP 8
- PRT files 86–9
- Psion 12–13
- Ptr 196–7, 200–3
- PtrZ 196–7
- Public Key Infrastructure (PKI)
 - 233
- public keys 233–5, 246
- Publisher Ids 236, 241–3
- publisher option, Symbian Signed
 - 240–3
- push email 21
- PushL 106–14, 119–20, 121,
- 207–8, 262–8, 273–4,
- 314–21, 374–7

- qikapplication.h 41–4
- qikappui.h 41–4
- QIK_COMMAND 46–56, 382,
- 392–405
- QIK_COMMAND_LIST 46–56,
- 382, 392–6

- QikCommand.rh 45–56
- QIK_CONTAINER... 392–6
- QIK_CONTROL 390–6
- QIK_CONTROL_COLLECTION
 - 389–96
- qikctl.lib 56
- QIK_DIALOG 388–405
- qikdlh.lib 403
- qikdocument.h 41–4
- qikon.rh 45–56
- QIK_SLOT_CONTENT 393–6
- QIK_SYSTEM_BUILDING_BLOCK
 - 393–6
- QIK_VIEW 46–56, 380–3
- QIK_VIEW_CONFIGURATIONS
 - 45–56, 375–7, 380–3
- QIK_VIEW_PAGES 46–56,
- 380–3, 394–6
- Quartz 15
 - see also* UIQ
- QueryWinL 402
- quick start guide, Symbian OS
 - development environment
 23–61
- QVGA 360–4
- QWERTY keyboards 18–19, 362,
- 364

- R (resource) classes, concepts
 - 96–100, 109, 120, 279
- radio 3–11, 64–5
- radio buttons 408
- Random Access Memory (RAM)
 - see also* memory...
 - capacity specifications 70–1
 - concepts 70–1, 74–5, 294–7
- RArray 209–13
- R_AVKON_SOFTKEYS_OK
 - _CANCEL 404
- R_AVKON_SOFTKEYS_OPTIONS
 - _EXIT 44–56, 383
- R_AVKON_SOFTKEYS_YES_NO
 - 403
- RBuF 168–70, 180, 184–6
 - see also* heap descriptors
- RChunk 72, 293–7
 - see also* chunks
- RConnection 221–3, 227, 357

- RCriticalSection 301
- RDebug 141, 273–5
- Read 97–100, 200–3, 292–7
- Read Only Memory (ROM) 11,
- 70–7, 124–5, 138–41, 217
- see also* memory; z: drive
- capacity specifications 70–1
- concepts 70–7, 124–5, 138–9
- executed-in-place code 70–7
- ReadDeviceData 224–5,
- 230–1
- ReadL 318–20
- ReadUserData 221–3, 230–1
- real-time operating system (RTOS)
 - 80–1
- ReAlloc 181–3
- reallyclean 128
- RecvFrom 340–1, 344–5
- RecvOneOrMore 340–1,
- 345–57
- reference platforms, Symbian OS
 - 13–15, 359–65
- registration files 48–9, 366–7,
- 384–5
- relocated data, concepts 73–7
- removable memory cards
 - see also* memory...
 - concepts 71
 - MMC storage media 16–22, 71
- Remove 212, 215
- Rendezvous 301–2, 310
- Replace 257
- request semaphores, asynchronous
 - functions 248–75, 300–1
- requisite lines, installation 156–7
- Research in Motion (RIM) 21
 - see also* BlackBerry
- Reset 200–3, 206, 212, 310–21
- Resize 200–3
- resource classes
 - concepts 96–100, 120
 - example 97–8
- resource directory 151, 231–2
- resource files
 - see also* RSS files
 - concepts 44–56, 128–31,
 - 141–4, 365–7, 377–411
 - definition 46, 377–8
 - format 253–4

- language support 385–7
- localization 366–7, 385–7, 409–11
- registration files 48–9, 366–7, 384–5
- SimpleEx example 380–3
- string-reading tips 386–7
- RESOURCE keyword, resource files 378–80, 405–6
- resource-limitations, smartphones 11–12
- RestoreL 370–7
- Resume 279–81, 288–91
- return codes, errors 101
- RFile 69–70, 97–100, 109–10, 113–14, 115, 303–4
 - see also* file server
- RFs 226–7, 232, 303–4
 - see also* file server
- RHandleBase 299–300, 307–10
- RHandleBase::Duplicate 299–300
- RHostResolver 335–57
 - see also* sockets
- Right 190
- RIM *see* Research in Motion
- RLibrary 117–18, 144
 - see also* dynamic link libraries
- RLS files 385–7, 388–9, 411
- RMessage 305–21
- RMutex 301
- robustness needs, smartphones 11–12
- ROM *see* Read Only Memory
- root certificates, Symbian Signed 236–7
- RPointerArray 209–13
- RProcess 97–100, 119, 224, 279–86, 292–7, 301–2, 309–10
 - see also* processes
- RProcess::Create 279–80
- RProcess::Logon 285–6
- RSC files 366–7, 386–7
- RSemaphore 97–100, 297–302
- RSessionBase 304–21, 335
 - see also* client–server... concepts 304–21, 335
 - methods 304–6, 320–1
 - r_SimpleEx_dialog 398–401
 - r_SimpleEx_form 398–401
 - r_SimpleEx_menu 45–56
 - RSocket 97–100, 115, 304–5, 335–57
 - see also* sockets
 - RSocketServ 335–57
 - see also* sockets
 - RSS files 39–56, 164, 365–7, 377–411
 - see also* resource files
 - RSS_SIGNATURE resource 45–56, 380–3
 - RTextBuff 307–10
 - RThread 97–100, 247, 254–3, 287–97, 301–2
 - see also* threads
 - RThread::Create 288–91
 - RThread::Kill 291–2
 - RThread::Logon 292
 - RThread::Open 289–90
 - RTimer 260–1, 262–8, 271–2
 - RTOS *see* real-time operating system
 - run area, virtual memory map 73–7, 286
 - RUNBOTH (RB) 156
 - RunConsoleL 163–4
 - RunDlgLD 395–6, 402–5
 - RunError 251–2, 253–4, 258–9
 - RUNINSTALL (RI) 156
 - RunL 82, 249–75, 292, 304–5, 311–12, 342–57
 - see also* active scheduler
 - concepts 249–75, 304–5, 311–12, 342–57
 - implementation 253, 262–8, 346–57
- RUNREMOVE (RR) 156
- RUNWAITEND (RW) 156
- RVCT compiler 132–5
- S60 14–20, 29–30, 31–56, 83–5, 113–14, 138–40, 151–60, 353–6, 359–416
 - see also* CAkn...
- characteristics 363–5
- classes 40–56, 367–411
- control structures 405–9
- data input 364–5, 387–8, 397–405
- dialogs 364–5, 387–405
- editions 14–15
- emulator 24–5, 32–8, 56–8, 138–40
- forms 397–401
- GUI architecture 14–20, 38–56, 83–5, 359–411
- header file 39–44, 151–60
- icons and captions 409–11
- package file 59–61, 151–60
- panics 113–14
- project build file 54–6, 128–31
- quick-start development
 - examples 24–7, 35–56
- resource file 44–56, 128–31, 387–405
- screens 363–5
- SDK 26–7, 35–56, 123–60
- stock dialogs 401–5
- view class 52–4, 374–7, 409
- S60Doc 126
- S60Ex 125
- Samsung 13–15, 20, 22, 364
 - Blackjack 20
 - SGH-i400 15, 364
 - SGH-i520 364
 - Symbian ownership 13–14
- sandbox *see* run area
- Save 397–401
- SaveFormDataL 397–401
- Scalable Vector Graphics Tiny (SVG-T) 366, 410–11
- screens 13–22, 83–5, 137–8, 303–4, 359–411
 - see also* graphical user interface framework
- concepts 13–22, 83–5, 137–8, 303–4, 359–411
- platforms 13–22, 359–65
- Series 60 (S60) 363–5
- specifications 16–22
- UIQ 360–3
- SD memory cards 17, 71

- SDKs *see* software development kits
- Secure Identifier (SID), platform security 129, 131, 232, 237–8
- SECUREID 129, 131, 151, 232, 238
- security issues *see* platform security
- segmented dynamic buffers, concepts 199–203, 206–8
- self-certification procedures, Symbian Signed 243
- self-signed applications 234–7, 324
 - see also* signed...
- semaphores
 - see also* synchronization
 - asynchronous functions 248–75, 300–1
 - concepts 77–9, 82, 97–100, 248–9, 297–302
 - creation 299–300
 - opening 299–300
 - protection uses 298–9
 - Symbian OS 300–1
 - uses 248–9, 298–9, 300–1
- Send 306–21, 339–57
- Sendo X 26–7
- SendReceive 306–21
- SendTo 340–1
- serial cable connection 9
- serial communications server, communications architecture 86–9
- Series 60 *see* S60 (S60) 14–20, 29–30, 32–56, 83–5, 113–14, 138–40, 151–60, 353–6, 359–411
- Series 80 (Nokia) 15, 18–19, 22, 26–7, 364
- servers
 - see also* file...; font and bitmap...; socket...; window...
 - active objects 303–4, 311–12, 313–21
 - asynchronous functions 303–4, 311–12, 313–21, 335–57
 - client–server model 68–70, 83–5, 292–3, 303–21, 326–7
 - concepts 68–70, 83–5, 292–3, 303–21
 - definition 303–4
 - ETEL server 86–9, 90
 - examples 306–21
 - execution flow 68–70, 303–4
 - GUI 303–4
 - implementation 312–21
 - kernel server 77–9
 - message-processing example 316–18
 - pointers 316–19
 - serial communications server 86–9
 - service-invoking methods 310–12, 349–53
 - shutdown issues 320–1, 334
 - sockets 86–9, 97–100, 115, 304–5, 323–4, 326–57
 - starting 307–10, 313–16
 - TCP/IP 326–57
 - TextBuffServ example 306–21
 - transient servers 320–1
 - types 68–9, 77–9, 83–5, 303–4
 - window server 83–5, 303–4
- servers-side code, sockets 328–57
- ServiceError 318
- ServiceL 306–21
- sessions, client–server model 304–21
- SetActive 253–7, 263–8, 270–5, 311–12, 349–57
- SetAddress 338–57
- SetArrayL 408
- SetItemDimmed 401
- SetLength 197, 215
- SetMax 197
- SetPenColor 396
- SetPort 338–57
- SetPriority 284, 290–1
- SetRect 374–7
- SetReserveL 200–3
- SetState 408
- SetTextL 406
- SetType 282–3
- shared code, concepts 65–8
- shared memory, concepts 65–8, 114–18, 292–7
- Sharp 22
- Short Messaging Service (SMS) 7–8, 12, 16–22, 64–5, 220–1, 239, 356
- SHUTDOWNAPPS (SH) 152–60
- SID *see* Secure Identifier
- Siemens 14–15
- SX1 26–7
 - Symbian ownership 13–14
- Signal 298–302
 - see also* synchronization
- signed applications 30, 60–1, 91, 160, 218–46, 324
 - see also* platform security
 - concepts 232–46
 - developer certificates 236, 244–6
 - DLLs 233–4, 239
 - freeware option 240–1
 - installation issues 236–7
 - publisher option 240–3
 - SIS files 233–5, 239–46
 - trusted/untrusted certificates 235–6
 - types of signing 234–7
 - UIDs/SIDs 237–8
- signsis 160, 234–5, 242, 246
- SIM *see* Subscriber Identification Module
- simple/compound controls, contrasts 393–4
- SimpleEx 38–56, 58, 151–60, 262–8, 353–6, 365–411
 - active objects example 262–8, 353–6
 - class-hierarchy diagrams 42–4
 - Draw 374–7
 - overview 39–40
- SimpleEx_app.cpp 49–50, 56
- SimpleEx.cpp 49–50
- SimpleEx_Doc.cpp 50–2
- SimpleEx.hrh 45–56, 382
- SimpleEx.mmp 54–6, 128–31, 385–7

- SimpleEx.pkg 59–61
- SimpleEx_reg.rss 48–56
- SimpleEx.rss 44–56, 380–3
- SimpleEx.sis 60–1
- SimpleEx_UI.cpp 50–2, 265–8
- SimpleEx_View.cpp 52–4, 267–8, 374–7
- single-shot grant notifications, capabilities 222–3
- SIS files 23–5, 39–56, 59–61, 149–60, 233–5, 239–46, 365–411
 - see *also* installation
 - concepts 23–5, 39–56, 59–61, 149–60, 239–41, 365–411
 - creation 150–1
 - embedded sis files 156
 - language support 157–60, 386–7
 - signed applications 233–5, 239–46
- SISAPP 153–60
- SISX files 59–61
- Size 171, 200–3
- smartphones
 - see *also* mobile phones
 - benefits 2–3
 - browsing 3–4, 8–9, 16–22
 - communication methods 3–11
 - concepts 1–22, 359–60
 - connectivity features 1, 9–10, 16–22, 64–5, 86–90
 - features 1–9
 - hardware limitations 359–60
 - historical background 2–3, 11
 - LBS 10–11
 - manufacturers 12–15
 - messaging 7–8, 16–22, 64–5, 323–4, 356
 - network protocols 3–11, 12, 16–22, 88–9, 323–57
 - operating systems 1, 11–15
 - PDAs 3–4, 7, 9–10, 20–2
 - resource-limitations 11–12
 - robustness needs 11–12
- SMS see Short Messaging Service
- socket 330–1, 337
- sockets
 - see *also* RSocket servers; TCP/IP
 - active objects 335–57
 - asynchronous functions 335–57
 - BSD socket 86–9, 323–4, 328–34
 - C++ 323–4, 329–57
 - classes 97–100, 115, 304–5, 334–57
 - client-side code 328–34
 - communications architecture 86–9
 - concepts 86–9, 97–100, 115, 304–5, 323–4, 326–57
 - connection 332–4, 335–57
 - creation 328–34, 337–57
 - destination addresses 338–9
 - examples 329–57
 - network programming 326–57
 - receiving data 333–4, 340–1, 344–5
 - remote web servers 339–40
 - sending data 333–4, 339–40, 344–57
 - servers-side code 328–34
 - shutdown issues 334
 - Symbian OS API 334–57
 - TCP/IP applications 323–57
 - weather-information example 345–57
- softkey items 360–3
 - resource files 44–56, 380–405
- software
 - see *also* applications; development...
 - C++ 27–30, 71–2, 84–5, 93–122, 166–8, 323–4, 329–57
 - developer prospects 1, 12
 - titles available 1, 12
- software development kits (SDKs) 2, 13–15, 23–61, 113–14, 123–60, 164, 409–10
 - see *also* development tools
 - build flow 39–56, 126–31
 - components 24–5, 123–60
 - concepts 2, 13–15, 23–7, 123–60, 164
 - directory structure 123–6
 - documentation directories 125–6
 - examples 26–7, 34–56, 123–60
 - getting 25–6
 - problems 37–8
- Sony Ericsson
 - Developer World 30
 - M600i 15, 27
 - P1i 15, 27
 - P800 15, 27
 - P900 15, 27
 - P990i 15, 16–17, 27, 70, 360
 - platform security 225–6
 - Symbian ownership 13–14
 - W950i 15, 27
 - W960i 15, 360
- Sort 211
- SOURCE 116–18, 119, 128–31, 141–4, 275, 278, 411
- source files, concepts 49–54, 128–31, 141–4, 365–411
- SOURCEPATH 116–18, 119, 128–31, 141–4, 275, 278, 411
- special handshakes, TCP 326
- specifications, mobile phones 16–22
- sprintf 194
- src directory 49–50, 54
- stack and heap chunk, concepts 72–3
- Standard Template Library (STL) 94–5
- Start 250–1, 254–7, 269, 273–4, 314–21
- START RESOURCE 129, 130, 411
- start-up code, ROM 70–1
- StartBackground 273–5
- StartL 306–21
- StartRunning 272–5
- StartServer 309–10, 313–16
- StartThreadL 287–8
- State 408

- state machines, active objects 342–56
- static data chunks, concepts 73–7
- static interface DLLs
 - see also* dynamic link libraries
 - concepts 66–8, 114–18
- static libraries
 - see also* import libraries
 - concepts 65–8, 114–18
 - creation 115
- status bars 360–5
- status checks, processes 285–6
- STL *see* Standard Template Library
- stock dialogs
 - see also* dialogs
 - concepts 401–5
- Stop 227, 254–7, 262–8, 269
- storage media 16–22, 71
 - see also* MMC...; removable memory cards
 - specifications 16–22
- StoreL 370–7
- stray-signal panics, active objects 269–70
- stricmp 187
- streaming media 323–4
- Streaming Media network service 327
- string literals, concepts 166–8, 171–2
- strings
 - see also* descriptors
 - binary data 165–6
 - concepts 165–8, 385–7
- STRUCT keyword, resource files 379–83, 391–6, 406–7
- sub-strings, descriptors 188–90
- Subscriber Identification Module (SIM) 5, 356
- suffixes, naming conventions 105–14, 121
- SurroundingsDD 224–5
- SVG-T *see* Scalable Vector Graphics Tiny
- SwEvent 224–5
- SWInstall 90–1
- switched processes, concepts 73–7, 82, 286
- symbian 123
- Symbian Developer Network 29–30
- Symbian Ltd 1, 12–15
 - see also* UIQ Technology AB
- Symbian OS
 - see also* operating systems; smartphones; sockets
 - application engines/ services/protocols 90–1
 - architecture 63–91, 409
 - basic data types 94–5
 - C++ concepts 93–122, 166–8, 323–4, 329–57
 - classes 95–100, 119–20, 165–215, 367–77
 - client–server model, concepts 68–70, 83–5, 292–3, 303–21
 - communications architecture 3–11, 16–22, 64–5, 86–90, 323–57
 - competitors 13–14, 20–2
 - components 63–4
 - concepts 1, 12–22, 23–31, 63–91
 - controls 405–9
 - development tools 2, 23–61, 123–60
 - DLLs 65–8, 78–9, 86–9, 114–18, 141–4
 - emulator 24–6, 31–8, 56–8, 118–19, 124–5, 126–31, 137–41, 286
 - Essential Booklets 30
 - flexible architecture 13–15
 - GUI architecture 13–14, 27–30, 38–56, 64–6, 83–5, 118–19, 125–6, 161–2, 303–4, 359–411
 - high performance graphics 85–6
 - historical background 12–15, 93–4
 - kernel 63–4, 72–9
 - memory 70–7
 - multitasking aspects 13–14, 64–5, 140, 250–1
 - naming conventions 68, 95–6, 119–21
 - network connections 356–7
 - ‘open’ aspects 1
 - overview 1–2, 13–15
 - owning manufacturers 13–14
 - phone specifications 16–22
 - platform security 2, 71–2, 76, 90–1, 217–46
 - platforms 13–22, 153–60, 359–65
 - programming basics 93–122
 - quick start guide 23–61
 - reference platforms 13–15, 359–65
 - sales 2
 - SDKs 2, 13–15, 23–61, 123–60, 164
 - semaphores 300–1
 - Socket API 334–57
 - statistics 2
 - TCP/IP 13–14, 64–5, 86–9, 323–57
 - templates 203–5
 - text console 161–5, 271
 - v6.0 26–7
 - v6.1 26, 35–6
 - v7.0 26–7, 31, 35, 89
 - v8.1 26, 77
 - v9.1 26–7, 225
 - v9.2 2, 26–7, 35–6, 66, 68, 89, 90, 118–19, 132, 135, 151, 153, 217, 236, 286
- Symbian OS C++ for Mobile Phones Volume 3* (Harrison) 409
- Symbian OS Platform Security* (Heath) 219, 227
- Symbian Signed 30, 60–1, 91, 218–46
 - see also* platform security
 - background information 238
 - basic capabilities 223
 - concepts 218–46
 - developer certificates 236, 244–6
 - extended capabilities 224–5
 - freeware option 240–1
 - official logo 238

- phone manufacturer capabilities
 - 220, 225–7, 243–4
- procedures 233, 236–7, 238–43
- process options 240–1
- Publisher Ids 236, 241–3
- publisher option 240–3
- root certificates 236–7
- self-certification procedures
 - 243
- submission procedures 242
- successful completion 243
- tests 238–43
- Symbian_Base 35–8
- synchronization
 - see also* critical sections;
 - mutexes; semaphores
- concepts 297–302
- threads 297–302
- synchronous functions, concepts
 - 247–8
- SyncML, application protocols
 - 64–5
- sys directory 226, 231–2
- SYSTEMINCLUDE 116–17, 119, 128–31, 141–4, 275, 278

- T (data type) classes, concepts
 - 96–100, 109–10, 120
- T-Mobile 356
- tabs 360–5
- TAny 95, 107–9, 204–5
- TARGET 55–6, 116–17, 119, 128–31, 141–4, 275, 278
- TARGETPATH 128–31, 278, 385, 411
- TARGETTYPE 55–6, 116–17, 128–31, 141–4, 275, 278
- TBool 95, 255–6
- TBuf 167–208, 263–8, 335–57, 380–3
 - see also* buffer descriptors
 - concepts 167–208, 380–3
 - memory layout 174
- TBufBase 169–71
- TBufC 167–208
 - see also* buffer descriptors
 - concepts 167–208
- HBufC 181
 - memory layout 176–7
- TBufCBase 169–71
- tbuffserver.h 312
- TC TrustCenter 233, 236, 241–3
- TCB *see* Trusted Computing Base
- TCE *see* Trusted Computing Environment
- TChar 95, 96, 192–3
- TCleanupItem 109–10
- TCP (Transmission Control Protocol) 323–57
- TCP/IP
 - see also* sockets
 - applications 323–57
 - client–server model 326–57
 - concepts 323–57
 - introduction 324–6
 - layering diagram 325
 - network connections 356–7
 - network programming 323–57
 - protocols 324–6
 - Symbian OS 13–14, 64–5, 86–9, 323–57
 - virtual connections 326–57
- tcpip6.prt 86–9
- TDbLQue 214
 - see also* linked lists
- TDbLQueLink 214
- TDes base class 169–71, 177, 187–98
- TDesC base class 167–73, 176–7, 187–98
 - see also* base classes;
 - descriptors
- Techview 15
- telephony server *see* ETEL server
- Telnet network service 326–7, 346–8
- template 204–5
- templates
 - arrays 205–6, 209–13
 - C++ 93–4, 109–10, 203–5
 - concepts 203–8, 209–13
- terminations
 - processes 284–5
 - threads 291–2
- text 52–4, 154–5, 159, 374–7
- TEXT... 154–5, 159
- text console, concepts 161–5, 271
- text notices, installation 154–5
- textbuffclient.h 307
- textbuff.h 313
- TextBuffServ 306–21
- TFindChunk 295–6
- TFindHandleBase 282–3
- TFindProcess 282–3, 295–6
- TFindSemaphore 299–300
- TFindThread 290, 295–6
- TFixedArray 205–6, 209–13
- thin templates 205–8
- third-party suppliers 1, 13–14, 16
- threads
 - see also* RThread
 - active objects 249–75, 300–1
 - cautionary uses 287
 - chunks 293–7
 - client–server model 303–21
 - concepts 64–5, 74–5, 77–9, 97–8, 277–8, 286–97
 - creation 287–92
 - definition 65
 - end-signaling method 292
 - executables 288
 - inter-thread communications 77–9, 281–2, 292–7, 318–20
 - multiple threads 65, 82, 140, 277–8, 286–92
 - opening methods 289–90
 - pre-emptive multithreading 65, 247, 250–1
 - priorities 290–1
 - Rendezvous 301–2, 310
 - running 287–9
 - starting 287–9
 - synchronization 297–302
 - terminations 291–2
- throw/catch exception C++ feature
 - 94–5, 102–3
- THUMB instruction set 135
- TIdentifyRelation 211
- timers, concepts 77–9
- TInt types 94–6, 104, 111–13, 204–5, 212–13, 214–15, 272, 282, 293–6, 297–9

- title bars 360–3
- TitleFont 374–7
- TLeave 111–13
- TLinearOrder 211, 213
- TLitC 171–2
- tool bars 359–411
- touch screens 15–17, 360–5
 - see also* pens
- TProcessId 281–2
- TProcessPriority 284
- TPtr
 - see also* pointer descriptors
 - concepts 168–215
 - memory layout 177–80
- TPtrC
 - see also* pointer descriptors
 - concepts 169–215
 - memory layout 177–80
- training, OS requirements 12
- transcoding features, WAP 8
- TRAP 101–14, 163
- trap mechanism, concepts
 - 101–14, 121, 162–5, 180–1,
 - 212–13, 253–4, 258–60,
 - 262–8, 314–21
- TRAPD 104–14, 314–21
- TReal types 95
- TRequestStatus 247–61,
 - 285–6, 336–57
 - see also* asynchronous functions
- TRes 169–71
- TrimAll 195–6
- TrimLeft 195–6
- TrimRight 195–6
- trust principle, platform security
 - 218
- Trusted Computing Base (TCB)
 - 90–1, 227, 229, 231–2, 243
- Trusted Computing Environment (TCE) 90–1
- trusted/untrusted certificates,
 - concepts 235–6
- TrustedUI 224–5
- TSocketAddr 341
- TSY files 88–9
- TSY modules, ETEL server 86–9
- TText types 95–6
- TTime 402–3
- TUint types 94–5, 191–2, 198,
 - 201–3
- TVersion 308–10
- TVwsViewId 54
- two-phase constructors, concepts
 - 112–14
- two-processor smartphone model
 - 80–1
- typedefs 95, 204–5
- UART 88–9
- UDEB 124–41, 164
- UDP (User Datagram Protocol)
 - 324–57
 - see also* sockets
 - client–server model 326–7
 - concepts 324–6
 - layering diagram 325
- _UHEAP_MARK 163–4
- _UHEAP_MARKEND 163–4
- UI *see* user interfaces
- UI classes, applications 38–56,
 - 365–411
- UI control framework *see* CONE
- UID 48–9, 116–18, 119,
 - 128–31, 141–4, 275, 282
- UID1 135–7, 141–4
- UID2 48–9, 135–7
- UID3 135–7, 151
- UIDs *see* unique identifiers
- UIKON, concepts 83–5
- uikon.rh 405
- UIQ 15–17, 26–7, 36–7, 38–56,
 - 83–5, 153–60, 353–6,
 - 359–411
 - see also* CQik . . . ; Motorola;
 - Sony Ericsson
 - characteristics 360–3
 - classes 41–56, 367–411
 - concepts 360–3
 - control structures 405–9
 - data input 362–3,
 - 387–405
- dialogs 363, 387–405
- emulator 57–8
- GUI architecture 15–17,
 - 38–56, 83–5, 359–411
- header file 41–4, 153–60
- package file 59–61, 153–60
- paper metaphor 362–3
- project build file 55–6
- quick-start development
 - examples 26–7, 36–7,
 - 38–56
- resource file 44–56, 387–405
- screens 360–3
- SDK 26–7, 36–7, 38–56, 123,
 - 125–60
- stock dialogs 401–3
- versions 15
- view architecture 409
- view class 53–4, 374–7, 409
- UIQ Developer Community 30
- UIQ Technology AB 15
 - see also* Symbian Ltd
- UIQ3SDK 123, 126
- UIQExamples 36–7, 125–6
- UMTS network protocol 6–7, 16
- Unicode 95, 165–6, 169–70,
 - 198–9, 378–80
- unique identifiers (UIDs) 38–56,
 - 116–18, 119, 128–31,
 - 135–7, 152–60, 237–8, 239,
 - 367–411
 - concepts 38–56, 128–31,
 - 135–7, 152–60, 237–8,
 - 368–77
 - getting 136–7
 - SIS files 152–60
 - vendor ID 137
- Unix 231, 247
- unsigned applications 234
 - see also* signed . . .
- UpperCase 194–5
- UREL 59–61, 124–38,
 - 152–60
- URLs 8
- USB connectivity 1, 9, 16–22,
 - 23–4, 64–5, 226, 334–5

- UseFont 374–7, 396
- User 94–5, 104–14, 115
- user interfaces (UI)
 - see also* graphical...; Series...; UIQ...
 - classes 38–56, 365–411
 - concepts 13–22, 38–56, 83–5, 137–8, 359–411
 - customization 83–5
 - OS requirements 12, 13–15, 83–5, 359–65
 - specifications 16–22
 - types 13–22, 359–65
- user library, concepts 78–9
- User::After 119, 251, 269, 271–5, 287–8
- User::AllocLC 110–11
- user.dll 161
- UserEnvironment 222, 230–1
- USERINCLUDE 116–17, 119, 128–31, 141–4, 275, 278
- User::InfoPrint 119, 287–8
- User::Leave 103–14, 212–13, 262–8, 279–81, 287
- User::LeaveIfError 104–14, 212–13, 262–8, 279–81, 287, 316–19
- User::LeaveIfNull 104–14
- User::LeaveNoMemory 104–14
- User::Locked... 301
- User::Panic 113–14, 313–21
- User::PrintInfo 173
- User::QueryVersion-Supported 309–10, 315–21
- User::WaitForRequest 248–9, 258–61, 270, 286, 292, 310, 335–57
- UTF-8 198–9, 378
- variables
 - global variables 118–19, 120
 - naming conventions 120
- vendor ID, concepts 137
- Verizon Wireless network 5
- Version 308–10
- VGA screens 15–17, 360–3
- video 4, 8, 11, 16–22
- video conferencing 4
- view architecture, concepts 409
- view classes 38–56, 360–5, 368–411
- ViewConstructFrom-ResourceL 54
- ViewId 54, 375–7
- virtual buttons, emulator 138–40
- virtual connections, TCP/IP 326–57
- virtual declarations, polymorphic DLLs 66–7
- virtual drives, emulator 138–41
- virtual functions 93–4, 98–100, 107–8, 117–18
- virtual keyboards 15–17, 138–40
- virtual memory addresses
 - concepts 71–7
 - memory map 73–7, 286
- VirtualKey 138–40
- viruses 218–19
- voice transfers 3–4
- voice-over-IP 17
- void data type 95, 257–8
- VPN 219
- VRTX 80
- W-CDMA technology 7
- w32.dll 83–4
- Wait 256–8, 260–1, 270, 298–302
 - see also* synchronization
- WaitForRequest 248–9, 256–7, 258–61, 270, 286, 292, 310, 335–57
- WAP browsers 8–9, 16–22, 24, 150, 323–4
- web browsing *see* browsing
- Wi-Fi 7, 16–22
- Wi-Fi network protocol 221, 226, 323–4, 357
- wildcard searches
 - descriptors 189–90
 - processes 282
- Win32 development tools 23–61
- window server
 - see also* servers
 - animation plug-ins 85–6
 - concepts 83–5, 303–4
- Windows 20–2, 23–61, 64–5, 118–19, 124–5, 137–41, 286
 - see also* Microsoft
 - 2000 23–4
 - CE 20–1
 - development package (Win32) 23–61
 - emulator 24–6, 31–8, 56–8, 118–19, 124–5, 137–41, 286
 - Mobile family 13, 20–1, 233–4
 - Vista 23
 - XP 23–4
- Windows development tools 23–61, 137–41
 - see also* development tools
 - components 24–6
 - concepts 24–6, 34–8, 137–41
 - debuggers 24–6, 127, 137–8
 - examples 34–56
 - getting 27–30
 - monopoly situation 30–1
 - problems 37–8
 - providers 27–30
 - quick test 34–8
- WindowTitle 138–40
- wins 36–8, 124–5
- winsb 37, 124–5
- winscw 28–30, 36–8, 56–8, 124–5, 127–35, 147–9, 163–4
- WLAN 89, 323
- WML 8
- Word 20

- WORD resource files 378–80,
391–6, 406–7
- World Wide Web 324
see also browsing; Internet
- wrapper classes 109–10, 117,
310–12, 405
- Write 97–100, 200–3
- WriteDeviceData 224–5,
230–1, 318
- WriteL 319
- WriteUserData 222
- WServ 90–1
- wunderground.com 346–57
- x86-based Windows binaries 28,
139–40
- xHTML 8
- z: drive
see also Read Only Memory
concepts 70–1, 124–5,
138–41
- Zero 197
- ZeroTerminate 196–7