

Symbian OS C++ for Mobile Phones

Volume 3



Application Development for Symbian OS v9

symbian

Richard Harrison
& Mark Shackman

Symbian OS C++ for Mobile Phones Volume 3

Richard Harrison, Mark Shackman

With

**Adi Rome, Alex Wilbur, Andrew Jordan, Douglas Feather,
Ernesto Guisado, Hassan Ali, Ioannis Douros, John Pagonis,
Lucian Piros, Mark Cawston, Martin Hardman, Mathew Inwood,
Rick Martin, Sanjeet Matharu, Tim Williams, Yang Zhang**

Reviewed by

**Graeme Duncan, Guanyun Zhang, Ian McDowall,
Jehad Al-Ansari, Jonathan Allin, Jo Stichbury,
Kostyantyn Lutsenko, Lane Roberts, Lars Kurth, Mark Jacobs,
Mark Welsh, Mathias Malmqvist, Matthew O'Donnell,
Rahul Singh, Ricky Junday, Robert Palmer,
Rosanna Ashworth-Jones, Sorin Basca, Tim Labeeuw,
Warren Day, Will Bamberg**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Symbian OS C++ for Mobile Phones

Volume 3

Symbian OS C++ for Mobile Phones Volume 3

Richard Harrison, Mark Shackman

With

**Adi Rome, Alex Wilbur, Andrew Jordan, Douglas Feather,
Ernesto Guisado, Hassan Ali, Ioannis Douros, John Pagonis,
Lucian Piros, Mark Cawston, Martin Hardman, Mathew Inwood,
Rick Martin, Sanjeet Matharu, Tim Williams, Yang Zhang**

Reviewed by

**Graeme Duncan, Guanyun Zhang, Ian McDowall,
Jehad Al-Ansari, Jonathan Allin, Jo Stichbury,
Kostyantyn Lutsenko, Lane Roberts, Lars Kurth, Mark Jacobs,
Mark Welsh, Mathias Malmqvist, Matthew O'Donnell,
Rahul Singh, Ricky Junday, Robert Palmer,
Rosanna Ashworth-Jones, Sorin Basca, Tim Labeeuw,
Warren Day, Will Bamberg**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Copyright © 2007 Symbian Software Ltd

Published by 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.

Library of Congress Cataloging-in-Publication Data

Harrison, Richard.

Symbian OS C++ for mobile phones / By Richard Harrison.

p. cm.

Includes bibliographical references and index.

ISBN 0-470-85611-4 (Paper : alk. paper)

1. Cellular telephone systems – Computer programs. 2. Operating systems (Computers) I. Title.

TK6570.M6H295 2003

621.3845'6 – dc21

2003006223

British Library Cataloguing in Publication Data

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

ISBN: 978-0-470-06641-6

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

About the Authors	xi
Acknowledgements	xix
Symbian Press Acknowledgments	xxi
About this book	xxiii
Glossary	xxvii
1 Getting Started	1
1.1 Using the Emulator	1
1.2 Hello World – Text Version	6
Summary	15
2 A System Introduction to Symbian OS	17
2.1 Symbian OS Basics	17
2.2 Framework Basics	30
2.3 APIs Covered in this Book	36
Summary	39
3 Symbian OS C++	41
3.1 Fundamental Data Types	41
3.2 Naming Conventions	43
3.3 Functions	49
3.4 APIs	51
3.5 Templates	55
3.6 Casting	56

3.7	Classes	57
3.8	Design Patterns	59
	Summary	60
4	Objects – Memory Management, Cleanup and Error Handling	61
4.1	Object Creation and Destruction	62
4.2	Class Categories in Symbian OS	66
4.3	Error Handling	70
4.4	The Cleanup Stack	76
4.5	Two-Phase Construction	81
	Summary	85
5	Descriptors	87
5.1	Overview	87
5.2	Anatomy of Descriptors	91
5.3	Literals	93
5.4	Stack Descriptors	93
5.5	Pointer Descriptors	95
5.6	Heap Descriptors	99
5.7	Narrow, Wide and Neutral Descriptors	108
5.8	Descriptors and Binary Data	109
5.9	Using Descriptors with Methods	110
5.10	Some Descriptor Operations	121
5.11	Correct Use of Descriptors	128
5.12	Manipulating Descriptors	138
	Summary	148
6	Active Objects	151
6.1	The Asynchronous Service	151
6.2	Multitasking and Pre-emption	156
6.3	A More In-depth Look at Active Objects	158
6.4	How It Works	167
6.5	Active Object Priorities	171
6.6	Active Object Cancellation	176
6.7	Starting and Stopping the Scheduler	181
6.8	Understanding a Stray Signal	182
6.9	Other Common Active Object Errors	187
6.10	Implementing State Machines	190
6.11	Long-Running Tasks and Active Objects	194
	Summary	200
7	Files and the File System	203
7.1	File-Based Applications	203
7.2	Drives and File Types	204

7.3	File System Services	205
7.4	Streams	216
7.5	Stores	224
	Summary	239
8	Interprocess Communication Mechanisms	241
8.1	Overview	241
8.2	Client–server IPC	247
8.3	Publish and Subscribe IPC	253
8.4	Message Queue IPC	257
8.5	Which IPC Mechanism Should You Use?	259
	Summary	261
9	Platform Security and Publishing Applications	263
9.1	Releasing an Application	263
9.2	How Does Platform Security Work?	264
9.3	How Do I Support Platform Security?	266
9.4	Preparing an Application for Distribution	268
9.5	Overview of Symbian Signed	272
9.5	Installing a SIS File	279
9.6	List of Capabilities	280
	Summary	281
10	Debugging and the Emulator	283
10.1	Using the Emulator	283
10.2	Emulator Debugging	289
10.3	Debugging on a Phone	306
10.4	Miscellaneous Tools	308
	Summary	309
11	The Application Framework	311
11.1	Symbian OS Application Framework	311
11.2	S60 and UIQ Platform Application Frameworks	313
11.3	A Graphical Hello World	315
	Summary	330
12	A Simple Graphical Application	331
12.1	Implementing the Game on S60	333
12.2	Differences for UIQ 3	355
	Summary	366
13	Resource Files	367
13.1	Why a Symbian-Specific Resource Compiler?	367
13.2	Source File Syntax	368
13.3	Bitmaps and Icons	372

13.4	Updating the Resource Files	377
13.5	Application Registration Files	378
13.6	Localizable Strings	379
13.7	Multiple Resource Files	382
13.8	Compiling a Resource File	383
13.9	The Content of a Compiled Resource File	385
13.10	Reading Resource Files	388
	Summary	389
14	Views and the View Architecture	391
14.1	The View Architecture	392
14.2	The <code>MCoableView</code> Interface	397
14.3	Introduction to the Example Application	398
14.4	Creating and Managing the Views	406
14.5	Implementing the <code>MCoableView</code> Interface	408
14.6	Command Menus	414
	Summary	421
15	Controls	423
15.1	What Is a Control?	423
15.2	Control Types	424
15.3	Control Layout	429
15.4	Handling Key and Pointer Events	432
15.5	Observing a Control	442
15.6	Drawing a Control	444
15.7	Backed-up Windows	451
15.8	Backed-up-Behind Windows	452
15.9	Dimmed and Invisible Controls	454
	Summary	455
16	Dialogs	457
16.1	What Is a Dialog?	457
16.2	Simple Dialogs	459
16.3	Complex Dialogs	462
16.4	Single-Page Dialogs	464
16.5	Multi-Page Dialogs	466
16.6	Dialog APIs	472
16.7	Stock Controls for Dialogs	475
16.8	Custom Controls in Dialogs	477
	Summary	480
17	Graphics for Display	481
17.1	Drawing Basics	482
17.2	The <code>CGraphicsContext</code> API	486

17.3	Drawing and Redrawing	492
17.4	Drawing Controls	499
17.5	Sharing the Screen	502
17.6	Support for Drawing in <code>CCoeControl</code>	514
17.7	Special Effects	521
17.8	Window Server Features	526
17.9	Device- and Size-Independent Graphics	529
	Summary	556
18	Graphics for Interaction	557
18.1	Key, Pointer and Command Basics	558
18.2	User Requirements for Interaction	559
18.3	Some Basic Abstractions	561
18.4	Processing Key Events	567
18.5	Processing Pointer Events	571
18.6	Window Server and Control Environment APIs	576
	Summary	582
19	Plug-ins and Extensibility	583
19.1	System Services	583
19.2	What Is a Plug-in?	587
19.3	The ECOM Library	593
19.4	Plug-ins in Symbian OS	603
	Summary	604
20	Communications and Messaging Services	605
20.1	Communications in Noughts and Crosses	605
20.2	Communication Between Controller and Transport	606
20.3	Serial Communications	609
20.4	Socket-based Communications	615
20.5	Messaging	632
20.6	Security	646
	Summary	649
21	Multimedia	651
21.1	The Multimedia Framework	651
21.2	The Image Conversion Library	682
21.3	Camera API	699
21.4	Tuner API	706
	Summary	712
22	Introduction to SQL RDBMS	715
22.1	Overview of RDBMS	715
22.2	SQL Basics	716

22.3	Symbian SQL Server Component Architecture	729
22.4	Symbian SQL Error Codes	751
	Summary	753
Appendix: Developer Resources		755
References		761
Index		763

About the Authors

Richard Harrison, Lead Author

Richard spent the first years of his working life teaching mathematics, physics and computer science. During that time he wrote a Forth language implementation for Acorn Computers, and wrote accompanying user manuals for the Acorn Atom and BBC Micro.

He joined Psion in 1983, and worked on a range of documentation and software projects. Amongst other roles, he was the principal designer and author of the Psion Series 3 word processor, and lead author for the Psion SIBO SDK.

Richard transferred to Symbian at its inception in 1998, when his initial responsibility was to build up and lead the System Integration team. More recently he has acted as adviser and author in the Symbian Press team.

Educated at Balliol College, Oxford with an MA in Natural Science (Physics), Richard also gained an MSc in Astronomy from Sussex University, and spent a further two years of postgraduate research in the Astronomy Group at Imperial College.

Outside work, Richard's interests include a fascination with lights in the sky – stemming from life-long loves for both Astronomy and Fireworks. These days, his interest in Astronomy tends to be of the armchair variety, but he still acts as an occasional firer on a range of public and private firework displays.

Mark Shackman, Lead Author

Mark graduated with a first-class honors degree in Computing Studies, followed by a Masters in Digital Systems and finally a Postgraduate

Certificate of Education. After six years of teaching mathematics, physics and computing and a spell at Morgan Stanley, he joined Psion Software in 1997 as a Technical Author working on SDK content and installation technologies.

After the formation of Symbian, Mark joined the Connectivity Engineering group, with sole responsibility for authoring, producing, delivering and supporting the Connectivity SDK. He also wrote a chapter in Symbian's first book, *Professional Symbian Programming*. In 2001 Mark moved to the Kits team, becoming Technical Architect shortly afterwards, with the responsibility of introducing both the new Package Manager Kit format and subsequently the Component-Based Releases.

In 2004, Mark transferred to the Symbian Developer Network, and now provides technical support to developers in the form of presentations, papers, books and tools. He specializes in platform security, has given presentations in the USA, Brazil, Israel, India, the Philippines, Singapore, at 3GSM & at the Smartphone Shows, and wrote for the *Symbian OS Platform Security* book.

Mark spends some of his spare time shooting things – but only from behind a Canon lens. He's only ever had two pictures published and firmly expects it to stay that way.

Adi Rome

Adi has a BA in Computer Science from Tel Aviv University. After a spell in teaching, she worked in Quality Assurance, moving to analysis, design and implementation of embedded systems, before joining Symbian in 2004. Having spent a while in the Multimedia team, becoming involved in all aspects of the development and delivery software lifecycle, Adi joined the Developer Services team as the multimedia guru. Within this team, she also provided technical consultancy across all areas of Symbian OS, with a special emphasis on hardware – she was responsible for communicating the Symbian roadmap to silicon and multimedia partners, and was Symbian's lead on the partner component validation program. Away from technical work, Adi has been involved in setting up the Symbian China partner-consulting organization and is in much demand for her presentation skills, having given sessions at a wide range of partner and industry events, including 3GSM World Congress.

Andrew Jordan

Andrew joined Symbian in late 2000 where he worked in the Connectivity team for a short time before moving to the Base Peripherals team. Here he worked on device and media drivers, the file server and file systems. Working mainly with Japanese customers for the Japanese market, Andrew

currently lives in Japan where he has been for the last two years having moved to a product development team.

Alex Wilbur

Alex graduated from the Royal Military College of Science in 1999 with a first-class honors degree in software engineering. A keen third-party software author for the Psion Series 5, Alex joined Symbian's PIM team in 1999.

2001 saw Alex joining Symbian's Professional Services department as a Senior Technical Consultant. Since then he has been living in Tampere, Finland at the heart of Nokia's S60 Research and Development site, consulting on all the S60-based smartphones, from the 7650 onwards.

Alex enjoys all aspects of home cinema, the Finnish outdoors and also spending time with his partner, Ina, and their daughter, Elli.

Douglas Feather

Douglas joined Symbian (then Psion) in 1994. He started by leading the writing of the text formatting engine for the Psion S5. He has also work in the Web Browser and Core Apps teams where he rewrote most of the Versit parser to greatly improve its performance. For seven of the last nine years, he has worked mainly on the Window Server, where he added full color support, fading support, a framework to allow digital ink to be drawn over the other applications, support for multiple screens and transparent windows.

Douglas has a BSc in mathematics from Southampton University and a PhD in number theory from Nottingham University. He is a committed Christian and can be regularly seen on a Sunday afternoon at Speaker's Corner (Hyde Park, London) engaging in theological debate to defend the biblical truths about Jesus Christ.

Ernesto Guisado

Ernesto is a senior C++ developer. Between 2001 and 2006, he worked at Symbian as developer, technology architect and technical lead for Symbian OS v9.1. In the process, he learned more about Platform Security and API compatibility than any sane person would want to. In his spare time he publishes articles for computing magazines and contributes to open source projects in C++, Perl and Ruby. He has a website and blog at <http://erngui.com>.

Hassan Ali

Hassan joined Symbian's Crystal team 2000. He started work on the 9210 project mainly on defects triage, true testing and Java compliance testing. He then moved onto Techview messaging and worked on the SyncML application, SyncML Ui Notifiers, messaging MTMs and Automated Smoke tests. Hassan then joined the Multimedia test team where he worked on automating the hardware testing and was a technical lead for Multimedia's EABI compiler migration. Currently he is working in Product Delivery as OS technical lead of maintenance products, where he automated the patch (ICD) delivery system. He is also involved in the Java automated build system and providing defect analysis support to the project office to help prioritize customer issues.

Ioannis Douros

Ioannis received his PhD in three-dimensional human body modeling from University College London, where he has also worked for a number of years as a research fellow in the areas of computer vision and intelligent systems.

He joined the Symbian ecosystem in 2004 and is currently a software engineer in the Converged Communications team. He is an Accredited Symbian Developer, as well as an unrepentant caffeine addict. He lives in rural Buckinghamshire with his partner, Sue.

John Pagonis

John joined Symbian in 1998, where he worked on the development of the Ericsson R380 smartphone and later for the team that developed the Symbian OS Bluetooth stack. Since then, John has worked on many areas of Symbian OS with his experience ranging from communication protocols, security, location-based services and operating system internals to software-engineering methodologies, developer consulting, team coaching and organizational improvements.

Currently, John is working as a consultant to developers, trying to keep them happy, among other ways, by writing articles and code and giving seminars and presentations to whoever wants to learn about developing for Symbian OS mobile phones. John is a visiting lecturer at City University in London and is (still) a PhD candidate at the University of Essex, from where he holds an MSc (Hons) in computer and information networks as well as a BEng (Hons) in computers and networks. John also believes that it is better when certain things are kept unread.

Lucian Piros

Since joining Symbian, in January 2006, Lucian has worked with the PIM and Internet team on various releases of contact model architecture and has been involved in the latest contact model architecture design. Before joining Symbian, Lucian worked for two years as independent developer on core Symbian OS applications for a number of UI platforms based on Symbian OS. He has eight years industrial experience, most of them spent as a C++ programmer writing telecommunications applications. Educated at Babes-Bolyai University, Cluj Napoca, with a BSc in mathematics and computer science, he enjoys spending his free time reading, traveling and cooking.

Mark Cawston

Mark joined Symbian after graduating from the Computer Science Department of the University of York. Since then he has worked across the operating system, from the kernel and device drivers, through file systems, databases, contacts and messaging engines, security, short-range connectivity and communications, up to the user interface and applications. He has combined a career developing the core Symbian OS functionality, with helping manufacturers to deliver phones powered by Symbian OS.

Martin Hardman

Martin joined Psion in 1994, remaining with the company for 10 years as it evolved into Symbian. During that time he worked in several areas and had several roles including Senior Technology Architect and System Architect. He was also Symbian's representative on the SyncML and OMA Data Synchronization and Device Management working groups. He graduated from South Bank University in London with a first-class degree in computing.

In 2004, he moved to Vancouver, where he now lives with his wife Hai Wei, to spend more time snowboarding. He works for Intrinsyc Software where he is still involved with Symbian development.

Mathew Inwood

Mathew joined Symbian's technical consulting group as a graduate in 2003. Since then, he has been involved in the development of the multimedia subsystem for the Symbian OS v9.1 range of Sony Ericsson

phones. He has a BA in computer science from Cambridge University, during which he worked as a summer intern at Amadeus in France. He is a keen paraglider pilot and tries to go flying as often as he can.

Rick Martin

Rick is a New Zealander who joined Symbian in 2005. One of Rick's hobbies is collecting qualifications. His first was in physics in the early 1980s. After that he graduated in software engineering, going on to qualify in french studies and classical history. His latest project is working toward an OU post-graduate degree in physical science, with a particular interest in quantum computing. On one particularly sad occasion, he sat and passed the Mensa exam.

Sanjeet Matharu

Sanjeet (Sanj) graduated from the University of Westminster in 1998 with a BSc in computing. He eventually found himself joining Symbian in 2000 as a software engineer working on test tools. He developed much of the core code for the Symbian Test Automation Tool (STAT) which is now used in a majority of integration testing for Symbian OS. He also worked on a number of other tools within the System Test team as a developer, technical architect and project manager culminating in the creation of the Symbian Test Network (involving many hours of reverse engineering and assembling rack-mount servers). This network is now used as a test harness for much of the Symbian OS testing.

In 2004, he moved into Marketing and had a direct involvement in the Symbian Signed program and Developer Certificates. Since then he has moved on to manage the Symbian Signed program and the Symbian Developer Network with his core focus being on after-market application developers and other developer programs such as Forum Nokia and Sony Ericsson Developer World. Sanj would like to thank his team for putting up with his continual demands to get things done and his wife, Jaskie, for just generally putting up with him.

Tim Williams

Tim joined Symbian when it took over Origin's Automation Technology group with which he had worked mainly on manufacturing control systems. An interest in user interfaces saw him work on the first S60 phone and Symbian's smartphone reference user interface. Currently a technical author he was previously a user-interface specialist in the

Symbian Licensee Technical Consulting group. Away from the office, he is normally riding his bike or coaching triathletes. He is married to Heather.

Yang Zhang

Yang has a BEng in communication systems, an MSc in physics and a PhD in engineering. He has contributed a number of works to IEEE journals, international conferences and book chapters. He is now working for Symbian after years of industrial experience in the telecommunications domain. His current interests include mobile social networking, mobile computing and mobile machine learning.

Acknowledgements

We would like to thank:

The authors, who toiled hard and long to give us their manuscripts, and the technical reviewers who corrected and enhanced the text.

Freddie Gjertsen, who gave us the opportunity to polish our egos by writing another book, and Laura Sykes.

Satu McNabb, for keeping our noses to the grindstone and making sure that the work at each stage was completed – more or less – on time.

Our colleagues, who answered our queries and bore additional workload as this volume demanded ever-increasing amounts of time.

All those at Wiley who coaxed us into agreeing to a challenging schedule for delivery of the text and then, following delivery, added their inimitable professional touches.

Our families and friends who, throughout the long process of preparing this book, saw much less of us than they had the right to expect

Symbian Press Acknowledgments

Symbian Press would like to thank Richard for not retiring too soon, and Mark for more than helping him to bear the heavy burden of working with us.

We'd also like to thank every member of the glittering crowd that contributed: the authors who worked so hard to create this book and the reviewers who shared their expertise by reviewing the content. A special thank you to Adi for saving us as disaster loomed – very much appreciated.

And last but not least, Satu, Freddie, Laura, Mark, Jo, Phil and everyone at Developer Product Marketing would like to wish Richard a relaxing retirement and every happiness in the future; it really has been a pleasure working with you.

About this book

Symbian OS C++ for Mobile Phones Volume 3 draws on the experience of Symbian's own engineers to provide a thorough grounding in writing C++ applications for mobile phones that use Symbian OS version 9.0 and beyond. It won't teach you *everything* you need to know about developing Symbian OS applications – no single book could do that. However, it will take you a long way along the road to being an effective Symbian OS developer, and give you a deep understanding of the fundamental principles upon which Symbian OS is based. The text is complemented with a specially developed suite of examples.

The book is broadly organized into four sections:

- **Section one** (Chapters 1 to 6) introduces Symbian OS itself, and describes the basic building blocks and usage patterns.
- **Section two** (Chapters 7 to 11) explains the core concepts, resources, APIs and programming idioms that you need to create, test and publish a simple GUI (graphical user interface) application.
- **Section three** (Chapters 12 to 18) provides a detailed description of the use of the Symbian OS graphical user interface to create non-trivial stand-alone applications. It looks deeply into the effective use of the available range of graphics APIs and helps you to ensure that your application code is as device-independent as possible.
- **Section four** (Chapters 19 to 22) provides an introduction to some of the most significant and useful of the Symbian OS system services. Starting with a discussion of extensibility and the use of plug-ins, this section continues with practically-based descriptions of the communications, multimedia and database services.

Symbian OS is used in a variety of phones with widely differing screen sizes. Some have full alphanumeric keyboards, some have touch-sensitive screens and some have neither. As far as possible, the material in this book is independent of any particular user interface. However, real applications run on real phones so, where necessary, we have chosen to use concrete examples based on both the S60 and UIQ user interfaces. Wherever relevant, the text explains the principal differences between these two user interfaces. This kind of information is invaluable for anyone who wishes to create versions of an application to run on a variety of Symbian OS phones.

Symbian OS C++ for Mobile Phones Volume 3 complements Symbian OS software development kits. When you've put this book down, the UIQ and S60 SDKs will be your first resource for reference information on the central Symbian OS APIs that we cover here. For more specialized and up-to-date information relating to a specific mobile phone, you will probably need to refer to a phone-specific SDK, available from the relevant manufacturer.

The SDKs contain valuable guide material, examples and source code, which together add up to an essential developer resource. We've pointed to these where they tie in with the book content. But as a general rule, look in the SDK anyway: you'll usually find additional information that explains things further than we could in this one book.

Who Is This Book For?

If you've programmed, at any level, in C++, it's for you. As a real and comprehensive system written in C++ from the ground up, and targeted at the high-growth area where computers and mobile communications converge, Symbian OS gives you unparalleled opportunities in mass-market, enterprise and system programming.

Besides C++ programmers, this book is of interest to other audiences:

- any other programmer or manager looking to exploit the potential of mobile solutions with Symbian OS technology
- consultants, trainers and authors thinking of basing their activity on Symbian OS technology
- anyone with an interest in system design, since Symbian OS is a full and interesting example in its own right.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

These boxes hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

We use several different fonts in the text of this book:

When we refer to words you use in your code, such as variables, classes and functions, or refer to the name of a file, we use this style: `iEikonEnv`, `ConstructL()`, or `e32base.h`.

URLs are written like this: ***www.symbian.com/developer***

And when we list code, or the contents of files, we use the following convention:

Lines that show concepts directly related to the surrounding text are shown on a gray background

We show commands typed at the command line like this:

```
abld build winscw udeb
```


Glossary

ACID	A database operation should be atomic, consistent, isolated, durable
ACS Publisher ID	Class 3 ID certificate
Actor	Something or someone (a system or end-user) with which or whom a use case scenario interacts
ANSI	American National Standards Institute
AppUi	A class that forms the core of an application
Automatic variable	a variable that is created automatically when it is required and destroyed when it goes out of scope
BLOB	Binary Large Object
Capabilities	API protection within Symbian OS
Chunk	A mechanism by which the kernel (memory model) allocates and manages memory, contains physical RAM pages mapped to virtual addresses
CLOB	Character Large Object
CONE	A control environment that provides the basic framework for controls
Context switch	A task-switching facility to switch between programs and keep track of the execution environment for the first program
Control	provides the principal means of interaction between an application and the user; each application view is a control, which forms the basis of all dialogs and menu panes

Co-operative multitasking	A multitasking environment in which multiple programs running on a CPU voluntarily yield control of the processor to each other
DBMS	Database Management System
DFC	A deferred function call that forms part of the driver non-ISR processing
Dialog	A specialized window-owning compound control.
DLL	Dynamic Linked Library
ECOM	A system service developed with the express purpose of enabling application extensibility
EKA2	The Symbian OS kernel architecture
Eshell	A simple command-line-mode shell
Event-driven Application	An application that is always waiting for the user to interact with the device and then carries out some operation in response to that interaction
FEP	Front-end processor
FIFO	First-in, first-out order
GC	Graphics context
GUI	Graphical user interface
IMEI	A unique number used by the GSM network to identify valid devices
IPC	Inter-Process Communication
ISR	Interrupt-Service Routine
MBM	Multi-bitmap file format; a graphics format used by Symbian OS
Metrowerks Target	A Symbian OS application that runs inside the OS and resident kernel and provides debugging services over serial communications (MetroTRK)
Non-pre-emptive Multitasking	see co-operative multitasking
Panic	A run-time failure in a program
Plug-in	A computer program that interacts with a main (or host) application (a web browser or an email program, for example) to provide a certain, usually very specific, function on-demand.
Pre-emptive Multitasking	A multitasking environment in which the operating system determines when and for how long a program has control of the processor

Process	A running instance of a program; unit of memory protection
RDBMS	A relational-database management system
Real-time Nanokernel	The core of the modern Symbian OS kernel architecture (EKA2)
Round robin	an operating-system scheduling algorithm which allocates equal processing time to all active processes
Scalable vector graphics	A format for icons, introduced by S60 3rd edition
SID	Secure ID
Store	A collection of streams, generally used to implement the persistence of objects
Stream	An external representation of one or more objects
Thread	The basic unit of execution – a sequence of instructions that can be executed
Two-phase construction	A mechanism that separates the safe constructions (which can be put into the constructor) from the unsafe constructions
Use case	A complete sequence of actions, initiated by an actor, that represents a particular way of using a system; often used for test scenarios
View server	A server with which each application may register its views
View	A class that implements an interface to the view architecture
Window server	A server that ensures that the correct window or windows are displayed, managing overlaps and exposing and hiding windows as necessary
XML	Extensible Markup Language

1

Getting Started

It seems to be traditional to start a book on computer programming with a ‘Hello World’ example and, although this book is more about an operating system than a programming language, we are following that tradition. In the process we introduce you to the emulator and to the tools for building C++ programs, so that by the end of the chapter you will know how to build and run a Symbian OS application. We don’t get too involved in describing Symbian OS programming conventions, API functions, and so forth; instead, we concentrate on the tools you need and how to use them, leaving the details until later chapters.

First we briefly describe the emulator. Most Symbian OS software is developed first on the emulator and only then on real target hardware. The emulator also includes a number of Symbian OS applications, and so mimics a real Symbian OS phone very closely. You will need to become familiar with the emulator and in the process we can take a look at the various graphical user interfaces (GUIs) used by Symbian OS.

Then we create a program. The easiest things to build are text-mode console programs, so that’s the form of the classic ‘Hello World’ application that we use. We demonstrate how to compile it for the emulator, and how to launch it using the Carbide.c++ IDE.

1.1 Using the Emulator

The emulator is a fundamental tool for all the Symbian OS SDKs, so it’s vital that you get to know how to use it.

If you are a newcomer to Symbian OS, the emulator offers an opportunity to get to know some Symbian OS basics from a user’s perspective, so we look at these straight away. Later, you’ll want to learn to make

effective use of it as a developer, so we cover the details of its operation in Chapter 10.

If you have some experience of Symbian OS, you may want to skip straight to Section 1.2 and start building an application.

Launching the Emulator

The first piece of software you need is a software development kit (SDK). There are a couple of Symbian OS v9 SDKs available, depending on the phone(s) you want to target. If you're unsure which SDK to select, we recommend starting with both a S60 v3 and a UIQ 3rd Edition SDK. You can obtain these via the links on the Symbian developer website (developer.symbian.com). Once you've installed your SDK, you can launch the emulator in any of the following ways:

- launch the executable `epoc.exe`, which you'll find in the directory `\epoc32\release\winscw\udeb` underneath the directory in which the SDK is installed
- from the Start menu, select either Programs, UIQ SDK or Programs, S60 Developer Tools, 3rd Edition SDK and select Emulator from the appropriate submenu.

However you choose to start it, and whichever emulator you're using (either for S60 or UIQ), the first thing you'll see in the emulator is the application launcher. As its name indicates, the application launcher enables you to launch applications. Its menus allow you to view or change system settings and it also has a control panel. It's very easy for end users to get to know the application launcher; you don't really need a manual. Just click with the mouse here and there and you'll soon find out what it has to offer.

GUI Style

If you've started up the UIQ emulator (shown in Figure 1.1a with the P990 extensions), as you browse around the application launcher you'll begin to see how UIQ is optimized for the pen-based mobile phone form factor. UIQ is designed as a 'read mostly' user interface, to be used mainly for browsing and for making a selection from a range of options with a single tap of a pen. Other GUIs – such as the S60 interface shown in Figure 1.1b and used, for example, on the Nokia Nseries phones – are optimized for the different hardware resources of the devices on which they are intended to run.

Although the various GUIs may have a superficially different appearance, they all rely on a common set of underlying features, some of which are briefly described in the next section.



Figure 1.1 UIQ (with P990 extensions) and S60 emulators

Screen layout

The UIQ screen layout, illustrated in the following diagram, includes the following areas (from top to bottom of the screen):

The *title bar* displays the name and icon of the current application. It also contains a View context area. Selecting the label opens the menu in Pen style. The width of the menu is adjusted to fit the longest label. The *View context area* can contain controls, for example icons, text labels and tabs. If there is enough space to the right of the tabs, other application-specific data or controls can be added.

The *menu bar* contains one or more menus, whose names and contents change from application to application, and also as you change view within a particular application. In UIQ, the menu bar usually contains two menus on the left and may optionally have a folder menu on the right.

The *application space* is the central area of the screen, where an application's view is displayed. Applications use this area in whatever way is appropriate to the information that they display.

Optionally, an application displays a *button bar* at the bottom of the application space. The most common use is to provide buttons to move

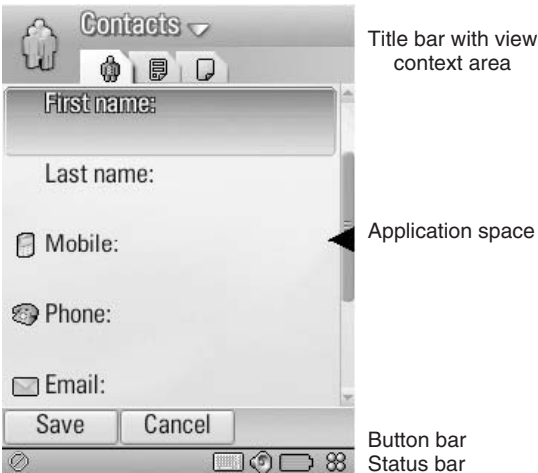


Figure 1.2 UIQ screen layout

between the application’s various views. In UIQ, a detail view, such as the one in Figure 1.2 that shows the detail of a single Contact entry, usually has a special button in the lower right corner to return you to the main view.

The *status bar* displays information such as battery charge, time of day, signal strength and notification of incoming messages. The P990i’s status bar includes a keyboard icon in the lower right corner, which is used to display a virtual keyboard for text input if you do not wish to use handwriting recognition.

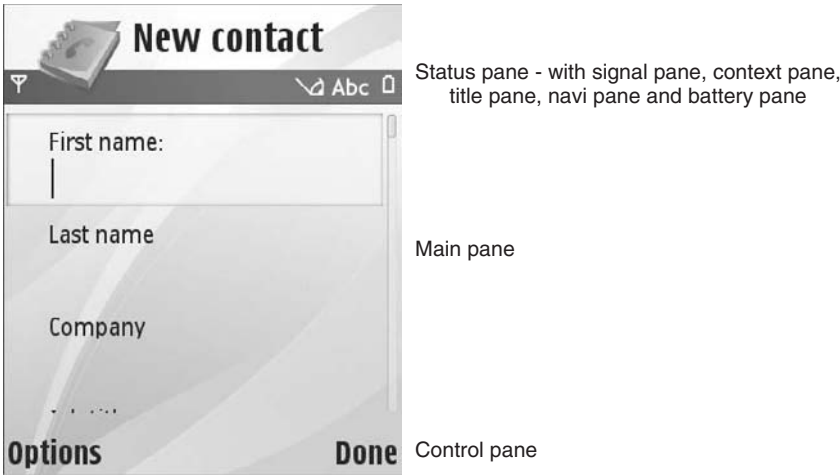


Figure 1.3 S60 screen layout

Also displayed in some views is the *application picker*, containing icons that allow you to switch applications. Selecting an icon brings the application it represents to the foreground. The application launcher icon brings the application launcher to the foreground, allowing you to launch applications that are not displayed on the application picker. If you wish, you can customize the application picker to launch your own preferred set of applications.

Most of these screen layout elements can be recognized in other GUIs used with Symbian OS, such as in Figure 1.3, though they may differ significantly in appearance or be located in different areas of the screen.

Menus

Figure 1.4 illustrates a set of menus in the Calendar application. The menu bar is different (but not very different) from menu bars in desktop GUIs.

Cascaded menu items can be used both to hide less common options and to reduce the vertical space required by menu panes. This feature is used sparingly in UIQ applications, where menu content changes with context and each menu is designed to contain as small and as simple a set of menu commands as possible. Cascaded menu items are used more frequently in the S60 interface, where they appear as in Figure 1.5.

In the UIQ emulator, use the menu with the pen (i.e. your PC's mouse) and you'll see some interesting visual cues to confirm the option you



Figure 1.4 Menu structure in the Calendar application



Figure 1.5 S60 cascaded menu items

selected: the option will flash very briefly before the menu disappears. It took a long time to get that effect just right!

As with all other elements of a Symbian OS GUI and applications, where a keyboard is available (including the keyboard of a PC running the emulator), you can drive the menus with the keyboard as well as with the pen. You can use the arrow keys and Enter to select items. You can also use cursor keys and the Confirm button on real target hardware and on emulators that support such features.

When writing an application you also have the option to assign a shortcut key to any menu item, which allows you to invoke the relevant function directly from a keyboard without going through the menus at all. Although they can be defined in any Symbian OS application, shortcut keys are clearly not usable on mobile phones without keyboards (except when the application is running on the emulator) so neither the UIQ nor the S60 user interfaces display shortcut key information in their menus.

1.2 Hello World – Text Version

Now that you've started to get to grips with the emulator, it's time to get your first Symbian OS C++ program running. Even though Symbian OS is primarily a system for developing GUI applications, the simplest kind of program uses a text interface, so for our first task we'll build a program

that writes ‘Hello world!’ to a text console. That will introduce you to the tools required for building applications for both the emulator and a real device, so that later on you’ll be ready for a program with a GUI.

If you want to follow this chapter through at your desktop with the SDK, make sure that you’ve installed all the tools you need. See the appendix for more information.

The Program: **HelloText**

Here’s the program we’re going to build. It’s your first example of Symbian OS C++ source code:

```
// helloworld.cpp

#include <e32base.h>
#include <e32cons.h>

LOCAL_D CConsoleBase* gConsole;

// Real main function
void MainL()
{
    gConsole->Printf(_L("Hello world!\n"));
}

// Console harness
void ConsoleMainL()
{
    // Get a console
    gConsole = Console::NewL(_L("Hello Text"),
                           TSize(KConsFullScreen, KConsFullScreen));
    CleanupStack::PushL(gConsole);

    // Call function
    MainL();

    // Pause before terminating
    User::After(5000000); // 5 second delay

    // Finished with console
    CleanupStack::PopAndDestroy(gConsole);
}

// Cleanup stack harness
GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;
    CTrapCleanup* cleanupStack = CTrapCleanup::New();
    TRAPD(error, ConsoleMainL());
    __ASSERT_ALWAYS(!error, User::Panic(_L("Hello world panic"), error));
    delete cleanupStack;
    __UHEAP_MARKEND;
    return 0;
}
```


Our main purpose here is to understand the Symbian OS tool chain, but while we have the opportunity, there are a few things to observe in the source code above. There are three functions:

- the actual ‘Hello world!’ work is done in `MainL()`
- `ConsoleMainL()` allocates a console and calls `MainL()`
- `E32Main()` allocates a trap harness and then calls `ConsoleMainL()`.

On first sight, this looks odd. Why have three functions to do what most programming systems can achieve in a single line? The answer is simple: real programs, even small ones, aren’t one-liners. So there’s no point in optimizing the system design to deliver a short, sub-minimal program. Instead, Symbian OS is optimized to meet the concerns of real-world programs – in particular, to handle and recover from memory allocation failures with minimal programming overhead. There’s a second reason why this example is longer than you might expect: real programs on a user-friendly machine use a GUI framework rather than a raw console environment. If we want a console, we have to construct it ourselves, along with the error-handling framework that the GUI would have included for us.

Error handling is of fundamental importance in a machine with limited memory and disk resources, such as those for which Symbian OS was designed. Errors are going to happen and you can’t afford not to handle them properly. We’ll explain the error-handling framework and its terminology, such as trap harness, cleanup stack, leave and heap marking, in Chapter 4.

The Symbian OS error-handling framework is easy to use, so the overheads for the programmer are minimal. You might doubt that, judging by this example! After you’ve seen more realistic examples, you’ll have better grounds for making a proper judgement.

Back to `HelloText`. The real work is done in `MainL()`:

```
// Real main function
void MainL()
{
    gConsole->Printf(_L("Hello world!\n"));
}
```

The `printf()` that you would expect to find in a C ‘Hello World’ program has become `Console>Printf()` here. That’s because Symbian OS is object-oriented: `Printf()` is a member of the `CConsoleBase` class.

The `_L` macro turns a C-style string into a Symbian OS-style descriptor. We’ll find out more about descriptors, and a better alternative to the `_L` macro, in Chapter 5.

Symbian OS always starts text programs with the `E32Main()` function. `E32Main()` and `ConsoleMainL()` build two pieces of infrastructure needed by `MainL()`: a cleanup stack and a console. Our code for `E32Main()` is:

```
// Cleanup stack harness
GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;
    CTrapCleanup* cleanupStack = CTrapCleanup::New();
    TRAPD(error, ConsoleMainL());
    __ASSERT_ALWAYS(!error, User::Panic(_L("Hello world panic"), error));
    delete cleanupStack;
    __UHEAP_MARKEND;
    return 0;
}
```

The declaration of `E32Main()` indicates that it is a global function. The `GLDEF_C` macro, which in practice is only used in this context, is defined as an empty macro in `e32def.h`. By marking a function `GLDEF_C`, you show that you have thought about it being exported from the object module. `E32Main()` returns a `TInt` integer. We could have used `int` instead of `TInt`, but since C++ compilers don't guarantee that `int` is a 32-bit signed integer, Symbian OS uses `typedefs` for standard types to guarantee they are the same across all Symbian OS implementations and compilers.

`E32Main()` sets up the error-handling framework. It sets up a cleanup stack and then calls `ConsoleMainL()` under a trap harness. The trap harness catches errors – more precisely, it catches any functions that leave. If you're familiar with exception handling in standard C++ or Java, `TRAP()` is like `try` and `catch` all in one, `User::Leave()` is like `throw`, and a function with `L` at the end of its name is like a function with `throws` in its prototype.

Here's `ConsoleMainL()`:

```
// Console harness
void ConsoleMainL()
{
    // Get a console
    gConsole = Console::NewL(_L("Hello Text"),
                           TSize(KConsFullScreen, KConsFullScreen));
    CleanupStack::PushL(gConsole);

    // Call function
    MainL();

    // Pause before terminating
    User::After(15000000); // 15 second delay

    // Finished with console
    CleanupStack::PopAndDestroy(gConsole);
}
```

This function allocates a console before calling `MainL()` to do the `Printf()` of the 'Hello world!' message. After that, it briefly pauses and then deletes the console again.

If we were creating this example for a target machine that had a keyboard, we could have replaced the delay code:

```
// Pause before terminating
User::After(15000000); // 15 second delay
```

with something like:

```
// Wait for key
console->Printf(_L("[ press any key ]"));
console->Getch(); // Get and ignore character
```

so that the application would wait for a keypress before terminating.

There is no need to trap the call to `MainL()`, because a leave would be handled by the `TRAP()` in `E32Main()`.

The main purpose of the cleanup stack is to prevent memory leaks when a leave occurs. It does this by popping and destroying any object that has been pushed to it. So if `MainL()` leaves, the cleanup stack will ensure that the console is popped and destroyed. If `MainL()` doesn't leave, then the final statement in `ConsoleMainL()` will pop and destroy it anyway.

In fact, in this particular example `MainL()` cannot leave, so the `L` isn't theoretically necessary. But this example is intended to be a starting point for other console-mode programs, including programs that do leave. We've left the `L` there to remind you that it's acceptable for such programs to leave if necessary.

If you're curious, you can browse the headers: `e32base.h` contains some basic classes used in most Symbian OS programs, while `e32cons.h` is used for a console interface and therefore for text-mode programs – it wouldn't be necessary for GUI programs. You can find these headers (along with the headers for all Symbian OS APIs) in `\epoc32\include` on your SDK installation drive.

The Project Specification File

As in all C++ development under Symbian OS, we start by building the project to run under the emulator (that is, for an x86 instruction set) using the Carbide.c++ compiler. We use a debug build so that we can see the symbolic debug information and to get access to some useful memory-leak checking tools. In Chapter 9 we'll build the project for a target Symbian OS phone, using an ARM instruction set. At that stage we'll use the release build, since that is what you would eventually do to create your final, usable, application.

For demonstration purposes we're actually going to build the project twice, because you can either compile the code from the command line or build it in the Carbide.c++ IDE.

Each type of build requires a different project file. To simplify matters, you put all the required information into a single generic *project specification file*, and then use the supplied tools to translate that file into the makefiles or project files for one or more of the possible build environments. Project specification files have a .mmp extension (which stands for 'makmake project'). The one for the HelloText project is as follows:

```
// helloworld.mmp
TARGET      HelloText.exe
TARGETTYPE  exe
SOURCEPATH  .
SOURCE      helloworld.cpp
USERINCLUDE .
SYSTEMINCLUDE \epoc32\include
LIBRARY     euser.lib
```

This is enough information to specify the entire project, enabling configuration files to be created for any platform or environment.

- The TARGET specifies the executable to be generated and the TARGETTYPE confirms that it is an EXE.
- SOURCEPATH specifies the location of the source files for this project.
- SOURCE specifies the single source file, helloworld.cpp (in later projects, we'll see that SOURCE can be used to specify multiple source files).
- USERINCLUDE and SYSTEMINCLUDE specify the directories to be searched for user include files (those included with quotes; the SYSTEMINCLUDE path is searched as well) and system include files (those included with angle brackets; only the SYSTEMINCLUDE path is searched). All Symbian OS projects should specify \epoc32\include for their SYSTEMINCLUDE path.
- LIBRARY specifies libraries to link to – these are the .lib files corresponding to the shared library DLLs whose functions you will be calling at run time. In the case of this very simple program, all we need is the E32 user library, euser.lib.

The Component Definition File

The Symbian OS build tools require one further file, the *component definition file*, to be present. This file always has the name bld.inf and contains a list of all the project definition files (frequently, there is only one) that make up the component. In more complex cases it will usually

contain further build-related information, but the one for `HelloText` is simply:

```
// BLD.INF
PRJ_MMPFILES
hellotext.mmp
```

Building from the Command Line

Once you've typed in the example code for the three files as listed above, we can begin to compile the application.

To start the command-line build, open up a command prompt, change to your installation drive and go to the source directory containing the code for this example. The first stage is to invoke `bldmake` by typing:

```
bldmake bldfiles
```

After a short pause, this command will return. By default, `bldmake` doesn't tell you anything. However, if you check the contents of the directory, you'll notice a new file, `abl1d.bat`, that is used to drive the remainder of the build process. You will also find that there is a new directory in the `\epoc32\build` directory tree, containing a number of generated files which relate to the various types of build that the build tools support.

Next, use `abl1d` to run the rest of the build by typing:

```
abl1d build winscw udeb
```

The `winscw` parameter specifies that we are building for the emulator, using the Carbide.c++ compiler, and the `udeb` parameter means we are using a (unicode) debug build. The command generates the following output:

```
make -r -f "\EPOC32\BUILD\HELLOTEXT\EXPORT.make" EXPORT VERBOSE=-s
Nothing to do
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" MAKEFILE VERBOSE=-s
perl -S makmake.pl -D \HELLOTEXT\HELLOTEXT WINSCW
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" LIBRARY VERBOSE=-s
make -s -r -f "\EPOC32\BUILD\HELLOTEXT\HELLOTEXT\WINSCW\
HELLOTEXT.WINSCW" LIBRARY
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" RESOURCE
CFG=UDEB VERBOSE=-s
make -s -r -f "\EPOC32\BUILD\HELLOTEXT\HELLOTEXT\WINSCW\
HELLOTEXT.WINSCW" RESOURCEUDEB
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" TARGET CFG=UDEB
VERBOSE=-s
make -s -r -f "\EPOC32\BUILD\HELLOTEXT\HELLOTEXT\WINSCW\
HELLOTEXT.WINSCW" UDEB
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" FINAL CFG=UDEB VERBOSE=-s
```

The build is split into six phases:

- The export phase copies exported files to their destinations. This typically includes copying public header files into the `\epoc32\include` directory. For many applications, as in the current case, this stage needs to do nothing.
- The makefile phase creates the necessary makefiles or IDE workspaces.
- The library phase creates import libraries.
- The resource phase creates the application's resources files, bitmaps and information files.
- The target phase creates the application's main executables.
- The final phase is present to perform any final actions that need to be done after the main executables have been created. For most applications, this phase does nothing.

These phases, and other possible options available with the `abld` tool, are fully described in the Build Tools Guide and Build Tools Reference sections of the Developer Library documentation supplied with Symbian OS SDKs. Typing `abld help` also gives a useful summary of the options available.

The result of running the `abld` tool is that the `HelloText` project is built into the emulator startup directory as `\epoc32\release\winscw`



Figure 1.6 HelloText emulator output

`\udeb\hellotext.exe`. To run the program under the UIQ emulator, you can start it right from there, using either the command prompt or Windows Explorer. The emulator will boot and you'll see 'Hello world!' on the screen for a few seconds, as illustrated in Figure 1.6.

On the S60 emulator, the program can be run in the same way, but is hidden behind the application picker display. To bring it to the foreground, once the main display is showing, press the Applications key for a couple of seconds to see the task list. Then (as `HelloText` is the only application you're running) press the Select soft key to show `HelloText` running. But be quick – you've only got 15 seconds!

Using Multiple SDKs

If you've installed more than one SDK, you need to ensure that the tools used when you build an application belong to the SDK that you're currently using. To help with this, there is a `devices` command in Symbian OS SDKs based on v7.0 and later. To list the SDKs which you currently have installed, just type `devices` at the command prompt. The SDK currently in use has 'default' appended to the name. To select a different SDK, type `devices -setdefault @<sdk>`, where `<sdk>` is the full name as copied from the original `devices` list.

Building in the Carbide.c++ IDE

Now we know our tool chain is working, let's build the project from the IDE and use the debugging tools on the example. First, start up the Carbide.c++ IDE and select File, Import and choose the option Symbian MMP File.

Browse to the directory containing the source code and select `hellotext.mmp`. Select the 'Emulator debug' build configuration for the appropriate SDK and then click on Finish; after a short time, the project will have been created.

You can build the project straight away by selecting Project, Build Project – the default target is `WINSCW UDEB`, as we used when building from the command line.

Once the code has been compiled, you can launch the emulator from the IDE by selecting Run, Run `HelloText...`. Don't forget that you'll need to bring `HelloText` to the foreground if you're using the S60 emulator. Alternatively, you can run the debugger on the code by selecting Run, Debug `HelloText...`. You can then use any of the usual debug techniques – run to cursor, step over a whole line of code, step into each of the functions on a line, step out of the current function, run to breakpoint, etc.

If you're curious, you might want to try debugging through line-by-line. You'll begin to get a feel for what's worth doing and what's not, and it will give you some insight into the system structure. On the other hand,

there's no need to jump in this deep right now. We'll explain what you really need to know through the next few chapters. The main point to note is that the Carbide.c++ IDE provides an excellent debugger, and as a Symbian OS developer you can take full advantage of it. We'll look at debugging in more detail in Chapter 10.

Summary

In this chapter, we've not gone very heavily into code, but have instead focused on the tools that come with the SDK and how to use them to build and test a simple project.

The topics we've looked at are:

- how to use the emulator
- a hint as to the support Symbian OS offers to help you code safely
- the basic structure of the project specification (MMP) file
- using the Carbide.c++ IDE and command-line tools
- building and running applications on the emulator.

2

A System Introduction to Symbian OS

Before we can start writing programs for Symbian OS, we need to appreciate the paradigms, idioms and philosophy that make up the architecture of Symbian OS and its C++ frameworks. So far, we've seen the basics of how to build programs for Symbian OS. Now we need some background information in order to understand how to write them.

In this chapter we'll introduce as many concepts as we need for the following chapters, without getting deeply involved in C++ code. All these issues are fundamental for appreciating the system design of Symbian OS and will be essential background as we move to look at areas such as coding conventions, the user library and other basic APIs in the next few chapters.

Since this chapter discusses the fundamentals of the system, the discussion is divided between the operating system basics and the framework principles that encompass them. Even though it is very rare for a domain-specific programming framework to be architected and developed synchronously with its underlying operating system, the two influence each other throughout and it is difficult to appreciate, or even explain, one without the other.

2.1 Symbian OS Basics

Symbian OS is intended to run on open (to third-party installable applications), small, battery-powered portable computers, that is, modern advanced mobile phones. This profoundly affects the design of the Symbian OS software system.

Symbian OS is designed for 32-bit CPUs, running at low speeds relative to those in desktops or workstations. Today's Symbian OS systems are based on XScale, ARM9 and ARM11-based CPUs, while older devices are powered by the ARM7 family of CPUs.

Symbian OS is stored in ROM, where the OS and all the built-in middleware and applications are kept. By comparison, in a PC only a small bootstrap loader and BIOS are built into ROM, with OS and applications loaded from hard disk. On a Symbian OS device, the system ROM is mapped as the Z: drive. Everything in the ROM is accessible, both as a file on Z: and by directly reading the data from ROM. Programs that exist in ROM are usually executed in place rather than being loaded into RAM first and then executed. Symbian OS products use anything between two and several dozen megabytes of ROM.

A mobile phone's system RAM is used by active programs and the system itself, as well as providing 'disk' space, accessed as the C: drive. Since the total RAM on a typical phone is somewhere between 4 MB and 32 MB, there is a real possibility that RAM may be exhausted. Symbian OS has been designed to cope with such situations, and this concern is pervasive throughout the Symbian OS C++ framework.

A typical Symbian OS product can use many I/O devices, including a screen with 'digitizer' for pen input, a keyboard which may be even more compact than those found on laptops (in modern Symbian OS phones, these are increasingly mobile phone-style keyboards), a memory card slot for additional 'disks' accessed as E:, a serial port for RS232, USB, Bluetooth, Wi-Fi, and IrDA.

Symbian OS is different, and there are good reasons why this is so [Mery 2000]. Having said that, most of the paradigms found in Symbian OS can be traced to, or observed in, other pre-existing academic or commercial systems. Nevertheless, it is the first time that an amalgamation of many such good ideas has been crafted into a successful mobile and open commercial system that interacts with so many people every day. Symbian OS did not just happen, it has grown out of more than 25 years' hard work and experience in the domain of battery-powered resource-constrained user-centered computing [Morris 2007, Tasker *et al.* 2000].

About the Kernel

As viewed from the perspective of the user and the applications programmer, a *kernel* manages the machine's hardware resources such as system RAM and hardware devices, while providing mechanisms for other software components to access these resources.

For much application programming it is not necessary to know the detailed mechanics of the underlying OS. In this chapter we give an overview of the kernel and what it does. We won't otherwise be covering how to program the kernel side, but [Sales 2005] provides an excellent grounding in this topic.

Symbian OS has a lightweight 32-bit pre-emptive multi-threaded real-time kernel with a hybrid design combining characteristics from both *microkernel* and monolithic kernel architectures [Tanenbaum and Woodhull 1997, Stallings 2000, Sales 2005].

These days, the Symbian OS microkernel is built as a personality on top of a real-time nanokernel (referred throughout as the nanokernel), (see Figure 2.1) which is responsible for primitives such as fast synchronization, timers, initial interrupt dispatching, and thread scheduling [Sales 2005]. The nanokernel is the core of modern Symbian OS kernel architecture (EKA2) and is a very small Real Time Operating System (RTOS), designed to underlie Symbian OS, and provide low interrupt and thread latency.

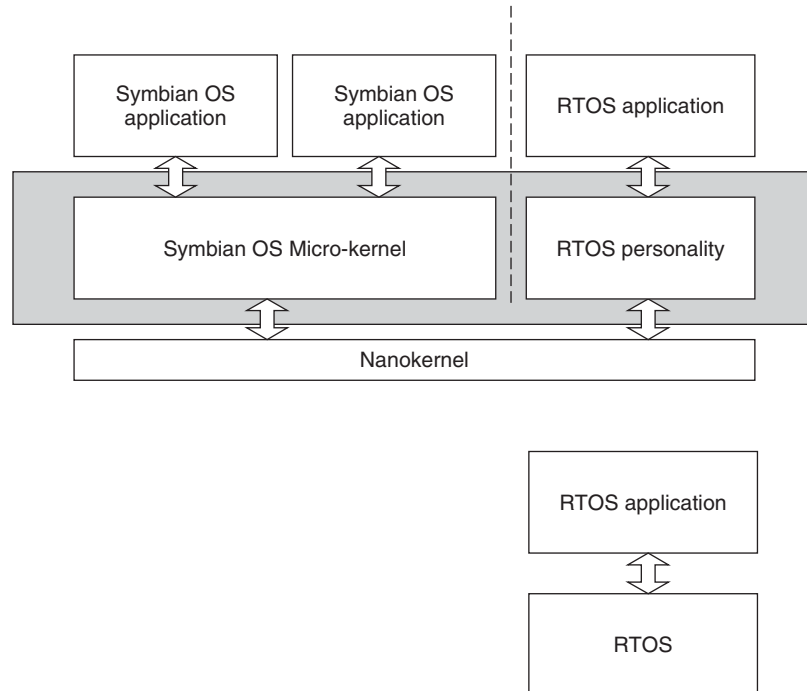


Figure 2.1 Symbian OS kernel architecture

In EKA2, the nanokernel does not deal with any dynamic memory management. Memory management is provided by the Symbian OS microkernel.

For the rest of this book, we will use the term 'kernel' to refer to the combination of nanokernel and microkernel that makes up the Symbian OS kernel, as it appears to applications and unprivileged programs (which are our primary concern).

The design of Symbian OS focuses on open mobile phones and therefore is, above all, optimized for efficient user-driven resource-constrained operation.

From microkernel architectures, the Symbian OS kernel borrows the following characteristics:

- several message-passing frameworks designed for the benefit of user-side servers
- networking and telephony stacks hosted in user-side servers
- file-systems hosted in a user-side server.

In the Symbian OS kernel, as in monolithic kernel architectures:

- device drivers are run kernel-side; not embedded in the kernel binary, though, but implemented as libraries that can be loaded and unloaded at run-time
- the scheduler and scheduling policy are implemented in the kernel
- memory management is implemented in the kernel.

In general, the kernel deals with core hardware resources such as:

- central processing unit (CPU) and memory management unit (MMU)
- memory management policies
- interrupt handling and management
- DMA channel management.

As mentioned above, some of the above responsibilities are shared between the nanokernel and the microkernel.

The microkernel nature of Symbian OS allows many OS services to be offered in the non-privileged mode of the processor. The kernel uses hardware-supported *privilege* to gain access to such hardware resources. That is, the CPU will perform certain privileged instructions only for the kernel. It runs other programs, so-called *user-mode* programs, without privilege, so that they can access system resources only through the kernel APIs.

In this text we refer to the non-privileged mode as *user-side* and to the privileged mode as *kernel-side* (ARM calls this mode *SVC*, for ‘supervisor’).

The boundary between the kernel and all other components is effectively a privilege boundary, which needs to be crossed every time a user-side program needs to communicate with the kernel as well as when one user-side process needs to communicate with another (since such communication is mediated by the kernel in all but exceptional cases).

Crossing from user side to kernel side

In Symbian OS, components never link to the kernel directly (that is, statically), as one would expect in more traditional embedded RTOSs. They have to interface to the kernel through a dynamically linked user library called `euser.dll`. This is always located at a known address, so that calls can be resolved at link/load time for more efficiency.

In `euser.dll` there are three families of exported calls:

- utility classes, methods and common constructs such as descriptors, array classes and math functions
- fast executive calls (system calls that execute with interrupts disabled)
- slow executive calls (system calls that execute with interrupts enabled).

Executive calls are the user library calls that allow a user thread to enter processor-privileged mode so that they can access hardware resources or kernel-space objects in a controlled and predefined way. An executive call switches control to the kernel executive. In that respect, the kernel executive is not a separate thread, but a collection of pre-programmed kernel functions that, while in kernel mode, execute in the context of the thread that calls them.

When programs call these functions, the user library causes a software interrupt (i.e., an ARM `SWI` instruction). This causes the processor to branch to the software interrupt handler routine, as defined at OS startup time. The software interrupt handler checks the type of executive call and branches to the correct kernel function accordingly (for a precise treatment of how this happens in EKA2 refer to [Sales 2005]).

From Symbian OS v8.1b onwards, executive calls can be pre-empted. This means that, while a thread is executing kernel-side (which may be blocking in the kernel), its system call may safely be interrupted. This flexibility is possible because each user thread has two stacks, a user stack for user-side operation and a kernel stack that is used when the thread has crossed to the kernel side. This gives much higher pre-emptibility, leading to much shorter latency during interrupts.

Executive calls may access and even modify kernel-side objects, as well as offer privileged access to hardware. On EKA2, executive calls can create and/or destroy kernel-side objects and, in general, perform allocations or de-allocations on the kernel heap.

Kernel threads

On EKA2, there are by default five threads that run in supervisor mode, as illustrated in Figure 2.2.

- The *Null thread* is the first thread to be scheduled. It has the lowest priority and is, amongst other things, responsible for placing the CPU in low-power mode when there is no other thread executing. This thread is also responsible for de-fragmenting physical RAM from time to time.
- The *Supervisor thread* is primarily responsible for asynchronously cleaning up resources after thread and process termination. This

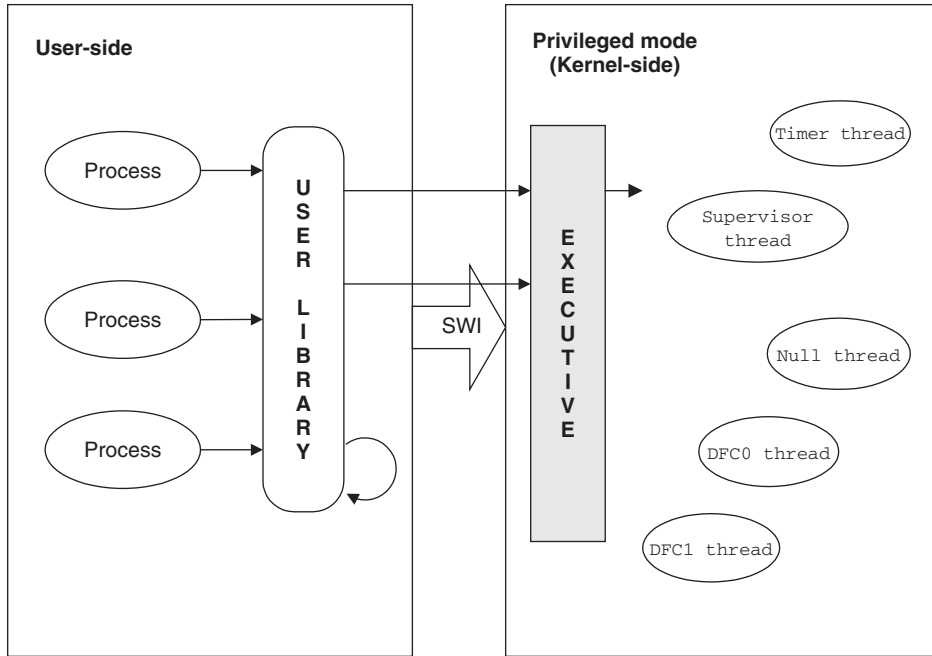


Figure 2.2 User-side processes and the kernel

thread is also responsible for the completion of property subscriptions (see Chapter 8).

- *DFC thread 0* is where most driver Deferred Function Calls execute. DFCs, which form part of driver non-ISR (Interrupt Service Routine) processing, are queued sequentially on this thread. Device drivers without stringent real-time requirements use this thread for their DFCs (for more information, see [Sales 2005]).
- *DFC thread 1*, which by default has the highest priority is usually responsible for running the nanokernel's timer DFCs.
- The *Timer thread* is, by default, used to manage the Symbian OS microkernel time queues.

These five threads run kernel-side by default, but they are not the only threads that can run in supervisor mode. In order to support other microkernel personalities, as well as device drivers and kernel extensions that need to run in supervisor mode, there may be other threads that also run kernel-side.

Threading and Process Model

As discussed already, Symbian OS is a pre-emptive multitasking operating system whose kernel runs in privileged mode, while all other tasks run in

non-privileged mode [Furber 2000]. All access to memory and memory-mapped hardware is protected through use of the MMU by the operating system. Kernel-side code can access all the memory belonging to any program, whereas a non-privileged program can directly access only the memory specifically mapped to it.

In Symbian OS the unit of memory protection is called the *process*, while the unit of execution is the *thread*. A process may contain one or more threads. In the case of the nanokernel, by default there are five threads, as discussed above. What gets scheduled in Symbian OS are the threads – not the processes, which are effectively memory-protection containers.

Memory management ensures that Symbian OS presents a virtual-memory machine model to all running programs. This means that all programs are presented with, and make use of, a linear virtual-memory environment facilitated by the use of the MMU. Virtual memory in this context does not mean that Symbian OS makes use of program swapping (for example, on hard disks), but that all programs during linking, locating and execution appear at the same virtual address; and that they cannot directly access each other's memory.

In general, each program runs as a separate Symbian OS process, within its own virtual address space, so the boundary between one application and another is a process boundary. Since Symbian OS makes use of hardware memory protection, one application cannot accidentally overwrite another's data. As discussed above, when user-side threads need to get privileged access or request kernel-side services, they go through a pre-programmed path, implemented by means of a library.

The thread is the fundamental unit of execution in Symbian OS. A process has one or more threads. Each thread executes independently of the others, but within the same address space. A thread can therefore change memory belonging to another thread in the same process – deliberately or accidentally. In consequence, threads are not as well isolated from each other as are processes.

Threads are pre-emptively scheduled by the Symbian OS kernel. The highest-priority thread that is eligible to run at a given time is run by the kernel. A thread that is ineligible is described as suspended. Threads may suspend themselves in order to wait for events to happen, and resume when one of those events occurs. Whenever threads suspend or resume, the kernel checks which one is the highest ready thread and schedules it for execution. This can result in one thread being scheduled even while another one is running. The consequent interruption of the running thread by the higher-priority thread is called pre-emption, and the possibility of pre-emption gives rise to the term pre-emptive multitasking.

The process of switching execution between one thread and another is known as *context switching*. As in any other operating system, the

scheduler in Symbian OS is written carefully to minimize the overheads involved with context switching.

Nevertheless, context switching is much more expensive in timing terms than, say, a function call. The most expensive type of context switch is between a thread in one process and a thread in another, because a process switch also involves many changes to the MMU settings, and various hardware caches may need to be flushed. It's much cheaper to switch context between two threads in the same process.

User Memory and Memory Management

All details of the system's memory and memory manipulations, including low-level memory management, are encapsulated in a kernel component called the memory model. For example, it is the memory model that implements all necessary operations for context switching, while the policy for doing so is implemented in the kernel scheduler. For further details on the internals of the memory model and its usage, see [Sales 2005].

Every user-side program on Symbian OS is instantiated in a process that has at least one thread, the main thread. When the main thread exits, the process terminates. A process can be a collection of many related threads. For a program to execute, every thread needs a stack and possibly a heap. As mentioned previously, in Symbian OS each user-side thread is assigned two stacks, one for supervisor-mode calls and one for normal execution. By default these stacks are 4 KB and 8 KB respectively.

While stacks are allocated by the kernel (via the memory model) on thread creation, the kernel does not create a heap for user-side threads. In EKA2, user-side threads are responsible for creating their own heaps on creation, or they may share an existing heap (usually the process's main thread heap), since all of the threads in a process share the same address space.

By default, a process's main thread is created with a heap; however, there are interfaces for a process to override the normal local heap creation mechanism and heap allocator functions. This is something that programmers may choose to do if they have specific memory allocation requirements (for example, for games or signal-processing algorithms) that would be better served by a custom memory allocator, rather than the default general-purpose one.

For every process, heaps, stacks, static data and any program instructions that are not run from ROM all need to be placed in RAM. The mechanism by which the Symbian OS memory model allocates and manages memory is called *chunks*. Chunks are containers for physical RAM pages mapped to virtual addresses. In terms of addressing and accessing, a chunk appears as a contiguous virtual memory region. Its apparent size may be larger than the physical memory actually committed for usage, thus not all of the virtual memory region is necessarily accessible.

In order to achieve the deterministic behavior necessary for real-time programming, there is a maximum number of chunks (16 by default) that may be mapped to each process. On initial creation, depending on the hardware used, a process will have all its user-mode stacks as well as its writable static data placed on the same chunk. Protection between the stacks used by different threads is achieved by leaving unmapped pages between adjacent stacks. At least one additional chunk is used for thread heaps, and another for storing any executable code that is not executed in place from ROM. If the process makes use of shared libraries that have writable static data, it is assigned yet another chunk for this data.

Chunks can be shared by being mapped onto more than one process. This can be used as a means of communication without copying data across process boundaries.

Symbian OS Servers

As with other microkernel architectures, a Symbian OS *server* is a program (usually without a user interface) that manages and mediates access to one or more resources or services.

A server provides a programming interface so that its *clients* can gain access to its services. A server's clients may be applications or other servers. Each server generally runs in its own process (although for performance reasons, closely related servers may run in the same process), so that the boundary between a server and its clients is a process boundary. This provides a strong assurance of server integrity. As shown in Figure 2.3, to communicate with a server, its clients need to employ client-server inter-process communication (IPC) mechanisms (see Chapter 8). Such IPC mechanisms are encapsulated in the client APIs of each server.

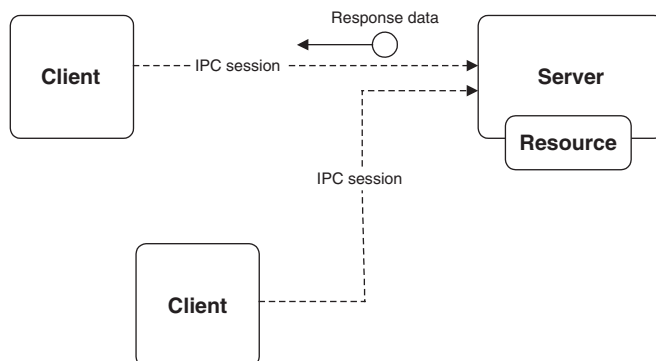


Figure 2.3 Client-server communications

The isolation between a server and its clients is of the same order as the isolation between a kernel and user-mode programs, but servers are much easier to program and work with. Symbian OS uses servers to

provide many services that, on monolithic kernel systems, are provided by the kernel or device drivers [Tanenbaum and Woodhull 1997].

Symbian OS Inter-Process Communication

The channel of communication between a client and a server is known as a *session*. Session-based communication comprises client requests and server responses. Within the kernel, which mediates all messages, such request–response pairs are associated through session objects. A client can have multiple sessions open with any given server if the server supports it. A session can be shared by more than one client thread to minimize the use of kernel resources, again if the server supports it.

Session-based client–server communication is aptly designed for event-driven user-initiated interactions, typically between an application that interacts with the user and a server that serializes many application requests to a resource. This is not the only IPC mechanism available: EKA2 programs may also make use of message queues, shared chunks and publish–subscribe to communicate with each other. Symbian OS IPC mechanisms are discussed in detail in Chapter 8.

Power Management

Power management is the single most important difference between a desktop and a portable system. Battery life, even with rechargeable batteries, makes a difference to how the user uses and thinks of the mobile phone. Symbian OS ensures that power is used efficiently.

Certain parts of the system should still be able to run while the phone is in low power mode and apparently off. For example, when an alarm is due, the phone should turn on so that the alarm can sound.

A mobile phone needs to react to an incoming call or SMS, which would not happen if it were fully turned off; the system should be able to respond by changing from low-power to active mode. Even if all power is removed suddenly, the system should do what it can to save critical information, so that there is a possibility of a warm boot rather than a cold boot when power returns.

As an application or server programmer, your task is easier than that of the kernel-side programmer, but you still have to be aware of power management issues. You have to write your programs efficiently, to make best use of the phone’s scarce resources. For example, as with all portable battery-powered devices, polling is considered a cardinal sin.

Power management becomes more complicated the deeper you delve into the system. For a device driver programmer, power management is more complex than simply having the phone on, off or in sleep mode. Even though the user may think the phone is off if the display is turned off, it may be processing or handling events behind the scenes.

Each hardware component should be responsible for its own power management. For example, a communications link driver should turn the physical device off if it's not needed. At times, the kernel will turn parts or even the whole of the CPU off, if all threads are waiting for an event. At the same time, every possible step must be taken to ensure that user data is retained, even in the most difficult power-loss situations.

Timers

Symbian OS timers are maintained on two levels, at nanokernel and microkernel level. Application developers are principally interested in the timer services provided by the microkernel, such as those offered by the `RTimer` class. As stated earlier, the Timer thread is the kernel thread responsible for managing timer queues and alarms, for example when requesting an alarm by means of a call to `RTimer::At()`.

Executables and Shared Libraries

In general, when a program is about to be executed by an operating system, it is loaded into memory (RAM). For the OS to be able to understand the executable image of a program, it must be packaged in a specific format known to the OS loader. On Symbian OS EKA2, this format is based on the ELF (Executable & Link Format) standard.

Different parts of the program are mapped to different parts of the memory which the operating system allocates for use by the process. An executable image must package such program information accordingly.

- The program's processor instructions are stored together in what is usually referred to as a text section. This section is used only to read the program instructions from.
- Data, such as constants and initialized global variables, is stored in the data section.
- In many cases, read-only data is stored separately from read/write data, in its own read-only data section.

Symbian OS programs come packaged in two forms, either in the so-called executable form (as EXE files) or in the shared library form. As the name implies, shared libraries are packaged so that a number of other executables, including other libraries, can reuse them concurrently, thus saving space at run time. EXEs are programs that are loaded and executed themselves, whereas shared libraries are only executed when used (also called linked) by other programs. Libraries are collections of functionality that is sufficiently generic to be reusable by other programs.

Libraries can be shared either at run time or at compile and link time (in the form of either source code or binary objects). It is very rare on Symbian

OS to find non-shared libraries, because they are not memory-efficient. Nevertheless, such libraries (also called static libraries) do exist.

Shared libraries are also referred to as DLLs (dynamically linked libraries). In Symbian OS these libraries come in two forms, the polymorphic interface form and the static interface form.

So, to clarify, there are two kinds of library:

- statically linked libraries, which usually come in one of two forms:
 - source code which compiles with your code, thus becoming part of your binary and not shared with other executables.
 - binary object (`.o` or `.obj`) files which, at link time, are combined in one monolithic executable with your program.
- dynamically linked, shared libraries, which also usually come in one of two forms:
 - static-interface, which typically offers a wide and fixed (i.e. static) API that can be shared by many other libraries or executables at run time, for example the Symbian OS system library called `euser.dll`
 - polymorphic-interface, which in most cases are plug-ins to some framework. Unlike the static interface form, which is normally loaded by the system, the executable that uses such a library has the responsibility of loading it.

Polymorphic-interface libraries adhere to a published interface, which typically uses a single factory export, declared using the macro `EXPORT_C` in Symbian OS C++. This means that it is relatively easy for many providers to implement the relevant services, and for users to select and load a particular implementation. For example, `ESock`, the Symbian OS sockets server, loads many protocol modules (`ESock` plug-ins) which all have the same factory method and thus provide the same `EXPORT_C` factory method for a framework to load (see Chapter 19).

The Symbian OS library exports are significant because the design of Symbian OS imposes a constraint that is, in most cases, alien to developers coming from other open-system platforms, such as PCs. With Symbian OS, linking is done using the ordinal position of the binary's exported function, rather than using the name of the exported function. This is achieved during the post-processing stage of the ELF binaries, where all symbols referring to the name of a function are removed. By doing so, the final image of an executable is trimmed and thus becomes more space-efficient. It does, however, mean that at run time the operating system cannot associate by name a caller's request for a particular function to the shared library that supplies it. This necessitates the link-by-ordinal constraint, and the consequential need to maintain the

public (mostly static-interface) shared-library function ordinals consistent between builds.

Writable Static Data Optimization

Another optimization that is recommended, but no longer enforced, on Symbian OS is that of not allowing writable static data in DLLs.

Every DLL that supports writable static data requires a separate chunk to be allocated in every process that uses the DLL. There are hundreds of DLLs in Symbian OS, and a typical application uses perhaps 60 of them. A user may have some 20 applications running concurrently, and around 10 system servers working on behalf of those applications.

The smallest unit of physical memory allocation in conventional MMUs is 4 KB (and a smaller amount wouldn't be sensible), so each use of a shared library that has even a single word of writable static needs at least an additional 4 KB chunk. In the above example, if each DLL used writable static data, the total additional memory usage would be 4 KB x (20 app processes + 10 server processes) x 60 DLLs each, which is about 7 MB of RAM, just for the writable static data!

If you have about 20 applications running on a personal computer, 7 MB isn't unacceptable – most of it is paged out to disk anyway – but for a Symbian OS device, with limited RAM and no swapping to secondary storage, this is quite an overhead.

Therefore, when developing shared libraries, we need to be aware that if we use writable static data, all the processes that link to the DLL will be taxed with an extra chunk that will occupy at least 4 KB. If you combine this with the fact that a process may not own more than 16 chunks, you can see that this can easily become problematic.

Hence it is recommended that shared library designs do not use writable static data, which translates to not using non-const global objects.

Files and the File Server

Files in Symbian OS are accessed and made available to programs via calls to the file server, called F32. Due to data caging (see Chapter 9), not all files are accessible to all programs for security reasons.

Platform Security

In the Platform Security environment, servers police specific client requests to ensure that the client's permitted capabilities are not exceeded. For example, if a client requests the ETel server to make a phone call, the server checks that the client has been granted the phone network capability. Framework classes are provided to make this checking simpler.

A server typically offers many functions to its clients, each of which can require different capabilities or none. The capabilities required may also

differ according to the data passed to the server function. For example, the file server's open file function does not require any capabilities if the client is opening its own files, but requires a system capability in order to open a system file.

The kernel adds client capabilities to the IPC message as it transfers the call from the client to the server side. It would be pointless to pass capabilities to the server directly from the client, as the server doesn't know whether or not the client is well-behaved.

2.2 Framework Basics

Symbian OS C++

The term 'Symbian OS C++' refers to the domain-specific C++ dialect and the accompanying frameworks, used to build Symbian OS and the software that runs on it.

Symbian OS C++ is 'different', using its own dialect of C++, and the reason is twofold. First, it is an accepted fact that, for a variety of reasons, C++ tends to differ from platform to platform [Henney 2001]. Second, and more importantly, this dialect, as well as the OS itself, has emerged from a vast amount of experience, starting more than twenty years ago at Psion, in the domain of small, resource-constrained, battery-powered portable and user-friendly computers. This experience has been captured in the domain-specific idioms and paradigms of Symbian OS and Symbian OS C++.

Symbian OS and Symbian OS C++ have, during their development, exerted a mutual influence in each other: some of the idioms and paradigms present in Symbian OS are reflected in Symbian OS C++, and vice versa.

Concurrency and Active Objects

Active objects (AOs) are the Symbian OS C++ paradigm for implementing concurrency within the environment of C++, without taxing the programmer with the difficulties of thread management and synchronization. Chapter 6 deals with active objects in more detail. This section places them in the context of the operating system and its frameworks.

Figure 2.4 illustrates a typical situation, where an application is making requests to a variety of system-supplied servers.

Since the C++ language does not support concurrency, Symbian OS C++ has to supply a model to embed concurrency. Unlike other systems, it allows for both pre-emptive multitasking, using threads, and non-pre-emptive multi-tasking within a thread, using active objects.

Programmers can use AOs in their applications both to request transactions to be serviced by microkernel servers (such as the window server) and to handle their completions, whether successful or not. This model

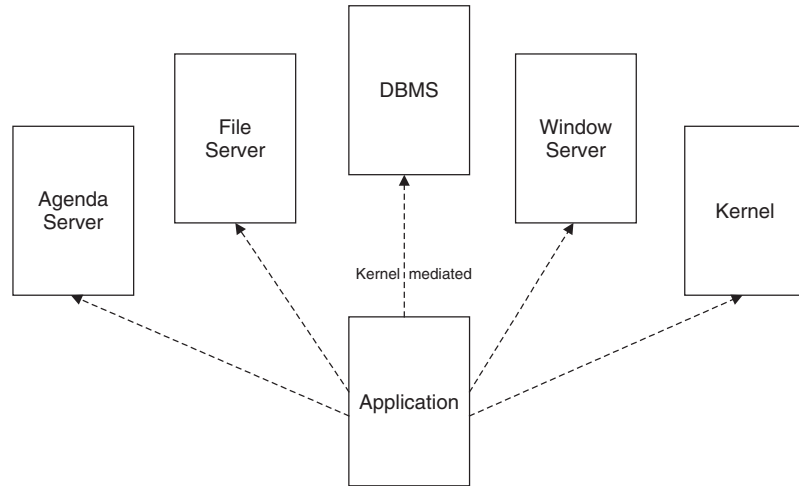


Figure 2.4 Typical concurrent processes

fits very well with the traditionally event-driven nature of Symbian OS, where the user is interacting with the system and these interactions flow from the user to the system and back.

In a monolithic and single-threaded OS, the programmer usually spawns threads in order to handle many events or requests asynchronously. In a microkernel OS such as Symbian OS, the concurrency of the servers hosting the requested services gives that asynchronicity for free. Within the single thread of a Symbian OS application, the programmer only needs to distribute requests to servers running concurrently in separate threads.

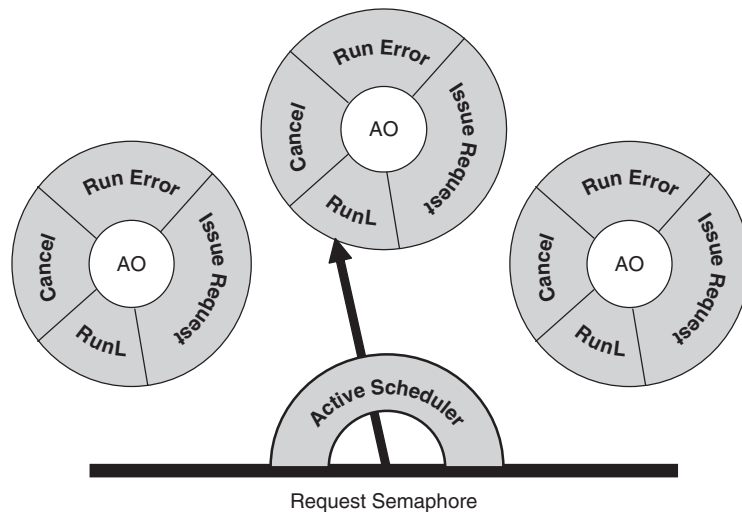


Figure 2.5 Scheduling of active objects

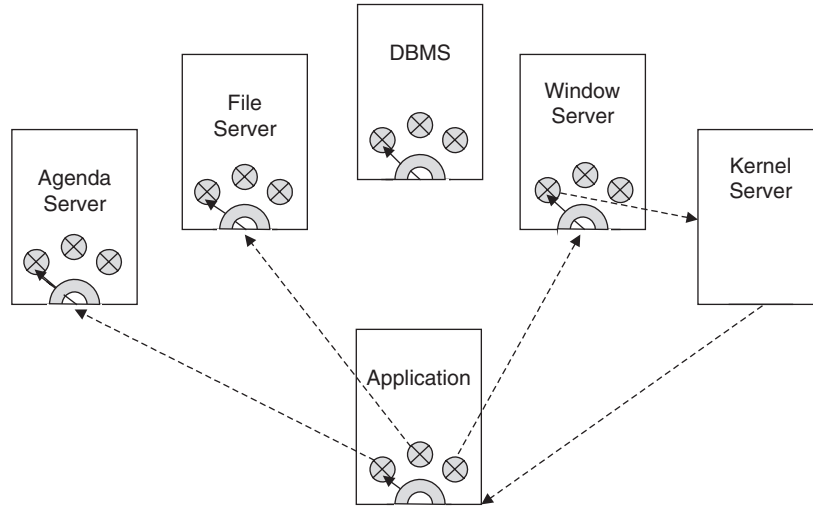


Figure 2.6 Active objects in concurrent processes

Active objects fit very naturally into such an environment. As illustrated in Figure 2.5, all Symbian OS active objects run under the control of the Active Scheduler, which is an AO queue management loop that waits on a special thread semaphore.

In general, the application and the servers with which it communicates will each own an active scheduler and a collection of active objects. Figure 2.6 illustrates this, and shows how active objects are used to implement the interactions of Figure 2.4.

In some circumstances a client may need to handle a service synchronously, and Symbian OS provides that functionality. However, the vast majority of services are handled asynchronously, with the client using the active object mechanism both to make a request and to handle the server's later request completion.

Event handling with active objects

An active object is normally responsible for asynchronously handling requests to one specific service provider. Each active object has a virtual member function called `RunL()`, which it must implement, as this is the function that gets called when the object's event happens. The implementation usually starts with some pre-processing to analyze the event. It may complete the handling of the event without calling any other functions, but in a framework `RunL()` usually calls one or more virtual functions that the programmer implements to provide specific behavior.

The most important frameworks are for GUI applications and servers. An application such as the one in our example in Figure 2.7 uses the

GUI framework. This framework analyzes input events, associates them with the correct control, and then calls virtual member functions, such as `OfferKeyEventL()` to handle a key press.

A server, for example the window server, uses the server framework to handle requests from client applications – including requests to draw on the screen, or to be notified about key events. Client requests are turned into messages, which are sent to the server. The server framework analyzes these messages, associates them with the correct client, and handles the client's request by calling `ServiceL()` on the server-side object that represents the client.

A server also uses its own active objects to handle events other than client requests – for instance, key events from the kernel.

Event-handling example

If a Symbian OS application is running and a key is pressed, a small cascade of events occurs. These events are handled by at least three threads, as illustrated in Figure 2.7.

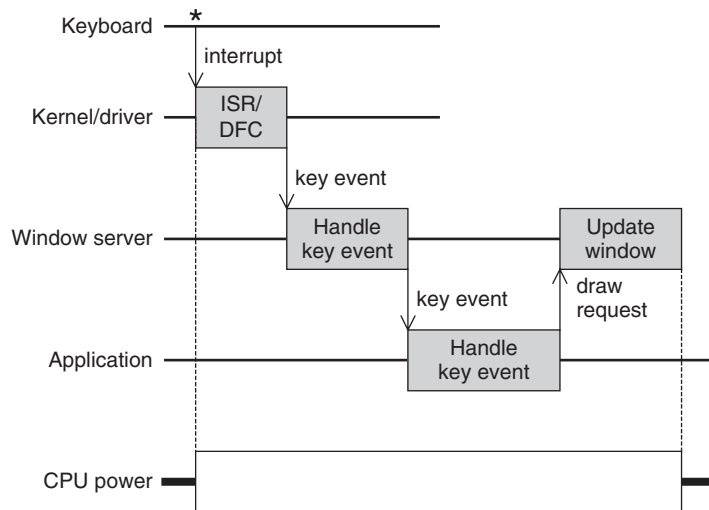


Figure 2.7 Events triggered by a key press

Let's look at this cascade from a couple of perspectives. Firstly, from the whole-system point of view:

- The I/O device responsible for looking after the keyboard generates an interrupt.
- The Interrupt Service Routine (ISR) translates this to an event. It interrogates the device, works out what ASCII key code to assign, and creates an event for whichever program is interested in raw key events – in any real Symbian OS system, that is the window server.

- The window server then works out which application is currently receiving keystrokes, and sends the event to the application.
- The Symbian OS application then handles the key – perhaps by adding text to the document, then updating the display.
- The window server updates the display in response to the application's request.

From the power-management point of view, power is needed for the CPU only while it's doing something. Power is turned on to handle an interrupt, and turned off again when no more threads are eligible to run.

You can also look at the tasks in the diagram and ask, 'What other events might this task have to handle?'

- The keyboard driver handles an interrupt, does minimal processing, and notifies a user-mode thread – in this case, the window server. The keyboard driver must also handle requests from the window server for key events. So the keyboard driver is an event-handling task that handles two types of event: requests from a user-mode thread and hardware events from the keyboard.
- The window server handles the key, does enough processing to identify the application that is currently taking keys, and then notifies the application. The window server, like the keyboard driver, also handles requests from the application for key presses, and the window server also performs screen drawing on behalf of all applications. So the window server is an event-handling task that handles these three event types (key events, requests to be notified about key events, and screen drawing) plus many more (for example, pointer events and requests to be notified about them).
- The application is an event-handling task that handles key events as well as others (for example, pointer events).

Symbian OS C++ Exception Handling

Native Symbian OS C++ programs use an exception-handling mechanism which is not compatible with the ISO C++ 0x exception-handling scheme (the design of Symbian OS predates ISO/IEC 14882 1998/2003).

The Symbian OS C++ exception-handling philosophy is to be 'in your face'. Symbian OS C++ programmers have to understand, handle and implement trapping, throwing and heap-allocated object tracking as part of their program's design, and not as an afterthought as is often the case in other frameworks.

Symbian did not implement exceptions in the 'standard way' for historic and efficiency reasons. When Symbian OS was being born, support in both free and commercial compilers for 'standard' C++ exceptions was

poor or, in many cases, absent. This was in 1990, when Bjarne Stroustrup called support for exceptions ‘experimental’ [BSI 2005]; C++ was first standardized only in 1998. In addition, the support that did exist for handling exceptions was inefficient and produced overly large binaries, a serious problem for the ROM-based products of the early nineties.

Symbian OS C++ makes use of the cleanup stack in order to keep track of heap-allocated objects in the face of exceptions. The cleanup stack was briefly introduced in Chapter 1 without much explanation; this chapter provides an overview of its purpose and use, and Chapter 4 covers the subject in depth.

Here is a brief example using the cleanup stack:

```
SomeMethodL()
{
    tmpObject = new (ELeave) CHeapObject(); // Note overloaded new
    CleanupStack::PushL(tmpObject);        //track object

    DoSomethingThatMayLeaveL(); // this call may throw an exception

    CleanupStack::Pop(tmpObject); // we know an exception was
                                // not thrown and we may remove
                                // the object from the stack

    ...
    // ...and pass ownership of that object
    // OR possibly even do the following instead
    CleanupStack::PopAndDestroy(tmpObject);
}
```

So how does the cleanup stack work?

- It stores pointers to objects that must be destroyed if an exception (a.k.a. Leave) occurs.
- These pointers are stored in nested levels.
- Each such level is marked by the TRAP macro, which functions rather like a C setjump/longjump with executive call.
- When a Leave occurs, the stack calls all destructors (and cleanup items) of objects belonging to the corresponding TRAP level.
- Also, when a Leave occurs, the stack unwinds to the point marked by the corresponding TRAP, returning an exception error code.

An exception is thrown in Symbian OS C++ by a call to `User::Leave()`, and caught by means of the TRAP macro.

```
CleanupStack::PushL(something); // will stay on the CleanupStack if
                                // it Leaves below
TRAPD(err, SomeMethodL());
if (err == KErrNone)
```

```
{
    //do something
}
else
{
    // start handling the exception
    // and possibly propagate it by another Leave
    User::Leave(err);
}
// possibly pop something
```

As is evident from the code snippet above, Symbian OS C++ propagates the exception context in the form of integers (in this case `err`), which differs from how standard ISO C++ works. Moreover, as in Objective-C, the cleanup stack mechanism does not cater by default for stack-allocated objects, whereas in ISO C++ all stack-allocated object destructors are called when an exception is thrown. This latter point, although an apparent limitation, makes throwing of exceptions much more efficient. On Symbian OS, this was included by design, because the stack is quite small (and doesn't grow dynamically). Thus, in Symbian OS, stack allocation of complex objects that own other resources, and hence need destructors, is in most cases not recommended.

Symbian OS C++ Naming Conventions

Because of the inefficiency of exceptions handling, and the need to be in complete control of a C++ object's lifecycle, Symbian chose a C++ naming standard that expresses both object ownership and exception-throwing (leaving) behavior. This standard is based on the Taligent coding standard [Taligent 1994].

In Symbian OS C++, the presence or absence of a trailing `L` in a method name tells you whether this method may leave (throw an exception) or not. This paradigm has immense value because it explicitly communicates to the user of a method the need to guard against any exceptions, and thus to keep track of objects that may need to clean up their resources. Similarly, starting identifier names with a prefix such as `i` or `a` communicates the scope of the corresponding objects to the programmer.

The coding standard is discussed in detail in Chapter 3.

2.3 APIs Covered in this Book

Symbian OS APIs are divided into categories that correspond to the different types of program:

- the kernel exposes an API through the executive calls of the user library

- system servers expose APIs through their client interfaces
- application engines expose APIs to the applications that use them
- middleware components are APIs in perhaps the purest and simplest sense
- other API types, such as device drivers, sockets protocol implementations, and printer drivers, are associated with particular system components.

These divide into several broad groupings:

Group	Description
Base	Provides the fundamental APIs for all of Symbian OS.
Middleware	Graphics, data, and other components to support the GUI, engines, and applications.
UI	The system GUI framework.
Applications	Application software can be divided into GUI parts (which use UIQ) and engines (which don't deal with graphics). Some applications are simply thin layers over middleware components; others have substantial engines.
Communications	Industry-standard communications protocols for serial and sockets-based communication, dial-up networking, TCP/IP, and infrared.
Language systems	The Java run-time environment.
Symbian OS Connect	Communications protocols to connect to a PC, and services such as file format conversion, data synchronization for contacts, schedule entries and e-mail, clipboard synchronization, and printing to a PC-based printer.

The following table shows the main C++ APIs that we will be covering in this book, and introduces the naming conventions that we use. The names include:

- a friendly title, which we'll normally use in the book unless there is a need to be more precise
- the shared library name: add `.dll` to this for the name of the DLL to use at run time, or add `.lib` for the name of the import library to specify in your MMP file

- the top-level project name in the source tree: this is the main system used internally by Symbian to refer to APIs, and usually corresponds to the shared library name. This is not always the case, since some projects produce more than one library, while others produce none at all.

For good measure the table includes a group category (base, middle-ware etc.) for each API.

The corresponding header files, with the names listed in the table, are located in the `\epoc32\include` directory. The Symbian OS C standard library is an exception to this rule, as its header files are isolated in their own directory, `\epoc32\include\libc`.

Title	DLL	Source	Group	Headers
User library	euser	E32	Base	e32def.h, e32std.h, e32base.h, e32*.h Utility and kernel-object APIs. See Chapter 4 for resource cleanup, Chapter 5 for strings and descriptors, Chapter 6 for active objects, and Chapter 8 for the client–server framework.
File server	efsrv	F32	Base	F32file.h File and device management. See Chapter 7.
GDI	gdi	GDI	Middleware	gdi.h Abstract graphical device interface. See Chapter 12 for an introduction to drawing, and Chapters 17 and 18 for other facilities, with the emphasis on device independence.
Window server	ws32	WSERV	Middleware	w32std.h Shares screen, keyboard and pointer between all applications. See Chapters 17 and 18.
CONE	cone	CONE	Middleware	coe*.h Control environment: works with window server to enable applications to use controls. See Chapters 17 and 18.
Stream store	estor	STORE	Middleware	s32*.h Stream and store framework and main implementations. See Chapter 7.

Title	DLL	Source	Group	Headers
Resource files	baf1	BAFL	Middleware	ba*.h Once grandly titled 'basic application framework library', its most useful aspect is resource files, though it also contains other APIs. See Chapter 13.
Application architecture	apparc	APPARC	Middleware	apa*.h Governs file formats and application launching. See Chapters 9 and 11.
Qikon/Avkon and Uikon	qik*/ avk* uik*	QIKON/ AVKON, UIKON and others	UI	qik*.h/ akn*.h eik*.h The system GUI. Qikon and Avkon provide UIQ-specific and S60-specific layers over Uikon.
Sockets server	esock	ESOCK	Comms	es_*.h Sockets-based communications using protocols such as TCP/IP, infrared and others. See Chapter 19.
Telephony server	etel	ETEL	Comms	etel*.h Voice, data, address book etc. on landline or mobile phones and modems. See Chapter 20.

While this book gives you a head start on the main APIs, it cannot provide detailed information on every Symbian OS API. The SDKs contain much additional valuable information and advice.

Summary

In this chapter we discussed the fundamentals of Symbian OS, covering the basic principles of both the operating system and the framework that encompasses it.

We looked at the distinct features that the Symbian OS EKA2 architecture borrows from microkernel, nanokernel and monolithic kernel philosophies. We also described the multi-threaded nature of the Symbian OS kernel and how programs access it by means of executive calls.

We examined the Symbian OS process model and memory management, and how Symbian OS servers communicate with clients through IPC. In addition, we discussed shared libraries and their distinction, based on the interface they provide.

Then we introduced a major Symbian OS C++ paradigm, that of active objects, and how concurrency is introduced in the C++ environment. We also looked at one of the most important idioms, that of Symbian OS C++ exceptions handling.

Finally we summarized the main APIs that are covered in this book, showing the broad groupings into which they fall.

3

Symbian OS C++

In the previous two chapters, we have looked at the fundamental conceptual elements of Symbian OS. The purpose of this chapter is to take a look at how Symbian OS approaches aspects of design and programming in C++.

The fundamental design decisions of Symbian OS were taken in 1994–5 and its toolchain for emulator and ARM builds was essentially stable by early 1996. Since then the compilers have changed, but the coding style still bears the hallmarks of the early GCC and Microsoft Visual C++ compilers.

3.1 Fundamental Data Types

Let's start with the basic types. `e32def.h` (in `\epoc32\include`) contains definitions for 8-, 16- and 32-bit integers and some other basic types that map onto underlying C++ types such as `unsigned int`, which are guaranteed to be the same regardless of the C++ implementation used (see Table 3.1).

For integers, use `TInt` unless you have good reason not to. Use unsigned integer types only for flags, or if you know exactly what you're doing with unsigned types in C++. Use specific integer widths when exchanging with external formats or when space optimization is paramount. `TInt64` is also available; it is a class defined in `e32std.h`, rather than a `typedef`. There is no `TUInt64`.

Do not use floating-point types unless you have to. Machines running Symbian OS do not usually include hardware floating-point units, so floating-point operations are much slower than integer operations. Most routine calculations in Symbian OS GUI or communications programs

Table 3.1 Data types.

Related types	Description
TInt8, TUInt8	Signed and unsigned 8-bit integers
TInt16, TUInt16	Signed and unsigned 16-bit integers
TInt32, TUInt32	Signed and unsigned 32-bit integers
TInt, TUInt	Signed and unsigned integers: in practice, this means a 32-bit integer
TReal32, TReal64, TReal	Single- and double-precision IEEE 754 floating-point numbers (equates to float and double); TReal equates to TReal64
TText8, TText16	Narrow (ASCII) and wide (Unicode) characters (equates to unsigned char and unsigned short int)
TBool	Boolean – equates to int due to the early compilers used; some code depends on this so it has not been changed with the new compilers
TAny	Equates to void and usually used as TAny* (a ‘pointer to anything’)

can be done by using integers. If you’re using floating-point numbers, use TReal for routine scientific calculations: conventional wisdom has it that TReal32 isn’t precise enough for serious use. Only use TReal32 when speed is of the essence and you know that it’s sufficiently precise for your problem domain.

Use TBool to specify a Boolean return value from a function, rather than TInt. This conveys more information to anyone trying to read your code. To represent Boolean values, do not use the TRUE and FALSE constants that are defined for historical reasons in e32def.h; rather, use ETrue and EFalse defined in e32std.h. Be aware that ETrue is mapped to 1 but C++ interprets any integral value as ‘true’ if it is non-zero, so never compare a value with ETrue:

```
TBool b = something();
if (b == ETrue)           // Bad!
```

Instead, just rely on the interpretation of Booleans in C++:

```
if (b) { ... };
```

And now, the most important rule of all:

Always use the Symbian OS typedefs rather than native C++ types, to preserve compiler independence.

The one exception to this rule is related to the C++ `void`, which can mean either ‘nothing’ (as in `void Foo()`) or ‘anything at all’ (as in `void* p`). Symbian C++ uses `void` for the ‘nothing’ case and `TAny*` for the ‘pointer to anything’ case:

```
void Foo();           // Returns no result
TAny* p;              // A pointer to anything
```

Fundamental types also include characters and text. We cover them in Chapter 5, along with descriptors, which are the Symbian OS version of strings.

3.2 Naming Conventions

Just like any system, Symbian OS uses naming conventions to indicate what is important and to make source code more readable. Symbian OS source code and all SDKs adhere to these conventions.

Class Names

A class name should be a noun, as a class represents an object. Class names use an initial letter to indicate the basic properties of the class; the main ones are shown in Table 3.2.

Some other prefixes are occasionally used for classes in rare circumstances. The only one we’ll encounter in this book is `HBufC`, for heap-based descriptors. Kernel-side programming uses `D` for kernel-side `CBase`-derived classes and `C`-style `structs` without any member variables use `S`. However, there are very few `C`-style `structs` in the later versions of Symbian OS – the trend is to replace them with `T` classes.

The distinction between `T`, `C` and `R` is very important in relation to cleanup and it is discussed in further detail in Chapter 4. In general, `T` classes do not require cleanup because they allocate no resources; `R` classes acquire resources that need to be closed or released; and `C` classes need to be deleted.

One thing we need to be honest about here is that some `M` classes are not pure interfaces, because some of them actually implement code. Nevertheless, the majority of these types are pure interfaces.

Table 3.2 Types of Class.

Prefix	Category	Examples	Description
T	Type	TDesC, TPoint, TFileName	T classes do not have a destructor; they act like built-in types. That's why the typedefs for all built-in types begin with T. T classes can be allocated automatically (if they're not too big), as members of other classes or on the heap.
C	Class	CActive, CBase	Any class derived from CBase. C classes are always allocated on the default heap. CBase's operator new() initializes all member data to zero when an object is allocated. CBase also includes a virtual destructor, so that by calling delete() on a CBase* pointer any C object it points to is properly destroyed.
R	Resource	RFile, RTimer, RWriteStream, RWindow	Any class that owns resources other than on the default heap is an R class. They are usually allocated as member or automatic variables. In a few cases, they can be allocated on the default heap. Most R classes use Close() to free their associated resources.
M	Mixin, interface	MGraphicsDeviceMap, MEikMenuObserver	An M class is an interface consisting of virtual functions. A class implementing this interface should derive from it. M classes are the only approved use of multiple inheritance in Symbian OS: they act in the same way as interfaces in Java. The old technical term was 'mixin', hence the use of M.
	Static class	User, Math, Mem, ConeUtils	A class consisting purely of static functions cannot be instantiated into an object. Such classes are useful containers of library functions.

Data Names

A data name should be a noun, as it represents an object. Data names, except automatic variables, also use a prefix letter to indicate their purpose (see Table 3.3).

Static members aren't used in native Symbian OS code (see the more general discussion of writable static data in Chapter 2). Global

Table 3.3 Data name prefixes.

Prefix	Category	Examples	Description
E	Enumerated constant	EMonday, ESolidBrush	Constants or values in an enumeration. If it has a name, the enumeration itself should have a T prefix, so that EMonday is a member of TDayOfWeek. Some #define d constants use an E prefix, in circumstances where the constants belong to a logically distinct set.
K	Constant	KMaxFileName, KRgbWhite	Constants of the #define or const TInt type. KMax -type constants tend to be associated with length or size limits: KMaxFileName, for instance, is 256 (characters).
i	Member variable	iDevice, iX	Any non-static member variable should have an i prefix. The i refers to an ‘instance’ of a class.
a	Argument	aDevice, aX	Any variable declared as an argument. The a stands for ‘argument’, not the English indefinite article so it is important to remember not to use an for words that begin with a vowel (i.e., aOrigin not anOrigin).
	Automatic variable	device, x	Any variable declared as an automatic (a variable that is created automatically when it is required and destroyed when it goes out of scope)

variables, such as `console`, have no prefix but are generally considered bad practice and are best avoided altogether. Some authors use initial capitals for global variables, to distinguish them from automatics; other authors use an initial lower case `g`. We have no strong views on the right way to do things here. This is mostly because in Symbian OS we prefer to avoid the issue by avoiding the use of global variables altogether.

The `i` convention is important for cleanup. The C++ destructor takes care of member variables, so you can spot over-zealous cleanup code, such as `CleanupStack::PushL(iMember)` by using this naming convention.

Function Names

As a matter of convention, you would normally use a verb for a function's name (as it represents an action). In addition, the initial letter should be uppercase: `Draw()`, `Intersects()`.

A leaving function should end with `L`: `CreateL()`, `AllocL()`, `NewL()`, `RunL()`. A leaving function is essentially used for light-weight exception handling. It may need to allocate memory, open a file or do some other operation that might fail (because there are insufficient resources or other environment-related conditions – not programmer errors). When you call a leaving function, you must always consider what happens both when it succeeds and when it leaves; you must ensure that both cases are handled. The cleanup framework in Symbian OS is designed to allow you to do this.

An `LC` function (`AllocLC()`, `CreateLC()`, `OpenLC()`, `NewLC()`) leaves something on the cleanup stack when it returns. Commonly, this is a pointer to an allocated object, but more generally it's a cleanup item. If the function fails, then it leaves.

Simple 'getter' functions get some property or member data of an object. Often a getter is used when the member is private. Its name should be a noun, corresponding with the member name: `Size()`, `Device()`, `ComponentControl()`.

Complex 'getter' functions get some property that requires more work and perhaps even resource allocation. Resource-allocating getters should certainly use `Get` as a prefix (`GetTextL()`); other than that the boundary between simple and complex getters is not fixed.

'Setter' functions set some property of an object: `SetSize()`, `SetDevice()`, `SetCommandHandler()`, `SetCharFormatL()`. Some setters simply set a member; some involve resource allocation, which may fail, and are therefore also `L` functions.

Macro Names

Symbian OS uses the usual conventions for C preprocessor macro names: upper-case letters and words joined with underscores, for example, `IMPORT_C` and `EXPORT_C`.

For build-dependent symbols, use two leading and trailing underscores (`__SYMBIAN32__`, `__WINS32__`). The symbols `_DEBUG` and `_UNICODE` are notable exceptions to this rule. Double underscores are also used to indicate guards.

Indentation, Line Breaks and Bracketing

In addition to the naming rules presented so far, there are a number of style rules that are intended to improve readability by ensuring some degree of conformity in the appearance of the source code. Please note that the

rules presented in this section make more sense when put together rather than when considered individually. Therefore, it makes little sense to try to memorize them all. They are given here for reference; the best way to get used to them is by day-to-day usage and by looking at the examples throughout this book (although this section contains an example that tries to include as many of them as possible).

- Use a new line when opening a curly bracket.
- The opening bracket should occupy a new line with nothing else on it and be indented by one tab space in comparison to the previous line. Throughout this book, because of the narrow line width, we have indented by two spaces rather than a full tab.
- Any code within curly brackets must be at the same indentation level as the brackets. (An exception to this is the access qualifiers for members in a class declaration.)
- The closing bracket must be at the same level of indentation as the opening bracket and must occupy a line on its own. (An exception to this is the closing bracket in a class declaration, which may be immediately preceded by its mandatory semicolon on the same line.)

In addition, the following rules apply for line breaks:

- There should be no more than one statement (e.g. function call or assignment or comparison) per line. As a rule of thumb, a line that contains an `if` statement or a curly bracket should contain no semicolons and a line that contains a `for` statement is the only one normally allowed to contain two semicolons. All other lines containing a statement would normally contain exactly one semicolon.
- A statement may be broken into multiple lines if it is too long to read comfortably (i.e. if it would otherwise require horizontal scrolling).
- A semicolon must always be followed by a line break.

There are also some rules regarding spacing, although they tend not to be followed as strictly as the rules for indentation and bracketing, so we do not present them here. Indentation and bracketing change the appearance of the source code and affect the readers' ability to find (easily) where a block starts and ends. The choice of whether to insert a space or not is unlikely to affect the overall appearance of the code and is largely a matter of personal taste.

The following is a piece of code from the noughts-and-crosses game that conforms to the naming, indentation and bracketing rules of this section. Do not worry too much about what it is supposed to be doing (in this case, nothing really), as at this stage the layout is the important thing to demonstrate:


```

class COandXEngine : public CBase
/*
  This engine class represents the board, a grid of tiles
  which can be modified by the controller.
*/
{
public:
    static COandXEngine* NewL();
    virtual ~COandXEngine();

private:
    COandXEngine();
    TInt TileState(TInt aX, TInt aY) const;

public:
    // reset
    void Reset();
    TInt SquareStatus(TInt aIndex) const;

    // stuff
    TBool TryMakeMove(TInt aIndex, TBool aCrossTurn);
    TInt GameWonBy() const;

    // persistence
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);

private:
    /*
      Each tile's value is represented by an integer,
      ETileBlank, ETileNought, or ETileCross.
    */
    TFixedArray<TInt, KNumberOfTiles> iTileStates;
};

TInt COandXEngine::TileState(TInt aX, TInt aY) const
/*
  Get the state of the tile at the supplied coordinates.
  @param aX      Tile X coordinate.
  @param aY      Tile Y coordinate.
  @return        Value of tile at (aX, aY).
*/
{
    ASSERT(aX >= 0 && aX < KTilesPerSide);
    ASSERT(aY >= 0 && aY < KTilesPerSide);

    return iTileStates[aY * KTilesPerSide + aX];
}

TInt COandXEngine::GameWonBy() const
/*
  Check if there is a full line of noughts or crosses.
  The line can be horizontal, vertical, or diagonal.

  @return  ETileNought if there is a line of noughts;
           ETileCross if there is a line of crosses;
           Zero if there is no complete line.
*/

```

```

{
const TInt KNoughtWinSum = KTilesPerSide * ETileNought;
const TInt KCrossWinSum = KTilesPerSide * ETileCross;

// is there a row or column of matching tiles?
for (TInt i = 0; i < KTilesPerSide; ++i)
{
    TInt rowSum = 0;
    TInt colSum = 0;
    for (TInt j = 0; j < KTilesPerSide; ++j)
    {
        rowSum += TileState(j, i);
        colSum += TileState(i, j);
    }

    if (rowSum == KNoughtWinSum || rowSum == KCrossWinSum)
        return rowSum / KTilesPerSide;

    if (colSum == KNoughtWinSum || colSum == KCrossWinSum)
        return colSum / KTilesPerSide;
}

// is there a diagonal of matching tiles?
TInt blTrSum = 0;    // bottom left to top right
TInt tlBrSum = 0;    // top left to bottom right
for (TInt i = 0; i < KTilesPerSide; ++i)
{
    tlBrSum += TileState(i,i);
    blTrSum += TileState(i,KTilesPerSide - 1 - i);
}

if (blTrSum == KNoughtWinSum || blTrSum == KCrossWinSum)
    return blTrSum / KTilesPerSide;

if (tlBrSum == KNoughtWinSum || tlBrSum == KCrossWinSum)
    return tlBrSum / KTilesPerSide;

return 0;
}

```

3.3 Functions

Function prototypes in C++ header files can convey a lot of information, including:

- whether the function is imported from a DLL (indicated by `IMPORT_C`), is inline and expanded from a header, or is private to a DLL
- whether the function is public, protected, or private in the C++ sense (you have to scan up the file to see this, but this information is effectively part of the prototype even so)
- whether the function is virtual (you have to scan down the base classes to be sure about this, but this information is part of the signature) and, if virtual, whether it is also pure virtual

- whether the function is `static`
- the return type (or `void`)
- the name – usually a good hint at what the function does
- whether the function can leave (L at the end of the name)
- the type and method of passing for all the arguments (optionally, with a name that hints at purpose, although the name is not formally part of the signature)
- whether there are any optional arguments
- whether it is `const`.

If a function and its arguments (and class) have been named sensibly, and if the right type of parameter passing has been used, you can often guess what a function does by looking at its prototype. For example, the following function is the basic function for writing data to a file – you can even guess that `TDesC8` is a type suitable for data buffers by looking at this signature:

```
TInt RFile::Write(const TDesC8& aBuffer)
```

The `TInt` return is an error code, while the `aBuffer` parameter is a descriptor containing the data and is not modified by the function.

Most of this is standard C++ material. The exceptions are leaving functions, which we’ve already mentioned (and explain fully in Chapter 4) and the naming conventions associated with DLLs. These are very important and aren’t covered by C++ standards; we cover the significance of `IMPORT_C` later in this chapter.

Each parameter’s declaration gives valuable information about whether that parameter is to be used for input or output, and a clue about whether the parameter is large or small. If a parameter is of basic type `X`, there are five ways to specify it in a signature, as shown in Table 3.4.

Table 3.4 Parameter specifications.

	By value	By & reference	By * reference
Input	X	const X&	const X*
Output		X&	X*

For input parameters, there is a fundamental distinction between passing by value and passing by reference. When you pass by value, C++ copies the object into a new stack location before calling the function. You should pass by value only if you know the object is small – a built-in

type, say, or something that is shorter than two machine words (64 bits). If you pass by reference, only a 32-bit pointer is passed, regardless of the size of the data.

If you pass by reference, you have to choose between `*` and `&` references. Usually an `&` reference is better. Effectively it treats the argument variable as a synonym of the variable in the calling function and operations affect the original variable contents directly. In contrast, a pointer (`*`) reference passes the address of the variable into the function, which then has to be dereferenced on every occasion using the appropriate operator (`->`). Use a pointer reference if you have to, especially where a null value is possible, or you're transferring ownership of the object. It's more usual to pass C class instances with `*` and to pass R and T class instances directly or with `&`.

You have to use an `&` reference for C++ copy constructors and assignment operators, but it's rare to need to code such things in Symbian OS. Some Symbian OS APIs use an `&` reference for C types to indicate that a null value is not acceptable.

3.4 APIs

If you have a component, then its API allows you to use it. In addition, its API may also allow it to call your code. In the old days, components were simple libraries; they would specify *library functions* that you would call to get the components to do what you wanted.

Event-driven GUI systems are often associated with *frameworks*, which call your code to allow it to do things supported by the framework. The component specifies *framework functions* and you implement them.

Framework functions used to be called *callbacks*: the basic theory was that your code was really in control, but the library needed to call it back occasionally so it could complete a function for the library. But the truth these days is that the framework is essentially in control and it enables you to do things. The framework functions are the main functions that allow you to do anything. 'Callback' is quite an inappropriate term for this. The word is not used for the major Symbian OS frameworks; it is still used for a couple of situations where the old callback scenario really applies and in relation to a couple of the oldest classes in Symbian OS.

We can loosely classify a class or even an entire API as either a *library API* or a *framework API*. A library API mainly contains functions that your code calls, while a framework API consists mainly of functions that call your code. Many APIs contain a good mixture of both: the GUI, for instance, calls your code so that you can handle events, but provides functions that your code calls to draw graphics.

Types of Function

We have defined library functions and framework functions. However, throughout the book, we use other terms to describe the role of different types of function.

Of course, there is the *C++ constructor*. We almost always use the full term, including ‘C++’, because, as we’ll see in Chapter 4, many classes also have a *second-phase constructor*, usually called `ConstructL()`. There’s only one *destructor*, though, so in that context we do not usually need to refer to it as the ‘C++ destructor’.

Convenience functions are trivial wrappers for things that could otherwise be done with a smaller API. If a class contains two functions `Foo()` and `Bar()`, which do all that the class requires, you may often find that code using your API contains sequences such as this:

```
x.Foo();  
x.Bar();
```

or this:

```
x.Foo(x.Bar());  
y = (x.Foo() + x.Bar()) / 2;
```

In that case, you may wish to code some kind of convenience function `FooAndBar()` that represents the sequence. This reduces code size, reduces mistakes, and makes code easier to read.

The cost of convenience is another function to design and document, and the risk of being tempted to produce many convenience functions which aren’t really all that necessary or even convenient – and then being forced to maintain them for ever more, because people depend on them. This is a fine judgment call: sometimes we provide too few convenience functions and sometimes too many.

DLLs and Other API Elements

An object-oriented system delivers APIs mainly as C++ *classes*, together with their member functions and data. Classes that form part of an API are:

- declared in header files
- implemented in C++ source files
- delivered in DLLs.

Library APIs (or the library parts of a framework API) are delivered in shared DLLs with a `.dll` file extension. The DLL's exported functions are made available in a `.lib` file to the linker at program build time.

Framework APIs are usually defined in terms of C++ classes containing virtual functions and an interface specification for the user to provide implementations for those virtual functions.

It's important to make sure that only the interface – not the implementation – is made available to programs that use the API. Classes that are not part of the API should not be declared in API header files, and their functions should not be exported from the DLLs that implement them. Functions and data that belong to the API classes, but which are not part of the API, should be marked `private`.

Besides classes, C++ APIs may contain entities such as: enumerations, constants, template functions and non-member functions.

Exported Functions

For a non-virtual, non-inline member function to be part of an API, it must be:

- declared public in a C++ class that appears in a public header file
- exported from its DLL.

You can see exported functions marked in their header files with `IMPORT_C`, like this:

```
class RTimer : public RHandleBase
{
public:
    IMPORT_C TInt CreateLocal();
    IMPORT_C void Cancel();
    IMPORT_C void After(TRequestStatus& aStatus,
        TTimeIntervalMicroSeconds32 anInterval);
    IMPORT_C void At(TRequestStatus& aStatus, const TTime& aTime);
    IMPORT_C void Lock(TRequestStatus& aStatus, TTimerLockSpec aLock);
};
```

The `IMPORT_C` macro says that the function must be imported from a DLL by the user of that API. In the corresponding implementation, the function is marked `EXPORT_C`, which means that it is exported from the DLL. A function without `IMPORT_C` is not exported from its DLL and cannot, therefore, be part of the public API. These macros are defined in `e32def.h`. Their implementations are compiler-dependent and differ between compilers.

Virtual and inline functions don't need to be exported – they form a part of the API, even without `IMPORT_C` in the header file.

If you are writing an API to be delivered in a DLL for use by other DLLs, you need to mark your `IMPORT_C`s and `EXPORT_C`s carefully.

If you are not writing APIs – or you are not encapsulating them in DLLs for export – then you do not need to worry about how to use `IMPORT_C` and `EXPORT_C`. It's enough to understand what they mean in the headers of an SDK.

Virtual Functions and APIs

C++ is not particularly well designed for API delivery. There is no way to prevent further override of a specific virtual function and there is no way to guarantee that a function is not virtual without looking down all the base classes to check for the `virtual` keyword. You can simulate Java's `final` at the class level, however, by making the constructor private.

The access control specifiers of C++ are not good for API delivery either. The meaning of `public` is clear enough, but `protected` makes a distinction between derived classes and other classes that doesn't put the boundary in the right place, since derivation is by no means the most important vehicle for code re-use in object orientation. The keyword `private` does not mean private when it comes to virtual functions: you can override private virtual functions whether or not this was intended by the designer of an intermediate class (derived from a base class).

C++ has no language support for packaging APIs, except classes and header files. This is why Symbian had to invent its own rules for DLLs. These design issues are most awkward when it comes to virtual functions. Best practice in Symbian OS C++ includes the following guidelines:

- declare a function virtual in the base class and in any derived class from which it is intended to further derive and override (or implement) this function
- when declaring a virtual function in a derived class, include a comment such as `// from CCoeControl`, to indicate from where the function was defined
- use `private` in a base class to indicate that your base class (or its friends) calls this function – this is usually the case for framework functions. If you don't like friends or the framework function is designed to be called from another class, then make it `public` in the base class
- use `private` in a derived class for a framework function that is implementing something in a framework base class.

These guidelines are, admittedly, incomplete and they are not always honored in Symbian OS code. But they're good for most cases.

Finally, there is another issue with virtual functions: if your class has virtual functions and you need to invoke the default C++ constructor from a DLL other than the one your class is delivered in, then you need to specify, and export, a default C++ constructor:

```
class CFoo : public CBase
{
public:
    IMPORT_C CFoo();
    ...
};
```

And then in the source code:

```
EXPORT_C CFoo::CFoo()
{
}
```

If you do not do this, a program that tries to create a default C++ constructor for your class cannot do it, because constructors need to create the virtual function table and the information required is inside your DLL. It will produce a link error.

3.5 Templates

Symbian OS uses C++ templates extensively, for collection classes, fixed-length buffers and utility functions. The use of templates in Symbian OS is optimized to minimize the size in ‘expanded’ template code – basically, by ensuring that templates are never expanded. The *thin-template pattern* is the key to this.

Symbian OS also uses numeric arguments in templates to indicate string and buffer sizes.

Thin-Template Pattern

The thin-template pattern uses templates to provide a type-safe wrapper round type-unsafe code. It works like this: code a generic base class, such as `CArrayFixBase`, which deals in ‘unsafe’ `TAny*` objects. This class is expanded into real code that goes into a DLL. Then, code a template class that derives from this one, and uses inline type-safe functions such as:

```
template <class T>
inline const T& CArrayFix<T>::operator[](TInt aIndex) const
{
    return (*(const T*)CArrayFixBase::At(aIndex));
}
```


This returns the item at position `aIndex` of type `const T&` in the `CArrayFix<T>` on which it is invoked. It acts as a type-safe wrapper around `At()` in the base class, which returns the pointer at position `aIndex` of type `TAny*`.

This code looks pretty ugly, but the good news is that application programmers do not have to use it. They can simply use the template API:

```
CArrayFix<TFoo>* fooArray;
...
TFoo foo = (*fooArray)[4];
```

The template guarantees that this code is type-safe. The fact that the `operator[]()` is expanded inline means that no more code is generated when the template is used than if the type-unsafe base class had been used.

Numbers in Templates

Sometimes, the parameter to a template class is a number, rather than a type. Here's the declaration of `TBuf`, a buffer of variable length:

```
template <TInt S> class TBuf : public TDes
{
...
};
```

You can then create a five-character buffer with:

```
TBuf<5> hello;
```

This uses the thin-template pattern too: here's the inline constructor:

```
template <TInt S>
inline TBuf<S>::TBuf() : TDes(0,S)
{
}
```

It calls the `TDes` base-class constructor, passing the correct parameters, and then completes the default construction of a `TBuf` (a couple of extra instructions).

3.6 Casting

Casting is a necessary evil. Old-style C provides casting syntax that enables you to cast any type to any other type. Over time, different casting patterns have emerged, including:

- cast away the `const` property (but don't change anything else)
- cast to a related class (rather than an arbitrary cast)
- reinterpret the bit pattern (effectively, old-style C casting).

C++ provides individual keywords for each of these types of cast: `const_cast<>()`, `static_cast<>()` and `reinterpret_cast<>()`. These should be used in preference to old-style C casting, as through doing this you get the benefit of C++ cast checking. Previous versions of the SDK defined macros which expanded to these keywords, for compatibility with older versions of GCC which did not support C++ casting. These are now deprecated – current compilers can use the standard C++ definitions.

3.7 Classes

As you'd expect, classes are used to represent objects, abstractions and interfaces. Relationships between classes are used to represent relationships between objects or abstractions. The most important relationships between classes are:

- *uses-a*: if class A 'uses-a' class B, then A has a member of type B, B&, `const B&`, B* or `const B*`, or a function that can easily return a B in one of these guises; A can then use B's member functions and data
- *has-a*: 'has-a' is like 'uses-a', except that A takes responsibility for constructing and destroying the B as well as using it during its lifetime
- *is-a*: if class A 'is-a' class B, then B should be an abstraction of A; 'is-a' relationships are usually represented in C++ using public derivation
- *implements*: if class A implements an *interface* M, then it implements all M's pure virtual functions; this is the only circumstance in which multiple inheritance is used in Symbian OS.

Sometimes, the distinction between abstract and concrete notions is blurred. `CEikDialog` is concrete as far as its implementation of `CCoeControl` is concerned but `CEikDialog` is an essentially abstract base class for user-specified dialogs or Uikon standard dialogs such as a `CEikInfoDialog`. Some classes (such as `CCoeControl` and `CEikDialog`) contain no pure virtual functions, but are still intended to be derived from. They provide fall-back functionality for general cases, but it is likely that one or two functions must be overridden to provide a useful class.

Interfaces

Symbian OS makes quite extensive use of interface classes (originally called mixins). An interface is an abstract base class with no data and only pure virtual functions.

APIs that have both library and framework aspects often define their library aspect by means of a concrete class and their framework by means of an interface class. To use such an API, you need to use the concrete class and implement the interface.

The Symbian OS Print API provides an example. In addition to library classes to start the print job, there are framework classes for printing part of a page and for notifying the progress of the job to an application. `MPrintProcessObserver` is the interface for notifying progress:

```
class MPrintProcessObserver
{
public:
    virtual void NotifyPrintStarted(TPrintParameters aPrintParams) = 0;
    virtual void NotifyBandPrinted(TInt aPercentageOfPagePrinted,
                                   TInt aCurrentPageNum, TInt aCurrentCopyNum) = 0;
    virtual void NotifyPrintEnded(TInt aErrorCode) = 0;
};
```

This interface definition includes functions for reporting the beginning and end of a print job and its progress at intervals throughout. The print-preview image is designed to be owned by a control in a print-preview dialog. It provides a standard print-preview image and its definition starts:

```
class CPrintPreviewImage : public CBase, private MPrintProcessObserver,
                           private MPageRegionPrinter
{
}
```

You can see that this uses C++ multiple inheritance, in this case inheriting from `CBase` and two interface classes. When using multiple inheritance in this manner, place the `CBase` (or `CBase`-derived) class first in the list. Otherwise, subtle details of the operating system will cause problems.

The only encouraged use of multiple inheritance in Symbian OS is to implement interfaces. Any other use is unnecessary and is strongly discouraged. Standard classes are not designed with multiple inheritance in mind.

Bad Practices

Many C++ features that look attractive at first sight are not used in Symbian OS – or, at least, they’re not encouraged in anything other than very specific situations:

- private inheritance: inheritance should only be used for ‘is-a’ relationships; private inheritance (the default in C++) is used to mean ‘has-a’, so that the private base class effectively becomes a private data member
- multiple inheritance: except in the case of interfaces, full-blown C++ multiple inheritance is more confusing than useful
- overriding non-trivial virtual functions: base classes with virtual functions should either specify trivial behavior (doing nothing, for example) or leave them pure virtual. This helps you to be clear about the purpose of the virtual function.
- ‘just-in-case’ tactics: making functions virtual ‘just in case’ they should be overridden or protected ‘just in case’ a derived class wishes to use them is an excuse for unclear thinking.

There may be times when these practices can be used for good reason. The thin-template pattern is really a C++ technical trick, so it’s fair game to use C++ technical tricks such as private inheritance to help implement it. But if you want your C++ to be a straightforward implementation of good object-oriented system design, you should use the object-oriented features of C++ rather than such technical tricks.

3.8 Design Patterns

Object orientation supports good design using the ‘uses-a’, ‘has-a’, ‘is-a’ and ‘implements’ relationships. Through good design, object orientation also supports good code re-use. That’s particularly attractive for Symbian OS, since minimizing the amount of code you require to implement a particular system is a very important design goal.

But code re-use isn’t the only form of re-use. Often, you find yourself doing the same thing again and again, but somehow you can’t abstract it into an API – even using the full power of object orientation and templates in C++. Or, you succeed in abstracting an API, but it’s more difficult to use the API than it was to write the repeated code in the first place.

This is a good time to think in terms of re-using *design patterns*. Design patterns are ways of designing things, rather than objects or APIs that you can re-use or glue together.

Symbian OS contains many frequently used design patterns. The most characteristic design patterns relate to cleanup (see Chapter 4) and active objects (see Chapter 6). However, most of the patterns used in Symbian OS are standard patterns used elsewhere in the software industry.

Summary

In this chapter, we have looked at the features of C++ that Symbian OS uses, those it avoids and those it augments. We've also seen the coding standards that are used in Symbian OS. Specifically, we've seen:

- the use of Symbian OS fundamental types to guarantee consistent behavior between compilers and platforms
- naming conventions – the core ones help with cleanup: `T`, `C` and `R` distinguish between different cleanup requirements for classes; `i` refers to member variables and `L` to leaving functions that may fail for reasons beyond the control of the programmer; `M` prefixes interfaces (consisting of pure virtual functions)
- good design of function prototypes – how to code input and output parameters and suggested use of references and pointers in Symbian OS
- the difference between library and framework DLLs – the former provide functions that you call, the latter functions for you to implement that the framework calls
- exporting non-virtual, non-inline functions using the `IMPORT_C` and `EXPORT_C` macros
- how to handle virtual functions
- templates and the thin-template pattern to keep code size to a minimum
- the four relationships between classes
- the use of mixins to provide interfaces and the only use of multiple inheritance in Symbian OS.

4

Objects – Memory Management, Cleanup and Error Handling

Before we head into the deeper aspects of Symbian OS, we need to spend some time looking at some of the basic operations, programming patterns and classes that are common to all applications, and indeed, to almost all code that runs in the system.

Object creation and destruction are intimately tied up with the issue of cleanup. It is important to make sure that your applications are coded in such a way that they do not leak memory even if a coding error occurs – a real issue for systems that may not be rebooted for long periods of time, if at all. This is why memory management (object creation and destruction), cleanup and error handling are discussed as a single subject. This chapter presents the basic patterns that are used over and over again: building blocks that allow you to build safe and efficient code.

Symbian OS uses object orientation, and is written in C++, with bits of Assembler thrown in at the lowest levels. This means that the vast majority of applications are written in C++. However, as discussed in Chapter 2, Symbian OS C++ is not exactly the same as C++ in other environments.

- C++ does more than Symbian OS requires – for example, full-blown multiple inheritance.
- C++ does less than Symbian OS requires – for example, it doesn't insist on the number of bits used to represent the basic types, and it knows nothing about DLLs.
- Different C++ communities do things differently because their requirements are different. In Symbian OS, large-scale system design is

combined with a focus on error handling and cleanup, and efficiency in terms of ROM and RAM budgets.

- Symbian OS is particularly concerned with the handling of errors that may occur during memory allocation. Because of the device constraints, it is important that these errors are handled in such a way that no memory remains allocated or orphaned afterwards.

4.1 Object Creation and Destruction

One of the fundamental characteristics of object-oriented systems is the creation and destruction of objects. Objects are created, have a finite lifetime, and are then destroyed.

We'll look first at the basics of object creation and destruction. In some ways this may give a misleading picture; the full picture will only emerge once we've looked at error handling and cleanup. This is because object creation, object destruction, error handling and cleanup are all intimately tied together with the aim of ensuring that:

- objects, once created, are always destroyed when no longer needed
- objects are never left half-created, that is with memory occupied or unavailable, if an error occurs during the creation process.

There are two places in Symbian OS that objects can be created: the heap and the program stack.

The Heap (Dynamic Objects)

All threads have an associated heap, termed the *default heap*, from which memory can be allocated at run time. This is where you put large objects and objects that can only be built at run time, including (but not limited to) dynamic variable-length strings. This is also where you put objects whose lifetimes don't coincide with the function that creates them; typically such objects become data members of the parent or owning object, with the relationship expressed as a pointer from the owning object to the owned object.

Memory is allocated from the thread's default heap, as and when required, using the C++ operator `new` or, very rarely, using user library functions such as `User::Alloc()`. If there is insufficient free memory, then an allocation attempt fails with an out-of-memory error.

In Symbian OS, classes that are intended to be instantiated on the heap are nearly always derived from the `CBase` class (see Section 4.2). All `CBase`-derived classes must be placed on the heap, but not all heap objects are necessarily `CBase`-derived.

Creating objects

The following code fragment shows a simple way of creating a heap-based object.

```
class CMyClass : public CBase
{
public:
    CMyClass();
    ~CMyClass();
    void Foo();
private:
    TInt      iNumber;
    TBuf<32> iBuffer;
}
CMyClass* myPtr = new CMyClass;
if (myPtr)
{
    myPtr->Foo(); // can safely access member data & functions
}
delete myPtr;
```

If there is insufficient memory to allocate the `CMyClass` object, then `myPtr` is `NULL`. If allocation succeeds, `myPtr` points to the new `CMyClass` object and the data members `iNumber` and `iBuffer` are guaranteed to be binary zeroes.

The `delete` operator causes the object's destructor to be called before the memory for the object itself is released back to the heap.

There's one very important variation on this:

```
CMyClass* myPtr = new (ELeave) CMyClass;
myPtr->Foo(); // can safely access member data & functions
delete myPtr;
```

The main difference here is that we have specified `ELeave` as part of the `new` operation. Instead of returning a `NULL` value when there isn't enough memory to create the `CMyClass` object, the operation leaves. As discussed in earlier chapters, the Symbian OS `Leave` corresponds to a standard C++ `throw`. Leaving is explained in detail in Section 4.3, but think of it for the moment as an operation where the function returns immediately.

If the `new` operation doesn't leave, then memory allocation for the new object has succeeded, the object has been created, and program control flows to the next C++ instruction, `myPtr->Foo()`. There is no need to check the value of `myPtr`; the fact that the `new` operation has returned means that `myPtr` will have a sensible value.

Conversely, if the `new` operation leaves, the next line (which would otherwise be unsafe) will not be executed.

Ownership of objects

In a typical object-oriented system such as Symbian OS, where objects are created dynamically, the concept of ownership is important. All objects need to be unambiguously owned so that it is clear who has responsibility for destroying them.

It is good practice to determine who owns each object at the design stage. Then it is, usually, just a matter of ensuring that the owner (typically a class) contains the implementation for both the creation and the destruction of the object.

Simply having a pointer to an object does not automatically imply ownership (that is, a ‘has-a’ relationship). Occasionally, a pointer to an object exists merely in order to facilitate access to its contents (that is, a ‘uses-a’ relationship). Symbian OS tries to address the possible ambiguity by wrapping non-owned objects into R classes (see Section 4.2).

The basic rules to observe are as detailed below:

- Use a destructor to destroy objects that you own. Never assume automatic cleanup for anything.
- Beware of any transfers in the ownership of objects. The owner of the object at the time of its destruction is not necessarily the same as at the time of its creation.
- Do not attempt to destroy objects that you do not own. In the case of a ‘uses-a’ relationship rather than a ‘has-a’ relationship, it is unsafe to delete the pointed-to object – the actual owner of the object will later attempt to delete it, with potentially disastrous results. This is known as *double deletion*.
- Don’t forget about objects – even by accident. Don’t allocate objects twice. It sounds obvious, but allocating an object a second time and putting the address into the same pointer variable into which you put the address of the first allocated object means that you lose all knowledge of that first object. There is no way that a class destructor or any other part of the C++ system can find this object, and it represents a *memory leak*.

Deleting objects

As we’ve seen, deleting an object is simply a matter of using the `delete` operator on a pointer to the object to be deleted. If a pointer is already zero, then calling `delete` on it is harmless.

However, you must be aware that `delete` does not set the pointer itself to zero. While this does not matter if you are deleting an object from within a destructor, it is very important if the deletion occurs anywhere else. Double deletion doesn’t always cause an immediate crash, and

sometimes it leaves side effects that only surface a long time after the real problem occurred. As a result, double deletions are very hard to debug.

On the other hand, double deletions are easy to avoid: just follow this little discipline:

C++ `delete` does not set the pointer to zero. If you delete any member object from outside its class's destructor, you must set the member pointer to `NULL`.

The Program Stack (Automatic Objects)

The stack is used to hold the C++ automatic variables for each function. It is suitable for fixed-size objects whose lifetimes coincide with the function that creates them.

In Symbian OS, the stack is a limited resource. A thread's stack cannot grow after a thread has been launched; the thread is panicked (terminated abruptly) if it overflows its stack. This means that the stack should only be used for small data items, for example strings of a few tens of characters. A good rule of thumb is to put anything larger than a file name onto the heap. However, it is quite acceptable to put pointers (and references) into the stack, even pointers to very large objects. This is because what actually goes into the stack is the memory address of the object, rather than the object itself.

You can control the stack size in an executable program, through the use of the `epocstacksize` keyword in the MMP file used to create the EXE. You can also control the stack size when you launch a thread explicitly from your program. However, creating large stacks eats into valuable resources, and is therefore considered bad practice.

Only Symbian OS T-class objects, as defined in Section 4.2, can safely be put onto the program stack. These include built-in types and classes that don't need a destructor because they own no external data. This means that they can be safely discarded, without the need for any kind of cleanup, simply by exiting from the function in which the automatic variable was declared.

In the following example function, a `TInt` and a `TBufC<16>` type are created as automatic variables, used in the body of the function, and then simply discarded, without any kind of cleanup, when the function exits.

```
void CMyClass::Foo()
{
    TInt myInteger;
    TBufC<16> buffer;
    ...
    // main body of the function
} // variables are safely discarded on exit from the function.
```

4.2 Class Categories in Symbian OS

Chapter 3 gave a brief overview of the Symbian OS class naming conventions. The class categories are based on:

- the likely size of a class object
- its likely location in memory (heap or stack)
- whether the object is going to own other objects or be owned by them.

In this section we present the characteristics of each class category in detail.

T and C Classes

The naming convention for classes has been chosen to indicate their main cleanup properties. This section presents the cleanup-related properties of C and T classes, which are quite similar.

Classes with names beginning with C are derived from CBase and allocated on the heap. They must therefore be cleaned up when they are no longer needed. Most C classes have a destructor.

A C class is typically referred to by a pointer stored in a member variable of some class that owns it, a member variable of a class that uses it, or an automatic variable.

If a C class is referred to only by a single automatic, in a function that might leave, then a copy of the pointer should be kept somewhere so that it can be properly deallocated if an error occurs. Symbian OS uses the cleanup stack for this purpose (see Section 4.4).

CBase offers two things to any C class:

- zero initialization, so that all member pointers and handles are initially zero, which is cleanup-safe
- a virtual destructor, so that CBase-derived objects can be properly destroyed from the cleanup stack.

By contrast, classes with names beginning with T are classes, or built-in types, that don't need a destructor because they own no data external to themselves. Examples of T types are:

- any built-in type (defined using typedef), for example TUint for an unsigned integer
- any enumerated type, for example TAmPm, which indicates whether a formatted time-of-day is am or pm; all enumerations begin with T, while enumerated constants, such as EAm and EPm, begin with E

- class types that do not need a destructor, for example `TBuf<40>` (a buffer for a maximum of 40 characters) and `TPtrC` (a pointer to a string of any number of characters). `TPtrC` contains a pointer, but it only uses (rather than owns) the characters it points to, so it does not need a destructor.

T classes do not own any data, so they don't need a destructor. They may have pointers, provided that these are 'uses-a' pointers rather than 'has-a' pointers.

T classes are normally allocated as automatics, or as member variables of any other kind of class.

It's possible (but rare) to allocate T-class objects explicitly on the heap. If so, you need to ensure that the heap cell is freed. You can push a T-class object to the cleanup stack as in the following example (see Section 4.4 for more detail).

```
TDes* name = new (ELeave) TBuf<40>; // TDes is a base of TBuf
CleanupStack::PushL(name);
DoSomethingL();
CleanupStack::PopAndDestroy();    // name
```

A T-class object can usually be assigned using a bit-wise copy. Therefore, T types do not need copy constructors or assignment operators (except in specialized cases, such as copying one `TBuf` to another, where maximum lengths of the `TBufs` may differ).

Since C-class objects reside on the heap and are referred to by pointers, they are passed by reference, that is, by copying the pointer. Thus C classes do not need a copy constructor or an assignment operator.

As a result, C++ copy constructors and assignment operators are extremely rare in Symbian OS.

R Classes

The next category is classes whose names begin with R, which stands for 'resource'. Usually, R-class objects contain a handle to a resource that's maintained elsewhere. Examples include `RFile` (maintained by the file server), `RWindow` (maintained by the window server), and `RTimer` (maintained by the kernel).

The R object itself is typically small; as a minimum, it contains only a resource handle. A function in an R class doesn't usually change the member data of the R class itself; rather, it sends a message to the real resource owner, which identifies the real object using the handle, performs the function, and sends back a result. Functions such as `Open()` and `Create()` allocate the resource and set the handle value. Typically,

a `Close()` function frees the resource and sets the handle value to zero. A C++ constructor ensures that the handle is zero to begin with.

A few R classes do not obey the conventions described here. We will point out such classes as we encounter them.

R classes are like T classes in some ways, and like C classes in others.

- Like T classes, R classes can be automatics or class members. Also like T classes, they can be copied (which just copies the handle), and the copy can be used like the original.
- As with T classes, it is very rare to refer to R classes using pointers. They are usually passed by value or by reference.
- Like C classes, R classes own resources: that is, they contain pointers to resources and are responsible for allocating and deallocating memory for them. Although R classes don't usually have a destructor, they do have a `Close()` function that has a similar effect, including setting the handle to zero, which is rather like zeroing the pointer to a C-class object. It's safe to call `Close()` twice – on an already closed R-class object it has no effect.
- Like C classes, R classes are expected to zero-initialize their handle value, so that functions can't be used until the handle is initialized. Note, however, that the zero-initialization must be explicitly coded, in their constructor (as is the case for all system R classes). There is no `RBase` corresponding to `CBase`.

R classes as member variables

R-class objects are often members of a C-class object. Consider the following example of a C class with a member variable `iFs` of type `RFs` (a file-server session handle). Its file deletion code could be written as follows:

```
case EDeleteFile:
{
    User::LeaveIfError(iFs.Connect());
    User::LeaveIfError(iFs.Delete(KTextFileName));
    iFs.Close();
}
```

It's important for the C++ destructor of the containing C class to include a call to `iFs.Close()`, so that the `RFs` is closed even if the delete operation fails.

M Classes

This particular type of class was designated in order to enable an easy mechanism for multiple inheritance. M classes are typically abstract, so they never get instantiated in their own right. Consequently, the case of memory being allocated for an M-class object is practically non-existent.

M classes are used for packaging *behavior* (rather than data) in a re-usable manner; the prefix M stands for 'mixin'. An M class can be combined any number of times with different base classes (typically C classes). The derived classes (also C classes) all share the behavior but also have their peculiar characteristics.

Usage of M classes for derivation is the only type of multiple inheritance allowed in Symbian OS.

A typical usage pattern for M classes is shown below. The mixin class contains some re-usable behavior that we want to pass down to other classes consistently.

```
class MSomeMixin
{
    virtual void CommonBehaviour1(TInt aParam) = 0;
    virtual void CommonBehaviour2(TInt aParam) = 0;
};
```

We have the following base classes:

```
class CBaseClassA : public CBase
{
public:
    TInt FuncA1();
    TInt FuncA2();
protected:
    void ConstructL();
};

class CBaseClassB : public CBase
{
    TInt FuncB1();
    TInt FuncB2();
protected:
    void ConstructL();
};
```

We can then define the following classes:

```
class CDerivedClassA : public CBaseClassA, MSomeMixin;
class CDerivedClassB : public CBaseClassB, MSomeMixin;
```

In this case, `CDerivedClassA` and `CDerivedClassB` are bona-fide C classes, because they ultimately derive from `CBase`. For memory allocation, creation and destruction, the same rules apply as for any other C class. However, they also both share the common functionality defined in `MSomeMixin`. This means that any changes in `MSomeMixin` will affect both derived classes.

It is of course possible to combine M classes with classes of other type, such as R classes.

```
class RBaseClassA;
class RBaseClassB;

class RDerivedClassA : public RBaseClassA, MSomeMixin;
class RDerivedClassB : public RBaseClassB, MSomeMixin;
```

In this case, the derived classes are also R classes, and are handled in the same way as all other R classes.

4.3 Error Handling

In machines with limited memory and resources, such as those for which Symbian OS is designed, error handling is of fundamental importance. Errors are going to happen, and you can't afford not to handle them correctly.

Symbian OS provides a framework for error handling and cleanup. It is a vital part of the system, with which you need to become familiar. Every line of code that you write – or read – will be influenced by thinking about cleanup. No other Symbian OS framework has as much impact. Because of this, we've made sure that error handling and cleanup are effective and very easy to do.

In addition to the error handling mechanisms discussed in this chapter, it is possible from Symbian OS version 8.1b onwards to use the standard C++ exception handling facilities. This makes it easier for developers to port their existing (non-Symbian OS) code into Symbian OS.

However, we still recommend that developers should use the richer, better tested and more tightly integrated set of Symbian OS-specific error-handling mechanisms discussed in this section.

What Kinds of Error Can the Framework Handle?

The first and most obvious type of error to consider is the out-of-memory error.

These days, desktop PCs come with at least 512 MB of RAM, virtual memory swapping out onto 60 GB or more of hard disk, and with users who expect to perform frequent reboots. In this environment, running out of memory is rare, so you can be quite cavalier about memory and resource management. You try fairly hard to release all the resources you can, but if you forget, then it doesn't matter too much: things will get cleaned up when you close the application, or when you reboot. That's life in the desktop world.

By contrast, Symbian OS phones have as little as 16 MB of RAM, and often no more than 32 MB, although there are now devices with 64 MB. By comparison with a PC this is small, and there is no disk-backed virtual memory, nor are the users used to having to reboot frequently.

There are some key issues here that don't trouble modern desktop software developers.

- You have to program efficiently, so that your programs don't use RAM unnecessarily.
- You have to release resources as soon as possible, because you can't afford to have a running program gobble up more and more RAM without ever releasing it.
- You have to cope with out-of-memory errors. In fact, you have to cope with potential out-of-memory for every single operation that can allocate memory, because an out-of-memory condition can arise in any such operation.
- When an out-of-memory situation arises that stops some operation from happening, you must not lose any data, but must roll back to an acceptable and consistent state.
- When an out-of-memory situation occurs partway through an operation that involves the allocation of several resources, you must clean up all those resources as part of the process of rolling back.

For example, consider the case where you are keying data into an application for taking notes. Each key you press potentially expands the buffers used to store the information. If you press a key that requires the buffers to expand, but there is insufficient memory available, the operation will fail. In this case, it would clearly be quite wrong for the application to terminate – all your typing would be lost. Instead, the document must roll back to the state it was in before the key was processed, and any memory that was allocated successfully during the partially performed operation must be freed.

The Symbian OS error-handling and cleanup framework is good for more than out-of-memory errors. Many operations can fail because of other environment conditions, such as reading and writing to files,

opening files, sending and receiving over communications sessions. The error-handling and cleanup framework can help to deal with these kinds of error too.

Even user input errors can be handled using the cleanup framework. As an example, code that processes the OK button on a dialog can allocate many resources before finding that an error has occurred. Dialog code can use the cleanup framework to flag an error and free the resources with a single function call.

Panics

There's just one kind of error that the cleanup framework can't deal with: programming errors. If you write a program with an error, you have to fix it. The best Symbian OS can do for you (and your users) is to kill your program as soon as the error is detected, with enough diagnostics to give you a chance to identify the error and fix it – hopefully, before you release the program. In Symbian OS, this is called a *panic*.

The basic function to use here is `User::Panic()`. This takes a string that we call the *panic category*, and a 32-bit integer number. The category and the number, in combination, serve as a way of identifying the programming error. The panic category should be no longer than 16 characters, but if it is longer, then only the first 16 characters are used.

On real hardware, after terminating the originating process, a panic simply displays a dialog titled 'Program closed', citing the process name, the panic category and the number passed in the `Panic()` function call.

A typical pattern is to code a global function (or a class static function, which is better) that behaves like this:

```
static void Panic(TInt aPanic)
{
    LIT(KPanicCategory, "MY-APP");
    User::Panic(KPanicCategory, aPanic);
}
```

You can call this with different integer values.

The reason for raising a panic, as we have said, is to deal with programming errors, and not environmental errors such as out-of-memory. A common use is for checking the parameters passed to a function call. For some parameters, there may be a defined range of valid values, and accepting a value outside that range could cause your code to misbehave. You therefore check that the parameter is within range, and raise a panic if it's not. Another common use is where you want to ensure that a function is called only if your program is currently in a particular state, and raise a panic if not.

A commonly used way of raising panics is to use *assert macros*, of which there are two, `__ASSERT_DEBUG` and `__ASSERT_ALWAYS`. The

first is only compiled into debug versions of your program, while the latter is compiled into both debug versions and production versions. As a general rule, put as many as you can into your debug code, and as few as you can into your release code. In other words, do your own debugging – don't let your users do it for you.

The following code shows the use of the `assert` macro.

```
enum TMyAppPanic
{
    ...
    EMyAppIndexNegative = 12,
    ...
}

void CMyClass::Foo(TInt aIndex)
{
    __ASSERT_ALWAYS(aIndex > 0, Panic(EMyAppIndexNegative));
    ...
}
```

The pattern here is `__ASSERT_ALWAYS(condition, expression)`, where the expression is evaluated if the condition is not true. `CMyClass::Foo()` should only be called with positive values of parameter `aIndex`, so we panic with code `EMyAppIndexNegative` if not. This gets handled by the `Panic()` function presented previously, so that if this code were executed on a production machine, it would show 'Program closed' with a category of `MY-APP` and a code of 12. The number 12 comes from the enumeration `TMyAppPanic` containing a list of panic codes.

Leave and the TRAP Harness

The majority of errors suffered by a typical program are environment errors, typically caused by a lack of suitable resources at some critical time, the most common of which is lack of memory. In such circumstances it is usually impossible to proceed, and the program must roll back to some earlier known state, cleaning up resources that are now no longer needed.

Leaving is part of that mechanism. A leave is invoked by calling the static function `User::Leave()` from within a *trap harness*. The trap harness catches errors – more precisely, it catches any functions that leave. If you're familiar with the exception handling in standard C++ or Java, `TRAP()` is like `try` and `catch` all in one, `User::Leave()` is like `throw`, and a function with `L` at the end of its name is like a function with `throws` in its prototype.

In a typical program, code can enter a long sequence of function calls where one function calls another which, in turn, calls another. If the

innermost function in this nested set of calls can't, for example, allocate memory for some object that it needs to create, then you might expect that function to return some error value back to its caller. The calling function has to check the returned value, and it in turn needs to return an error value back to its caller. If each calling function needs to check for various return values, the code quickly becomes intractably complex.

Use of the trap harness and `leave` can cut through all this. `User::Leave()` causes execution of the active function to terminate, and to return through all calling functions, until the first one is found that contains a `TRAP()` or `TRAPD()` macro. Just as important, it rolls back the stack frame as it goes.

You can find the `TRAP()` and `TRAPD()` macros in `e32std.h`. It is not important to understand their implementations; the main points are as follows.

- `TRAP()` calls a function (its second parameter) and returns its leave code in a 32-bit integer (its first parameter). If the function returns normally, without leaving, then the leave code will be `KErrNone` (defined as zero).
- `TRAPD()` defines the leave code variable first, saving you a line of source code, and then essentially calls `TRAP()`.

For example, in an application having word processing-type behavior, a key press might cause many things to happen, such as the allocation of undo buffers and the expansion of a document to take the new character or digit. If anything goes wrong, you might want to undo the operation completely, using code such as the following.

```
TRAPD(error, HandleKeyL());
if(error)
{
    RevertUndoBuffer();

    // Any other special cleanup
    User::Leave(error);
}
```

If the `HandleKeyL()` function leaves, `error` contains the leave value, which is always negative. If the function does not leave, `error` contains 0.

On detecting a leave in `HandleKeyL()`, this code fragment performs some specialized cleanup, then calls `User::Leave()` itself. The Symbian OS framework provides a `TRAP` statement outside the user application code, which catches the leave and posts the error message.

While leaving can save a lot of hard coding effort, it does present a problem. A typical program allocates objects on the heap, quite often

lots of them. It is also common behavior for a function to store a pointer to a heap object in an automatic variable, even if only temporarily. If, at the time a leave is taken, the automatic variable is the only pointer to that heap object, then knowledge of this heap object will be lost when a leave is taken, causing a memory leak. As Symbian OS doesn't use C++ exceptions, it needs its own mechanism to ensure that this does not happen. This mechanism is the cleanup stack (see Section 4.4 below).

You don't often need to code your own traps, partly because the Symbian OS framework provides a default `TRAP` macro for you, and partly because routine cleanup is handled by the cleanup stack. Sometimes, though, you'll need to perform a recovery action or some non-routine cleanup. Or you'll need to code a non-leaving function, such as `Draw()`, that allocates resources and completes successfully if possible, but otherwise traps any leaves and handles them internally. In these cases, you must code your own trap.

While `Draw()` functions do not normally need to allocate any resources, some are rather complicated and it may be appropriate to code them in such a way that they do make allocations. In this case, you have to hide the fact from the Symbian OS framework by trapping any failures yourself.

```
virtual void Draw(const TRect& aRect) const
{
    TRAPD(error, MyPrivateDrawL(aRect));
}
```

In this case, we choose to keep quiet if `MyPrivateDrawL()` failed. The failing draw code should take some graceful action, such as drawing in less detail, or blanking the entire rectangle, or not updating the previous display.

Don't use traps when you don't have to. Here's a particularly useless example:

```
TRAPD(error, FooL());
if(error)
    User::Leave(error);
```

The net effect of this code is precisely equivalent to:

```
FooL();
```

However, it's more source code, more object code, and more processing time whether or not `FooL()` actually does leave.

4.4 The Cleanup Stack

The cleanup stack addresses the problem of cleaning up objects which have been allocated on the heap, but whose owning pointer is an automatic variable. If the function that has allocated the object leaves, then the object needs to be cleaned up.

In the following example, the `UseL()` function allocates memory and might leave.

```
case ECmd3:
{
    CX* x = new (ELeave) CX;
    x->UseL();
    delete x;
}

...

void CX::UseL()
{
    TInt* pi = new (ELeave) TInt;
    delete pi;
}
```

This code can go wrong in two places. First, the allocation of `CX` might fail – if so, the code leaves immediately with no harm done.

If the allocation of the `TInt` fails in `UseL()`, then `UseL()` leaves, and the `CX` object, which is pointed to only by the automatic variable `x` in the case statement, can never be deleted. The memory is orphaned – a memory leak has occurred.

Using the Cleanup Stack

After the line `CX* x = new (ELeave) CX;` has been executed, the automatic `x` points to a cell on the heap. After the leave, the stack frame containing `x` is abandoned without deleting `x`. That means that the `CX` object is on the heap, but no pointer can reach it, and it will never be destroyed.

The solution is to make use of the cleanup stack. As its name implies, the cleanup stack is a stack of entries representing objects that are to be cleaned up if a leave occurs. It is accessed through the `CleanupStack` class. The class has `push-` and `pop-`type functions that allow you to put items onto the stack and remove them. In a console application the cleanup stack must be explicitly provided, but in a standard GUI application it is provided by the UI Framework.

C++'s native exception handling addresses the problem of automatics on the stack by calling their destructors explicitly, so that a separate cleanup stack isn't needed. As discussed in Chapter 2, C++ exception

handling was not standardized or well supported when Symbian OS was designed, so it wasn't an option to use it.

Stack-frame object destructors are not called if a leave happens, in other words objects on the stack frame are not cleaned up. Only objects on the cleanup stack are cleaned up.

Here's how we should have coded the previous example:

```
case ECmd3:
{
    CX* x = new (ELeave) CX;
    CleanupStack::PushL(x);
    x->UseL();
    CleanupStack::PopAndDestroy(x);
}
```

The cleanup stack class, `CleanupStack`, is defined in `e32base.h`. With these changes in place, the sequence of events is:

- Immediately after we have allocated the `CX` and stored its pointer in `x`, we also push a copy of this pointer onto the cleanup stack.
- We then call `UseL()`.
- If this doesn't fail (and leave), our code pops the pointer from the cleanup stack and deletes the object. We could have used two lines of code for this (`CleanupStack::Pop()`, followed by `delete x`), but this is such a common pattern that the cleanup stack provides a single function to do both.
- If `UseL()` does fail, then as part of leave processing, all objects on the cleanup stack in the current trap harness, including our `CX` object, are popped and destroyed.

We could have done this without the aid of the cleanup stack by using code like this:

```
case ECmd3:
{
    CX* x = new (ELeave) CX;
    TRAPD(error, x->UseL());
    if(error)
    {
        delete x;
        User::Leave(error);
    }
    delete x;
}
```

However, this is much less elegant. The cleanup stack works particularly well for long sequences of operations, such as:

```
case ECmd3:
{
    CX* x = new (ELeave) CX;
    CleanupStack::PushL(x);
    x->UseL();
    x->UseL();
    x->UseL();
    CleanupStack::PopAndDestroy();
}
```

Any one of the calls to `UseL()` may fail, and it would look very messy if we had to surround every `L` function with a trap harness just to address cleanup. It would also reduce efficiency and increase the size of the code to unacceptable levels.

The cleanup stack is a stack like any other, and you can add more than one item to it. So for example, we can have:

```
case ECmd3:
{
    X* x1 = new (ELeave) CX;
    CleanupStack::PushL(x1);
    CX* x2 = new (ELeave) CX;
    CleanupStack::PushL(x2);
    x1->UseL();
    CleanupStack::PopAndDestroy(2);
}
```

The call to `PopAndDestroy(2)` causes the last two items to be removed from the stack and destroyed. When you do this, you must be careful not to remove more items from the cleanup stack than you put onto it, otherwise your program will panic.

Don't Use the Cleanup Stack Unnecessarily

A common mistake when using the cleanup stack is to be overenthusiastic, putting all possible pointers to heap objects onto the cleanup stack. This is wrong. You need to put a pointer to an object onto the cleanup stack only to prevent an object's destructor from being bypassed. If the object's destructor is going to be called anyway, then you must not use the cleanup stack.

If an object is a member variable of another class, it will be destroyed by the class's destructor, so you should never push a member variable to the cleanup stack.

According to Symbian OS coding conventions, member variables are indicated by an `i` prefix, so code such as the following is always wrong.

```
CleanupStack::PushL(iMember);
```

This is likely to result in a double deletion, once from the cleanup stack and once from the class's destructor. Following the coding conventions makes it very easy to spot and prevent coding mistakes such as the one above.

Another way of thinking about this is that once a pointer to a heap object has been copied into a member of some existing class instance, then you no longer need that pointer on the cleanup stack. This is because the class destructor takes responsibility for the object's destruction (assuming, of course, that the destructor is correctly coded to do this).

What if `CleanupStack::PushL()` Itself Fails?

Pushing to the cleanup stack potentially allocates memory, and therefore may itself fail! You don't have to worry about this, because such a failure will be handled properly. But for reassurance, here's what happens under the covers.

Symbian OS addresses this possibility by always keeping at least one spare slot on the cleanup stack. When you do a `PushL()`, the object you are pushing is first placed onto the cleanup stack (which is guaranteed to work, because there was a spare slot). Then a new slot is allocated. If that fails, the object you just pushed is popped and destroyed.

The cleanup stack actually allocates more than one slot at once, and doesn't throw away existing slots when the contents are popped. So pushing and popping from the cleanup stack are very efficient operations.

Since the cleanup stack is used to hold temporary objects, or objects whose pointers have not yet been stored as member pointers in their parent object during the parent object's construction, the number of cleanup stack slots needed by a practical program is not large; more than ten would be very rare. So the cleanup stack itself is very unlikely to be a contributor to out-of-memory errors.

CBase and the Cleanup Stack

In the earlier examples, when we push `x` to the cleanup stack, we are invoking the function `CleanupStack::PushL(CBase* aPtr)`, because `CX` is derived from `CBase`. When a subsequent `PopAndDestroy()` happens, this function can only call the destructor of a `CBase`-derived object.

As mentioned in Section 4.2, `CBase` has a virtual destructor. This means that any object derived from `CBase` can be pushed onto the

cleanup stack and, when it is popped and destroyed, the correct destructor function is called.

The cleanup stack and C++ destructors make it very easy for a programmer to handle cleanup. Use the cleanup stack for heap objects pointed to only by C++ automatics; use the destructor for objects pointed to by member variables. You very rarely need to use `TRAP()`. The resulting code is easy to write, compact, and efficient.

R Classes on the Cleanup Stack

Sometimes you need to create and use an R-class object as an automatic variable rather than as a member of a C class. In this case, you need to be able to push it to the cleanup stack. There are two options available:

- use `CleanupClosePushL()`, `CleanupReleasePushL()` or `CleanupDeletePushL()`
- do it directly: make a `TCleanupItem` consisting of a pointer to the R-class object, and a static function that will close it.

If you have an item with a `Close()` function, then `CleanupClosePushL()`, a global non-member function, will ensure that `Close()` is called when the item is popped and destroyed by the cleanup stack. C++ templates are used for this, so `Close()` does not have to be virtual.

The following code demonstrates how to use `CleanupClosePushL()`:

```
case ECmdDeleteFile:
{
    RFs fs;
    CleanupClosePushL(fs);
    User::LeaveIfError(fs.Connect());
    User::LeaveIfError(fs.Delete(KTextFileName));
    CleanupStack::PopAndDestroy(&fs);
}
```

You just call `CleanupClosePushL()` after you've declared your object, and before you do anything that could leave. You can then use `CleanupStack::PopAndDestroy()` to close the object when you've finished with it.

`CleanupClosePushL()` pushes a `TCleanupItem` onto the cleanup stack. A `TCleanupItem` is a general-purpose object encapsulating an object to be cleaned up and a static function that will close it. When `CleanupStack::PopAndDestroy()` is called, the cleanup operation is performed on that object. In this particular example, the object is `fs` and the cleanup operation is the function `Close()` that is called on it.

There are two other related functions:

- `CleanupReleasePushL()` works like `CleanupClosePushL()` except that it calls `Release()` instead of `Close()` during object cleanup. You should use this when your object is a resource that you are holding (rather than a handle).
- `CleanupDeletePushL()` is used for pushing heap-allocated, T-class objects onto the cleanup stack. In contrast with R-class objects, T-class objects do not normally have `Close()` or `Release()` member functions, but they do have a class destructor. Therefore, `CleanupDeletePushL()` is equivalent to the `CleanupStack::PushL(TAny*)` overload except that, `CleanupDeletePushL()` calls the class destructor. Another frequent use is when we have a pointer to an M-class object that needs to be placed on the cleanup stack.

You can also create your own `TCleanupItem` and push it onto the cleanup stack if the system-supplied cleanup functions are insufficient. The `TCleanupItem` class is defined in `e32base.h`, and contains a pointer and a cleanup function. Anything pushed to the cleanup stack is actually stored in a cleanup item. `CleanupStack::PushL(CBase*)`, `CleanupStack::PushL(TAny*)` and `CleanupClosePushL()` simply create appropriate cleanup items and push them.

4.5 Two-Phase Construction

Two-phase construction is a pattern for preventing memory leaks that would otherwise occur during the construction of complex objects. If a constructor allocates memory (e.g. for a member variable), and the allocation fails and leaves, the object being constructed is not yet on the cleanup stack, and is therefore orphaned.

We need a mechanism for separating the safe parts of construction from the unsafe parts, and delaying the unsafe parts until the object has been pushed onto the cleanup stack. This mechanism is the two-phase construction pattern.

Separating Safe from Unsafe Constructions

The following example has a class called `CY`, containing a member variable which points to a `CX`. Using conventional C++ techniques, we allocate the `CX` from the `CY` constructor:

```
class CY : public CBase
{
public:
```

```

    CY();
    ~CY();
public:
    CX* iX;
};
CY::CY()
{
    iX = new (ELeave) CX;
}
CY::~~CY()
{
    delete iX;
}

```

We can then make a call to `CX::UseL()`, using cleanup-friendly code as follows:

```

case ECmd4:
{
    CY* y = new (ELeave) CY;
    CleanupStack::PushL(y);
    y->iX->UseL();
    CleanupStack::PopAndDestroy();
}

```

In this example we have used C++ constructors in the usual way, and used the cleanup stack properly. Even so, this code isn't cleanup-safe. It makes three allocations as it runs through.

- The command handler allocates the `CY`: if this fails, everything leaves and there's no problem.
- The `CY` constructor allocates the `CX`: if this fails, the code leaves, but there is no `CY` on the cleanup stack, and the `CY` object is orphaned.
- `CX::UseL()` allocates a `TInt`: by this time, the `CY` is on the cleanup stack, and the `CX` will be looked after by `CY`'s destructor, so if this allocation fails, everything gets cleaned up nicely.

The trouble here is that the C++ constructor is called at a time when no pointer to the object itself is accessible to the program. This code:

```

CY* y = new (ELeave) CY;

```

is effectively expanded by C++ to:

```

CY* y;
CY* temp = User::AllocL(sizeof(CY)); // Allocate memory
temp->CY::CY();                      // C++ constructor
y = temp;

```

The problem is that we get no opportunity to push the `CY` onto the cleanup stack between allocating the memory for the `CY`, and the C++ constructor, which might leave. And there's nothing we can do about this.

It is a fundamental rule of Symbian OS programming that no C++ constructor may contain any functions that can leave.

To get around this, we need to provide a separate function to do any initialization that might leave. We call this function the *second-phase constructor*, and the Symbian OS convention is to call it `ConstructL()`. The second-phase constructor must only be called after the object under construction has been pushed onto the cleanup stack.

Class `CZ` is like `CY` but uses a second-phase constructor:

```
class CZ : public CBase
{
public:
    static CZ* NewL();
    static CZ* NewLC();
    void ConstructL();
    ~CZ();
public:
    CX* iX;
};

void CZ::ConstructL()
{
    iX = new (ELeave) CX;
}
```

`CZ::ConstructL()` performs the same task as `CY::CY()`, so the leaving function `new (ELeave)` is now in the second-phase constructor `ConstructL()`. This is now cleanup-safe.

Wrapping Up `ConstructL()` in `NewL()` and `NewLC()`

Working with the two-phase constructor pattern can be inconvenient, because the user of the class has to remember to call the second-phase constructor explicitly.

To make this easier and more transparent, we use the `NewL()` and `NewLC()` patterns. The `CZ` class has a static `NewLC()` function that's coded as follows:

```
CZ* CZ::NewLC()
{
    CZ* self = new (ELeave) CZ;
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}
```

Because `CZ::NewLC()` is static, you can call it without any existing instance of `CZ`. The function allocates a `CZ` with `new (ELeave) CZ` and then pushes it onto the cleanup stack so that the second-phase constructor can safely be called. If the second-phase constructor fails, then the object is popped and destroyed by the rules of the cleanup stack. If all goes well, the object is left on the cleanup stack – that’s what the `C` in `NewLC()` stands for. `NewLC()` is useful if, on returning, we want to store the reference to the new `CZ` in an automatic variable.

Often, however, we don’t want to keep the new `CZ` on the cleanup stack, and we use the `NewL()` static function. This can be coded as follows:

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

This is exactly the same as `NewLC()`, except that we pop the `CZ` object off the cleanup stack after the second phase constructor has completed successfully.

The two implementations are almost the same, and `NewL()` is commonly implemented in terms of `NewLC()`, like this:

```
CZ* CZ::NewL()
{
    CZ* self = CZ::NewLC();
    CleanupStack::Pop();
    return self;
}
```

One interesting thing to note in the `NewL()` implementation is that we have popped an item from the cleanup stack without destroying it. All we’re doing here is putting a pointer onto the cleanup stack only for as long as there is a risk of leaving. The implication is that on return from the `NewL()`, ownership of the object is passed back to the caller of `NewL()`, who then has to take responsibility for it.

`CZ::NewLC()` and `CZ::NewL()` are known as *factory functions* – static functions that act as a replacement for normal constructors.

It is good practice to make the class constructors private, in order to ensure that users always use the `NewL()` and `NewLC()` functions, thus ensuring that their code is always cleanup-safe.

Summary

In this chapter we have explained the principles of memory management and cleanup, but we could summarize all of it in a single rule:

Whenever you see something that might leave, think about what happens (a) when it doesn't leave, and (b) when it does.

In practice it's better to operate with a few safe patterns, so below is a more detailed summary of the rules we've seen.

- Always delete objects your class owns, from the class destructor.
- Don't delete objects that you don't own (those that you merely use).
- Don't allocate twice (this will cause a memory leak).
- Don't delete twice (this will corrupt the heap).
- When you delete outside of the destructor, immediately set the pointer to zero.
- When you are reallocating, you must use the sequence 'delete, set pointer to zero, allocate', just in case the allocation fails.
- Use `new (ELeave)` rather than plain `new`.
- Use `L` on the end of the name of any function that might leave.
- Use traps only where you need to – for instance, when a function can't leave and must handle errors privately.
- Push an object to the cleanup stack if (a) that object is only referred to by an automatic pointer, and (b) you are going to call a function that might leave during that object's lifetime.
- Never push a member variable to the cleanup stack – use the `iMember` naming convention to help you spot this error.
- Give all heap-based classes a name beginning with `C`, and derive them from `CBase`. Trust `CBase` to zero-initialize all data, including all pointers. Utilize `CBase`'s virtual destructor for cleanup purposes.
- Never leave from a C++ constructor.
- Never allocate memory from within a C++ constructor.
- Put construction functions that might leave into a second-phase constructor such as `ConstructL()`.

- Instead of expecting your user to call `ConstructL()` and/or a C++ constructor explicitly, use `NewL()` and `NewLC()` to wrap up allocation and construction. You can enforce this by making the C++ constructors private.

It doesn't take long to become familiar with these rules and to work with them effectively. In addition, it is easy to trap many forms of misbehavior by using the emulator debug keys provided by the UI and the heap checking provided by CONE.

5

Descriptors

One of the most common and recurring programming requirements in computing is the handling of strings. In Symbian OS, strings are handled using a set of classes known as descriptors. They offer a safe and consistent mechanism for dealing not only with strings but also with binary data.

When you encounter descriptors for the first time you may suffer culture shock and confusion at the bewildering number of descriptor classes to choose from. When you first start to experiment with them your code may not compile when you think it should; or it may panic when executed; or you may not get the behavior you were expecting. Frustration can easily set in and it may be tempting to use standard C string-handling functionality instead. Don't. Descriptors are one of the most fundamental and heavily used entities in Symbian OS and it is essential to become familiar with them as their use is unavoidable, even if developing the simplest of native Symbian OS applications.

This chapter introduces you to the various types of descriptors and provides guidance on their usage.

5.1 Overview

Symbian OS was based upon an earlier operating system known as EPOC16 (also known as SIBO) which was written in C and used standard C-like string-handling mechanisms. The development of EPOC16 was often held up for days as obscure defects and system lockups had to be debugged. In many cases, the root cause of these defects was as a consequence of memory overruns. When Symbian OS was being designed, it was deemed vitally important that it would not suffer from the same problems and a decision was made to impose a systematic low-level solution – the outcome being the creation of the descriptor classes which were designed to offer protection against memory overruns.

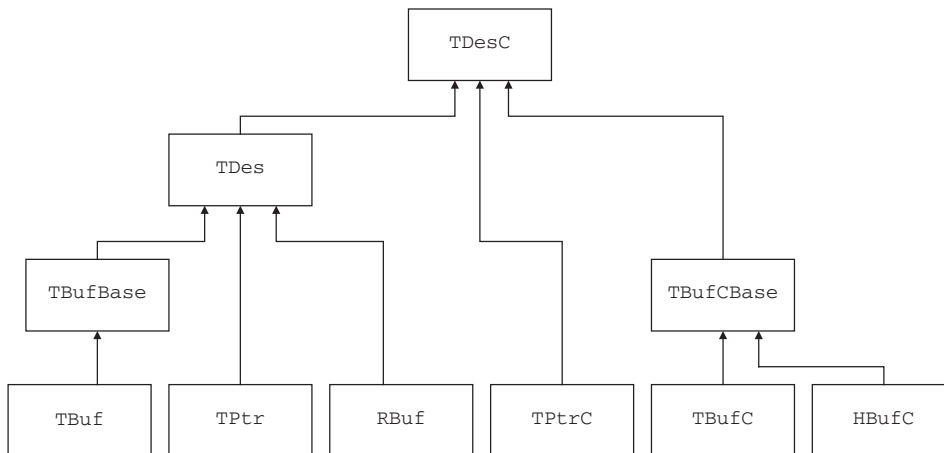


Figure 5.1 Descriptor hierarchy

The descriptor classes inherit from the base class `TDesC`, as shown in Figure 5.1, and they can be classified according to how their data is stored and accessed:

- stack descriptors store their data as part of themselves
- pointer descriptors point to data stored elsewhere
- heap descriptors store their data in the heap.

In addition there is a related type known as a literal. You frequently see literals referred to as literal descriptors; because they are stored in the program-binary code for an executable, they are also sometimes referred to as program-binary descriptors.

Literals aren't actually descriptors as they do not inherit from the base descriptor class `TDesC`; however they are closely related and have an `operator()()` method for converting them to a descriptor.

There are four 'abstract' classes: `TDesC`, `TDes`, `TBufBase` and `TBufCBase`. These are not abstract in the typical C++ sense, as descriptors do not have pure virtual methods; however, they are abstract in the sense that they are not intended to be instantiated.

`TBufCBase` and `TBufBase` are an implementation convenience and are not discussed further here. There are six concrete descriptor types: `TBuf`, `TPtr`, `RBuf`, `TPtrC`, `TBufC` and `HBufC`, which, along with the literal type, can be instantiated and used directly in your programs to store or point to your string or binary data.

Table 5.1 shows these types along with the approximate standard C equivalent for the benefit of readers familiar with string handling in that language.

Table 5.1 Descriptor types

Type	Constness	Name	Approximate C equivalent
Literal	Not modifiable	TLitC	static const char []
Stack	Modifiable	TBuf	char []
	Not directly modifiable	TBufC	const char []
Pointer	Modifiable	TPtr	char* (pointing to non-owned data)
	Not modifiable	TPtrC	const char* (pointing to non-owned data)
Heap	Modifiable	RBuf	char* (pointing to owned data)
	Not directly modifiable	HBuFC	const char* (pointing to owned data)

The C in the type name stands for ‘constant’, however the term does not mean ‘constant’ in the usual C++ meaning of the term. TBuf, TPtr and RBuf are all directly modifiable, while TLitC and TPtrC are not modifiable at all, however TBufC and HBuFC are not directly modifiable but they are indirectly modifiable.

Descriptors were designed to be as efficient as possible and a manifestation of this is in TBufC and HBuFC’s apparently contradictory ‘constant yet modifiable’ nature.

If you need a truly constant descriptor then use TLitC, TPtrC or TBufC as appropriate according to your intended usage. If you need a modifiable descriptor then choose between a TPtr, TBuf, RBuf or HBuFC.

From an efficiency point of view, there are basically three points at which a decision can be made:

- deciding whether to use a stack or heap descriptor to store the data (pointer descriptors do not store any data)
- if using a heap descriptor, deciding whether to use an HBuFC or RBuf
- if using a writable stack descriptor, deciding whether to use the directly-writable TBuf or the indirectly-writable TBufC.

It is regarding this last choice that the constness or efficiency of the descriptor classes come into play. Perhaps the simplest and best advice

regarding these aspects is: unless you are using large numbers of stack descriptors, do not worry about whether a TBufC or TBuf is more efficient.

Here's a preview of using descriptors, with a variation on a traditional example: the string 'Hello World!' is displayed seven times to the text console:

```
#include <e32base.h>
#include <e32cons.h>

void HelloWorldL(void)
{
    // Create a literal descriptor
    _LIT(KTxtHelloWorld, "Hello World!");

    const TInt KHelloWorldLength = 12;

    // create a TBuf
    TBuf<KHelloWorldLength> tbuf(KTxtHelloWorld);

    // create a TBufC
    TBufC<KHelloWorldLength> tbufc(KTxtHelloWorld);

    // create an HBufC
    HBufC* hbufc = KTxtHelloWorld().AllocLC();

    // Create an RBuf
    RBuf rbuf;
    rbuf.CreateL(KHelloWorldLength, KTxtHelloWorld);
    rbuf.CleanupClosePushL();

    // create a TPtrC
    TPtrC tptrc(tbufc);

    // create a TPtr
    TPtr tptr = hbufc->Des();

    // display "Hello World!" 7 times to the text console
    CConsoleBase* console = Console::NewL(KTxtHelloWorld,
        TSize(KConsFullScreen, KConsFullScreen));
    console->Printf(KTxtHelloWorld);
    console->Printf(tbuf);
    console->Printf(tbufc);
    console->Printf(tptr);
    console->Printf(tptrc);
    console->Printf(*hbufc);
    console->Printf(rbuf);
    console->Getch(); // pause until user enters input
    CleanupStack::PopAndDestroy(2);
    delete console;
}

GLDEF_C TInt E32Main()
{
    static CTrapCleanup* cleanup = CTrapCleanup::New();
```

```
TRAPD(ret, HelloWorldL());
delete cleanup;
return (0);
}
```

5.2 Anatomy of Descriptors

Before describing how to use the descriptors, we take a look at their internal structure as this helps in understanding their differences.

As we have already seen, all the descriptors derive from the abstract base class `TDesC`. This class defines two property values: one for storing the type of the descriptor-memory layout (stored in half a byte) and one for storing the length of the descriptor (stored in three and a half bytes¹). `TDes` inherits from `TDesC` and additionally stores the maximum length of the data that is stored or pointed to by the descriptor.

As indicated in Table 5.2, the type field is not used to store the actual type of the descriptor, instead it is used to store the type of memory

Table 5.2 Types of descriptor layout

Type	Descriptors	Meaning
0	HBufC, TBufC, TLitC	Array-type storage without maximum length
1	TPtrC	Pointer-type storage without maximum length
2	TPtr, RBuf	Pointer-type storage with maximum length, pointing to data
3	TBuf	Array-type storage with maximum length
4	TPtr, RBuf	Pointer-type storage with maximum length, pointing to a descriptor.

¹ As the type and the length are stored as bit-fields, if you are visually inspecting a descriptor object's length in an IDE, you must take this into consideration. For example, with the following code:

```
_LIT(KTxtHello, "Hello");
TBuf<5> buf(KTxtHello);
```

The value of the `buf` object's `iLength` data member (where the type and length information is held) is 805306373. In binary this is 00110000000000000000000000000000101 – 0011 represents the type and 101 the length.

layout of the descriptor, determining how and where the descriptor data is stored and whether the maximum length is stored. This allows concrete descriptors to be passed as TDesC or TDes parameters without the overhead of using virtual methods because the TDesC::Ptr() method uses a switch statement to identify the type of descriptor and thus knows where the data is located. This requires that TDesC::Ptr() has hard-coded knowledge of the memory layout of all its subclasses which consequently means that you can't create your own descriptor class deriving from TDesC. The actual descriptor memory layouts are shown in Figure 5.2.

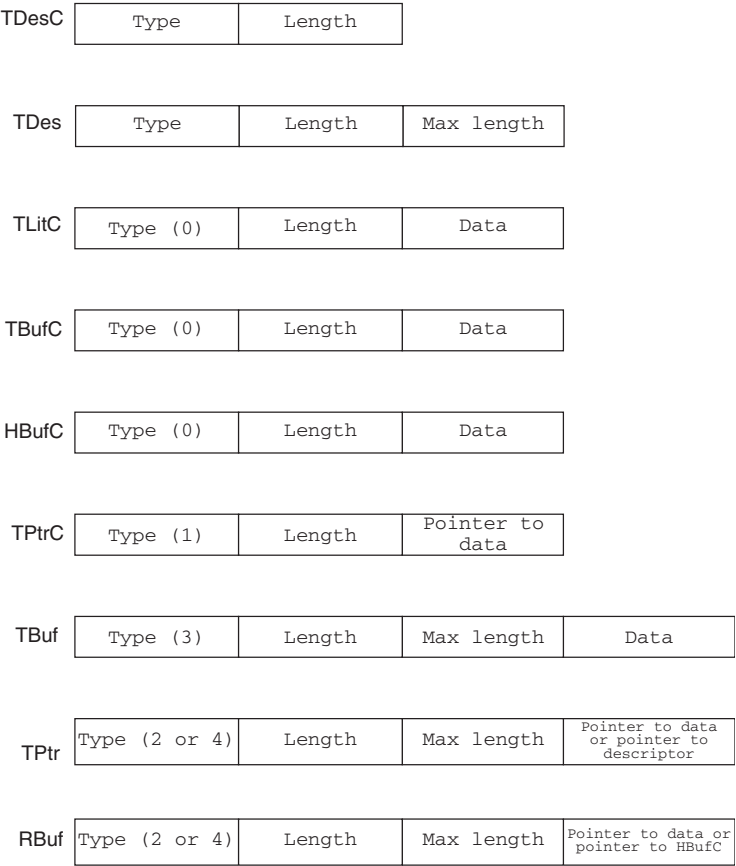


Figure 5.2 Descriptor memory layouts

The length field is used to store the length of the data contained or pointed to by the descriptor. This removes the need for the descriptor data to be delimited with a '\0' character as it is in C, a consequence of which is that it allows binary data to be stored as well as string data. It also means that data beyond the bounds of the descriptor cannot be accessed by mistake.

The maximum length field is used to store the maximum capacity of the data which can be held. This is used to prevent memory overruns. When an attempt is made to write or access data beyond a descriptor's boundary, a User-11 panic is generated.

5.3 Literals

We look at literals first because they are used in most of the code examples within this chapter.

A program binary consists of the compiled and linked DLLs and EXEs that make up a program. A literal string is one which is included in the program binary. As a program binary is executed either directly from ROM or from read-only RAM, any literals it contains are also read-only.

In standard C, a string literal would be included in the program binary by declaring the variable as a static constant array and initializing it with character data:

```
static const char KTxtHelloWorld[] = "Hello World!";
```

In Symbian OS, literal strings are included in the program binary using the `TLitC` class, however you never actually use this class directly, instead you use it indirectly via the macro `_LIT` as follows:

```
_LIT(KTxtHelloWorld, "Hello World!");
```

This adds the string literal 'Hello World!' of type `TLitC` to the program binary and associates the symbol `KTxtHelloWorld` with it so that it can be referenced.

The binary layout of `TLitC` is designed to be identical to a `TBufC` (Figure 5.2), which allows a `TLitC` to be treated as a descriptor.

5.4 Stack Descriptors

There are two stack descriptors: `TBuf` and `TBufC` can be used in a similar way to that in which a `char []` or `const char []` array would be used in C. The descriptor data is stored within the descriptor objects as part of their member data. As their name suggests, they are typically declared on the stack (they can be allocated on the heap, but read Section 5.11 before doing this).

There are a number of ways they can be created; here they are constructed from a literal:

```
_LIT(KTxtHello, "Hello");
TBuf<11> buf(KTxtHello);    // equivalent to char buf[12]2
TBufC<5> bufC(KTxtHello);   // equivalent to const char bufC[6]
```

The classes are a templated array and take an integer value that defines the maximum length of data that the descriptor can hold. In this case the `buf` variable can store a maximum of 11 items and the `bufC` variable a maximum of five. The contents of the `KTxtHello` literal are physically copied into each descriptor’s array space.

In both these examples, there is sufficient space for the data to be copied, but if `KTxtHello` had instead contained ‘Hello!’ and thus had been of length six, then there would be a panic during the construction of the `bufC` variable.

The `TBufC<5>` object layout is shown in Figure 5.3. The template parameter value defines the length of the data area available; the length of the data copied is also five; it could have been less than five, but definitely not more than five.

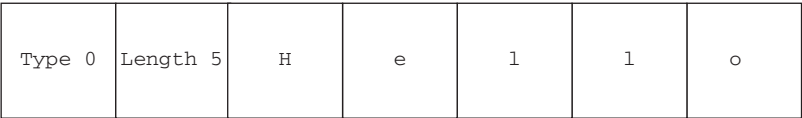


Figure 5.3 TBufC<5> layout

Compare this with the layout for the `TBuf<11>` object in Figure 5.4. The length is still five but the maximum length is eleven and there is room to insert, or append, another six items.



Figure 5.4 TBuf<11> layout

Once you have a `TBuf` object you can use a variety of methods to manipulate its content. The following code illustrates appending text to the `buf` variable created above:

```
_LIT(KTxtSpace, " ");
_LIT(KTxtWorld, "World");
buf.Append(KTxtSpace); // buf contains "Hello " and is 6 long
buf.Append(KTxtWorld); // buf contains "Hello World" and is 11
bufC.Append(KTxtSpace); // Error: Append() is not a TbufC method
```

² Notice how, in the equivalent C code, the length of the array is 1 more than in the corresponding descriptor code. This is because C strings need to store the NULL terminator.

It is not possible to `Append()` data to the `bufC` variable because `TBufC` objects cannot be directly manipulated (however, there is an indirect way which is described in Section 5.5).

Once the `buf` variable contains the data 'Hello World', then both its length and maximum length data members are 11 and thus it is not possible to append any more data to it:

```
_LIT(KTxtExclamation, "!");
buf.Append(KTxtExclamation); // buf's length cannot exceed 11
                             // so the code will panic
```

See Section 5.12 for a full list of methods for manipulating and accessing descriptor data.

As already mentioned, stack descriptors are usually declared on the stack; often they are embedded in other classes. Allocating them directly on the heap is possible though not recommended for the reasons outlined in Section 5.11. A common situation where a `TBuf` is used is when you need to construct a descriptor, use it, and then discard it all within one method and you do not want to bother with memory allocation. `TBufC` is usually embedded in other objects.³

A `TBufC` uses four bytes less RAM than a `TBuf` (as it doesn't store a maximum length), therefore using a `TBufC` saves four bytes of storage per object. Even if you need to modify the contents, you can still use a `TBufC` as it can be modified via its `Des()` method; however a call to `Des()` takes up space in ROM.

If you are using only a small number of stack descriptors then the recommendation is to simply use whichever is easiest to use to suit your purposes. If you are using many of them then decide if it is more desirable to save ROM space or RAM space.

5.5 Pointer Descriptors

While `TBuf` and `TBufC` are implemented as templated arrays and contain a physical copy of data, `TPtr` and `TPtrC` instead contain a pointer to data and can be used in similar circumstances to those in which `char*` or `const char*` are used in C.

A pointer descriptor object can either go onto the program stack or it can be a data member of some class, but the data pointed to can be anywhere that is addressable by your code. If a pointer descriptor points

³ Some people are confused about whether a `TBuf` or `TBufC` declared as a data member inside a class is on the heap or the stack – if they are data members of a `CBase`-derived class then they are allocated on the heap because all `CBase`-derived classes are, by definition, allocated on the heap.

to data allocated in the heap, it is not responsible for the allocation or de-allocation of that memory. You must of course ensure that any data the pointer descriptor is pointing to exists for the duration of the pointer descriptor.

TPtrC Descriptors

A TPtrC is similar to a `const char*` in C. It can be set to point to many things: descriptor data, portions of descriptor data, raw memory, or a literal string located in ROM. Here are some examples (the first of which is illustrated in Figure 5.5):

```
// 1 Construction of a TPtrC from a literal
_LIT(KTxtHello, "Hello");
TPtrC ptr(KTxtHello); // ptr points to "Hello" in the program binary

// 2 Setting a TPtrC to point to data in a stack descriptor
_LIT(KTxtWorld, "World");
TBufC<5> buf(KTxtWorld);
ptr.Set(buf); // ptr points to "World" in the buf descriptor variable

// 3 TPtrC set to point to raw data held in a C-style array
const TText array[6] = L"Hello";
ptr.Set(array, 5); // ptr points to "Hello" in the array variable

// 4 TPtrC set to point to a portion of data held in a
// descriptor
ptr.Set(buf.Mid(1,2)); // ptr points to "or" in the buf
                        // descriptor variable
```

Code example 3 above shows how TPtrC is useful if you have some data that is not in the form of a descriptor, but you have to pass that data to a method expecting a descriptor argument – after being set, the `ptr` variable can be passed directly to the method.

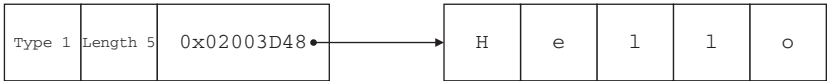


Figure 5.5 TPtrC layout

Type 2 TPtr Descriptors

TPtr is a modifiable pointer descriptor class allowing access to, and modification of, the character or binary data it points to. There are two flavors of TPtr: type 2 points to *data* (which may or may not be held within a descriptor) and type 4 points to a *descriptor*.

All `TPtr`s are type 2 unless they are created using the `HBufC::Des()` or `TBufC::Des()` methods. In practice, type 2 `TPtr`s are not used particularly often and there are not many good real-world examples of their usage within the Symbian OS code base. An example usage would be if you wanted to modify the contents of raw memory or data as a descriptor. The following code shows how a type 2 `TPtr` can be used to point to non-descriptor data and then write to that data as if it were a descriptor:

```
TText cStyleArray[] = L"Hello";4

// Creates a pointer to the array of data. The 2nd parameter is
// the length of the data pointed to and the 3rd parameter is
// the maximum length of the data pointed to.
// sizeof(cStyleArray) returns 12 which is the number of bytes,
// however the conceptual length of the data is 6, hence the
// division by 2.
TPtr ptrToRawData(cStyleArray, sizeof(cStyleArray)/2,
                  sizeof(cStyleArray)/2);

// copy "World" into the cStyleArray memory location
_LIT(KTxtWorld, "World");
ptrToRawData = KTxtWorld; // cStyleArray now contains "World"

// TPtrs provide safety when writing to raw memory as an attempt
// to write beyond the end of the data storage area results in
// a panic
ptrToRawData += KTxtWorld; // this will panic!
```

In the above example, the `TPtr` is created using the constructor `TPtr::TPtr(TUint16 *aBuf, TInt aLength, TInt aMaxLength)` which sets the length and maximum length (see Figure 5.6).⁵



Figure 5.6 `TPtr` (type 2) pointing to raw data ‘Hello’

Type 4 `TPtr` Descriptors

While a type 2 `TPtr` points to data, a type 4 `TPtr` points to another descriptor. A type 4 `TPtr` can only be created by calling `HBufC::Des()`

⁴ `L` indicates to the compiler that “Hello” should be treated as a wide string, do not confuse it with `_L`.

⁵ There is also a constructor that doesn’t take the length as a parameter: `TPtr::TPtr(TUint16 *aBuf, TInt aMaxLength)`. If you use this, be aware that the length of the descriptor is 0 after construction.

or `TBufC::Des()`. It is used to indirectly modify the contents of these otherwise constant descriptors.

```
_LIT(KTxtHello, "Hello");
_LIT(KTxtString, "Overwrite me");
HBufC* hbuf = KTxtString().AllocL();
// hbuf contains "Overwrite me"
TPtr ptr = hbuf->Des(); // points to the hbuf contents
ptr = KTxtHello;        // hbuf now contains "Hello"
delete hbuf;
```

This code copies the string 'Hello' into the `hbuf` variable. Changes to the length of a type 4 `TPtr` also changes the length of the `HBufC` it points to, so in the example above, after the assignment, the length of both `hbuf` and `ptr` is five (see Figure 5.7).

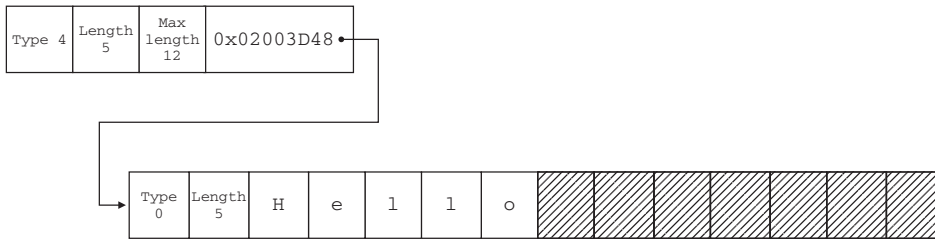


Figure 5.7 `TPtr` (type 4) pointing to an `HBufC` descriptor

Be aware that if you have more than one type 4 `TPtr` pointing to the same `HBufC` or `TBufC`, then modifications made via one are not reflected in the others.

If you are using `TPtr`'s assignment operator there is some unexpected behavior that you need to be aware of – see Section 5.11.

Using `TPtr::Set()`

`TPtr` and `TPtrC` both have a `Set()` method for changing what they point to:

```
_LIT(KHello, "Hello");
_LIT(KWorld, "World");
TPtrC hello(KHello); // contains "Hello"
TPtrC world(KWorld); // contains "World"
world.Set(hello);    // world now contains "Hello"
```

Note that if you wanted to indicate that the data the `TPtrC` points to should not be changed then you can declare it to be `const`. Doing

so typically generates a compiler warning if an attempt is made to call `Set()`.

If you have a `TPtr` (of either type) that you wish to declare now but initialize with data later then the `Set()` method is useful:

```
TPtr ptr(NULL, 0);
...
ptr.Set(hbuf->Des());
```

It is also useful if the `TPtr` is a member of a class:

```
CSomeClass::CSomeClass() : iPtr(NULL, 0)
{
}
...
iPtr.Set(hbuf->Des());
```

5.6 Heap Descriptors

The heap descriptors are `HBufC` and `RBuf` and their data is stored on the heap. They are responsible for the allocation and de-allocation of this memory. The heap descriptors are used in some situations where `malloc()` is used in C.

The `RBuf` descriptor is new and was introduced into Symbian OS v8.0 (it is documented in the Symbian Developer Library from v8.1) and so may not be present if you are using an older SDK.

These types are useful when the amount of data to be held by the descriptor cannot be determined until run time or if a large amount of data is required to be used as a local variable⁶ (the stack size is limited in Symbian OS and you should always strive to minimize its usage).

With both `RBuf` and `HBufC` descriptors, if the heap size needs to expand, this is not done automatically and you must dynamically resize the descriptor.

With both `HBufC` and `RBuf` descriptors, you are responsible for memory management so always be aware of cleanup issues to ensure that there is no risk of memory leaks. `HBufCs` are placed on the cleanup stack using an overload of `CleanupStack::PushL()`, while `RBufs` have their own method for pushing to the cleanup stack: `RBuf::CleanupClosePushL()`.

⁶ If you have a very large amount of data that you need to hold and manipulate in memory, consider using the `CBufSeg` class. With `HBufC` and `RBuf`, the data is held as one contiguous memory cell within the heap; with a `CBufSeg`, it is held as individual cells linked together. See the SDK for further details.

HBuFC Descriptors

An HBuFC can be created as follows:

```
_LIT(KTxtHello, "Hello");
HBuFC* buffer = HBuFC::NewL(32);
*buffer = KTxtHello;
delete buffer;
```

This creates a memory cell on the heap with a maximum size of at least 32 and then copies the string 'Hello' into it.⁷

In C, the equivalent code would be:

```
static char KTxtHelloWorld[] = "Hello";
char* buffer = (char*)malloc(32);
strcpy(buffer, KTxtHelloWorld);
free buffer;
```

The stack descriptors, TBuf and TBufC, are templated, and the template value determines the length of the data area. If you try to assign data that is longer than the template value, the descriptor protects itself by panicking. The HBuFC descriptor is slightly different. It is allocated as a single cell from the heap (see Figure 5.8).

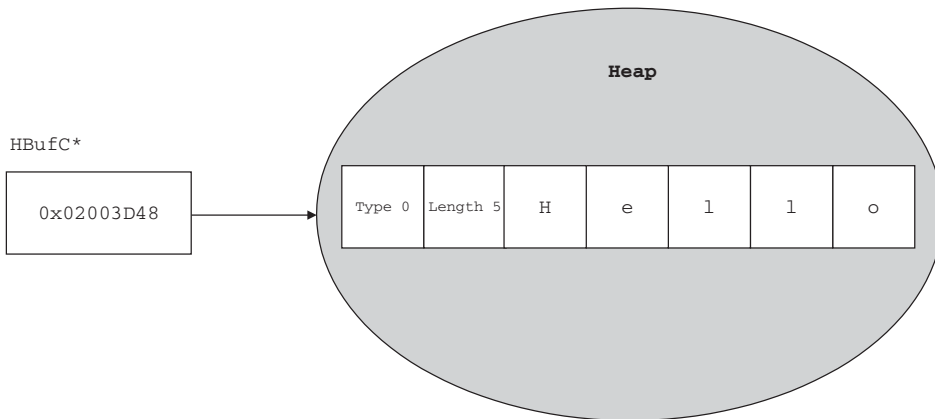


Figure 5.8 HBuFC layout

To save memory, it does not store its maximum length; instead, it finds out its maximum length by finding out the size of the heap cell it is stored in. Heap cells have a granularity that depends on the specific hardware to

⁷ For illustrative purposes this code is not as efficient as it could be, instead `HBuFC* buffer = KTxtHello().AllocL()` could have been used; see the section on creating an HBuFC from another descriptor.

which Symbian OS has been ported. When allocating memory, Symbian OS always rounds up to the next nearest granularity boundary, so if you create an `HBufC` of a specific length, you may end up with a bigger descriptor than you asked for. This becomes important in cases where, for example, you pass the descriptor to some method whose purpose is to return data in that descriptor. Do not assume that the length of data returned in the descriptor is the same as the length you used to allocate the descriptor.

As you can see from the code fragment, you need a pointer to an `HBufC` type in your code; this can go onto the program stack or it can be a data member of some class. An `HBufC` is created via one of its overloaded `NewL()` or `NewLC()` methods, in a similar way to `CBase`-derived objects.

The `Des()` method

You can't directly modify an `HBufC` type; however, you can use the `Des()` method to create a `TPtr` through which you can change the `HBufC`. An example is given in Section 5.5.

There are some common misuses of the `Des()` method – see Section 5.11).

Creating an `HBufC` from another descriptor

Programs often need to create an `HBufC` from an existing descriptor. For example, you might want to pass data into a method and then do some processing on that data. Typically you do not know the length of the data being passed to your method, and so the easiest way to handle this is to create an `HBufC` descriptor within your method. We use the descriptor method `Alloc()` or, more usually, one of its variants: `AllocL()` and `AllocLC()`. Here's a simple example:

```
_LIT(KTxtHelloWorld, "Hello World!");

myFunctionL(KTxtHelloWorld);
void myFunctionL(const TDesC& aBuffer)
{
    HBufC* myData = aBuffer.AllocLC();
    ...
}
```

`AllocL()` (and the `Alloc()` and `AllocLC()` variants) creates an `HBufC` that is big enough to contain `aBuffer`'s data and then copies the data from `aBuffer` into `myData`. `AllocL()` does three jobs in one: it works out the length of the source descriptor, allocates the heap descriptor, and then copies the data into the heap descriptor.

Changing the size of an *HBufC*

Heap descriptors can be created dynamically to whatever size is required. However, they are not automatically resized when additional storage space is required, so you must ensure the buffer has sufficient memory before calling a modification operation. If necessary, you can reallocate the buffer to expand it to a new size using one of the `ReAllocL()` methods. This may change the location of the heap cell containing the descriptor data in memory, so if there is a `TPtr` pointing to the *HBufC* it must be reset (see Section 5.11).

RBuf Descriptors

An *RBuf* is similar to an *HBufC* in that it can be created dynamically and its data is stored on the heap. However, as it derives from *TDes*, it has the benefit that it is directly modifiable, which means you do not have to create a `TPtr` around the data in order to modify it. Thus *RBuf* is preferable to *HBufC* when you know you need to dynamically allocate a descriptor and modify it.

An *RBuf* can create its own buffer. Alternatively, it can take ownership of a pre-allocated *HBufC* or a pre-allocated section of memory. The memory must be freed by calling the `Close()` method. Note that *RBuf* does not automatically reallocate its memory cell if it needs to expand so, as with an *HBufC*, you still need to manage the memory for operations that may need the descriptor to expand.

*RBuf*s are not named *HBuf*, because *HBufC* objects are directly allocated on the heap (H stands for heap⁸); *RBuf* objects are not – the R prefix indicates that the class owns and manages its own memory resources. Unlike *HBufC* objects, *RBuf* objects can have their maximum length set to the exact number you specify.



Figure 5.9 *RBuf* (type 2) layout

There are two types of *RBuf* object, as illustrated in Figures 5.9 and 5.10, one that points to a memory area and one that points to an *HBufC*:

```
_LIT(KTxtHello, "Hello");
RBuf type2RBuf;
type2RBuf.CreateL(5);
```

⁸ *HBufC* is not called *CBufC*, as it doesn't inherit from *CBase*. It is a notable exception to the Symbian OS naming convention for classes.

```

type2RBuf = KTxtHello;
type2RBuf.Close();

HBufC* hBuf = KTxtHello().AllocLC();
RBuf type4RBuf(hBuf); // assign ownership of hBuf to type4RBuf
CleanupStack::Pop(hBuf); // remove hBuf from the cleanup stack
type4RBuf.Close(); // delete hBuf

```

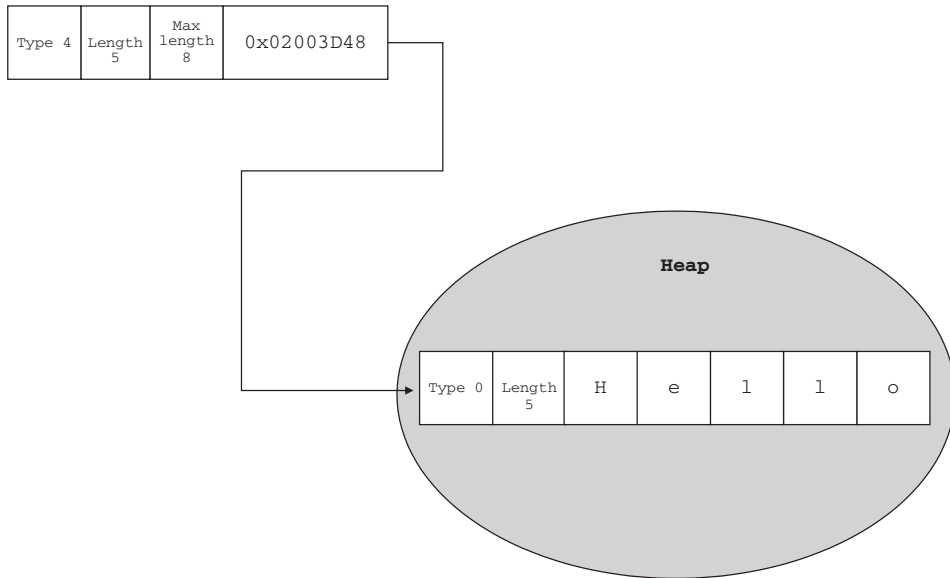


Figure 5.10 RBuf (type 4) layout

Construction

On construction, an RBuf object can allocate its own buffer or take ownership of a pre-existing section of allocated memory or a pre-existing heap descriptor.

For example, to construct an RBuf with a zero length, resizable buffer that owns no allocated memory, use:

```
RBuf myRBuf;
```

To have an RBuf take ownership of an existing heap descriptor, use:

```

HBufC* myHBufC = HBufC.NewL(20);
RBuf myRBuf (myHBufC);

```

In practice, your code may appear more along the lines of:

```
RBuf resourceString (iEikEnv->AllocReadResourceL (someResourceId));
```


A final option allows the contents of an RBuf's data to be set from a stream which contains the length of the data followed by the data itself. Both the stream and the maximum permitted size of the stream are specified as parameters to the `CreateL()` method – there is no `Create()` method available for setting an RBuf from a stream:

```
RBuf myRBuf;
RReadStream myStream;
TInt maxSizeOfData = 30;
myRBuf.CreateL(myStream, maxSizeOfData);
```

Note that `Create()` orphans any data already owned by the RBuf, so `Close()` should be called where appropriate to avoid memory leaks.

Assignment

An RBuf can take ownership of a pre-allocated heap descriptor, use:

```
HBufC* myHBufC = HBufC::NewL(20);
RBuf myRBuf.Assign(myHBufC);
```

To transfer ownership of another RBuf, the RBuf is specified as the parameter to the `Assign()` method:

```
RBuf myRBuf1;
RBuf myRBuf2;
HBufC* myHBufC = HBufC::NewL(20);
myRBuf1.Assign(myHBufC); // take ownership of heap memory
myRBuf2.Assign(myRBuf1); // take ownership of another RBuf
myRBuf2.Close();
```

Finally, the RBuf can take ownership of pre-allocated memory on the heap by specifying the heap cell and the maximum size of the data:

```
TInt maxSizeOfData = 20;
RBuf myRBuf;
TUint16* pointer = static_cast<TUint16*>(User::AllocL (maxSizeOfData*2));
myRBuf.Assign(pointer, maxSizeOfData);
```

For this call, the current size of the descriptor is set to zero. To specify a different size, insert this value as the second parameter. As an example, the last line of the code snippet above could be replaced by:

```
TInt currentSizeOfData = maxSizeOfData / 2;
myRBuf.Assign(pointer, currentSizeOfData, maxSizeOfData);
```

Note that `Assign()` orphans any data already owned by the `RBuf`, so `Close()` should be called where appropriate to avoid memory leaks.

The `Swap()` method allows the contents of two `RBufs` to be exchanged.

Reallocation

Having created an `RBuf`, the data space in the descriptor can be resized if the data should exceed the size of the descriptor. This is achieved by using the `ReAlloc()` method:

```
myRBuf.CleanupClosePushL();
...
const TInt newLength = myRBuf.Length() + appendBuf.Length();
if (myRBuf.MaxLength() < newLength)
{
    myRBuf.ReAlloc(newLength);
}
myRBuf.Append(appendBuf);
...
CleanupStack::PopAndDestroy(); // calls myRBuf.Close();
```

If the `ReAlloc()` method is used on an `HBufC` object, the change in location of the heap cell means that any associated `HBufC*` and `TPtr` variables need to be updated. This update isn't required for `RBuf` objects.

A corresponding `ReAllocL()` method is available that acts similarly but leaves on failure.

Destruction

Regardless of the way in which the buffer has been allocated, the `RBuf` object is responsible for freeing memory when the object itself is closed. Both the `Close()` and the `CleanupClosePushL()` methods are available and should be called appropriately. See the example for `ReAlloc()` above.

Other methods

We have discussed the methods introduced in the `RBuf` class. Since `RBuf` inherits from `TDes`, all the methods from `TDes` and `TDesC` are available to `RBuf` and can be used as described in the Symbian Developer Library documentation.

```
_LIT(KTextHello, "Hello");
_LIT(KTextWorld, " World");
RBuf myRBuf;
myRBuf.CleanupClosePushL();
myRBuf.CreateL(KHello());
```

```

iMyRBuf.ReAllocL(KHello().Length() + KWorld().Length());
myRBuf.Append(KWorld);
CleanupStack::PopAndDestroy() // calls myRBuf.Close();

```

Migration from HBufC

It is desirable to migrate code that uses HBufC* and TPtr so that it uses just RBuf.

- It makes the code easier to read and understand and hence to maintain.
- It reduces the possibility of errors.
- The object code is slightly smaller as a TPtr doesn't have to be created around the object in order to modify it, as it does with an HBufC.

Since it is possible to create an RBuf from an existing HBufC, it's easy to move code over to this new class. For example, when Symbian was migrating code internally, the changes were mainly from code of the form:

```

HBufC* mySocketName;
...
if (mySocketName==NULL)
{
    mySocketName = HBufC::NewL(KMaxName);
}
TPtr socketNamePtr (mySocketName->Des());
aMessage.ReadL(aMessage.Ptr0(), socketNamePtr);
// where aMessage is of type RMessage2

```

to code like this:

```

RBuf mySocketName;
...
if (mySocketName.IsNull())
{
    mySocketName.CreateL(KMaxName);
}
aMessage.ReadL(aMessage.Ptr0(), mySocketName);

```

An RBuf can directly replace an HBufC, so you can use it to call predefined APIs that return HBufC s. For example:

```

HBufC* resString = iEikonEnv->AllocReadResourceLC (someResourceId);

// increase size of label, copy old text and append new text
labelLength += 4;

```

```
HBufC* label = HBufC::NewLC(labelLength);
TPtr labelPtr(label->Des());
labelPtr.Copy(*resString);
labelPtr.Append(_L("-Foo"));
//... etc.
SetLabelL(ELabel, *label);
CleanupStack::PopAndDestroy(2);
```

converts to:

```
RBuf resString (iEikonEnv->AllocReadResourceL(someResourceId));
resString.CleanupClosePushL();
// Use modifiable descriptor to append new text...
resString.ReAllocL(resString.Length() + 4);
resString.Append(_L("-Foo"));
//... etc.
SetLabelL(ELabel, resString);
CleanupStack::PopAndDestroy() // calls resString.Close();
```

5.7 Narrow, Wide and Neutral Descriptors

So far the descriptors we have seen are known as neutral descriptors. They are neutral in that their class names do not indicate if they store wide or narrow data. There are actually two sets of descriptor classes: a set of narrow classes for storing 8-bit data and a set of wide classes for storing 16-bit data. Their names end in either 8 or 16 to explicitly indicate what type of data they store, narrow or wide.

If you specifically need to use narrow or wide data, you can use the narrow or wide descriptor classes directly instead of the neutral descriptors.

So are the neutral descriptors wide or narrow? The answer is that they may be either depending upon a compile-time flag. If you look in the `e32std.h` header file, you'll be able to see the `typedef`s and the build flag, the `_UNICODE` macro. Here is how it is done for `TDesC` and `TBufC`:

```
#if defined(_UNICODE)
    typedef TDesC16 TDesC;
#else
    typedef TDesC8 TDesC;
#endif
template <TInt S>
#ifdef _UNICODE
    class TBufC : public TBufCBase16
#else
    class TBufC : public TBufCBase8
#endif
```

Here is a list of the neutral, narrow and wide classes involved in this scheme.

Neutral	Narrow	Wide
<code>_LIT</code>	<code>_LIT8</code>	<code>_LIT16</code>
<code>TDesC</code>	<code>TDesC8</code>	<code>TDesC16</code>
<code>TDes</code>	<code>TDes8</code>	<code>TDes16</code>
<code>TPtrC</code>	<code>TPtrC8</code>	<code>TPtrC16</code>
<code>TPtr</code>	<code>TPtr8</code>	<code>TPtr16</code>
<code>TBufC</code>	<code>TBufC8</code>	<code>TBufC16</code>
<code>TBuf</code>	<code>TBuf8</code>	<code>TBuf16</code>
<code>HBufC</code>	<code>HBufC8</code>	<code>HBufC16</code>
<code>RBuf</code>	<code>RBuf8</code>	<code>RBuf16</code>

You can find these definitions in `e32def.h`, `e32std.h`, `e32des8.h` and `e32des16.h`.

Symbian OS was designed from the beginning to support worldwide locales and although early versions used narrow 8-bit characters, its architects planned for 16-bit characters from the beginning. It was decided to concentrate effort first on developing Symbian OS with narrow strings and in the future it would be changed to deal with wide strings. The technical foundation of the Symbian OS strategy was to program using the neutral versions of descriptors, then by simply changing the compilation flag, it would be possible to rebuild Symbian OS with 16-bit or 8-bit characters.

Early versions of Symbian OS, up to and including v5.0, used narrow characters. The next version, v5.0U (the U stands for Unicode) and subsequent versions used wide characters. Although the 16-bit descriptors are called 'Unicode', they are in fact UTF-16, which is an encoding of Unicode.

Today, Symbian OS is always built using 16-bit wide characters but it is conventional to use the neutral classes unless there is an instance where you need to explicitly use a narrow or wide descriptor. Code all your general-purpose text objects to use the neutral variants. Where you are using descriptors to refer to binary data, code specifically using 8-bit classes, for example, `TDesC8`.

5.8 Descriptors and Binary Data

You can use descriptors to store and manipulate binary data just as you can string data. The main reason you can do this is because descriptors include an explicit length, rather than using `NULL` as a terminator.

The data APIs and string APIs offered by descriptors are almost identical.

- Use `TDesC8`, `TDes8`, `TBuf8`, `TPtr8`, etc. rather than their equivalent `TDesC`, `TDes`, `TBuf` and `TPtr` character classes.
- Use `TUInt8` or `TInt8` for bytes, rather than `TText`.
- Locale-sensitive character-related methods, such as `FindF()` and `CompareF()`, aren't useful for binary data.
- You can use `Size()` to get the number of bytes in a descriptor, whether it's an 8-bit or 16-bit type.

APIs for writing and reading, or sending and receiving, are often specified in terms of binary data. Here, for example, is part of the `RFile` API:

```
class RFile : public RFsBase
{
public:
    IMPORT_C TInt Open(RFs& aFs, const TDesC& aName, TInt aFileMode);
    IMPORT_C TInt Create(RFs& aFs, const TDesC& aName, TInt aFileMode);
    IMPORT_C TInt Replace(RFs& aFs, const TDesC& aName, TInt aFileMode);
    IMPORT_C TInt Temp(RFs& aFs, const TDesC& aPath,
                      TFileName& aName, TInt aFileMode);
    IMPORT_C TInt Write(const TDesC8& aDes);
    IMPORT_C TInt Write(const TDesC8& aDes, TInt aLength);
    IMPORT_C TInt Read(TDes8& aDes) const;
    IMPORT_C TInt Read(TDes8& aDes, TInt aLength) const;
    ...
};
```

The `Open()`, `Create()`, `Replace()` and `Temp()` methods take a filename, which is text, and is specified as a `const TDesC&`.

The `Write()` methods take a descriptor containing binary data to be written to the file. For binary data, these classes are specified as `const TDesC8&`, the '8' indicating that 8-bit bytes are intended, regardless of the size of a character. The version of `Write()` that takes only a descriptor writes its entire contents (up to `Length()` bytes) to the file. Many APIs use descriptors exactly like this, for sending binary data to another object.

5.9 Using Descriptors with Methods

This section describes the most appropriate types to use when passing descriptors to, and returning them from, methods.

Descriptors as Parameters

The following guidelines should be followed when using descriptors as method parameters:

- To pass a descriptor as an input parameter into a method (i.e., to pass descriptor data that is read within the method but is not changed), use `const TDesC&` as the parameter type.
- To pass an in-out parameter to a method (i.e., to pass descriptor data that you want to change within your method), use `TDes&` as the parameter type.

The following global method, `appendText()`, is an example of the use of `const TDesC&` and `TDes&` parameters:

```
void appendText(TDes& aTarget, const TDesC& aSource)
{
    aTarget.Append(aSource);
}

_LIT(KTxtHello, "Hello");
_LIT(KTxtWorld, " World!");
TBuf<12> helloWorld(KTxtHello);
appendText(helloWorld, KTxtWorld);
```

When the call to `appendText()` returns, `helloWorld` contains the text 'Hello World!'.

The important point to note with this example is that any descriptor type can be passed as the `aSource` parameter. If we had prototyped the method in such a way that `aSource` was, for example, of type `TBuf` or `TPtr&`, then we would need to add variants of the method to cope with all possible permutations, which is clearly wasteful.

While any descriptor type can be passed into a method taking `const TDesC&`, only `TPtr`, `TBuf` or `RBuf` can be passed into methods taking `TDes&`.

The next example shows first how an `RBuf` can be passed to the `appendText()` method and then how an `HBufC` can be passed to `append` itself to itself. Note that as an `HBufC` cannot be passed as a `TDes&` parameter, a `TPtr` to the data in the `HBufC` must be passed instead:

```
TInt maxLen = KTxtHello().Length() + KTxtWorld().Length();
RBuf helloBuf;
RBuf worldBuf;
helloBuf.CreateL(KTxtHello, maxLen);
helloBuf.CleanupClosePushL();
worldBuf.CreateL(KTxtWorld); // "Hello"
appendText(helloBuf, worldBuf); // "Hello World!"
CleanupStack::PopAndDestroy(&helloBuf);
worldBuf.Close();

HBufC* heapBuf = HBufC::NewL(maxLen);
heapBuf->Des() = KTxtHello();
// heapBuf contains "Hello"
```



```
appendText(heapBuf->Des(), *heapBuf);
           // heapBuf now contains "HelloHello"
delete heapBuf;
```

An empty descriptor can be passed to a method expecting a `const TDesC&` using the `KNullDesC` literal. There are also `KNullDesC8` and `KNullDesC16` variants. All three define an empty or null descriptor literal (in `e32std.h`):

```
_LIT(KNullDesC, "");
_LIT8(KNullDesC8, "");
_LIT16(KNullDesC16, "");
```

When declaring method parameters you must declare `TDesC` and `TDes` as pass by reference and not by value and you should also not omit `const` when using `TDesC` – see Section 5.11.

Generally, method parameters should be either `const TDesC&` or `TDes&` but there may be occasions when you need to use other types.

Using RBuf& as a method parameter

If you need to create a heap buffer within your method and pass ownership to the calling code, then the choice is between an `HBufC` and an `RBuf`. The standard pattern for creating an `HBufC` is to return it from the method as described in Section 5.9, however this is not possible for an `RBuf` (as its copy constructor is protected); instead, it may be achieved as follows:

```
void FooL(RBuf& aBuf)
{
    aBuf.CreateL(KMaxLength); // potential memory leak
    // fill with data
}

RBuf buf;
FooL(buf);
```

However calling `CreateL()` twice causes a memory leak. In the above code, if the `RBuf` has already been created, a leak results. An alternative therefore is to call `ReAllocL()`; note how this works in both situations below:

```
void copyTextL(RBuf& aDes)
{
    _LIT(KTxtHello, "Hello");
    if (aDes.MaxLength() < KTxtHello().Length())
    {
        aDes.ReAllocL(KTxtHello().Length());
    }
}
```

```

    }
    aDes = KTxtHello;
}

RBuf rBuf1;
FooL(rBuf1);
rBuf1.Close();

RBuf rBuf2;
rBuf2.CreateL(3);
rBuf2.CleanupClosePushL();
FooL(rBuf2);
CleanupStack::PopAndDestroy();

```

RBuf::ReAllocL() does not suffer from the same potential problems as HBufC::ReAllocL(), described in Section 5.11.

Using TPtrC as a method parameter

TPtrC is frequently used as a return type from a method, however there may be occasions when the method return is being used for something else and it is necessary to use the TPtrC as a parameter instead:

```

__LIT(KTxtHelloWorld, "Hello World!");
TInt CSomeClass::GetWorld(TPtrC& aWorldOut)
{
    iBuf = KTextHelloWorld;      // iBuf may be a TBufC<x>
    aWorldOut.Set(iBuf.Mid(6,5));
    return KErrNone;
}

CSomeClass* someClass = CSomeClass::NewL();
TPtrC ptr;
TInt ret = someClass->GetWorld(ptr);    // ptr contains "World"
delete someClass;

```

Note that TPtrC and const TPtrC& are not used as method parameters.

Using const TDesC* as a method parameter

There may be occasions when a descriptor parameter can have a NULL input, that is when no descriptor object is passed (as opposed to when an empty descriptor is passed). In these circumstances, const TDesC* could be used:

```

class TSomeClass
{
public:
    void Foo(const TDesC* aDes = NULL);
};

```

```
...
someClass.Foo(&someDescriptor.Left(x)); // be wary!
```

However if you do this, you should be wary of calling the method with a temporary object as C++ does not guarantee what the behavior is when passing a temporary object by `const*`. Instead you could consider overloading the method in combination with using `inlines`:

```
class TSomeClass
{
public:
    inline void Foo();
    inline void Foo(const TDesC& aDes);
private:
    void DoFoo(const TDesC* aDes);
};

inline void TSomeClass::Foo()
{ DoFoo(NULL); }
inline void TSomeClass::Foo(const TDesC& aDes)
{ DoFoo(&aDes); }
```

Using HBufC*& as a method parameter

Usually, if a method needs to create an `HBufC` and pass ownership of it to the caller this is done by returning it, however there may be circumstances when something else must be returned from the method. In these circumstances, this can be accomplished using a parameter instead:

```
TInt FooL(HBufC*& aBuf)
{
    ASSERT(aBuf == NULL);
    aBuf = KTxtHelloWorld().AllocL();
    return KErrNone;
}

HBufC* buf = NULL;
FooL(buf);
delete buf;
```

Note that if the `HBufC` needs to be created before being passed to a method and assigned within the method, then this can also be accomplished using an `HBufC*&` parameter, though if you need to use `ReAlloc()` within the method, then caution should be exercised (see Section 5.11).

Returning Descriptors from Methods

Which descriptor type to use as a return from a method depends upon:

- whether a descriptor or raw data is being returned

- whether the entire part, or just a portion, of the descriptor or raw data is being returned
- whether the descriptor or raw data being returned is constant or modifiable
- whether a heap descriptor is being created within the method and ownership is to be transferred to the caller.

Returning an entire constant descriptor

If you are returning an entire descriptor that exists while the caller of the method uses it and the descriptor is not to be modified, then you should return `const TDesC&`:

```
const TDesC& CSomeClass::ReturnConstantEntireDescriptor() const
{
    return iDescriptor;
}

CSomeClass* someClass = CSomeClass::NewL();
TPtrC p = someClass->ReturnConstantEntireDescriptor();
// use p
delete someClass;
```

Returning a partial constant descriptor

If you are returning part of a descriptor that exists while the caller of the method uses it and the descriptor is not to be modified, then you should return `TPtrC`:

```
TPtrC CSomeClass::ReturnConstantPartialDescriptor() const
{
    return iDescriptor.Right(4); // return the rightmost 4 chars
}

TPtrC p = someClass->ReturnConstantPartialDescriptor();
// use p
```

A `TPtrC` occupies 8 bytes, which means it is officially allowed to be passed to and returned from methods by value (as opposed to by reference) according to the Symbian OS coding standards.

In this example, `return iDescriptor.Right(4)` creates and returns a temporary `TPtr` object, however it is safe to do so as the method return type is a `TPtrC`; it is returning by value and not by reference. If the return type was a reference to one of the descriptor types, then random stack data would be returned.

Returning an entire non-constant descriptor

If you are returning an entire descriptor that exists while the caller of the method uses it and the descriptor is to be modified, then you should return `TDes&`:

```
TDes& CSomeClass::ReturnWritableEntireDescriptor()
{
    return iDescriptor;
}

someClass->ReturnWritableEntireDescriptor() = KTxtHello;
```

Returning a partial non-constant descriptor

If you are returning part of a descriptor that exists while the caller of the method uses it and the descriptor is to be modified, then you should return `TPtr&`:

```
TPtr& CSomeClass::ReturnWritablePartialDescriptor()
{
    iTPtr.Set(iDescriptor.MidTPtr(1, 5);
    return iTPtr;
}

someClass->ReturnWritablePartialDescriptor() = KTxtHello;
```

In this example, assuming `iDescriptor` is of type `TBuf<x>`, then it is not possible to return `iDescriptor.MidTPtr(1, 5)` as that would result in a reference to a temporary `TPtr` being returned. Similarly, it is not sufficient to assign the result of `iDescriptor.MidTPtr(1, 5)` into a locally declared `TPtr` as that too would result in a reference to a temporary object being returned. Hence, it is necessary to have a `TPtr` data member and return a reference to it.

Note that the `MidTPtr()`, `LeftTPtr()` and `RightTPtr()` methods of `TDes` all return a `TPtr` while the `Mid()`, `Left()` and `Right()` methods of `TDesC` return a `TPtrC`.

Returning constant raw data

If you are returning in its entirety raw data that exists while the caller of the method uses it and the data is not to be modified, then you should return `const TDesC&` or `TPtrC` or, as raw data is being dealt with, more likely the narrow variants `TDesC8` or `TPtrC8`:

```
class CSomeClass : public CBase
{
public:
```

```
// methods omitted for brevity
private:
    TText8 iRawData[KMaxLength];
    mutable TPtrC8 iPtrC;
};

TPtrC8 CSomeClass::ReturnConstantEntireRawData() const
{
    return iRawData;
}
```

If the return type is TPtrC8 then the raw data can be returned directly; if the return type is const TDesC8& then the raw data must first be converted to a descriptor via a TPtrC8 or TPtr8:

```
const TDesC8& CSomeClass::ReturnConstantEntireRawData() const
{
    iPtrC.Set(iRawData, KMaxLength);
    return iPtrC;
}
```

Note that iPtrC has been declared as mutable, otherwise a compilation error results when Set() is called as ReturnConstantEntireRawData() is a const method.

Returning a portion of constant raw data

If you are returning a portion of raw data that exists while the caller of the method uses it and the data is not to be modified, then you should return TPtrC:

```
TPtrC8 CSomeClass::ReturnConstantPartialRawData() const
{
    // if iRawData contains "Hello World!",
    // return a pointer to "World!"
    iPtrC.Set(iRawData, KMaxLength);
    return iPtrC.Right(6);

    // or
    iPtrC.Set(&iRawData[5], 6);
    return iPtrC;
}

TPtrC8 ptr = someClass->ReturnConstantPartialRawData();
// ptr points to "World!"
```

Returning modifiable raw data

If you are returning in its entirety raw data that exists while the caller of the method uses it and the data is to be modified, then you should return

TDes&. You need to construct a non-temporary TPtr around the raw data and return that:

```
class CSomeClass : public CBase
{
public:
// methods omitted for brevity
private:
    TText8 iRawData[KMaxLength];
    TPtr8 iPtr;
};

TDes8& CSomeClass::ReturnWritableEntireRawData()
{
    iPtr.Set(iRawData, dataLength, KMaxLength);
    return iPtr;
}
```

TPtr does not have a default constructor so, when it is used as a data member of a class, it must be initialized in the constructor initialization list (see Section 5.11 for more information).

Returning a portion of modifiable raw data

If you are returning raw data that exists while the caller of the method uses it, and the data is to be modified, then you should return TDes&. You need to construct a TPtr around the raw data and return that:

```
TDes8& CSomeClass::ReturnWritablePartialRawData()
{
    iPtr.Set(iRawData, dataLength, KMaxLength);
    iPtr = iPtr.RightTPtr(6);
    return iPtr;

    // or
    iPtr.Set(&iRawData[5], 6);
    return iPtr;

    // as always when returning references, be careful not to
    // return a reference to a temporary object, so avoid code
    // such as the following:
    // iPtr.Set(iRawData, dataLength, KMaxLength);
    // return iPtr.RightTPtr(6);
    // RightTPtr() returns a TPtr object
}
```

Returning a constant descriptor by value

If the data to be returned has to be constructed and has a known maximum length which is less than or equal to 256⁹ then return a TBufC or TBuf. If the data is large, create and return a heap descriptor.

⁹ Returning a descriptor by value like this consumes stack space. Objects bigger than 256 bytes should not be placed on the stack.

```

const TInt KBufferSize = 64;
typedef TBuf<KBufferSize> TMyBufferType;

class CSomeClass : public CBase
{
public:
    TMyBufferType ReturnByValue() const;
    ...
private:
    TMyBufferType iBuf;
};

TMyBufferType CSomeClass::ReturnByValue() const
{
    return iBuf;
}

CSomeClass* someClass = CSomeClass::NewL();
TMyBufferType myBuffer(someClass->ReturnByValue());

```

Creating a heap descriptor and returning ownership

If the descriptor has to be constructed and its length is long or not fixed then allocate and return an HBufC:

```

HBufC* CSomeClass::GetSomeTextL()
{
    HBufC* hBuf = iEikonEnv->AllocReadResourceL(someResourceID);
    return hBuf;
}

HBufC* hBuf = GetSomeTextL();
...
delete hBuf;

```

The caller is responsible for taking ownership of the returned heap-based descriptor and must be responsible for deleting it when it is finished with.

If the heap descriptor is not written to or is written to infrequently then use HBufC; however if the heap descriptor is to be written to frequently then an RBuf is preferable as it is more efficient.

An RBuf cannot be returned from a method as its copy constructor is protected, thus the following code generates a compilation error:

```

RBuf FooL()
{
    _LIT(KTxtHello, "Hello");
    RBuf buf;
    buf.CreateL(buf);
    return (buf);
}

```


You could attempt to return a pointer to an `RBuf` instead but then making your code leave-safe becomes more complicated and messy. Anyway, R-type objects are intended to be used on the stack and not the heap. Instead you should use an `RBuf` as a parameter.

Returning a reference to a re-allocatable writable descriptor

If you want to return a writable descriptor, where you require reallocation but do not want to transfer ownership, then return an `RBuf&`:

```
class CSomeClass : public CBase
{
public:
    RBuf& ReturnWritableReallocatableDescriptor();
private:
    RBuf iBuf;
};

RBuf& CSomeClass::ReturnWritableReallocatableDescriptor()
{
    return iBuf;
}

RBuf& buf = someClass->ReturnWritableReallocatableDescriptor();
buf.ReAllocL(KNewMaxSize);
```

You can also return an `HBufC*`:

```
class CSomeClass : public CBase
{
public:
    HBufC* ReturnWritableReallocatableDescriptor();
private:
    HBufC* iBuf;
};

HBufC* CSomeClass::ReturnWritableReallocatableDescriptor()
{
    return iBuf;
}

HBufC* buf = someClass->ReturnReallocatableDescriptor();
if (buf != NULL)
{
    delete buf;
    buf = NULL;
}
buf.ReAllocL(KNewMaxSize);
```

Note that both these examples are a violation of object-oriented principles as the private internals of a class are exposed for manipulation.

5.10 Some Descriptor Operations

Descriptors and the Text Console

Symbian OS provides a text console class called `CConsoleBase` which allows you to output formatted text to the screen and accept keyboard input. This can be useful when first experimenting with descriptors.

The following code shows how a literal and stack descriptor can be displayed using `CConsoleBase::Printf()`. The code displays 'Hello World!' to the console twice, then pauses until the user presses a key.

```
#include <e32base.h>
#include <e32cons.h>

TInt E32Main()
{
    _LIT(KTxtConsoleTitle, "Hello Title");
    CConsoleBase* console = Console::NewL(KTxtConsoleTitle,
        TSize(KConsFullScreen, KConsFullScreen));

    _LIT(KTxtHelloWorld, "Hello World!\n");
    console->Printf(KTxtHelloWorld);

    TBufC<20> constantBuffer(KTxtHelloWorld);
    console->Printf(constantBuffer);

    console->Getch(); // get a input character from the user
    delete console;
    return 0;
}
```

Any of the descriptor types can be passed directly to `Printf()` to be displayed.

`Printf()` can also take a formatting string similar to the `sprintf()` method in standard C. To display C-style strings, use `%s` as the formatter; to display descriptors, use `%S`. You must also pass the address of descriptors as follows:

```
TText array[] = L"Hello World!\n";
TBufC<20> constantBuffer(KTxtHelloWorld);
HBufC* hBuf = KTxtHelloWorld().AllocL();
_LIT(KFormatTxt, "%s%S%S");
console->Printf(KFormatTxt, array, &constantBuffer, hBuf);
delete hBuf;
```

This displays 'Hello World' three times. Note how the address of `constantBuffer` is passed as a parameter (leaving off the `&` would result in an exception); but `hBuf` is already an address so prefixing with `&` is not necessary.

This same pattern of using %s for C-style strings and %S for descriptor strings is used in several other places within Symbian OS where formatting functionality is used.

CConsoleBase doesn't have any input method other than Getch() which gets a single character at a time. If you want to use it to get strings from the console, you must construct a loop to suit your purposes:

```
TBuf<20> buf;
TKeyCode c = console->Getch();
    // TKeyCode is declared in e32keys.h
while(c != EKeyEscape )
{
    console->Printf(_L("%c"), c);
    buf.Append(c);        // overflow checking omitted for clarity
    c = console->Getch();
}
```

Converting a Descriptor to a Number

The contents of a descriptor can be converted into a number using the Val() method of the TLex class. For example:

```
_LIT(KNumber, "12345");
TLex lex(KNumber);
TInt value = 0;
User::LeaveIfError(lex.Val(value));
    // the number 12345 is assigned to value
```

The Val() method has many overloads for conversion to different numerical types, with or without limit checking, including options for conversion from decimal, binary, octal, etc. See your SDK for further details.

The TLex class provides a range of functionality intended to aid lexical processing of text data. Consult your SDK for further details.

Converting a Number to a Descriptor

A numeric value can be converted into a descriptor using the Num() method:

```
TInt minusThirtyFour(-34);
TBuf<8> des;
des.Num(minusThirtyFour);
```

This creates a descriptor of length three, containing the characters '-', '3', '4'.

There are several overloads of `Num()` for conversions from different numerical types and including options for conversion from decimal, binary, octal, etc. There are also overloaded `AppendNum()` methods for appending a number to a descriptor.

Externalizing and Internalizing Descriptors

Descriptors have their own `operator<<` and `operator>>` methods defined; they externalize a compressed form of the length. It is a common mistake to externalize the length and data of the descriptors explicitly, as in the following code:

```
void CSomeClass::ExternalizeL(RWriteStream& aStream) const
{
    // externalize an HBufC
    TInt size = iData->Length();    // iData is an HBufC*
    aStream.WriteUInt32L(size);
    aStream.WriteL(iData->Des(), size);
    // unnecessary call to HBufC::Des()

    // externalize a TBuf<>
    size = iBuffer.Length();    // iBuffer is a TBuf<>
    aStream.WriteUInt32(size);
    aStream.WriteL(aData.Ptr(), size);
}
```

The correct way to write this code is:

```
void CSomeClass::ExternalizeL(RWriteStream& aStream) const
{
    // externalize an HBufC
    aStream << *iData;    // iData is an HBufC*

    // externalize a TBuf<>
    aStream << iBuffer;    // iBuffer is a TBuf<>
}
```

The corresponding internalization code is:

```
void CSomeClass::InternalizeL(RReadStream& aStream) const
{
    // internalize into an HBufC
    delete iData;    // assuming iData is already allocated
    iData = NULL;
    iData = HBufC::NewL(aStream, KMaxDataLength);

    // internalize into a TBuf<>
    aStream >> iBuffer;
    // should be large enough to accommodate all the data
}
```

`HBufC::NewL(RReadStream &aStream, TInt aMaxLength)` takes a read stream as a parameter and reads up to `aMaxLength` of descriptor data into an `HBufC*` that it creates.

Similarly to `HBufC::NewL()`, `RBuf` also has a `CreateL()` variant that takes an `RReadStream` method parameter.

To use the `operator<<` and `operator>>` methods of descriptors, you must link to `estor.lib` in the MMP file. Remember that, as with all streaming operators, they may leave.

Putting Binary Data into a `_LIT`

The `\x` escape sequence can be used to place binary data into a `_LIT`. The following code produces a literal of length five containing the sequence of bytes `0x01 0x02 0x03 0x45 0xEF`:

```
_LIT8(KNarrowLiteral, "\x01\x02\x03\x45\xEF");
TPtrC8 narrowPtr(KNarrowLiteral);
```

With wide literals, you may need to be concerned with byte ordering depending upon your intended usage, because `\x12` in a Unicode string actually means `0x0012`; on Symbian OS, it is stored as `0x12 0x00`. The following code produces a literal of length two containing the sequence of bytes `0x34 0x12 0xFF 0x33`:

```
_LIT(KWideLiteral, "\x1234\x33FF");
TPtrC widePtr(KWideLiteral);
```

Conversions Involving Descriptors and Strings

The following examples use `TDes8::Copy(const TDesC16&)` and `TDes16::Copy(const TDesC8&)` to convert between narrow and wide descriptors.

`TDes8::Copy(const TDesC16&)` strips out alternate wide characters while `TDes16::Copy(const TDesC8&)` pads each character with a trailing zero as part of the copy operation. These methods thus form a rudimentary way of copying and converting when the character set is encoded with one byte per character and the last byte of each wide character is NULL-padded.

To perform full conversions in both directions where the data is encoded in UTF-16, UTF-7 and UTF-8 see the `CCnvCharacterSetConverter` and `CnvUtfConverter` classes. Further details can be found in your SDK.

From a wide descriptor to a narrow descriptor

```
__LIT(KTxtWideHello, "Hello");
TBuf<5> wide(KTxtWideHello);
TBuf<5> narrow;
narrow.Copy(wide); // Uses TDes8::Copy(const TDesC16&)
```

Note that there is also a method `TDes8::Append(const TDesC16&)`.

From a narrow descriptor to a wide descriptor

```
__LIT8(KTxtNarrowHello, "Hello");
TBuf8<5> narrow(KTxtNarrowHello);
TBuf<5> wide;
wide.Copy(narrow);
```

If you are using Symbian OS v8.1 or later then you may also use `TBuf8::Expand()` which extends each character with zero to 16 bits.

There is no corresponding `TDes16::Append(const TDesC8&)`.

From a narrow string to a narrow descriptor

```
const unsigned char* constNarrowString = ...
unsigned char* narrowString = ...

// conversion to TPtrC8
TPtrC8 ptrc(constNarrowString, User::StringLength(constNarrowString));

// conversion to TPtr8
TInt stringLen = User::StringLength(narrowString);
TPtr8 ptr(narrowString, stringLen, stringLen);

// conversion to TBuf8
TBuf8<5> buf(narrowString);
```

From a narrow string to a wide descriptor

You can first convert to a narrow descriptor and then use `TDes16::Copy(TDesC8&aDes)` to subsequently convert from a narrow descriptor to a wide descriptor:

```
unsigned char* narrowString = ...
TInt stringLen = User::StringLength(narrowString);
TPtr8 ptr(narrowString, stringLen, stringLen);
TBuf<5> buf;
buf.Copy(ptr);
```

From a wide string to a wide descriptor

```
unsigned short int* wideString = ...
TInt stringLen = User::StringLength(wideString);
TPtr ptr(wideString, stringLen, stringLen);
```

From a wide string to a narrow descriptor

You can first convert to a wide descriptor and then use `TDes8::Copy (const TDesC16&aDes)` to subsequently convert to a narrow descriptor:

```
unsigned short int* wideString = ...
TInt stringLen = User::StringLength(wideString);
TPtr ptr(wideString, stringLen, stringLen);
TBuf8<5> buf;
buf.Copy(ptr);
```

From a narrow descriptor to a narrow string

```
_LIT8(KTxtNarrowHello, "Hello");
TBuf8<5> narrow(KTxtNarrowHello);
TText8 string[6];
Mem::Copy(string, narrow.Ptr(), narrow.Size());
string[narrow.Size()] = '\\0'; // append NULL terminator
```

From a narrow descriptor to a wide string

```
_LIT8(KTxtNarrowHello, "Hello");
TBuf8<5> narrow(KTxtNarrowHello);
TBuf<5> wide;
wide.Copy(narrow);
TText string[6];
Mem::Copy(string, wide.Ptr(), wide.Size());
string[wide.Length()] = '\\0'; // append NULL terminator
```

Be careful not to confuse `Size()` with `Length()`. The `Size()` method must be used with `Mem::Copy()`, otherwise only half the descriptor is copied; `Length()` must be used when indexing into the array otherwise memory beyond the array's bounds is accessed.

From a wide descriptor to a wide string

```
_LIT(KTxtHello, "Hello");
TBuf<5> wide(KTxtHello);
```

```
TText string[6];
Mem::Copy(string, wide.Ptr(), wide.Size());
string[wide.Length()] = '\0'; // append NULL terminator
```

From a wide descriptor to a narrow string

```
_LIT(KTtxtHello, "Hello");
TBuf<5> wide(KTtxtHello);
TBuf8<5> narrow;
narrow.Copy(wide);
TText8 string[6];
Mem::Copy(string, narrow.Ptr(), narrow.Size());
string[narrow.Length()] = '\0'; // append NULL terminator
```

Using Collapse() and Expand()

From Symbian OS v8.1 onwards, these methods may be used to aid with conversion operations between narrow and wide descriptors.

Expand() widens a narrow descriptor while Collapse() narrows a wide descriptor:

```
void TakeNarrowDes(const TDesC8& aDes)
{
    // aDes contains "AABB"
    TInt size = aDes.Size(); // 2
    TInt len = aDes.Length(); // 2
}

_LIT(KTtxtWideData, "\x11AA\x22BB");
TBuf<5> wideBuf(KTtxtWideData); // contains AA11BB22
TInt wideSize = wideBuf.Size(); // 4
TInt wideLength = wideBuf.Length(); // 2
TakeNarrowDes(wideBuf.Collapse()); // now contains AABBBB22
wideSize = wideBuf.Size(); // 4
wideLength = wideBuf.Length(); // 2
```

The Collapse() method returns a TPtr8 where every second byte has been stripped from the original wide data, hence the aDes parameter of TakeNarrowDes() is of length and size 2 and contains AABB. Note that the length and size of wideBuf are unaffected by calling Collapse() despite its contents having been altered. Thus it is not meaningful to use a descriptor directly once its Collapse() method has been called, instead it should be accessed via the returned TPtr8.

```
void TakeWideDes(const TDesC& aDes)
{
    TInt size = aDes.Size(); // 10
    TInt len = aDes.Length(); // 5
}
```



```

_LIT8(KTxtNarrowHello, "Hello");
TBuf8<10> narrowHello(KTxtNarrowHello);
// must contain space for expansion
TInt narrowSize = narrowHello.Size();    // 5
TInt narrowLen = narrowHello.Length();   // 5
TakeWideDes(narrowHello.Expand());

narrowSize = narrowHello.Size();    // 10, twice previous size
narrowLen = narrowHello.Length();   // 10, twice previous length

```

With expansion the descriptor is modified and an extra zero extension byte is inserted after each existing byte. The descriptor must have a maximum length at least twice its current length. Notice the difference in lengths between that of `aDes` and that of `narrowHello` after `Expand()` has been called.

If you use this function, you must be aware that if the descriptor is a pointer descriptor then the memory pointed to (i.e., the result of calling `TDesC::Ptr()`) must be at an even-numbered address because the ARM processor can only access 16-bit quantities at even-numbered addresses.

5.11 Correct Use of Descriptors

Do Not Declare **TDesC** or **TDes** Variables

Neither `TDesC` nor `TDes` provide any storage space for data or for a pointer to data (see Figure 5.2) thus you should never instantiate a `TDesC` or `TDes`. The following code illustrates a common mistake:

```

void Foo(const TDesC& aDes)
{
    TBuf<32> buf(aDes); // buf contains random data
}

_LIT(KTxtHelloWorld, "Hello World!");
TDes des(KTxtHelloWorld);    // these compile but neither hold
TDesC desC(KTxtHelloWorld);  // nor reference any actual data
Foo(desC);

```

Always Pass **TDes** and **TDesC** Parameters by Reference

When declaring method parameters, you must declare `TDesC` and `TDes` as pass-by-reference and not pass-by-value. The reason is that these classes do not contain any actual string data (see Figure 5.2) and if you pass them by value, then the rules of C++ mean that static binding is being used and the consequence is that polymorphism won't work. The

end result is that your code compiles but you are not actually passing any data. For example, in the following code the `buffer` variable contains random stack data:

```
void Foo(const TDesC aString) // error, should be const TDesC&
{
    TBufC<10> buffer(aString); // buffer will contain random data
    ...
}
```

Do Not Omit `const` when Declaring `TDesC` Parameters

It is a mistake to omit the `const` keyword when passing a `TDesC`. The following code may generate a compilation error depending upon the compiler used. Even if the method declaration does not generate a compilation error, attempting to pass a literal to that method does, so always use `const`:

```
void Foo(TDesC& aString); // possible compilation error
...
Foo(KNullDesC);           // definite compilation error
```

Do Not Set the Size of a Stack Descriptor at Run Time

A frequent mistake when first using stack descriptors is to attempt to define their size at run time:

```
TInt length = someValue;
TBuf<length> myDescriptor;
```

This generates a compilation error because the stack classes are templated and thus their size must be determined at compilation time and not at run time.¹⁰

Do Not Place Large Stack Descriptors on the Stack

If you find yourself in the situation where you want to define the size of a stack descriptor at run time, you may be tempted to declare a stack descriptor with a large enough maximum size to account for the total amount of data your descriptor may hold:

¹⁰ Depending upon the compiler used, the compilation error may be confusing. Some compilers may indicate that the error is in the class declaration of `TBuf` within the `e32std.h` header file.

```
TBuf<5000> myDescriptor;
```

This may be wasteful because it would be consuming 10 000 bytes of descriptor stack space regardless of how much data the descriptor actually contained (see Section 5.7 if you were expecting it to use 5000 bytes of stack space).

Stack descriptors are typically used on the stack, however, as with any type, large ones should not use the stack due to the limited default stack size.¹¹ If you find yourself wanting to use a large stack descriptor see the next section.

A Symbian OS rule of thumb is that if something is more than 256 bytes then it shouldn't go on the stack (the stack size is pretty limited: the default is 8 KB). So, for example, if you want to use a `TBuf<256>` as a stack variable, you should consider turning it into an `RBuf` or `HBufC` instead.

Symbian OS defines a number of classes and `typedef`s that are descriptors or contain descriptors and that could be used inefficiently if used on the stack. It is advisable therefore to be aware of the amount of space the following types consume:

- a `TEntry` consumes 552 bytes
- a `TFileName` consumes 520 bytes
- a `TFullName` consumes 520 bytes
- a `TName` consumes 264 bytes.

Do Not Allocate a Large Stack Descriptor Directly on the Heap

If you find yourself in the situation where you want to place a large descriptor on the stack, you may be tempted to allocate the descriptor on the heap, either directly as in this code or indirectly by making it a member variable of a `CBase`-derived class.

```
TBuf<5000>* myDescriptor = new(ELeave) TBuf<5000>;  
// allocation on the heap
```

While this moves the descriptor data-storage area from the stack to the more plentiful heap, it is still wasteful of memory if the actual amount of data you store in the descriptor is much less than 5000 characters.

If you find yourself in this situation then the solution is to use one of the heap descriptors, `HBufC` or `RBuf`, instead.

¹¹ The stack in Symbian OS is a limited resource and should not be used to hold data objects as large as this, even if only temporarily. (If your code executes on the emulator but generates a Kern Exec 3 panic on the hardware, a large descriptor on the stack may be the cause.)

Be Wary of `TPtr::operator=()`

If you find yourself using `TPtr`'s assignment operator be aware that attempting to assign a type 4 `TPtr` into a type 2 `TPtr` results in a panic. Assigning a type 4 `TPtr` to a type 2 `TPtr` does not result in the type changing, which may lead to problems.¹²

```
TText* helloText = (TText*)L"Hello";
TBufC<8> worldBuffer(_L("World\0"));
TInt helloTextLength = User::StringLength(helloText); // length is 6
TPtr type2Ptr(helloText, helloTextLength, helloTextLength);
TPtr type4Ptr = worldBuffer.Des();
type2Ptr = type4Ptr;      // program crash
type4Ptr = type2Ptr;      // type4Ptr remains a type 4 TPtr
type2Ptr.Set(type4Ptr);   // type2Ptr is now a type 4 TPtr
```

Be Wary of `TPtr`'s Non-conformance to C++ Conventions

You should be aware that there is some unexpected behavior with `TPtr`s because their copy constructors behave differently from their assignment operators. This can be confusing because it is conventional in C++ to ensure that these two operations produce the same result, however `TPtr` breaks with this convention.

Instead of copying the members of a descriptor, which would result in two `TPtr`s pointing to the same data, the `TPtr` type 4 assignment operator is defined to copy the contents of one `TDesC` into where the `TPtr` points:

```
_LIT(KWorld, "World");
TText helloBuffer[5] = {'h','e','l','l','o'};
TPtr memoryPtr(helloBuffer, 5); // points to "hello"
TBufC<5> worldBuf(KWorld);
TPtr worldPtr(worldBuf.Des()); // points to "World"
memoryPtr = worldPtr;          // "World" is written to helloBuffer
```

Note that this behavior doesn't occur if the assigning `TPtr` points to memory as opposed to a descriptor (i.e., a type 2 `TPtr`):

```
TText helloBuffer[5] = {'h','e','l','l','o'};
TPtr memoryPtr(helloBuffer, 10); // points to "hello"
```

¹² This example also illustrates the `User::StringLength()` method, which returns the length of a C-style NULL-terminated string. There are two overloads: `User::StringLength(const TUint8 *aString)` and `User::StringLength(const TUint16 *aString)`. In the example code, the `L` macro is used to indicate to the compiler that "Hello" is a wide string; without it `helloTextLength` would be 3. Note that `L` is not a macro defined in Symbian OS and should not be confused with `_L`.

```
TText worldBuffer[5] = {'W','o','r','l','d'};
TPtr worldPtr(worldBuffer,5);    // points to "World"
memoryPtr = worldPtr;    // "World" is not written to helloBuffer
```

Use **HBufC::Des()** Correctly

There is a common mistake that many people tend to make once they become familiar with the `Des()` method. When they have an `HBufC` that is to be passed to a method that takes a `const TDesC&` type, they assume that they first have to create a `TPtr` using `Des()` and pass this into the method. While this works, it is far simpler and much more efficient to simply dereference the heap descriptor; for example, if you have a function `foo()` prototyped as:

```
void foo(const TDesC& aDescriptor);
```

then the following code works fine:

```
HBufC* buffer = HBufC::NewL(256);
...
foo(*buffer);
```

Similarly, people frequently also call the `Des()` method unnecessarily when returning an `HBufC` data from a method, for example, in the following, `iData->Des()` should be simply `*iData`:

```
const TDesC& CSomeClass::Data() const
{
    if (iData)                // iData is an HBufC*
        return iData->Des();    // should use return *iData instead
    else
        return KNullDesC;13
}
```

Use **Allloc()** When Creating an **HBufC** from a Descriptor

When you have a descriptor and you wish to create an `HBufC` and copy the contents into it, a common mistake is to code it as follows:

```
void myFunctionL(const TDesC& aBuffer)
{
    HBufC* myData = HBufC::NewL(aBuffer.Length());
```

¹³ Note that, with some compilers in some circumstances, an error may be generated when using `KNullDesC`; if this occurs, use `KNullDesC()` instead.

```
TPtr ptr = myData->Des();
ptr.Copy(aBuffer);
...
}
```

Instead use the `Alloc()`, `AllocL()` or `AllocLC()` methods:

```
void myFunctionL(const TDesC& aBuffer)
{
    HBufC* myData = aBuffer.AllocLC();
    ...
}
```

Be Wary of `HBufC::ReAllocL()` and `HBufC::ReAlloc()`

If you wish to resize an `HBufC` you should be aware that `ReAllocL()` may return a different address from the current `HBufC` pointer:

```
_LIT(KTxtHello, "Hello");
_LIT(KTxtGreeting, "How are you?");

HBufC* heapBuf = KTxtHello().AllocLC();
    // place it on the cleanup stack
TPtr ptr(heapBuf->Des());
// ptr = KTxtGreeting; // would panic because the
                        // maximum length is exceeded

heapBuf = heapBuf->ReAllocL(KTxtGreeting().Length());
    // So need to reallocate the heapBuf variable
CleanupStack::Pop();
    // The address of heapBuf may have changed during ReAllocL
CleanupStack::PushL(heapBuf);
    // so need to re-push it to the cleanup stack
ptr.Set(heapBuf->Des()); // and reset the pointer
ptr = KTxtGreeting;
    // Now the assignment can be made safely
...
CleanupStack::PopAndDestroy();
```

The call to `ReAllocL()` creates a new heap descriptor and, if successful, copies the original data into it and returns the new address. The original copy is deleted. Of course, if there is insufficient memory, then the method leaves. After the reallocation has taken place, `heapBuf` may point to a different heap cell. There may be times when it points to the original heap cell, but it is safer to assume that it doesn't. This is particularly important if you use the `Des()` method, in which case you need to re-Set() any `TPtr` returned by `Des()`.

This example also shows how you must pay attention if you are pushing `HBufC`s to the cleanup stack and you reallocate them. If the memory address changes during the reallocation then an out-of-date memory

location may now be held on the cleanup stack; this must be addressed by popping it off the stack and pushing a new address on. Note that it would have been incorrect to call `CleanupStack::Pop(heapBuf)` because the system uses the parameter to `Pop()` as a check that the expected item is being popped off the stack. However, if `heapBuf`'s address has changed, this results in a panic. Similarly calling `PopAndDestroy()` with the old value of `heapBuf` on the stack results in a `User-44` panic if its address changes.

Be careful not to `Pop()` from the cleanup stack before the reallocation, if the code from above is rearranged as follows and the call to `ReAllocL()` fails, there will be a memory leak:

```
CleanupStack::Pop();
heapBuf = heapBuf->ReAllocL(KTxtGreeting().Length());
CleanupStack::PushL(heapBuf);
```

Don't forget that `ReAlloc()` and `ReAllocL()` return a `HBufC*` which must be assigned to a variable. The following code also causes a `User-44` panic if the address of the allocated cell is changed; this is because the old address is being pushed onto the cleanup stack:

```
_LIT(KTxtHello, "Hello");
HBufC* hbuf = KTxtHello().AllocL();
hbuf->ReAllocL(50); // should be hbuf = hbuf->ReAllocL(50);
CleanupStack::Pop(hbuf);
CleanupStack::PushL(hbuf);
```

Be aware also that if `ReAlloc()` is being used and the allocation fails, then `NULL` is returned so this should be checked for:

```
HBufC* heapBuf = KTxtHello().AllocL();
heapBuf = heapBuf->ReAlloc(KSomeLength);
TPtr ptr = heapBuf->Des(); // Kern Exec 3 if ReAlloc() fails
                        // to allocate memory
```

Additional care must be taken if re-allocation is occurring within a method. For example, consider the following code:

```
void copyTextL(HBufC*& aDes)
{
    _LIT(KTxtHello, "Hello");
    if (aDes->Des().MaxLength() < KTxtHello().Length())
    {
        aDes = aDes->ReAllocL(KTxtHello().Length());
        // may change the address of the parameter
        aDes->Des() = aSource;
    }
}
```

```
HBufC* hBuf = HBufC::NewL(3);
...
copyTextL(hBuf);
```

The call to `ReAllocL()` may change the address of the parameter, but the `copyTextL()` function cannot know if the `HBufC` has been allocated to the cleanup stack or not. If it has been pushed and the call to `ReAllocL()` changes the address of `aTarget` then if the calling function calls `CleanupStack::PopAndDestroy()` there will be a `User-44` panic. The `copyTextL()` method could pop `aTarget` off the stack and then back on again in case the calling code had pushed it to the cleanup-stack; if the calling code hadn't pushed it to the cleanup-stack there will be an `E32User-CBase 63` panic.

The `copyTextL()` method could specify as a pre-condition that `aTarget` must not be on the cleanup-stack before the function is called, but then the calling code would have to `TRAP` the call to `copyTextL()` in the event of a leave. However specifying such a pre-condition inevitably leads to complexity, is error prone and inelegant. In such a situation, it is preferable to use an `RBuf` instead, as described in Section 5.9.

Be Aware of the Difference Between %s and %S when Formatting

If you are passing a descriptor to a method that takes a formatting specifier, then be aware that C-style strings are formatted using `%s` and descriptors are formatted using `%S`; the address of the descriptor must be passed as the argument. The following code causes an exception:

```
_LIT(KTxtHelloWorld, "Hello World!");
TBufC<20> constantBuffer(KTxtHelloWorld);
_LIT(KFormatTxt, "%S");
console->Printf(KFormatTxt, constantBuffer);
// should be &constantBuffer
```

Use _LIT Instead of _L

In older versions of Symbian OS, the `_LIT` macro and `TLitC` class didn't exist. The only way of creating a literal was to use the `_L` macro that produces a `TPtrC` from a literal value and can be used without a name, so you can write the following in your program:

```
TPtrC helloRef(_L("hello"));
```

While this works, it is inefficient and the `_LIT` macro was created as a more efficient alternative. When using `_LIT`, no run-time temporary

and no inline constructor code are generated and so the ROM budget is significantly reduced for components that contain many string literals.

The use of `_L` is deprecated. It is nevertheless still acceptable to use it in test programs where it is convenient as it doesn't have to be declared separately:

```
User::Panic(_L("Test code panic"), KMyTestCodeError);
```

You may occasionally see the macro `_S`. Like `_L`, it too is deprecated for production code but it is still used in test code. It is similar to the `_L` macro in the way the string is stored in the program binary but it does not construct a temporary `TPtrC` around the string.

```
TText* text = _S("Hello World\n");
```

`_L`, `_S` and `_LIT` are defined in `e32def.h` as:

```
#if defined(_UNICODE)
#define _L(a) (TPtrC((const TText *)L ## a))
#define _S(a) ((const TText *)L ## a)
#define _LIT(name,s) static const TLitC<sizeof(L##s)/2>
                        name={sizeof(L##s)/2-1,L##s}
#else
#define _L(a) (TPtrC((const TText *)a))
#define _S(a) ((const TText *)a)
#define _LIT(name,s) static const TLitC<sizeof(s)>
                        name={sizeof(s)-1,s}
#endif
```

Avoid Declaring Literals in Header Files

When using literals, try not to declare them in a header file because all CPP files that include it generate a new copy, leading to code bloat. Instead put your `_LITs` in the CPP file and have a method return a reference to it:

```
_LIT(KTxtHelloWorld, "Hello World!");
EXPORT_C const TDesC& CMyClass::GetHelloText() const
{ return KTxtHelloWorld; }
```

Initialize `TPtr` Class Data Members

`TPtr` does not have a default constructor, so when it is used as a data member of a class, it must be initialized in the constructor initialization list, otherwise a compilation error is generated.

```

class CSomeClass : public CBase
{
    ...
    TPtr iPtr;
};

CSomeClass::CSomeClass() : iPtr(0,0)
{

```

Perform Bounds Checking

Descriptors perform checks when data is being inserted or appended and so they panic if the resulting length would be greater than the maximum length, therefore you should always take care to ensure there is sufficient space for the operation to succeed:

```

_LIT(KSomeText, "Some Text");

void Foo(TDes& aDescriptor)
{
    if (aDescriptor.Length() + KSomeText().Length() <=
        aDescriptor.MaxLength())
        aDescriptor.Append(KSomeText);
    else
        // handle error situation
}

```

In order to keep the example code small, the examples in this chapter do not contain any explicit bounds checking but that doesn't mean you should not do it.

Implicit bounds checking is performed if you attempt to access an item beyond the end of the descriptor. The following code generates a panic:

```

_LIT(KTxtHello, "Hello");
TBufC<5> buf(KTxtHello);
TChar ch = buf[KTxtHello().Length()];

```

You should also be mindful not to confuse the `Length()` and `Size()` methods. `Length()` returns the conceptual length of a string while `Size()` returns the total number of bytes it occupies. These are not always the same – a narrow string's size is the same as its length but a wide string's length is half its size.

```

"Hello"    // this narrow string's conceptual length is 5,
           // one byte per character
L"Hello"   // this wide string's conceptual length is also 5,
           // but with two bytes per character its size is 10

```

5.12 Manipulating Descriptors

This section provides a brief summary of the descriptor methods available. It is not possible to describe the API in full here due to its large size. For more information, consult your SDK.

The methods use some descriptor-specific naming conventions:

- a method name containing an uppercase F, such as `LocateF()`, indicates *folding*, a process that sets everything to upper case and removes accents
- a method name containing LC, such as `CopyLC()`, indicates lower case, provided the descriptor is not a heap-based descriptor (`RBuf` or `HBuFC`); for heap-based descriptors, LC follows the standard Symbian convention, as in `AllocLC()`, to mean leave and push to the cleanup stack
- a method name containing UC, such as `CopyUC()`, indicates upper case
- a method name containing an uppercase C, but not preceded by an uppercase L, such as `FindC()` indicates *collation*, the process of removing any differences between characters so that they can be put in a sequence that allows for straightforward comparisons.

Folding

Folding is a relatively simple way of normalizing text for comparison by removing case distinctions, converting accented characters to characters without accents, etc. Folding is used for tolerant comparisons, i.e. comparisons that are biased towards a match.

For example, the file system uses folding to decide whether two file names are identical or not. Folding is locale-independent behavior, and means that the file system, for example, can be locale-independent.

Variants of member methods that fold are provided where appropriate. For example, `TDesc16::CompareF()`¹⁴ for folded comparison.

It is important to note that there can be no guarantee that folding is in any way culturally appropriate. It should not be used for comparing strings in natural language; collation is the correct functionality for this.

Collation

Collation is a much better and more powerful way to compare strings and produces a dictionary-like ('lexicographic') ordering. Folding cannot

¹⁴ Be aware that `CompareF()` is much slower than `Compare()`.

remove accents or deal with correspondences that are not one-to-one (such as the mapping from German upper case *ß* to lower case *ß*). In addition, folding cannot optionally ignore punctuation.

For languages using the Latin script, for example, collation is about deciding whether to ignore punctuation, whether to fold upper and lower case, how to treat accents, and so on. In a given locale there is usually a standard set of collation rules that can be used. Collation should always be used for comparing strings in natural language.

Variants of member functions that use collation are provided where appropriate. For example, `TDesC16::CompareC()` for collated comparison.

Basic Functions

Use `Length()` to find the logical length of data, for example how many characters it is, and `Size()` to find out how many bytes the data occupies. Be careful not to use `Size()` where you really mean `Length()`.

`TDes` provides `MaxLength()`, which tells you the maximum length of the data area. If the descriptor is intended to hold a string, then this is the maximum number of characters that the descriptor can hold. Any manipulation method that exceeds this causes a panic.

If you write your own string-handling methods, you normally construct them using the descriptor library methods. In some circumstances, you may wish to access descriptor contents directly, and manipulate them using C pointers; `Ptr()` allows you to do this – but be careful. In particular, make sure that you honor `MaxLength()` if you're modifying descriptor data, and make sure that any method you write panics if asked to do something that would overflow the allocated `MaxLength()`; the standard way of ensuring this is to call `SetLength()` before modifying the content, which panics if the length exceeds `MaxLength()`.

Manipulating Data

- `TDes::Copy()` copies data to the descriptor starting at the beginning, while `Append()` can copy additional data to the descriptor, starting where the existing data stops. There are variants of the `Copy()` method that perform case or accent folding when copying character data.
- `Insert()` inserts data into any position, pushing subsequent data towards the end of the descriptor data area.
- `Delete()` deletes any sequence of data, moving down subsequent data to close the gap.
- `Replace()` overwrites data in the middle of the descriptor data area.

Substring Methods

The `TDesC` methods `Left()`, `Right()`, and `Mid()` are especially applicable to strings. They let you construct `TPtrC` descriptors that represent portions of existing descriptor data. For example, you might have a stack descriptor containing text that needs parsing. Following the parse operation you might have a number of `TPtrC` descriptors, each of which represents syntactically significant parts of the buffer content. The original descriptor data is not changed, deleted or copied.

Formatting

`TDes::Format()` is a bit like `sprintf()`: it takes a format string and a variable argument list and saves the result in the descriptor. Methods such as `AppendFormat()` are similar but append the result to existing descriptor data. Methods such as `Format()` are implemented in terms of `AppendFormat()`.

Many lower-level methods exist to support `AppendFormat()`: various `Num()` methods convert numbers into text and corresponding `AppendNum()` methods append converted numbers onto an existing descriptor. For simple conversions, the `AppendNum()` methods are much more efficient than using `AppendFormat()` with a suitable format string.

In C, scanning methods are provided by `sscanf()` and packaged variants such as `scanf()`, `fscanf()` and so on. Similar methods are available in Symbian OS through `TLex` and associated classes which scan data held in descriptors. These methods are relatively specialized and it was not thought appropriate to implement them directly in `TDesC`.

TDesC Methods

All descriptors are derived from the abstract base class `TDesC` which provides the following methods for accessing the data but doesn't provide any methods for altering the descriptor contents.

Name	Description
<code>Alloc()</code> <code>AllocL()</code> <code>AllocLC()</code>	Creates and returns an <code>HBufC*</code> initialized with a copy of the descriptor's data.
<code>Compare()</code> <code>CompareF()</code> <code>CompareC()</code>	Compares the descriptor's data with a supplied descriptor.
<code>Find()</code> <code>FindC()</code> <code>FindF()</code>	Searches for the first occurrence of a specified data sequence, with the search beginning at the start of the descriptor.

Name	Description
<code>HasPrefixC()</code>	Compares a possible prefix against the start of the descriptor, using a collated comparison.
<code>Left()</code>	Returns a <code>TPtrC</code> representing the leftmost <code>n</code> characters.
<code>Length()</code>	Returns the length of the data. Do not confuse it with <code>Size()</code> . For narrow descriptors, they return the same value; for wide descriptors, the size is twice the length.
<code>Locate()</code> <code>LocateF()</code> <code>LocateReverse()</code> <code>LocateReverseF()</code>	Searches for the first occurrence of a character within the descriptor's data, starting the search from the leftmost or rightmost position.
<code>Match()</code> <code>MatchC()</code> <code>MatchF()</code>	Searches the descriptor's data for a match with a match pattern supplied in the specified descriptor.
<code>Mid()</code>	Returns a <code>TPtrC</code> representing the middle <code>n</code> characters from position <code>m</code> .
<code>operator<()</code> <code>operator<=()</code> <code>operator>()</code> <code>operator>=()</code> <code>operator==()</code> <code>operator!=()</code>	Returns a <code>TBool</code> indicating the result of the comparison between the descriptor and a specified descriptor.
<code>operator[]()</code>	Returns a <code>const TUInt&</code> to a specified single data item within the descriptor data. Iterating over a descriptor using <code>operator[]</code> is expensive; consider using C++ pointer arithmetic and the <code>TDesC::Ptr()</code> function instead.

Name	Description
<code>Ptr()</code>	Returns a <code>const TUInt*</code> to the data represented by the descriptor.
<code>Right()</code>	Returns a <code>TPtrC</code> representing the rightmost <code>n</code> characters.
<code>Size()</code>	Returns the size of the data in bytes; do not confuse it with <code>Length()</code> .

TDes Methods

Descriptors that derive from `TDes` are directly modifiable. `TDes` provides the methods to alter the contents of these descriptors, summarized below.

Name	Description
<code>Append()</code>	Appends a character, a descriptor, or data onto the end of the descriptor.
<code>AppendFill()</code>	Appends and fills the descriptor with a specified character.
<code>AppendJustify()</code>	Appends data and justifies it.
<code>AppendFormat()</code> <code>AppendFormatList()</code>	Formats and appends text onto the end of the descriptor.
<code>AppendNum()</code> <code>AppendNumUC()</code> <code>AppendNumFixedWidth()</code> <code>AppendNumFixedWidthUC()</code>	Converts a number into a character representation or fixed-width character representation and appends to the end of the descriptor.
<code>Capitalize()</code>	Capitalizes the content of the descriptor.
<code>Collapse()</code>	Narrows each character.

Name	Description
<code>Collate()</code>	Performs collation on the content of the descriptor (this method is deprecated).
<code>Copy()</code> <code>CopyC()</code> <code>CopyCP()</code> <code>CopyF()</code> <code>CopyLC()</code> <code>CopyUC()</code>	Copies data into the descriptor with various options. <code>TDes16::Copy(const TDesC8&)</code> converts from a narrow descriptor to a wide descriptor; with <code>TDes8::Copy(const TDesC16&)</code> each double-byte value can only be copied into the corresponding single byte when the double-byte value is less than decimal 256. A double-byte value of 256 or greater cannot be copied and the corresponding single byte is set to a value of decimal 1.
<code>Delete()</code>	Delete <i>n</i> characters of data from the descriptor starting at position <i>m</i> .
<code>Expand()</code>	Zero-extends each narrow character to 16 bits (only available for <code>TDes8</code> and from Symbian OS v8.1)
<code>Fill()</code> <code>FillZ()</code>	Fills the descriptor with a specified <code>TChar</code> or with binary zeros.
<code>Fold()</code>	Performs folding on the contents of the descriptor.
<code>Format()</code> <code>FormatList()</code>	Formats and copies text into the descriptor.
<code>Insert()</code>	Inserts data into the descriptor from position <i>n</i> .

Name	Description
<code>Justify()</code>	Copies data into the descriptor and justifies it, replacing any existing data.
<code>LeftTPtr()</code>	Returns a <code>TPtr</code> to the leftmost portion of the descriptor of length <code>n</code> .
<code>LowerCase()</code>	Converts the contents of the descriptor to lower case.
<code>MaxLength()</code>	Returns the maximum length of the descriptor.
<code>MaxSize()</code>	Returns the maximum size of the descriptor.
<code>MidTPtr()</code>	Returns a <code>TPtr</code> to a portion of the descriptor starting at position <code>p</code> and of length <code>l</code> .
<code>Num()</code> <code>NumUC()</code> <code>NumFixedWidth()</code> <code>NumFixedWidthUC()</code>	Converts a number into a character representation (with fixed width) and copies it to the descriptor.
<code>operator=()</code> <code>operator+=()</code>	Copies or appends data into the descriptor.
<code>operator[]()</code> <code>operator[]() const</code>	Returns a <code>TUint&</code> or <code>const TUint&</code> to a specified data item in the descriptor.
<code>PtrZ()</code>	Appends a NULL terminator onto the end of the descriptor's data and returns a pointer to the data; this allows the pointer to be used directly as a C string. The length of the descriptor must be less than its maximum length to allow for the addition of the terminator.

Name	Description
Repeat()	Copies data with repetition into the descriptor, either from another descriptor or a specified memory location.
Replace()	Replaces data in the descriptor.
RightTPtr()	Returns a TPtr to the rightmost n characters of the descriptor.
SetLength()	Sets the length of the data represented by the descriptor.
SetMax()	Sets the length of the data to the maximum length of the descriptor.
Swap()	Swaps the data represented by the descriptor with the data represented by a specified descriptor.
Trim() TrimAll() TrimLeft() TrimRight()	Presents various options for deleting white space within the descriptor.
UpperCase()	Converts the contents of the descriptor to upper case
Zero()	Sets the length of the data to zero.
ZeroTerminate()	Appends a zero terminator onto the end of this descriptor's data.

HBuFC Methods

Name	Description
Des()	Creates and returns a modifiable pointer descriptor (TPtr) for the data represented by the descriptor, thus allowing modification.

Name	Description
New() NewL() NewLC() NewMax() NewMaxL() NewMaxLC()	Creates an empty HBufC and returns a pointer to it. With New(), NewL() and NewLC(), the length of the data is set to 0; while with NewMax(), NewMaxL() and NewMaxLC(), the length is set to the maximum length that was passed as a parameter.
NewL() NewLC()	Creates an HBufC and internalize data into it from a specified RReadStream before returning a pointer to it.
ReAlloc() ReAllocL()	Creates and returns a new HBufC* which is an expansion or contraction of the original.
operator=()	Copies data into the descriptor, replacing any existing data.

RBuf Methods

Name	Description
Assign()	Assigns or transfers ownership of an HBufC*, a specified memory location, or another RBuf to the descriptor.
CleanupClosePushL()	Pushes an instance of the RBuf to the cleanup stack.
Close()	Frees any allocated memory owned by the RBuf.
Create() CreateL() CreateMax() CreateMaxL()	Creates the descriptor, allocating sufficient memory to contain the data up to a specified length. Overloads exist to assign data to the descriptor from various specified sources including an RReadStream.
operator=()	Assignment operator.

Name	Description
<code>RBuf()</code>	Creates the descriptor, optionally transferring ownership from a specified <code>HBufC*</code> .
<code>ReAlloc()</code> <code>ReAllocL()</code>	Resizes the memory holding the descriptor data.
<code>Swap()</code>	Swaps the contents of the descriptor with another <code>RBuf</code> .

TBuf Methods

Name	Description
<code>TBuf()</code>	Various options for creating a <code>TBufC</code> .
<code>operator=()</code>	Various options for assignment to the descriptor.

TBufC Methods

Name	Description
<code>TBufC()</code>	Various options for creating a <code>TBuf</code> .
<code>operator=()</code>	Various options for assignment to the descriptor.
<code>Des()</code>	Creates and returns a modifiable descriptor (<code>TPtr</code>) to the data, allowing a means of altering it.

TPtr Methods

Name	Description
<code>operator=()</code>	Various options for copying data into the descriptor. This operator does not do what you might expect for a standard C++ assignment operator (see Section 5.7).

Name	Description
<code>TPtr()</code>	Constructs the descriptor from data in a specified location in memory.
<code>Set()</code>	Sets the descriptor to an existing descriptor or to a specified memory location (which may be in ROM or RAM).

TPtrC Methods

Name	Description
<code>TPtrC()</code>	Constructs the descriptor from another descriptor or from data in a specified location in memory.
<code>Set()</code>	Sets the descriptor to an existing descriptor or to a specified memory location (which may be in ROM or RAM).

Summary

In this chapter we've seen how Symbian OS handles strings and binary data using descriptors. We covered a lot of material and there is much to remember. The main summary points are:

- descriptors are used for storing both string and binary data
- use the `_LIT` macro to store data in the program binary
- use a `TBuf` or `TBufC` when you need to construct a descriptor, use it and discard it, provided that the maximum size is not too large
- use a `TPtrC` when you need to get a substring of another descriptor or when you need to pass data, not in the form of a descriptor, to a method expecting a descriptor
- use a `TPtr` when you want to modify a `TBufC`, `HBufC` or some raw data as though it were a descriptor
- use a `TPtrC` or `TPtr` when you need to refer to raw data or an existing descriptor

- use an `HBufC` or `RBuf` when the length of the descriptor is unknown at compile time or when the maximum length is large
- use `const TDesC&` and `TDes&` to pass descriptors to methods
- use `const TDesC&`, `TDes&`, `TPtr` or `HBufC*` to return descriptors from methods.

6

Active Objects

In the past, programs were written so that, every so often, the program would decide to check for user input and would then process it. Essentially the application would *poll* to see if the user had initiated an action and then respond by carrying out some task. However, with modern GUI systems, the user is in control – their input is the focus of the application's existence. In this paradigm, the application can be thought of as being *event-driven* – it is always waiting for the user to interact with the device and then carries out some operation in response to that interaction.

In GUI systems, application programs spend the majority of their time waiting for events, for example, keyboard input, pointer input, completion of an I/O request, timer events, etc. These events and the services associated with them are provided by *asynchronous service providers*. Application programs then simply *respond* to these events.

Symbian OS provides some fundamental building blocks for event-handling systems, known as active objects. Here, we try to give a flavor of how they work and why they are important.

6.1 The Asynchronous Service

At the heart of what we call event-driven programming is the concept of the asynchronous service.

When a program requests a service of some other component or part of the system, that service can be performed either *synchronously* or *asynchronously*. A synchronous service is the 'usual' pattern for function calls; when the function returns, the service requested has either been performed or some kind of error has been returned. The program then continues executing based upon the outcome of the synchronous operation.

An asynchronous service is *requested* by a function call, but completion occurs later and is indicated by some kind of signal. In this way, the

program flow is split into two entities. First, the operation is requested and then later, when the outcome of the operation is known, the program continues execution based upon this outcome.

It is normal, though not mandatory, that the requesting entity is partitioned in some way from the component that implements the service being requested, known as the *service provider*. For example, the requesting component and service provider may be in different threads.

Between the issue of the request and the receipt of the signal, the request is said to be *pending*. The requesting program may do other processing while the request is pending but until the signal is received the requested operation cannot be assumed to have been completed. Therefore, ultimately the requesting entity waits, or sleeps, until the signal is received. The operating system wakes up the requesting entity when completion of its pending request is signaled.

This is the crux of a typical event-driven program; it makes requests for keyboard input, for pen events, timer events, etc. and, once it has nothing else to do, it sits and waits. When an event occurs, the program comes out of its wait state, responds to the event in whatever manner is appropriate, typically issues a request for another similar event, and then waits again.

Figure 6.1 shows the typical pattern in a simplistic way. Ignoring start up, shut down, and many other issues, there's a central core that waits; when an event occurs, it calls the code that can deal with that event; control then flows back to the central core.

Consider Figure 6.1 representing code running in a single thread. If only one event-handling section of code can run at any one time, such as the code that runs when handling a key event, and whilst that code

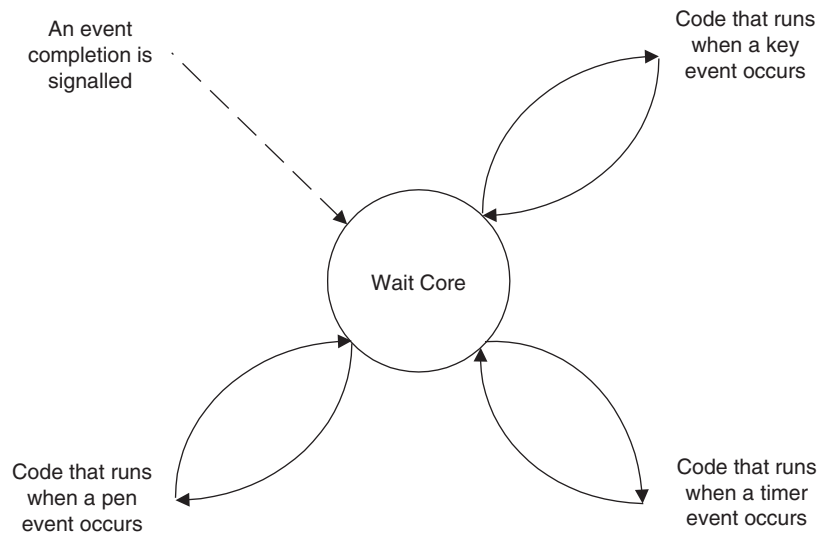


Figure 6.1 Overview of an event-driven program

is running nothing else in that thread can run until control flows back to the central wait core, then this represents non-pre-emptive multitasking within the context of a single thread.

To help illustrate these concepts further, let's consider a simple use case where the thread-event handling is implemented in a non-pre-emptive multitasking way. The use case describes a program that requests to be informed when:

- a keypress event occurs
- a ten-minute timer expires.

Specific actions might be carried out when these events are triggered, such as updating the screen or playing a sound.

There are four key actors involved in the use case: a keyboard object, a timer object, an initialization object and a wait object.

Let's consider the key roles of each actor in turn:

- the `Initialization` object represents the logic associated with the preparatory stage of the program; it controls the creation and initialization of the other objects
- the `Timer` object understands how to request that an event occur at a specific time; it wants to perform some action when the timer event occurs
- the `Keyboard` object understands how to request an event when the user presses a key on the keyboard; it wants to be able to perform some action, for example, updating the display when a key is pressed
- the `Waiter` object is responsible for detecting when an event has occurred and then ensuring that any resultant action required to handle the event takes place.

Figure 6.2 represents the execution of the program.

Initialization Phase

1. At the beginning, the initialization code prepares the other objects to carry out the use case.
2. Firstly, the initializer instructs the keyboard object to make an asynchronous request to receive any future key events. After the asynchronous request is issued, the keyboard object's request is pending.
3. The control returns to the initialization object. Since requests are asynchronous, the initialization code can continue to interact with other objects.

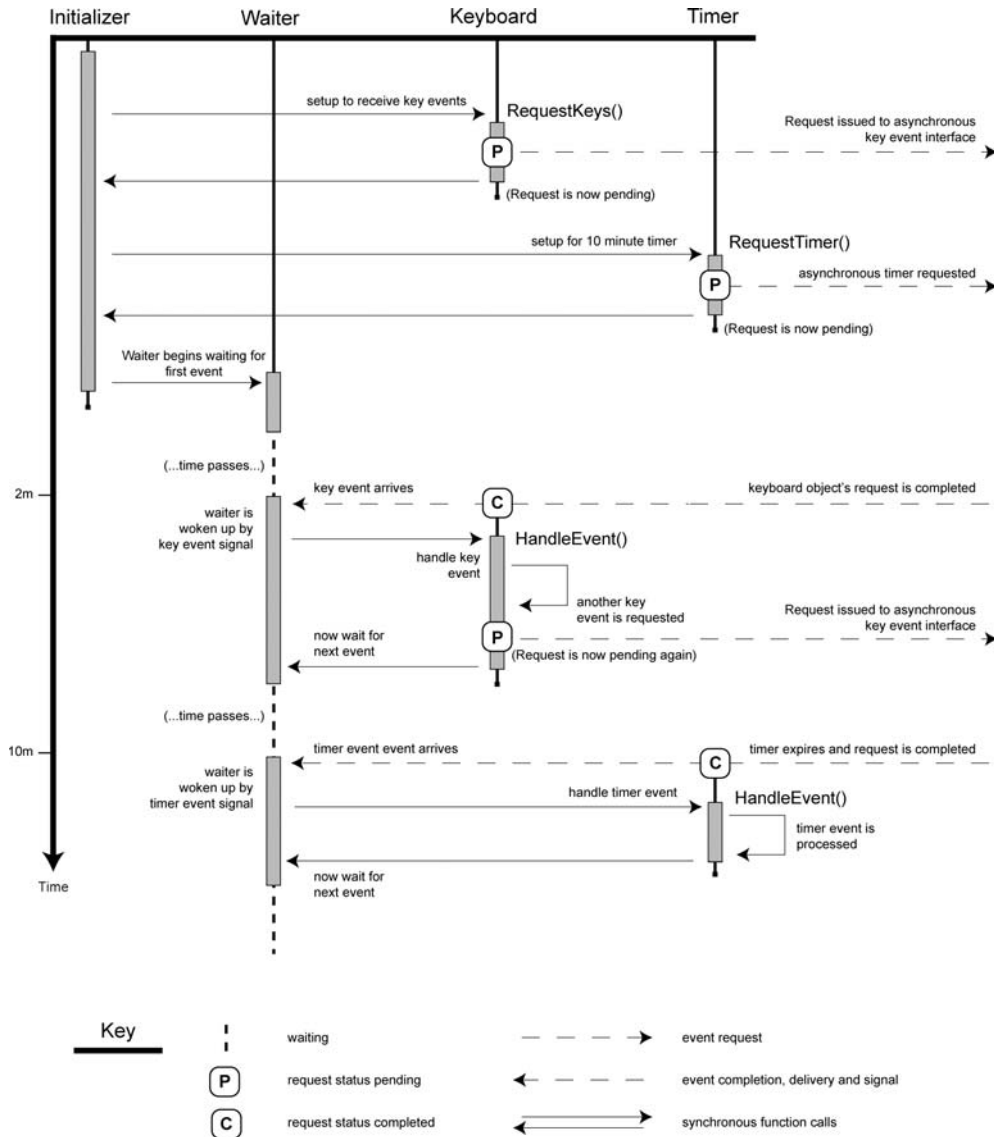


Figure 6.2 An event-driven program

- Next, a call is made to prepare the timer object. The timer requests that an event occurs in ten minutes. Again, the timer's request becomes pending.

Waiting Phase 1

- The objects have been prepared to receive the events when they occur. The program now enters the waiting phase. In this state, the

initialization object finishes its task by requesting that the waiter object start to wait for event-completion signals. At this point, the program is now truly asynchronously event-driven. It is waiting for some external input, such as a keypress to continue execution of the program's code.

The Keyboard Event Dispatch Phase

6. In this example, we can see that a key event occurs at the two-minute mark. The act of completing the keyboard object's asynchronous request moves its status from pending to completed, as represented by the C symbol.
7. At this point, the wait object is signaled that an asynchronous request has been completed and therefore it awakens.
8. The waiter object is able to identify that a keyboard event has occurred and therefore it activates the keyboard object to process the event.
9. After the keyboard object has processed the event, it makes another asynchronous request to receive any future key events. Again its request becomes pending, represented by the P symbol.

Waiting Phase 2

10. Now that the key event has been fully processed, program execution returns to the waiter object. Again, it begins to wait for the signal that another request has been completed.

The Timer Event Dispatch Phase

11. At the tenth minute of execution, the timer event occurs. The timer object's status changes from pending to completed.
12. The waiter is signaled about the event. This causes it to awaken.
13. The waiter detects the completion of the timer event and acts accordingly, asking it to handle the event.
14. The timer object processes the event, perhaps playing a tone.

Waiting Phase 3

15. Again, since the event has been fully handled, the timer returns program execution control to the waiter. It waits for a signal that another request has been completed.

The above logic illustrates several of the key concepts implemented by Symbian's active object framework. The code that runs when an event

occurs is represented by an active object. Typically the active object also initiates its own asynchronous request, as was the case in this example.

The decision as to which active object should be called upon to process an event is taken by the waiter, called the *active scheduler*. The active scheduler is responsible for listening to a signal that informs it when a request has been completed, and then it decides which active object is responsible for handling the associated event.

Symbian OS provides a framework for asynchronous requests, active objects and an active scheduler at its lowest level. Each thread can contain only one current active scheduler, but the scheduler can contain many active objects. We look at this in more detail in the following sections.

Asynchronous requests are, more often than not, handled by servers. While it is useful to know this, it is not a prerequisite to understanding the way in which active objects work.

6.2 Multitasking and Pre-emption

Symbian OS implements pre-emptive multitasking so that it can run multiple applications and servers concurrently. Active objects are used to implement non-pre-emptive multitasking within the context of a single thread.

In Figure 6.2, we assumed that events, or request completions, occur in some kind of predictable order and that our thread gets a chance to deal with an event before the next event occurs. Of course, in reality it doesn't happen like that. The timing of events is unpredictable, and when our thread gets a chance to run, there may be more than one event ready to be handled. This is why active objects, like threads, have priorities that affect their scheduling. On completion of an active object's execution, such as handling a timer event, control returns to the active scheduler, which then schedules further active object calls according to the following rules:

- if there is just one object eligible to run, run it
- if there is more than one eligible object, choose the one with the highest priority
- if there are no eligible objects, then wait for the next event and decide what to do based on these rules.

The function that gets called to handle an event is the active object's `RunL()`. The act of calling a `RunL()` is something that we will refer to as a `RunL()` *dispatch*.

Some events are more important than others. It is much better to handle events in priority order than in a simple first-in, first-out (FIFO) order. Events that control the thread (keypress events to an application, for example) can be handled with higher priority than others (for example, some types of animation). But, once a `RunL()` has started – even for a low-priority event – it runs to completion. No other `RunL()` can be called until the current one has finished. That is not a problem provided all your event handlers are short and efficient.

Non-pre-emptive multitasking is surprisingly powerful. Actually, there should be no surprise about this; it's the natural paradigm to use for event handling. For example, the window server handles key and pointer events, screen drawing, requests from every GUI-based application in the system, and animations including a flashing text cursor, sprites and self-updating clocks. It delivers all this sophistication using a single thread with multitasking based on active objects.

In many systems, the preferred way to multitask is to multithread. In Symbian OS, the preferred way to multitask is to use active objects.

In a truly event-handling context, active objects give only gains over using threads: you lose no functionality, as events occur sequentially and can be handled in priority order. You gain convenience over threads because you know an active object cannot be pre-empted; you do not need to use mutexes, semaphores, critical sections, or any kind of synchronization to protect against the activities of other active objects in your thread. Your `RunL()` is guaranteed to be an atomic operation. It is also more efficient; you do not incur the overhead of a context switch, nor the per-thread memory overhead of both user-side and kernel-side stacks, when switching between active objects.

Non-pre-emptive multitasking in Symbian OS is not the same as co-operative multitasking. In co-operative multitasking, one task says: 'I am now prepared for another task to run' for instance by using `Yield()` or some similar function. What this really means is, 'I am a long-running task, but I now wish to yield control to the system so that it can get any outstanding events handled if it needs to'. Active objects do not work like that – during `RunL()`, your event-handling code is the only code that executes within your thread. Only after `RunL()` has finished can other events be handled by your application.

All multitasking systems require a degree of co-operation so that tasks can communicate with each other where necessary. Active objects require less co-operation than threads because they are not pre-emptively scheduled. They can be just as independent as threads. As mentioned earlier, each thread can contain an active scheduler. The thread's active scheduler manages active objects independently of one another, just as

the kernel scheduler manages threads independently of one another. It is worth mentioning that, whilst active objects themselves are non-pre-emptive, the threads in which they reside are scheduled pre-emptively.

6.3 A More In-depth Look at Active Objects

An event-handling thread has a single active scheduler, which is responsible for deciding the order in which events are handled. It also has one or more active objects, derived from the `CActive` class, which are responsible both for issuing requests (which later result in an event happening) and for handling the event when the request completes. When an event occurs, the active scheduler calls `RunL()` on the active object associated with the completed event.

The active scheduler is, in effect, a ‘wait loop’ or wait-object that waits for events. On receipt of a signal marking the occurrence of an event, the active scheduler decides which event, of the many it may be expecting, has occurred and then *dispatches* the `RunL()` function of the appropriate active object.

Symbian OS makes extensive use of active objects, particularly at the UI layer. For example, the UI framework always maintains an outstanding request to the window server for user input and other system events. When an event occurs and this request is completed, it calls the appropriate application UI and control functions to ensure that the application handles the event properly. So, ultimately, all application code is handled under the control of an active object’s `RunL()`.

Because this framework is provided for you, you can get started with Symbian OS application programming without knowing exactly how active objects work. However, for more advanced GUI programming, and for anything to do with servers or any service that is requested as an asynchronous request, you do need to know how they work in detail. With this understanding, you can not only use the large array of asynchronous services provided by the operating system, but you can also create your own.

The best way to understand how to use active objects is to look at a simple example. The example shown in Figure 6.3 demonstrates the use of two active objects. Figure 6.3 shows what it looks like when you launch the program and press the ‘Options’ key to display its menu.

The Set hello menu item triggers a Symbian OS timer. Three seconds later, that timer completes, causing an event. This event is handled by an active object, which puts a message saying "Hello world!" on the screen. During that three-second period, you can select the Cancel menu item to cancel the timer, so that the message never appears.

The Start flashing menu item starts to flash the message and Stop flashing stops it. The flashing is implemented by an active object that



Figure 6.3 Two active objects on a menu

creates regular timer events and then handles them. Its event handling changes the visibility of the text and redraws the view.

The two active objects represent two different styles of usage:

- a one-off event is generated and its completion handled
- an event is generated, and as part of the handling of the completion of that event, a further asynchronous request is issued and therefore another event is generated so that we have a continuous cycle, broken only by selecting the Stop Flashing menu item.

The Set Hello Menu Item

The behavior that we see when the Set hello menu item is selected is handled by the `CDelayedHello` class:

```
class CDelayedHello : public CActive
{
public:
    // Construct/destruct
    static CDelayedHello* NewL();
    ~CDelayedHello();
    // Request
    void SetHello(TTimeIntervalMicroSeconds32 aDelay);
private:
    // Construct/destruct
    CDelayedHello();
    void ConstructL();
    // from CActive
    void RunL();
    void DoCancel();
    TInt RunError(TInt aError);
private:
    RTimer iTimer;                // Has
    CEikonEnv* iEnv;              // Uses
};
```


From the declaration, you can see that CDelayedHello:

- ‘is-a’ active object, derived from CActive
- ‘has-a’ event generator, an RTimer object
- defines a request function, SetHello(), that requests an event from the RTimer
- implements the RunL() function to handle the event generated when the timer request completes
- implements the DoCancel() function to cancel an outstanding timer request
- overrides the default implementation of RunError() to handle the case if RunL() leaves.

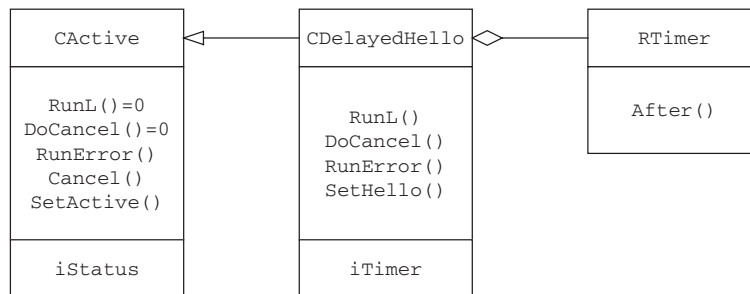


Figure 6.4 The basic pattern of an active object

All active object classes share this basic pattern (see Figure 6.4). They are derived from CActive and implement its RunL(), DoCancel() and RunError() functions. They include an event generator and at least one request function.

Construction

Construction of a CDelayedHello active object is implemented by the NewL() static function, which follows the standard pattern that we saw when we looked at error handling and cleanup. Here it is:

```

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

CDelayedHello::CDelayedHello() : CActive(CActive::EPriorityStandard)
{
    CActiveScheduler::Add(this);
}
  
```

The C++ constructor is required for any derived active object class. Inside it, you call the `CActive` constructor to specify the active object's *priority*. The active scheduler uses this value to decide which active object is to be dispatched first if it finds that more than one event is ready to be handled. You should specify `CActive::EPriorityStandard` here unless there are good reasons to specify something lower or higher (we look at those reasons later). The new object adds itself to the active scheduler, so that the active scheduler can include it in event handling. Adding the object to the active scheduler does not allocate memory; which is why we can do this in the C++ constructor – it cannot leave.

```
void CDelayedHello::ConstructL()
{
    iEnv = CEikonEnv::Static();
    User::LeaveIfError(iTimer.CreateLocal());
}
```

The second-phase constructor, the `ConstructL()` function, gets a pointer to the UI environment, and then uses `iTimer.CreateLocal()` to request that the kernel create a kernel-side timer object, which we access through the `RTimer` handle. If there is any problem here, it leaves.

Destruction

```
CDelayedHello::~~CDelayedHello()
{
    Cancel();
    iTimer.Close();
}
```

Before we destroy the active object, the destructor must cancel any outstanding requests for timer events. We do this by calling the standard `CActive` function `Cancel()`. It works by checking to see whether a request for a timer event is outstanding and, if so, calls `DoCancel()` to handle it. `Cancel()` should be called before other resources are destroyed in case they are used by the service provider or the `DoCancel()` method.

An active object must implement a `DoCancel()` function and must also call `Cancel()` in its destructor.

The destructor closes the `RTimer` object, which destroys the corresponding kernel-side object. After this, the base `CActive` destructor is invoked and this removes the active object from the active scheduler.

Requesting and handling events

`SetHello()` requests a timer event after a given delay:

```
void CDelayedHello::SetHello(TTimeIntervalMicroSeconds32 aDelay)
{
    _LIT(KDelayedHelloPanic, "CDelayedHello");
    __ASSERT_ALWAYS(!IsActive(), User::Panic(KDelayedHelloPanic, 1));
    iTimer.After(iStatus, aDelay);
    SetActive();
}
```

Every line in this function is important.

- First, we assert that no request is already outstanding (that `IsActive()` is false). The client program must ensure that this is the case, either by refusing to issue another request when one is already outstanding or by canceling the previous request prior to issuing a new one. We must do this because an active object has no way of handling multiple concurrent requests.
- Then, we request the timer to generate an event after `aDelay` microseconds. The first parameter to `iTimer.After()` is a `TRequestStatus&` which refers to the `iStatus` member that we inherit from `CActive`. As we explain below, `TRequestStatus` plays a key role in event handling.
- Finally, we indicate that a request is outstanding by calling `SetActive()`.

This is the standard pattern for active object request functions: assert that no request is already active (or, in rare cases, cancel it prior to issuing a new request); issue a request, passing the active object's `iStatus` to some function that generates an event; then call `SetActive()` to indicate that the request has been issued.

You can deduce from this that since each active object has only a single `iStatus` member, it can be responsible for only one outstanding request at a time. You can also deduce that all request functions take a `TRequestStatus&` parameter. A `TRequestStatus` object is an important partner and we look at it in more detail in Section 6.5.

Our UI program calls this function from `HandleCommandL()`:

```
case EActiveHelloCmdSetHello:
{
    iDelayedHello->Cancel();           // just in case
    iDelayedHello->SetHello(3000000); // 3-second delay
    break;
}
```

In other words, it cancels any request so that the assertion in `SetHello()` is guaranteed to succeed and then requests a message to appear after three seconds.

When the timer event occurs, it is handled by the active object framework and results in the active object's `RunL()` being called:

```
void CDelayedHello::RunL()
{
    iEnv->InfoMsg(R_ACTIVEHELLO_TEXT_HELLO);
}
```

Clearly, this code is very simple; it's a one-line function that produces an information message with the usual greeting text.

The degree of sophistication in an active object's `RunL()` function can vary enormously from one active object to another. The application framework's `CCoeEnv::RunL()` function initiates an extremely sophisticated chain of processing. In contrast, the function above is a simple one-liner. In all cases, recall that no other asynchronous request can be dispatched by the active scheduler until the `RunL()` function runs to completion. The `RunL()` function should therefore be kept as short as possible.

Error handling for `RunL()`

`CActive()` provides the `RunError()` virtual function, which the active scheduler calls if `RunL()` leaves. You implement this to handle the leave. This means that it is generally unnecessary to TRAP errors from inside `RunL()` methods. Instead, you should implement the `RunError()` virtual function and let the active scheduler take care of trapping your `RunL()` call.

The protocol here is that your function must return `KErrNone`, which tells the active scheduler that the leave, or error, has been handled. A default implementation is provided by `CActive` which just returns the leave code:

```
EXPORT_C TInt CActive::RunError(TInt aError)
{
    return aError;
}
```

Only a return value of `KErrNone` informs the active scheduler that the leave has been handled. If the leave has not been handled then the active scheduler calls the central error handler, `CActiveScheduler::Error()`.

In UI threads, the application framework installs its own `CActiveScheduler`-derived class. This subsequently displays the leave as an error dialog. For example, in S60 it appears as a system error.

Let's take a look at our example `RunError()` implementation:

```
TInt CActiveHello::RunError(TInt aError)
{
    return KErrNone;
}
```

Clearly, in this example, `RunL()` cannot leave, and all we do is return `KErrNone` to illustrate the point. However, in more sophisticated applications, it is entirely possible that the function could leave and `RunError()` would then need to do some real work.

If an active object's `RunL()` calls a leaving function, then that active object should provide an override of the `RunError()` function.

Canceling a request

If your active object can issue requests, it must also be able to cancel them. `CActive` provides a `Cancel()` function that checks whether a request is active and, if so, calls `DoCancel()` to cancel it.

You should not call `DoCancel()` directly from elsewhere in your code. Instead, you should call `Cancel()` and let the framework decide if a call to `DoCancel()` is required.

As the implementer of the active object, you have to implement `DoCancel()`:

```
void CActiveHello::DoCancel()
{
    iTimer.Cancel();
}
```

There is no need to check whether a timer request is outstanding, because `CActive` has already checked this by the time `DoCancel()` is called. By canceling the request, we ensure that no call to `RunL()` occurs. If you use the UI's cancel function, then the text does not appear.

It is also worth mentioning again that it is best practice to always call `Cancel()` in your active object's destructor. `DoCancel()` should not leave or allocate resources.

There is an obligation for the asynchronous service provider to provide both the request function and corresponding cancel function. When requesting an asynchronous service, you call your service provider's request function. Correspondingly, when implementing your `DoCancel()` function, you must call the service provider's cancellation function.

The Start flashing menu item

The Start flashing menu item causes the text in the center of the display to start flashing and Stop flashing stops it. The flashing is implemented by an active object that creates regular timer events and then handles them. Its event handling changes the visibility of the text and redraws the view.

The behavior that we see when the Start flashing menu item is selected is handled by the `CFlashingHello` class. This is slightly more complex than `CDelayedHello` because the `RunL()` function not only handles the completion of a timer event, but it must also re-issue the timer request to ensure that a subsequent timer event occurs and `RunL()` is called again. There is also added complexity because it has drawing code.

Construction

This is the same as the Set Hello example, except that now we need to supply a pointer to the application view object, because we do some drawing.

The second-phase constructor, the `ConstructL()` function, saves the pointer to the application view object and creates the kernel-side timer object in exactly the same way as we did in the Set Hello example.

Requesting and handling events

The `Start()`, `RunL()` and `DoCancel()` functions of `CFlashingHello` show how you can maintain an outstanding request. Here's `Start()`:

```
void CFlashingHello::Start(TTimeIntervalMicroSeconds32 aHalfPeriod)
{
    _LIT(KFlashingHelloPeriodPanic, "CFlashingHello");
    __ASSERT_ALWAYS(!IsActive(), User::Panic(KFlashingHelloPeriodPanic, 1));

    // Remember half-period
    iHalfPeriod = aHalfPeriod;
    // Hide the text, to begin with
    ShowText(EFalse);
    // Issue request
```

```
iTimer.After(iStatus, iHalfPeriod);  
SetActive();  
}
```

This is called from `HandleCommandL()` when you select the Start flashing menu item; it begins by asserting that a request is not already active and ends by issuing a request – just as before. Since a whole series of requests are issued, `Start()` doesn't merely pass the half-period parameter to the `iTimer.After()`, but stores it as a member variable for later use.

`Start()` also starts off the visible aspect of the flashing process, by hiding the text (which is visible until `Start()` is called).

When the timer completes, `RunL()` is called:

```
void CFlashingHello::RunL()  
{  
    // Change visibility of app view text  
    const TBool textIsShowing = iAppView.TextIsShowing();  
    ShowText(!textIsShowing);  
    // Re-issue request  
    iTimer.After(iStatus, iHalfPeriod);  
    SetActive();  
}
```

`RunL()` changes the visibility of the text to implement the flashing effect. Then, it simply renews the request to the timer with the same `iHalfPeriod` parameter as before. As always, the renewed request is followed by `SetActive()`.

In this way, events are generated and handled and then generated again in a never ending cycle. The only way to stop the flashing is to issue a call to `Cancel()` which, as usual, checks whether a request is outstanding and, if so, calls the `DoCancel()` implementation:

```
void CFlashingHello::DoCancel()  
{  
    // Ensure text is showing  
    ShowText(ETTrue);  
    // Cancel timer  
    iTimer.Cancel();  
}
```

We make sure the text is showing, and then cancel the timer. `ShowText()` is a simple utility function that sets the visibility of the text; it simply changes `iShowText` in the application view and redraws the application view.

Because `DoCancel()` contains drawing code, it must be executed when there is an environment in which drawing is possible. This

means that we must cancel the flashing text before calling `CEikAppUI::Exit()`. We destroy the active object from the destructor of its owning class – this is the right place to destroy it, since the destructor is called in cleanup situations, while the command handler is not.

The general rule is that you should always be careful about doing anything fancy from an active object's `DoCancel()`. Nothing in a `DoCancel()` should leave or allocate resources and `DoCancel()` should complete very quickly.

6.4 How It Works

In this section, we look at the way an event-handling thread, with an active scheduler and active objects, works. The asynchronous request idiom and active object framework is built into Symbian OS at the very lowest levels, namely the user-side library, `euser.dll`, and the kernel. Every user-side thread can have a single installed active scheduler and any number of active objects registered with that active scheduler.

The kernel's representation of a user-side thread, called `DThread`, contains a pointer to the thread's active scheduler. Whilst the kernel makes no use of this pointer itself, it is provided so that it is possible to access the active scheduler from a static user function. We have already used this functionality in our earlier examples:

```
CDelayedHello::CDelayedHello() : CActive(CActive::EPriorityStandard)
{
    CActiveScheduler::Add(this);
}
```

Calling the static `CActiveScheduler::Add()` function makes a kernel-executive call to request the active-scheduler-object pointer for the thread:

```
EXPORT_C void CActiveScheduler::Add(CActive *aRequest)
{
    CActiveScheduler *pS=Exec::ActiveScheduler();
    __ASSERT_ALWAYS(pS!=NULL, Panic(EReqManagerDoesNotExist));
    __ASSERT_ALWAYS(aRequest, Panic(EReqNull));
    __ASSERT_ALWAYS(!aRequest->IsAdded(), Panic(EReqAlreadyAdded));
    pS->iActiveQ.Add(*aRequest);
}
```

This makes it convenient to access the active scheduler from anywhere in your code without having to pass around a pointer to the object explicitly. In fact, there are several static functions exposed by the `CActiveScheduler` class which all follow this pattern.

The relevant part of the class definition is as follows:

```
class CActiveScheduler : public CBase
{
public:
    IMPORT_C CActiveScheduler();
    IMPORT_C ~CActiveScheduler();
    IMPORT_C static void Install(CActiveScheduler* aScheduler);
    IMPORT_C static CActiveScheduler* Current();
    IMPORT_C static void Add(CActive* aActive);
    IMPORT_C static void Start();
    IMPORT_C static void Stop();
private:
    TPriQue<CActive> iActiveQ;
};
```

The scheduler implements an ordered queue of CActive objects. The ordering is based upon the priority that you assign to the active object when it is constructed. Higher-priority objects are inserted near the start, or head, of the queue.

The kernel also provides all user threads with a *request semaphore*. This is a standard counting semaphore object that is incremented by the kernel each time a request in that thread has been completed.

Each active object is associated with just one object that has request functions – functions that take a TRequestStatus& parameter. When a program calls an active object's request function (SetHello() in the example we saw earlier), the active object passes the request on to the asynchronous service provider. It passes its own iStatus member as the TRequestStatus& parameter to the request function and, having called the request function, it immediately calls SetActive().

The TRequestStatus is a relatively simple class that encapsulates a completion code and a series of binary flags that describe the state of a request. Valid binary bit flags for the request are ERequestPending, for a request that is issued but has not yet been completed, and EActive, which is closely related to the CActive::IsActive() and CActive::SetActive() functions, meaning that the object is able to accept events.

Here is the class definition of TRequestStatus from e32cmn.h:

```
class TRequestStatus
{
public:
    inline TRequestStatus();
    inline TRequestStatus(TInt aVal);
    inline TInt operator=(TInt aVal);
    inline TBool operator==(TInt aVal) const;
    inline TBool operator!=(TInt aVal) const;
    inline TBool operator>=(TInt aVal) const;
    inline TBool operator<=(TInt aVal) const;
    inline TBool operator>(TInt aVal) const;
```

```

inline TBool operator<(TInt aVal) const;
inline TInt Int() const;
private:
enum
{
    EActive          = 1, //bit0
    ERequestPending  = 2, //bit1
};
TInt iStatus;
TUint iFlags;
friend class CActive;
friend class CActiveScheduler;
friend class CServer2;
};

```

The `TRequestStatus` class features a range of operator overloads to enable value comparison against the request completion code, `iStatus`. Additionally, it has several friends, including `CActive` and `CActiveScheduler`, permitting direct access to the `iStatus` and `iFlags` member variables by these objects.

Before starting to service a request function, the asynchronous service provider sets the value of `TRequestStatus` to `KRequestPending` (which is defined as `-KMaxTInt`). This is required of all asynchronous service providers. The vast majority of such entities are servers and therefore the Symbian OS client-server framework ensures that when an asynchronous request is issued to a server, the request status is automatically set to pending. This happens on the client-side, prior to the server receiving the client's request.

The `TRequestStatus` assignment operator ensures that when a new value is assigned to a request status, the flags are updated accordingly:

```

inline TInt TRequestStatus::operator=(TInt aVal)
{
    if(aVal == KRequestPending)
        iFlags|=TRequestStatus::ERequestPending;
    else
        iFlags&=~TRequestStatus::ERequestPending;
    return (iStatus=aVal);
}

```

If the value is equal to `KRequestPending` then the `ERequestPending` flag is set within the request status, otherwise it is cleared. In this way, the request object's state is always kept up-to-date which, as we shall see, allows the framework to use this information during the `RunL()` dispatch phase.

As we saw in our earlier example, after an asynchronous request has been issued, the active object should always immediately indicate that it is active by calling `SetActive()`. The implementation of `SetActive()` is as follows:

```

EXPORT_C void CActive::SetActive()
{
    __ASSERT_ALWAYS(! (iStatus.iFlags & TRequestStatus::EActive),
                    Panic (EReqAlreadyActive));
    __ASSERT_ALWAYS(IsAdded(), Panic (EActiveNotAdded));
    iStatus.iFlags |= TRequestStatus::EActive;
}

```

The function first checks that the object's `TRequestStatus` flags do not already indicate that the object is active. This is a programming error that would result in a request being issued to the service provider whilst an existing request is already pending or awaiting `RunL()` processing. Secondly, an additional check is made to ensure that the active object has added itself to the active scheduler. If the object has not been added to the active scheduler then the scheduler will not be able to locate the object when the request is eventually completed. Finally, because `CActive` is a friend of `TRequestStatus`, `CActive` is permitted to access the `TRequestStatus::iFlags` member directly, which it does in order to set the `EActive` bit. This is a hint to the framework so that it knows that the object is now ready to accept an event.

When the asynchronous service provider finishes processing the request, it generates an event. This means that it signals the thread's request semaphore and posts a completion code (such as `KErrNone` or any other standard error code – anything except `KRequestPending` is permissible) into `TRequestStatus`. The act of completing an asynchronous request increments the requesting thread's request semaphore value by one.

The active scheduler is responsible for detecting the occurrence of an event, which it then associates with the active object that originally requested it, and calls `RunL()` on that active object.

The active scheduler calls `User::WaitForAnyRequest()` to detect an event. This function's behavior is linked to the current counter value for the thread's request semaphore. The counter value is decremented by one and, if the resultant request semaphore value is zero or above, then the call returns immediately, indicating that an event has occurred. If the semaphore counter value drops below zero as a result of decrementing the value by one, then the thread is suspended until an event occurs and the request semaphore is signaled to indicate an event completion.

When the thread is resumed as a result of an asynchronous request being completed, the active scheduler then scans through all its active objects searching for the object that should handle the event. The criteria the active scheduler applies to locating the correct active object are fairly straightforward.

- The scheduler starts at the beginning of its active object queue, looking at the highest-priority active object in the entire active scheduler.

- It checks to see if the object is active by inspecting its request status flags. As we saw earlier, the flags are updated when `SetActive()` is called.
- If the object is active and its `iStatus` value no longer equals `KRequestPending`, then this implies that the object made an asynchronous request that has now been completed by the service provider. If the object is active and the `iStatus` value is still `KRequestPending`, then it would imply that the object has made an asynchronous request that has not yet been completed.
- If the object is not active then the next object in the queue is selected and the same checks are applied once more.
- Having located the active object that should handle the event, the active scheduler then clears its request status flags (`CActive::iStatus.iFlags`) to indicate that the object is no longer active or has any request pending. It then calls `RunL()` on the specified active object.
- When the call to `RunL()` has completed, the active scheduler issues another call to `User::WaitForAnyRequest()` and waits for the next event to occur. This implies that the scheduler handles precisely one event per `User::WaitForAnyRequest()`. If more than one event is outstanding, there's no problem; the next `User::WaitForAnyRequest()` completes immediately without suspending the thread and the scheduler finds the active object associated with the completed event by applying the same criteria.

Given this description, you might expect writing an active object to be difficult. In fact, as we've already seen with `CDelayedHello`, it is not. You simply have to:

- issue request functions to an asynchronous service provider, remembering to call `SetActive()` after you have done so
- handle completed requests with `RunL()`
- cancel requests with `DoCancel()`
- set an appropriate priority
- handle leaving from `RunL()` with `RunError()`.

6.5 Active Object Priorities

As we have now seen, the active-scheduler framework provides non-pre-emptive multitasking within the context of a single thread. Whilst one

`RunL()` call is being processed, no other `RunL()` call within the same thread can execute or be pre-empted. This means that it is important to understand the importance of choosing priorities for active objects.

If the thread's request semaphore is signaled more than once (i.e. two or more events occur simultaneously), before the active scheduler can dispatch an active object's `RunL()`, it has to decide which active object to service first. The queue of active objects is ordered by the priority of those objects, with the highest-priority object at the head of the queue. The active scheduler always starts looking for an active object to run at the head of the queue, working through the queue until it finds an object that satisfies the criteria we described in Section 6.4.

For this reason, if two objects are ready to run, one with a low priority and another with a high priority, the active scheduler calls `RunL()` on the higher priority object first. If two active objects with the same priority are both ready to run, then the object which was added to the active scheduler first, via `CActiveScheduler::Add()`, is the first object to have its `RunL()` method called.

The scheduler does not use a round-robin algorithm. If two or more active objects with the same priority are ready to run, the offer to run always goes to the one that is higher in the scheduler's queue.

The `CActive` class defines a basic set of priority values, encapsulated by the `TPriority` enumeration:

```
// Defines standard priorities for active objects.
enum TPriority
{
    // A low priority, useful for active objects representing
    // background processing.
    EPriorityIdle = -100,

    // A priority higher than EPriorityIdle but lower than
    // EPriorityStandard.
    EPriorityLow = -20,

    // Most active objects will have this priority.
    EPriorityStandard = 0,

    // A priority higher than EPriorityStandard; useful for
    // active objects handling user input.
    EPriorityUserInput = 10,

    // A priority higher than EPriorityUserInput.
    EPriorityHigh = 20
};
```

The application framework (CONE) defines an additional set of priorities, `TActivePriority`, in `coemain.h`:

```
// UI Control framework active object priorities.
// These are in addition to the values contained in the
// TPriority enum in class CActive.
enum CActivePriority
{
    // 300
    CActivePriorityClockTimer=300,
    // 200
    CActivePriorityIpcEventsHigh=200,
    // 150
    CActivePriorityFepLoader=150,
    // 100
    CActivePriorityWsEvents=100,
    // 50
    CActivePriorityRedrawEvents=50,
    // 0
    CActivePriorityDefault=0,
    // 10
    CActivePriorityLogonA=-10
};
```

Most applications use `CActive::EPriorityStandard` as their standard priority value, but if you need to use non-zero values for your own active objects, then you need to look at the enumeration values so that you can ensure you fit in with them.

Since a `RunL()` dispatch cannot be pre-empted by any other active object in the same scheduler, this means that it is possible for one slow, low-priority, active object `RunL()` to delay the `RunL()` of a higher-priority object. For example, the Control Environment installs a high-priority active object to receive key, pen and other kinds of events from the Window Server. This object needs to have a high priority to ensure that it runs as soon as possible after an event occurs, so as to maintain a high degree of responsiveness. However, if the active scheduler contains an object that takes a long time to finish its `RunL()` processing, then the user's key or pen input cannot be processed until that slow `RunL()` completes. For this reason, it's important to ensure that `RunL()` calls complete quickly, no matter where they are within the scheduler's queue.

Having the wrong priority for your active object can also lead to active object starvation. If an object that receives a large number of events is always in a ready-to-run state because its request has been completed, then if its priority is very high it may prevent other ready-to-run objects lower down the priority list from being dispatched. This is a common problem when using timer active objects, or `CIdle`, if their priorities are too high. In `e32base.h`, the `CIdle` class provides a ready-made wrapper for idle-time processing. It is an active object that 'calls back' a function when no other active object is ready to run. The idle object completes its own request status immediately when it is started, so that it is always ready to run. You should generally create a `CIdle` object

with a priority of `CActive::EPriorityIdle`, which ensures that the object appears near the end of the scheduler queue. When no higher-priority object is ready to run, the idle-time processing function is called. However, if you give the idle object a higher priority, it may end up running ahead of other active objects, preventing them from having their `RunL()` functions called.

If you find that your `RunL()` is not being called when you think it should be, check the priority of your object and think about the priority of other objects within your scheduler. It is possible that the object's priority is too low or that some other object's priority is too high.

Even though the priority of an active object is defined during its construction, it is possible to adjust the priority afterwards, providing the object is not active. The `CActive` base class provides a `SetPriority()` method which can be used to adjust the priority after it has been added to the scheduler. Whilst its use is fairly uncommon, it can be used as a means of counteracting the event starvation problem, essentially creating a manual round-robin scheduling algorithm.

In Symbian OS communications programming, it is common for one active object to handle outgoing data transfer and another active object to handle incoming data. In this situation, it can be dangerous to give one object a fixed higher priority than the other, since it may mean that a flood of incoming data prevents the outgoing data-transfer active object from having a chance to run. Generally, it is desirable to have the objects close to one another in priority, but you may still need to take further action to ensure that both objects have an equal chance of having their `RunL()` methods called.

In Figure 6.5, both objects have the same priority of zero but we know that the order in which the `RunL()` functions for these objects are called is based upon the order in which they were added to the scheduler. In the case where both are ready to handle an event, one or other object always runs before the other, which may lead to event starvation. In this

Active Scheduler		
	Priority	Ready
...		
reader (RX)	0	Yes
writer (TX)	0	Yes
...		
...		

Figure 6.5 Active objects in a communications program

example, the reader (RX) object was added to the scheduler before the writer (TX) object and therefore, if both are ready to run, the `RunL()` method of the reader object is always called first. If the reader is receiving a large amount of data, but an important outgoing transmission packet needs to be sent, then the sending of the outgoing packet does not occur until the reader object is no longer ready to run.

One way to counteract this problem is to have one object adjust its priority when its `RunL()` method is called, so that it pivots around a central priority point (see Figure 6.6). For example, the reader object could alternate between the priority values of zero and minus one. The first time `RunL()` is called, the object adjusts its priority by calling `SetPriority(-1)`.

Active Scheduler		
	Priority	Ready
...		
writer (TX)	0	Yes
reader (RX)	-1	Yes
...		
...		

Figure 6.6 Active objects with dynamic reader-priority adjustment

In this way, even though the reader is still ready to run almost immediately, the `RunL()` method of the writer is now called first and it can send that critical packet. The next time the reader's `RunL()` method is called, it toggles its priority back to zero.

A simpler alternative to priority toggling is to keep both objects at the same priority, but adjust the position in which they appear relative to other objects of the same priority in the active scheduler's queue. When the object's `RunL()` is called, the object removes itself from the active scheduler, by calling `CActive::Deque()` and then calls `CActiveScheduler::Add()` again. This effectively accomplishes the same result, since adding an object to the scheduler inserts it after any other objects of the same priority. However, whilst this is perhaps simpler to write, it relies on an implementation detail of the active scheduler and therefore is less transparent than explicit priority adjustment.

It is worth pointing out that active objects and the adjustment of their priorities cannot guarantee a specific response time. The overall idea is that of co-operative event handling – where it is possible that one object may take an unbounded amount of time before it finishes its `RunL()` processing. If your code must have a response within a certain period of time of an event occurring then you need to use the Symbian OS pre-emptive thread system in conjunction with suitable thread and process priorities.

6.6 Active Object Cancellation

All asynchronous service providers must implement cancel functions corresponding to their request functions. All active objects that issue request functions must also provide a `DoCancel()` function to cancel the request that they originally issued.

A cancel is actually a request for *early completion*. Every asynchronous request that is issued must be completed precisely once – whether normally or by a cancel method. The contractual obligation of a cancel function is to:

- execute synchronously
- return quickly
- complete the original asynchronous request, usually with a status of `KErrCancel`.

Whilst active objects must implement the `DoCancel()` method, it should never be called directly. Instead you call the `Cancel()` method that is inherited from `CActive`.

```
EXPORT_C void CActive::Cancel()
{
    if (iStatus.iFlags & TRequestStatus::EActive)
    {
        DoCancel();
        User::WaitForRequest(iStatus);
        iStatus.iFlags &= ~(TRequestStatus::EActive |
                           TRequestStatus::ERequestPending);
    }
}
```

The method behaves as follows:

- It checks whether the object is active by inspecting the underlying request status flags and checking for the `EActive` bit. If an object is not active, then there is no request to cancel and hence no call to `DoCancel()` is needed.
- If the object is active, a call is made to the pure virtual `DoCancel()` function, which must ask for early completion of the original asynchronous request.
- Then, `User::WaitForRequest()` is called but unlike the active scheduler, which waits for any request to complete, this time a call is made to the overload that takes a `TRequestStatus&` argument: the `iStatus` of the active object. This means, ‘Wait until this specific request is completed.’ As long as `iStatus` is set to

`KRequestPending` then the request has not yet been completed. Once the request semaphore for the thread is signaled and `iStatus` is no longer `KRequestPending`, this line returns. It is the call to `User::WaitForRequest()` that prevents `RunL()` from being called when an active object is cancelled. This line ‘eats’ the event associated with cancellation so that when the active scheduler waits for the next event, it is oblivious to the cancellation of this active object.

If your call to `Cancel()` appears to hang, then the reason is most probably because your call to `DoCancel()` call didn’t result in `iStatus` being completed.

- Finally, the last line updates the underlying request status flags to indicate that the object is no longer active and doesn’t have an outstanding asynchronous request.

If you accidentally call `DoCancel()` instead of `Cancel()`, your `RunL()` method may be called unexpectedly.

In normal circumstances, most asynchronous requests complete normally and only the occasional request needs to be canceled. Despite their relative infrequency, it is important to ensure that cancellations work cleanly and successfully under all possible conditions. In fact, when creating an active object, you need to consider all the possible ways that a request can complete. The remainder of this section examines, in turn, each of the principal ways that completion may occur. It is not altogether surprising that the majority of them relate to cancellation.

Handling a Request that Cannot Run

In some cases it is possible to make a request that does not actually start. For example, there might be insufficient memory or resources to execute the request, or the client might supply incorrect parameters to the request. In such a situation, the implementer of the asynchronous service should inform the requester that the request could not be carried out. There are several possible ways to achieve this, such as returning an error code or leaving. However, these techniques can lead to complex client code that needs to TRAP calls to the request function or check for any returned errors before calling `SetActive()`. This is often error prone.

It is preferable to complete the client’s request by setting the status to an error code (see Figure 6.7). In this way, `RunL()` is called normally and

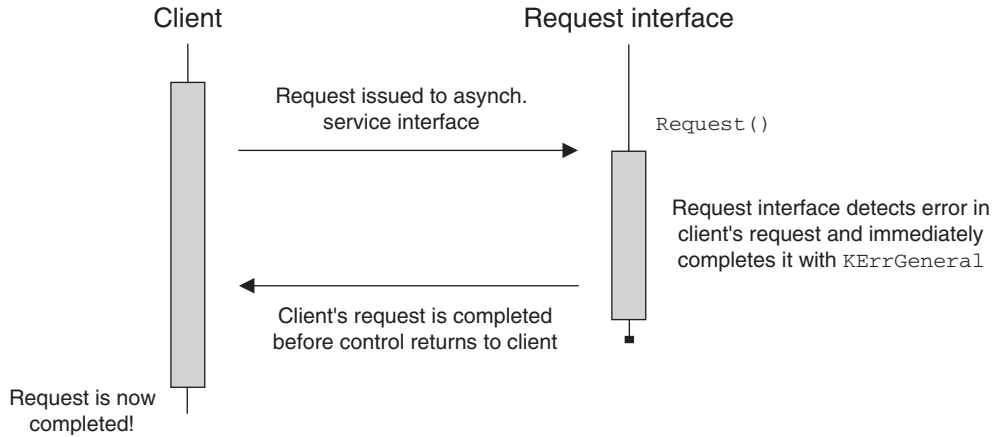


Figure 6.7 Request unable to start

the client's implementation of `RunL()` should check for error conditions by inspecting the value of `iStatus`.

Let's consider a trivial example of a very simple asynchronous request to divide one number by the other. The function treats a request to divide by zero as an error and so checks for this prior to initiating the asynchronous request. The request function might look something like this:

```

void RAsyncDivider::Divide(TRequestStatus& aStatus, TInt aDividend,
                          TInt aDivisor)
{
    aStatus = KRequestPending;
    if (aDivisor == 0)
    {
        TRequestStatus* status = &aStatus;
        User::RequestComplete(status, KErrArgument);
    }
    else
    {
        // Calculate result etc.
    }
}

```

The service provider sets the client's request status to `KRequestPending`. It then checks the request arguments to make sure that they are valid. If the client attempts to use a divisor of zero then the client's request status is immediately completed with an error prior to carrying out any asynchronous operation. In this way, the client receives an event notification and, if active objects are involved, the client active object's `RunL()` method is called with `iStatus` equal to the `KErrArgument` error code.

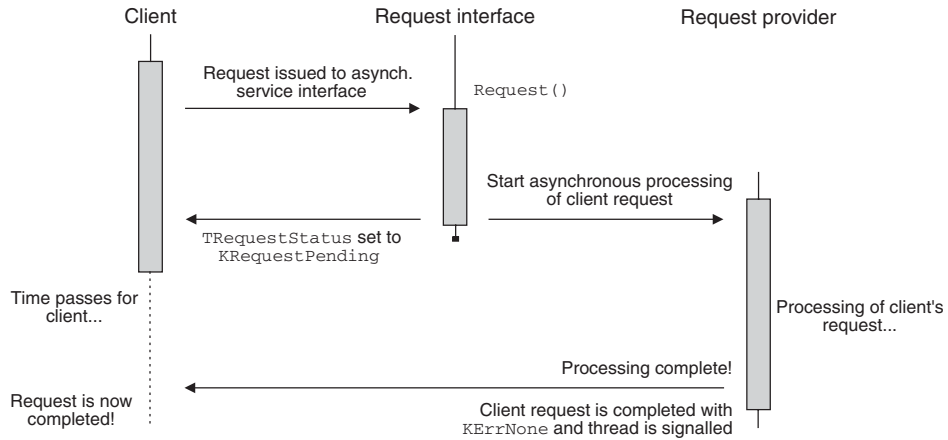


Figure 6.8 Normal request processing

Completing a Request after Normal Processing

An asynchronous request is issued, is processed, and an event is signaled some time after the processing is complete.

In this common scenario (see Figure 6.8), the request service provider has time to process the client's request. When the service provider is

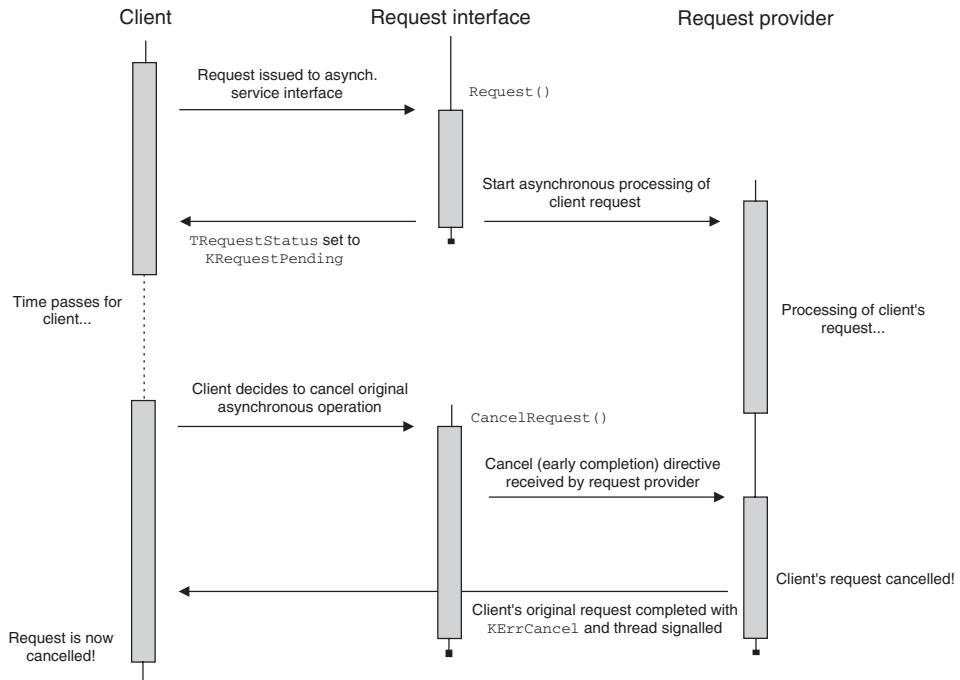


Figure 6.9 Early request cancellation

ready, it completes the client's request with a completion code, typically `KErrNone`. This also signals the client's thread that an event occurred.

Canceling a Request Early

The request is cancelled before the service provider has finished processing the original request.

The request provider receives the cancellation request whilst it is still carrying out the task. It must immediately complete the client's request status with a suitable value (usually `KErrCancel`). Figure 6.9 illustrates the standard cancellation of an asynchronous request.

Canceling a Request Late

An attempt is made to cancel the request, but the request completes normally before the service provider processes the cancellation.

In Figure 6.10, the client asked to cancel the request, but the request had already been completed normally by the service provider. In this case, the service provider must ignore the cancellation request. When `CActive::Cancel()` calls `User::WaitForRequest(iStatus)`,

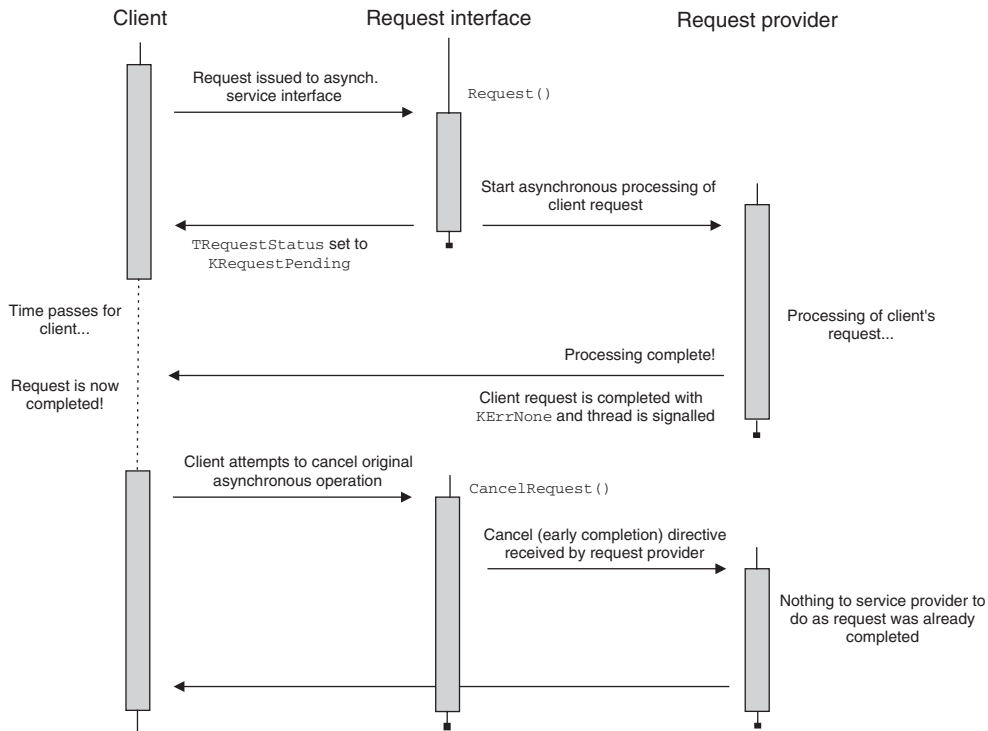


Figure 6.10 Late request cancellation

its `iStatus` value is already something other than `KRequestPending` and the thread's semaphore count is already more than zero as a result of the normal completion, hence this call returns immediately.

When implementing a Symbian OS server, be sure that it can cope with late cancellation.

Canceling a Request when the Service Provider Terminates Unexpectedly

It is entirely normal to use the client–server framework to make asynchronous requests from client to server. If the server unexpectedly terminates whilst a client's request is pending, the kernel ensures that the client's request is completed with the error code `KErrServerTerminated`. In this way, the client is not left waiting for a request to complete even though the server has died.

Canceling a Request when Asynchronous Server Resources are Exhausted

When using the client–server framework, each server may support a fixed number of simultaneous asynchronous requests per connection. If the number of asynchronous requests for a session have been used by other asynchronous operations, then the kernel cannot set up the asynchronous request and completes the request status with `KErrServerBusy`.

6.7 Starting and Stopping the Scheduler

Application programmers rarely have to call the `Start()` and `Stop()` functions of the active scheduler. The Control Framework, CONE, does that for you.

The active scheduler's wait loop is started by issuing `CActiveScheduler::Start()`. Clearly, before this happens, at least one request function should be issued, so that the first `User::WaitForAnyRequest()` completes. From that point on, any completed event causes the `RunL()` function of one of your active objects to be called.

A `RunL()` function can stop the active scheduler by issuing `CActiveScheduler::Stop()`. When that `RunL()` returns, the function call to `CActiveScheduler::Start()` completes.

Stopping the active scheduler brings down the thread's event-handling framework, which is not something you should do lightly. Only do it from the main active object that controls the thread.

6.8 Understanding a Stray Signal

If the active scheduler cannot find the active object associated with an event, this indicates a programming error known as a *stray signal* and the active scheduler panics the thread with the code `E32USER-CBase 46`.¹

In this situation, the active scheduler is told that a request was completed by the act of the thread's request semaphore being incremented, however, the scheduler cannot find the associated active object on which to call `RunL()`.

There are several programming errors which can lead to a stray signal being reported. Stray signals can be quite common when you first start working with active objects. The guidelines in the following sections can help you to understand why you see a stray signal panic and what you can do to solve it.

Forgetting to Set an Object as Active

If you make an asynchronous request but forget to indicate that the active object is now active then the scheduler cannot locate the active object when the request is completed. When the scheduler iterates through all the active objects in its queue, it can see that the request status value is not `KRequestPending` but, because the object wasn't 'tagged' as active, it doesn't know that a request was ever issued. The scheduler treats this as a programming error and therefore panics.

You should normally call `SetActive()` in your code immediately after making your asynchronous request. In particular, you must ensure that you do not call any potentially leaving functions between making the request and setting your object active. In this way, the request phase contractual obligations become atomic.

Commonly, the failure to set an object as active is seen when an object is written so that it always makes an asynchronous request but only calls `SetActive()` when the object is not already active:

```
RequestService(iStatus);  
if (!IsActive())  
    SetActive();
```

With the above implementation, the asynchronous request is always issued but `SetActive()` is only called if the object has no outstanding request already being processed. This can lead to the case where `iStatus` is used for two requests simultaneously.

¹ `E32USER-CBase` is the category of panic for user-originated programming errors when working with `CBase`-derived objects from the `euser.dll` library. The panic number, 46, is specifically used to indicate that a stray signal has occurred.

Forgetting to Set a Request as Pending

When creating an asynchronous service, the service provider has a contractual obligation to ensure that the client's `TRequestStatus` is correctly initialized to `KRequestPending` prior to its completion.

When the scheduler is searching for an active object upon which to call `RunL()`, it explicitly checks the active object's request status flags to ensure that the object has a request pending. If you have called `SetActive()` but the request status was not set to `KRequestPending` during the request-initialization phase, then the scheduler is unable to identify it and treats it as a stray-signal panic. For asynchronous client-server requests, the Symbian OS framework ensures this takes place automatically.

Making a Second Request

As discussed earlier, there is a one-to-one relationship between active objects and asynchronous request functions. Since each active object contains precisely one `TRequestStatus` data member, it can make only one asynchronous request at any given time. If an existing request is still pending at the time a new request is issued then effectively two service providers are competing for the same request status. This causes no problem to the service provider, but it results in the request semaphore for the thread being signaled twice, once for each event completion. For the first request that is completed, a `RunL()` dispatch is made without error and the object is set as being inactive. However, when the scheduler next calls `User::WaitForAnyRequest()`, it returns immediately and attempts to locate the relevant active object. This time the scheduler panics with a stray signal since it fails to find an appropriate active object upon which to call `RunL()`.

Completing a Request Twice

If your code creates an asynchronous service within its own thread, then care must be taken to ensure that you complete the `CActive::iStatus` member only once.

Let's look at a simple example in which a `RunL()` method completes the object's request status immediately, thereby requesting another `RunL()` call as soon as possible. The core implementation of such a method might look something like this: ²

```
void CRunAsSoonAsPossible::RunL()
{
    // do some processing to handle original event
```

² This code is an oversimplification, for illustrative purposes only. See Section 6.11 for a description of the potential dangers of such code and how to use it safely.


```

...
// Now request RunL() be called again as soon as possible
TRequestStatus* myStatus = &iStatus;
User::RequestComplete(myStatus, KErrNone);
SetActive();
}

```

The precise action of `User::RequestComplete()` is explained in Section 6.10. For now, it is sufficient to know that the effect of this call is to simulate the making of an asynchronous request and its successful completion, thereby allowing the scheduler to call the object's `RunL()` again.

A common question in such a situation is, 'How do I implement `DoCancel()` for this kind of object?' There is usually confusion here because it is not always clear what asynchronous request was issued. This often leads to creating incorrect code as follows:

```

void CRunAsSoonAsPossible::DoCancel()
{
    // INCORRECT: Try to cancel my request
    TRequestStatus* myStatus = &iStatus;
    User::RequestComplete(myStatus, KErrCancel);
}

```

A possible problem would occur if the object were cancelled. In this situation, `iStatus` has been completed twice: once from the `RunL()` call to `User::RequestComplete()` and again from within `DoCancel()`. Since a call to `User::RequestComplete()` simulates a request completion, the active scheduler is confused into believing that two active objects are ready for dispatch.

In this particular case, the correct implementation for `DoCancel()` is simply to leave the method body empty:

```

void CRunAsSoonAsPossible::DoCancel()
{
    // CORRECT: No asynchronous request to cancel since iStatus
    // was already completed immediately by the RunL() function.
}

```

The active object's request status was completed by `RunL()` so no further processing is needed here.

Completing a Request with `KRequestPending`

The `KRequestPending` constant has special meaning for the active object framework and should not be used outside it. If a request is

completed with `KRequestPending` then the thread's request semaphore is incremented, indicating that an event is ready for processing. However, the active scheduler only calls `RunL()` on an active object when its request status value (completion code) is not `KRequestPending`. This can mean that the scheduler reaches the end of its active object queue without locating a suitable object to dispatch.

Setting the Status of a Completed Request to `KRequestPending`

Normally in Symbian OS, asynchronous requests are made within the context of calls between a client and a server. In this situation, the service provider's client API should provide an asynchronous request function that the client can call. The implementation of the service provider's client-side asynchronous request function should ultimately end up calling either:

```
void RSessionBase::SendReceive(TInt aFunction,
                              TRequestStatus& aStatus) const;
```

or

```
void RSessionBase::SendReceive(TInt aFunction, const TIPCArgs& aArgs,
                              TRequestStatus& aStatus) const;
```

In either case, the implementation of these methods automatically sets the client's request status to `KRequestPending` before dispatching the request to the server.

In some situations, the server may process the client's request before control returns to the client's request function. Alternatively, there may be insufficient resources to deliver the request asynchronously to the server. In either of these situations, the client's request status may be completed already by the time the `SendReceive()` function call returns and therefore `aStatus` already contains a valid completion code.

If your code attempts to set `aStatus` to `KRequestPending` after calling the service provider's client API, then you risk overwriting the request completion code. From the active scheduler's perspective, this would make your active object appear as though it has made a request, but it has not yet been completed. This ultimately creates a stray signal panic within your thread.

If you are creating asynchronous APIs outside the client-server framework, you may need to assign a request status the value of `KRequestPending` in order to satisfy the active scheduler that a request has been initiated. Take care to do this immediately before initiating the request to the underlying service provider.

Using `RThread::RequestSignal()` and the Active Object Framework

Calls to `RThread::RequestSignal()` increment the request semaphore count for your thread. If used in conjunction with the active-scheduler framework, it may confuse the scheduler into believing it has an active object that is ready to run when no such object exists.

Using Active Objects and Calling `User::WaitForRequest()`

A common mistake is to use `TRequestStatus` objects outside the active object framework to initiate multiple simultaneous asynchronous requests and then wait for one to complete synchronously. For example, starting a thread and a 5-second timer in parallel. If the thread does not exit within five seconds, then something must have gone wrong.

The code might look like this:

```
TRequestStatus timerStatus;  
TRequestStatus threadStatus;  
// start thread  
// start timer  
User::WaitForRequest(timerStatus, threadStatus);
```

Symbian OS provides an overload of `User::WaitForRequest()` that allows you to achieve exactly this. The above pseudo-code says, ‘I want to wait for either my timer request or my thread request to complete but I do not care which one completes first.’

Until at least one of the requests has been completed, the thread is suspended. Once a request is completed and the thread’s request semaphore is signaled, the thread awakes. The usual cause of a stray signal in this kind of scenario is that only one request completed and therefore one was still pending.

Let’s assume that the thread request was completed, but the timer request is still pending. This means that at some time in the future, the timer’s request will expire. If control returns to the active scheduler, the thread’s request semaphore is signaled when the timer request expires. However, there is no active object upon which to call `RunL()` because we didn’t use active objects to make the timer request. This means that the active scheduler cannot locate any suitable active object to dispatch and panics with a stray signal.

In this particular case, the problem can be solved by checking which request is pending and then issuing a cancel request to the appropriate service provider. When canceling the five-second timer, for example, you must also call `User::WaitForRequest(timerStatus)` to ensure that you ‘eat’ the associated timer completion event, otherwise the

scheduler will be confused and think the event is associated with an active object. Mixing active objects with calls to the `User::WaitForRequest()` functions is not recommended.

Removing an Uncompleted Active Object from the Queue

When the request semaphore for your thread is signaled, the active scheduler assumes that the event signal relates to an active object. If you call `CActive::Deque()` to remove your active object from the active scheduler before its request completes, by calling `CActive::Deque()`, then the active scheduler cannot find your object.

Calling a Leaving Function after an Asynchronous Request

This is really a variation on forgetting to set an object as active.

In the following code, the implementation of the service provider calls a leaving function after an asynchronous request has been initiated. With an `aDelay` argument of zero, the asynchronous request is initiated and the asynchronous service provider leaves with `KErrArgument`. This means that the call to `SetActive()` is skipped and when the asynchronous request eventually completes, the active scheduler raises a stray-signal panic since it cannot identify that the object actually made a request.

```
void RAsyncProvider::NotifyAfterDelayL(TRequestStatus& aStatus,
                                      TTimeIntervalSeconds aDelay)
{
    aStatus = KRequestPending;
    SendReceive(ENotifyOnDelay, aStatus);
    if (aDelay == 0)
    {
        // Using a delay of zero is invalid
        User::Leave(KErrArgument);
    }
}

void CMyDelayActiveObject::RequestDelayL()
{
    iDelayProvider.NotifyAfterDelayL(iStatus, TTimeIntervalSeconds(0));
    SetActive();
}
```

6.9 Other Common Active Object Errors

There are some other frequent mistakes made when working with active objects. This section considers some of the more common problems that have been observed.

Calling `DoCancel()` Directly

The `DoCancel()` function is meant to be called indirectly via the public non-virtual `CActive::Cancel()` function.

If you call `DoCancel()` directly, then you are not allowing the active object framework a chance to ‘eat’ the request semaphore signal that occurs when the asynchronous request is cancelled. This confuses the scheduler into believing that your active object is now ready to run, thereby causing the `RunL()` method to be called even though you intended to cancel it. In this situation, when `RunL()` is called, `iStatus` is likely to be set to an error value (typically `KErrCancel`).

Failing to Call `Cancel()` from Within a Destructor

Active objects should almost always call `Cancel()` in their destructor. If your object happens to be active at the time of destruction, failing to call `Cancel()` causes the `CActive` base class destructor to panic your thread with `E32USER-CBase 40`.

In addition, `Cancel()` should usually be called prior to executing any other code within your destructor, such as deleting class members. This ensures that all resources required to fully cancel any outstanding asynchronous request are still available when the virtual `DoCancel()` method is invoked.

Checking `IsActive()` before Calling `Cancel()`

```
if (IsActive())  
{  
    Cancel();  
}
```

The framework `CActive::Cancel()` function already does this internally, so there is no added value from doing it yourself.

Failing to Implement `DoCancel()` Correctly

If your active object makes an asynchronous request, then your `DoCancel()` method should typically cancel that request by calling your service provider’s synchronous cancel function.

If you provide an empty implementation of `DoCancel()` then should the active object’s `Cancel()` method be called when the object is active, your thread will be blocked until the asynchronous service provider completes the request status. As we saw, `CActive::Cancel()` waits for your object’s `iStatus` to be completed, hence the thread is blocked until this occurs.

Omitting Error Handling

A common problem, most often seen when using asynchronous notification APIs, is that the `RunL()` method of the active object does not check the `iStatus` completion code for an error. Instead, the error is ignored and the active object just re-issues the same request to the same asynchronous service provider. In such a case, each subsequent request is likely to complete with an error and the resulting ‘flat-spin’ cycle can continue indefinitely.

If the priority of the active object is high, then it may well cause other objects to be starved of active scheduler time. Even if the active object has a suitably low priority, the process can quickly result in a drained battery, as the CPU will have little time to sleep.

Using a **TRAP** Macro Inside `RunL()`

The active scheduler places a **TRAP** harness around all calls to an active object’s `RunL()` method. If the `RunL()` method should leave, then, as we have discussed, `RunError()` is invoked to handle the error.

Whilst not technically a mistake, adding additional **TRAP** statements inside `RunL()` may be an unnecessary overhead. A more refined solution may be to remove the **TRAP** and handle error situations via `RunError()`.

Calling Leaving Code in a Self-destructing `RunL()` Method

If your `RunL()` implementation deletes its own active object instance, for example, by calling `delete this`, then logically this should typically be the last line within the `RunL()` method.

Were the `RunL()` method to delete the object and somehow leave, then the active scheduler **TRAPs** the leave and attempts to call `RunError()` on the object in question. Since the object has deleted itself, this results in an exception.

Using `operator=()` to Assign **TRequestStatus**

The `CActive::iStatus` member, an instance of the **TRequestStatus** class, is rarely meant to be assigned a value directly. There is one exception: the asynchronous service provider typically sets a **TRequestStatus** argument to `KRequestPending` during the initiation of an asynchronous request.

However, it is rarely appropriate to assign a **TRequestStatus** object a value directly. Instead, the request status should be assigned a value as part of the request completion phase. For components that implement their own thread-local active objects, a request status should be completed via `User::RequestComplete()`.

Leaving Asynchronous Methods

It is generally better to make asynchronous methods non-leaving. This avoids overly complex client code whereby clients need to TRAP calls to the function and handle scenarios in the case of a leave. Furthermore, for clients of such methods, there is always some uncertainty as to whether the function left before or after the asynchronous request was issued.

Returning Error Values from an Asynchronous Method

Should an error situation exist, it is preferable to report the error via the `TRequestStatus` argument rather than explicitly returning an error code.

```
void RAsyncProvider::NotifyAfterDelay(TRequestStatus& aStatus,
                                     TTimeIntervalSeconds aDelay)
{
    if (aDelay == 0)
    {
        // Using a delay of zero is invalid
        TRequestStatus* status = &aStatus;
        User::RequestComplete(status, KErrArgument);
    }
    else
    {
        // continue with normal asynchronous processing
    }
    // ...
}
```

If we assume that the client of this API is using an active object, then the above implementation means that clients of this API need to handle error situations only via their `RunL()` function.

An alternative, less robust, implementation of the request function might return an error value under certain circumstances (such as parameter validation) and additionally handle other potential errors during the request processing (for example, due to lack of memory). In such a scenario, the client must handle errors in two places: the direct return value from the asynchronous request function and also from within the `RunL()` implementation.

6.10 Implementing State Machines

Active objects can be used to implement state machines. As an example, say we wish to amalgamate the `CDelayedHello` and `CFlashingHello` functionality of Section 6.3 and so produce the effect of the message flashing on and off once followed by displaying the message. This cycle repeats

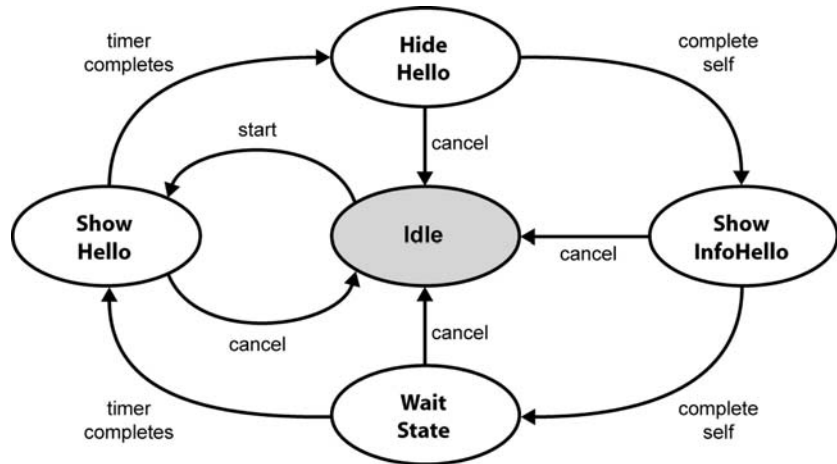


Figure 6.11 State machine

until cancelled. This behavior can be represented by the state diagram in Figure 6.11.

The state machine is initially in the `Idle` state and starting the machine moves it into the `Show Hello` state. The greeting is shown and a timer is started. When the timer completes, it notifies the state machine and it moves to the `Hide Hello` state – the greeting is removed. The state machine then moves itself into the `Show InfoHello` state. Here a message is produced showing the greeting text. Again the state machine moves itself into the `Wait State` where it starts a timer. When the timer completes, it notifies the state machine, moves to `Show Hello` and the cycle repeats. Canceling the state machine can be done in any state and simply moves the state machine into the `Idle` state.

This is a very simple state machine but it demonstrates the idea. The `CMultiPartHello` object implements the state machine. It is an active object that maintains an outstanding request, once it has been started, until it is cancelled.

```

class CMultiPartHello : public CActive
{
public:
    static CMultiPartHello* NewL(CActiveHelloAppView* aAppView);
    ~CMultiPartHello();
    void Start(TTimeIntervalMicroSeconds32 aDelay);
private: // From CActive
    void RunL();
    void DoCancel();
    TInt RunError(TInt aError);
private:
    CMultiPartHello();
    void ConstructL(CActiveHelloAppView* aAppView);
    void CompleteSelf();

```



```

void ShowText(TBool aShowText);
private: // Enums
enum THelloState
{
    EIdle      = 0,
    EShowHello,
    EHideHello,
    EShowInfoHello,
    EWaitState
};
private:
    RTimer iTimer;
    TTimeIntervalMicroSeconds32 iDelay;
    THelloState iState;
    CEikonEnv* iEnv;
    CActiveHelloAppView* iAppView;
};

```

The `CompleteSelf()` function makes the active object eligible for `RunL()` next time control is returned to the active scheduler.

```

void CMultiPartHello::CompleteSelf()
{
    TRequestStatus* pStat = &iStatus;
    User::RequestComplete(pStat, KErrNone);
    SetActive();
}

```

The `User::RequestComplete()` function results in three actions:

- `iStatus` is set to `KRequestPending`, which ensures that the `TRequestStatus` flags are correctly updated in accordance with active object and active scheduler protocols
- the request status is changed from `KRequestPending` to the supplied completion status code, in this case, `KErrNone`
- `RThread::RequestComplete()` is called, resulting in a kernel executive call which ultimately increments the thread's request semaphore by one.

The object then sets itself active. The net effect is to make it appear as if the active object issued a request that has been completed. Therefore the active scheduler can call `RunL()`.

`RunL()` implements the behavior for the different states.

```

void CMultiPartHello::RunL()
{
    THelloState nextState = iState;
    switch (iState)
    {

```

```

case EShowHello:
{
    ShowText(ETTrue);
    // issue request
    iTimer.After(iStatus, iDelay);
    SetActive();
    nextState = EHideHello;
    break;
}
case EHideHello:
{
    ShowText(EFalse);
    CompleteSelf();
    nextState = EShowInfoHello;
    break;
}
case EShowInfoHello:
{
    iEnv->InfoMsg(R_ACTIVEHELLO_TEXT_HELLO);
    CompleteSelf();
    nextState = EWaitState;
    break;
}
case EWaitState:
{
    // issue request
    iTimer.After(iStatus, iDelay);
    SetActive();
    nextState = EShowHello;
    break;
}
default:
    break;
}
iState = nextState;
}

```

When `RunL()` is entered, the `switch` statement performs the desired behavior and the next state is set. The next state is performed when `RunL()` is next called. This can be due to either an asynchronous request being completed (the timer in this example) or by the state machine completing itself.

You could argue that there is no need for the `CompleteSelf()` functionality – the states can be amalgamated. Applying this to our example would mean that `Hide Hello`, `Show InfoHello` and `Wait State` could be amalgamated into a single state. So why not do this? One reason would be to allow control to be returned to the active scheduler and therefore maintain responsiveness, that is, to allow higher-priority active objects, such as the Control Environment's user input handler, to be given some processor time. Also, state machines are not usually as simple as this and the processing path can vary depending on some condition. For example in a given state A, the next state may be state B (which uses a `CompleteSelf()` call to get there) or state C (which is entered once an asynchronous request completes) depending on some condition.

Checking the state need not only be done in the `RunL()`. For example, the `DoCancel()` function in our example checks to see if the timer needs to be cancelled.

```
void CMultiPartHello::DoCancel()
{
    switch( iState )
    {
        case EHideHello:
        case EShowHello:
        {
            iTimer.Cancel();
            break;
        }
        default:
            break;
    }
    ShowText(ETTrue);
    iState = EIdle;
}
```

Looking back at `RunL()`, we can see that the timer was started in the Show Hello state and then the state changed to Hide Hello. Similarly in Wait State, the state was changed to Show Hello once the timer was started. The timer is only cancelled if the state machine is in the Hide Hello or Show Hello states.

6.11 Long-Running Tasks and Active Objects

Sometimes, you want to be able to implement a long-running task alongside your application's (or server's) main task. In other operating systems you might implement such a task with a background thread. In Symbian OS, the best way to implement such a task is with a low-priority active object that runs in the idle time of other event-handling active objects. It is essentially a low-priority state machine that always completes itself. The paradigm for a long-running task is:

- Design a state machine to do the background task through a series of states.
- In the `Start()` function, use a self-complete function to make your object eligible for its `RunL()` function to be called the next time control returns to the active scheduler.
- When `RunL()` is called, you should do some processing for the long-running task. Remember that you should keep the processing to a realistic amount so as not to block higher-priority active objects from having a chance to handle their events.

- If the task is not complete, then it needs to self-complete to continue processing. If the task is complete, it should simply stop and not complete any more.
- In `DoCancel()`, you may wish to destroy any intermediate data associated with the long-running task. You do not need to issue a call to `User::RequestComplete()` because you already did that from either `Start()` or `RunL()`. Remember, any request you issue should be completed precisely once, so you do not need to complete it again.

A very simple long-running task would be a Fibonacci number generator. Starting with the numbers 0 and 1, each successive number is the sum of the two preceding numbers (see Figure 6.12).



This is a pure virtual ‘mixin’ class, which the client of the active object must implement. In this particular case, we implement it inside our main view control so that we can draw the generated number, `aResult`, to the display. To report the calculated number, the active object calls the `HandleFibonacciResultL()` function. The secondary purpose of the mixin is to provide a means of telling the user interface (UI) that the active object is about to restart calculation again from the beginning of the sequence, once 80 numbers have been calculated. The engine can inform the UI that calculations are going to be restarted by calling the `HandleFibonacciCalculatorResetL()` function.

Next, we declare the active object:

```
class CFibonacciGenerator : public CActive
{
public:
    // Construct/destruct
    static CFibonacciGenerator* NewL(MFibonacciResultHandler&
                                     aResultHandler);

    ~CFibonacciGenerator();
    // Request
    void Start();
private:
    // Construct/destruct
    CFibonacciGenerator(MFibonacciResultHandler& aResultHandler);
    void ConstructL();
    // from CActive
    void RunL();
    void DoCancel();
    TInt RunError(TInt aError);
    // Internal methods
    void Reset();
    void CompleteSelf();
private:
    MFibonacciResultHandler& iResultHandler;
    TUint64 iMostRecentResult;
    TUint64 iOldestResult;
    TUint iSequenceNumber;
};
```

The static constructor, `NewL()`, has a reference to the new result handler interface so that when a number is calculated the handler can receive the generated number.

```
CFibonacciGenerator::CFibonacciGenerator
(MFibonacciResultHandler& aResultHandler)
    : CActive(EPriorityIdle),
      iResultHandler(aResultHandler)
{
    CActiveScheduler::Add(this);
}
void CFibonacciGenerator::ConstructL()
{
    // Empty on this occasion
}
```

The constructor of the class has several purposes. First, it defines the active object priority as `CActive::EPriorityIdle`, so that this object's `RunL()` only executes when there are no higher-priority events to be handled. Secondly, it stores a reference to the result handler in the `iResultHandler` data member so that we can use this to report a calculation result later on. Finally, it adds the object to the active scheduler. The second-phase constructor, `ConstructL()`, is empty in this instance since no further object initialization is required.

The `Start()` function is responsible for starting the idle-time calculation.

```
void CFibonacciGenerator::Start()
{
    Reset();
    CompleteSelf();
}

const TUint KFibonacciSeedNumberFirst = 0;
const TUint KFibonacciSeedNumberSecond = 1;
void CFibonacciGenerator::Reset()
{
    iSequenceNumber = 0;
    iOldestResult = KFibonacciSeedNumberFirst;
    iMostRecentResult = KFibonacciSeedNumberSecond;

    // Notify our result handler that we have restarted
    // our calculation
    TRAP_IGNORE (iResultHandler.HandleFibonacciCalculatorResetL());
}
```

`Start()` calls the `Reset()` function, which initializes the seed numbers to their starting values of 0 and 1, as well as resetting the sequence number to 0. The `iSequenceNumber` contains a counter value that indicates how many numbers in the sequence we have calculated so far. A Fibonacci sequence begins with the numbers zero and one, so we must always ensure we inform the UI of these two numbers before attempting to calculate the third. We also make a call to our result handler to inform it that we are about to begin calculating the sequence again from the start, using `iResultHandler.HandleFibonacciCalculatorResetL()`. This allows the UI to reset itself ready for the start of a new sequence.

`Start()` then calls the `CompleteSelf()` utility function to complete the active object's request status and set the object as active.

```
void CFibonacciGenerator::CompleteSelf()
{
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, KErrNone);
    SetActive();
}
```

We do not need to include a line that sets the value of `iStatus` to `KRequestPending`. The call to `User::RequestComplete()` does that for us. After `CompleteSelf()` has finished, the object is ready to run and, providing the active scheduler doesn't have a higher-priority object to dispatch, it calls the `RunL()` function.

```
const TUint KMaxNumberOfIterationsBeforeRestarting = 80;
void CFibonacciGenerator::RunL()
{
    // Update the number of numbers we have generated so far
    ++iSequenceNumber;
    // Declare calculation result variable
    TUint64 result = 0;
    switch(iSequenceNumber)
    {
        case 1:
            result = iOldestResult;
            break;
        case 2:
            result = iMostRecentResult;
            break;
        default:
            result = iOldestResult + iMostRecentResult;
            break;
    }
}

// Inform result handler about the next number
iResultHandler.HandleFibonacciResultL(result);
if (iSequenceNumber > KMaxNumberOfIterationsBeforeRestarting)
{
    // Restart from the beginning once the numbers get very large...
    Reset();
}
else if (iSequenceNumber > 2)
{
    // Update state for next time around
    iOldestResult = iMostRecentResult;
    iMostRecentResult = result;
}
// Request next call to RunL()
CompleteSelf();
}
```

This function is a bit longer than our earlier examples, so let's break it up into sections.

- First, we're keeping track of how many numbers we have generated in the overall sequence. The main purpose of this is to ensure we can tell when we have generated the third (or greater) number and when we've generated a total of 80 numbers. The first two numbers in the sequence are always 0 and 1 so we do not actually calculate them. However, for

the sequence to be complete, we must make sure we inform the result handler about their values before we begin calculating subsequent numbers, hence we increase the `iSequenceNumber` counter value each time we successfully ‘calculate’ a number.

- Next we declare a local variable, `result`, to hold our calculated value. We then switch on the value of our `iSequenceNumber` counter. If `RunL()` has only been called once, then we must report the `iOldestResult` (zero) to the result handler. If `RunL()` has been called twice, we must report the `iMostRecentResult` (one) to the calculation handler. For any call to `RunL()` greater than two, we use the default case and calculate a real value by adding `iMostRecentResult` and `iOldestResult`.
- After calculating the next number in the sequence, we must inform our result handler what the number was. Using our mixin interface, we call `iResultHandler.HandleFibonacciResultL()` and pass the variable `result` as an argument. The implementation of the `MFibonacciResultHandler` class is the main view, and it then updates the screen with the result of the calculation. Note that the call is a leaving function. This means we must expect that the call to the calculation handler may leave so we must cope with this eventuality by implementing a `RunError()` function.
- Assuming that the result handler does not leave, the next action depends on how many numbers in the sequence we have calculated so far. If we have reached 80 numbers, we call `Reset()` to start the process again from the beginning. For sequence numbers greater than two, we must update the `iMostRecentResult` and `iOldestResult` variables. We always discard the oldest number, so it is replaced with the value from the previous time `RunL()` was called. We then update `iMostRecentResult` with the value we just calculated.
- Finally, since this object is going to continue to calculate the next number in the sequence until it is cancelled, we call the `CompleteSelf()` function once again.

The `DoCancel()` function is straightforward since we completed our request immediately from within both `Start()` and `RunL()`. The request status should only be completed once, so there is nothing more for us to do here.

```
void CFibonacciGenerator::DoCancel()
{
}
```


Finally, we must attempt to cope with the possibility that `RunL()` may leave. As is good practice for all active objects, we implement `RunError()`:

```
TInt CFibonacciGenerator::RunError(TInt aError)
{
    Reset();
    return KErrNone;
}
```

In the case of an error, we reset our internal state back to its default values and indicate that we have handled the leave by returning `KErrNone` to the active scheduler.

You'll notice here that we didn't hard-code our `RunL()` method to write directly to the screen or display an information message. Instead we chose to add more flexibility to our class by using an interface to handle the result of the calculation. In this way, we could create different implementations of the `MFibonacciResultHandler` class that behave in different ways, depending on the output type. You'll see this pattern a lot when working with active objects, particularly in the UI framework, since it abstracts the complexity of the active objects.

Since our active object had a priority of `EPriorityIdle`, it doesn't prevent the other active objects that run within the application's thread from being dispatched. This means it is still possible for the application to handle user input, such as displaying a menu, whilst the calculation is going on in the background. This object's `RunL()` processing is very fast and, even though it is active all of the time, it doesn't cause a significant delay to the dispatch of higher-priority objects. However, this kind of always-active object quickly drains the device's battery if left running for a long time since it prevents the CPU from going to sleep.

Summary

Symbian OS has a highly responsive pre-emptive multi-threaded architecture. However, most application and server tasks are by nature event handlers and active objects are a very suitable paradigm with which to handle events.

Because they are non-pre-emptive, it's very easy to program with active objects: you do not need to code mutual exclusion with other threads trying to access the same resources.

Active objects also use fewer system resources than full-blown threads: by default, thread overheads start at around 4 KB on the kernel side and 8 KB on the user side for just the thread's stacks, whereas active objects need be only a few bytes in size. Additionally, switching between active

objects is much cheaper than switching between threads, even in the same process. The time difference can be up to a factor of 10.

This combination of ease of use and low resource requirement is a major factor in the overall efficiency of Symbian OS. However, performance can be impaired if there are many active objects since the active scheduler needs to iterate through the list of active objects to find which object's `RunL()` to call.

7

Files and the File System

This chapter provides an overview of a range of file-related topics, from the file system itself to resource and bitmap files. With such a wide range of topics, it isn't possible to give a detailed description. The intention, instead, is to provide the essential information, with some practical examples that allow you to start making effective use of the system.

7.1 File-Based Applications

Almost all applications use files for data storage. The way files are used depends on the data the application needs to save to persistent memory.

- A Jotter application is clearly file-based. You can save textual data into a file and load data from a file to display.
- A simple game allows you to save your current status and resume playing at some later time. The status information may not be complicated and is discarded once the player has resumed playing or saves the game again after some progress.
- An application such as a calendar is file-based, but you don't load and save the whole file at once. Instead, you use the file as a database and load and save individual *entries*. For efficiency, you also maintain some index data in RAM (for example, on the C: drive). This index may or may not be saved to file as well.
- An application such as Messaging is again file-based, making use of database files where appropriate and separate data files for content data such as text and attachments. However, it saves attachment data to media and relies on another application, such as a photo viewer, to read photo data.

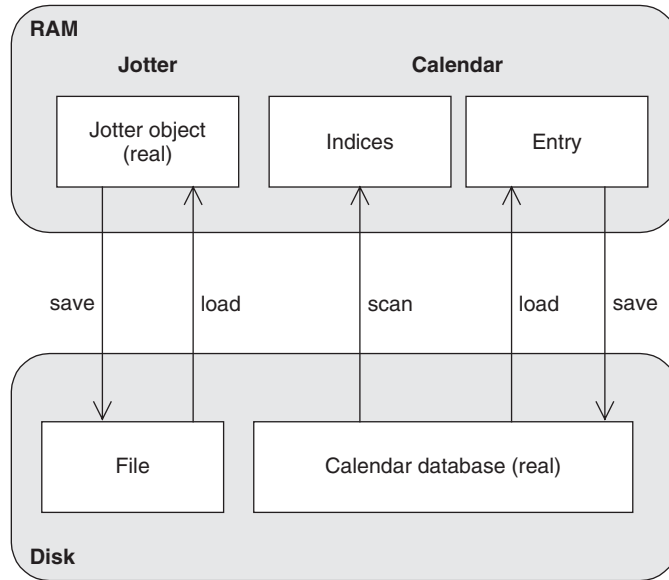


Figure 7.1 File and database applications

Figure 7.1 illustrates the difference between file and database applications.

In load–save applications, we tend to think that the ‘real’ data is in RAM: the file is just a saved version of the real data created or edited by the application. In database applications, we tend to think that the ‘real’ document is the database: each entry is just a RAM copy of something in the database. The disciplines for managing the documents are therefore quite different.

7.2 Drives and File Types

From the perspective of the Symbian OS phone user, there are usually two types of file:

- document files contain the end-user’s data
- system files are mainly stored in the phone’s ROM image (the Z: drive).

For most end users, it is better to pretend that there is only one type of file – document files – and hide the system files altogether. In Symbian OS phones, the user is not exposed to files directly; if it is possible to browse files with a file manager application, system drives are typically hidden. Most user data files are browsed via the application that presents the data to the user.

The drive configuration of a Symbian OS phone may vary from device to device. Feature phones may have large drives for storing music or video data. Some devices have drives that are accessible using USB Mass storage via a host computer. The underlying media and file-system type assigned to a particular drive letter is specific to the UI and device. However, some details are consistent across all phone devices: drive `Z:` is the ROM drive; it contains binaries and system resource files that consist of the operating system and its preinstalled applications.

Symbian OS v9 introduced platform security, which has several implications for the use of files and file systems when developing applications. The concept of data caging, a key part of platform security, allows each application to own a private directory. Application developers should use the private directory to store information private to an application (see Section 9.3 for further details).

7.3 File System Services

This section describes the basic low-level services that underlie all other Symbian OS file-based services. These services are provided by a system server referred to as the file server or `F32`. Most `F32` APIs can be found in the public header file `f32file.h`.

File Specifications

As with most computer operating systems, Symbian OS files are identified by a file name, which may be up to 256 characters in length. A file specification consists of:

- a device, or drive, such as `C:`
- a path, such as `\Document\Unfiled`, where the directory names are separated by backslashes (`\`)
- a file name
- an optional file name extension, separated from the file name by a period (`.`).

All file systems supplied by Symbian support the VFAT specification in terms of character legality for file and directory names.

The file server supports up to 26 drives, from labeled from `A:` to `Z:`. On Symbian OS phones, the `Z:` drive is always reserved for the system ROM and the `C:` drive is always an internal read–write drive, although on some phones it may have limited capacity. Drives from `D:` onwards may be internal, or may contain removable media or a drive used for USB Mass Storage access. You should not assume that you can write to all such drives; many phones have one or more read-only drives, in addition to the `Z:` drive.

Subject to the overall limit on the length of a file specification, a directory name, file name or extension may be of any length. The file system preserves the case of such names, but all operations on the names are case-independent. Clearly, this means that you can't have two or more files in the same directory whose names differ only in the cases of some of their letters. File specifications for searching APIs may contain the wild cards '?' (a single character) and '*' (any sequence of characters) in any component other than the drive letter.

Although most Symbian OS applications do not do so, you are free to include a file name extension in the specification of any file. Symbian OS applications do not always rely on the extension to determine the file's type. Instead, they use one or more UUIDs, stored within the file, to ensure that the file type matches the application. For applications such as music players or photo editors, the file being accessed may be subject to open standards, such as MP3 or Jpeg, and therefore UUIDs may not be used.

Manipulating File Names

File names may be constructed and manipulated using the `TParse` class and its member functions. For example, the following code sets an instance of `TParse` to contain a file specification:

```
_LIT(KFileSpec, "C:\\Private\\<application_UUID>\\application.dat");
TParse fileSpec;
fileSpec.Set(KFileSpec, NULL, NULL);
```

where `<application_UUID>` is the 32-bit UUID assigned to the application.

Following this code, you can call `TParse`'s getter functions to determine the various components of the file specification. For example, `fileSpec.Drive()` contains the string `"C: "` and `fileSpec.Path()` contains `"\\Private\\<application_UUID> "`.

The `Set()` function takes three text parameters: the first is a reference to a `TDesC`, containing the file specification to be parsed; the second and third parameters are pointers to two other `TDesC` descriptors, and either or both may be `NULL`. If present, the second parameter (the related file specification) is used to supply any missing components in the first file specification and the third parameter should point to a default file specification, from which any components not supplied by the first and second parameters are taken. Any path, file name or extension may contain the wildcard characters '?' or '*', respectively representing any single character or any character sequence.

A `TParse` owns an instance of `TFileName`, which is a `TBuf<256>`. This is a large object, with file names specified using 16-bit Unicode characters, and its use should be avoided if possible. If you can, create a

smaller buffer to contain the file specification and use a `TParsePtr` (referencing a modifiable buffer) or a `TParsePtrC` (referencing a constant buffer).

File Server Sessions

The Symbian OS file server provides the basic services that allow user programs to manipulate drives, directories and files, and to read and write data in files.

As with all servers, the file server uses session-based communication to convert a client-side operation into a message that is sent to the server. The requested function is performed in the server, and then any result is passed back to the client. In order to use the file server, you first need a connected file server session, represented by an instance of the `RFs` class.

The general pattern, ignoring error handling, for using the file server is:

```
RFs session;
session.Connect() ;
...
session.Close() ;
```

Between connecting and closing the `RFs`, you can use it to open any number of files or directories, or to perform any other file-related operations. If you wish, you may keep a file server session open for the lifetime of your application. It is preferable that you make sure that all open file-based resources are correctly closed before you close the session. In any case, when the session is closed, the server cleans up any server-side resources associated with the session.

In fact, in a GUI application, you don't need to open a file server session, since the control environment already has an open `RFs` that you can access with `iCoeEnv->FsSession()`. Opening a file server session is an expensive operation so, if at all possible, you should use the existing session, rather than creating your own.

The `RFs` class provides many useful operations related to the file system, including:

- making, removing and renaming directories using `MkDir()`, `MkDirAll()`, `RmDir()` and `Rename()`
- deleting or renaming files using `Delete()` and `Rename()`
- reading and changing directory and file attributes by means of `Att()`, `SetAtt()`, `Modified()` and `SetModified()`
- notifying changes with `NotifyChange()` and `NotifyChangeCancel()`
- manipulating drives and volumes using `Drive()`, `SetDriveName()`, `Volume()` and `SetVolumeLabel()`

- peeking at file data without opening the file by using `ReadFileSection()`
- adding and removing file systems with `AddFileSystem()`, `MountFileSystem()`, `DismountFileSystem()` and `RemoveFileSystem()`
- querying and creating the private path for an application using `PrivatePath()` and `CreatePrivatePath()`

These functions, and their use, are documented in the SDKs.

Most RFS-related functions are stateless – that is, the results of a function call don't depend on any functions previously called. That's why you can usually use the control environment's RFS for your own purposes. However, RFS does have one item of state: its current directory. When you open an RFS, its current directory is set to a default location, the root of the C: drive (`c:\.`), but you can use `SetSessionPath()` to change the current directory. The session path must consist of at least a drive and root, as the default location does, and any number of subdirectories can also be specified. All subsequent file server operations involving file names use the session path unless the file name is fully specified, starting with a drive letter. The current session path for an RFS can be retrieved using `SessionPath()`.

The current directory includes the drive as well as directory names. There is no concept of one current directory per drive.

If you manipulate, or rely on, the current directory, make sure you use your own RFS rather than sharing one. In that case, you must also handle errors associated with the session. How to handle them depends on the way that the session is stored. If it is declared as member data in one of your classes, with a class definition containing a line such as:

```
RFS iFs;
```

then it is sufficient to connect the file session (from a function that allows leaves to occur) with:

```
User::LeaveIfError(iFs.Connect() );
```

and to close the session with:

```
iFs.Close()
```

in the class destructor.

If you declare the file server session on the stack, you need to take a little more care. You need to place the session on the cleanup stack and the best way is to use `CleanupClosePushL()`, as follows:

```
RFs myFs;
User::LeaveIfError(myFs.Connect() );
CleanupStack::CleanupClosePushL(myFs);
...
// file session operations that may leave
...
CleanupStack::PopAndDestroy() ;
```

Remember that connecting a file server session is an expensive, time-consuming operation, so you should only do it if there is a good reason not to share the one that exists in the control environment.

`RFs` contains functions to manipulate a file system. However, the contents of more than one individual file or directory may be opened at once. If you had to have a separate file server session open for each file opened, this would place a large burden on the file server. To avoid this issue, files and directories are treated as Symbian OS subsessions. Each subsession can be opened against a given session: the file server session is passed in as a parameter to the `Open()` function of the subsession and can maintain the state of the file or work through a directory listing.

Directories

A directory contains files and other directories, each of which is represented by a *directory entry*. In Symbian OS, a `TEntry` object is used to contain directory-entry information. `TEntry` objects contain members representing the size, attributes (hidden, read-only), modified date and name of a file or directory.

The `RDir` class allows you to open a directory and read the entries it contains. There are several different ways of reading the contents of a directory. The most straightforward, but not necessarily the most efficient, way is illustrated in the following example:

```
void ReadDirContentsL(RFs& aFs, const TDesC& aDirName)
{
    RDir myDir;
    TInt err;
    User::LeaveIfError(myDir.Open(aFs, aDirName,
                                KEntryAttNormal| KEntryAttDir));
    TEntry currentEntry;
    FOREVER
    {
        err = myDir.Read(currentEntry);
        if (err)
        {
            break; // No more entries, or some other error
        }
    }
}
```

```

    // Process this entry
    }
    myDir.Close()
    if (err != KErrEof) // EOF signifies no more entries to read
    {
        User::LeaveIfError(err);
    }
}

```

Each call to `Read()` reads a single entry, which is useful in cases where memory usage needs to be minimized. It is, however, inefficient to make multiple calls to a server function, so `RDir` also supplies an overload of `Read()` that, in a single call, reads several entries into a `TEntryArray`. The number of entries contained in each `TEntryArray` cannot be determined before a call to `Read()` because the length of each file or directory name is likely to be different. As with many other functions that involve communication with a server, two further overloads supply asynchronous versions of the two types of `Read()`, to be used from an active object.

In addition to `RDir`'s `Read()` functions, `RFs` provides a range of `GetDir()` functions that, in a single call, reads an entire directory's content, or all entries matching the search criteria, into a `CDir`. This can simplify reading a directory contents, however, it still makes use of `RDir` and so may be less efficient than a function written for the requirements of particular application.

As you can see from the following code, which creates a list of entries similar to that of the previous example, `GetDir()` also allows you to specify a sort order for the resulting list.

```

void ReadDirContentsL(RFs& aFs, const TDesC& aDirName)
{
    CDir* dirList;
    User::LeaveIfError(aFs.GetDir(aDirName, KEntryAttNormal,
                                ESortByName, dirList));

    // Process the entries
    delete dirList;
}

```

You can change the attributes of directory entries, including the hidden, system, read-only and archive bits. The only bit with a really unambiguous meaning is read-only. If a file is specified as read-only, then you cannot write to it or erase it. The other attributes are supported for strict compatibility with VFAT but they are not important in Symbian OS, so it is probably best not to use them.

Symbian OS maintains directory entry timestamps in UTC, not local time, so that there is no risk of confusion caused by a time zone change. All time parameters are returned by the file server in UTC; it is a task for the application writer to convert them to local time.

Files

Within a file server session, an individual file is represented by an `RFile` object, which provides the means of opening and manipulating the file. For example, to open a file and append data to it, you could use the following code:

```
TInt WriteToFileL(RFs& aFs, const TDesC& aFileName, const TDesC8& aData)
{
    RFile file;
    TInt err = file.Open(aFs, aFileName, EFileWrite);
    if (err == KErrNotFound) // file does not exist, so create it
    {
        err = file.Create(aFs, aFileName, EFileWrite);
    }
    User::LeaveIfError(err);
    CleanupStack::CleanupClosePushL(file);
    User::LeaveIfError(file.Seek(ESeekEnd, 0));
    User::LeaveIfError(file.Write(aData));

    CleanupStack::PopAndDestroy() ; // Closes the file
}
```

This code attempts to open an existing file, but if the file does not exist, it is created. In addition to `Open()` and `Create()`, `RFile` provides two other means of opening a file:

- `Replace()`, which deletes an existing file and creates a new one
- `Temp()`, which opens a temporary file and allocates a name to it.

In the example above, the file is opened as a binary file, with non-shared write access. A file can be opened as a text file and a variety of other access modes are supported, including shared write and exclusive or shared read. The mode indicators can be divided into the following logical groups. Subject to the specific restrictions stated below, the file mode is specified by ORing together up to one value from each group.

- Read or write access:
 - `EFileRead`: The file is opened for reads but not writes.
 - `EFileWrite`: The file is opened for reads or writes; this mode is not compatible with `EFileShareReadersOnly`.
- Text or binary:
 - `EFileStream`: The file is opened in binary mode; this is the typical case for native Symbian OS applications.
 - `EFileStreamText`: The file is opened in text mode.

- Shared access:
 - `EFileShareExclusive`: If the file is opened with this mode, no other program can access the file until it is closed. If the file has already been opened by another application, in any mode, attempting to open the file in this mode will fail.
 - `EFileShareReadersOnly`: The file may be opened by others only for reading. This flag is incompatible with `EFileWrite`.
 - `EFileShareAny`: The file may be shared with others for reading and writing. This flag is incompatible with `EFileShareReadersOnly`.
 - `EFileShareReadersOrWriters`: The file may be shared for reading and writing. In this mode, the file does not care if it is opened for reading or writing. It is compatible with files opened with `EFileShareReadersOnly`.
- Read completion:
 - `EFileReadAsyncAll`: Read requests should not complete until all requested data becomes available. This flag cannot be combined with `EFileShareExclusive` or `EFileShareReadersOnly` as this would prevent other clients writing data to the file.

If you don't specify otherwise, a file is, by default, opened as a binary file, with no shared access.

You normally specify the access mode when you open the file, although you can use `ChangeMode()` to change it while the file is open. You must have appropriate capabilities to change the access mode of the file. When a file is opened by more than one process with shared write access, you can use `Lock()` to claim temporary exclusive access to a region of the file and `UnLock()` it after the update is made.

When a file is first opened, the current read and write position is set to the start of the file. You can use `Seek()` to move to a different position; the code example above uses it to move to the end of the file, ready to append more data. `RFile` provides a variety of overloaded `Write()` functions, all of which write 8-bit data from a `TDesc8`. Half of them are synchronous functions, such as the one used in the above example and the others are asynchronous, for use from an active object.

There is a corresponding variety of overloaded `Read()` functions, again supplied in both synchronous and asynchronous versions, all of which read 8-bit data into a `TDesc8`.

There is also provision for opening a file in a mode designed for streaming media, `EFileReadAsyncAll`. This mode allows the file to have an outstanding read operation beyond the current length of the file. A read operation for a file opened in this mode cannot complete until sufficient data has been written to file. Corresponding functions exist to

allow reads that may not complete (for example, if the length of streamed data is unknown) to be canceled. Also note that only this type of read operation can be canceled; conventional asynchronous file reads and writes cannot be canceled.

Unless you are reading or writing fairly small amounts of information, you should always use the asynchronous functions to ensure that the application remains responsive during potentially long read or write sequences.

Sharing File Handles

For file data that may be shared between more than one application there are several approaches you could take. First, the file containing the data could be created in a public area of a file system. This is a legitimate place to store a file, however it offers no protection to the data stored in. Any other application can open the file and read its contents.

The second approach is suitable for a set of applications that share data stored in files. The application that ‘owns’ the data can create a file in its own private directory, as mentioned above. To share the data with another application, a handle to this file may be shared. This requires that application X provides an API that application Y can call. The API implemented by application X returns a handle to the shared file. Application Y can then use this handle to access the data contained in the file.

File-sharing APIs are provided in the `RFile` class. One process must act as a server, the other as a client. This is normally a simple decision as one process or application is likely to own a file that another process requests access to, although files can be shared in either direction.

In the example below, the server process opens the file with the access mode it wishes the client to use. The server then transfers the handle to the client via an IPC call, specifying the message number in which the handle is stored.

```
GetFileHandleWriteL(RFs& afs, const RMessage2& aMsg)
{
    User::LeaveIfError(afs.ShareProtected() );
    RFile file;
    User::LeaveIfError(file.Open(afs, KFileName, EFileWrite));
    CleanupStack::CleanupClosePushL(file);
    User::LeaveIfError(file.TransferToClient(aMsg, 0));
    CleanupStack::PopAndDestroy() ; // results in RFile::Close()
    aMsg.Complete(r);
}
```

In the above code, the file is shared with write access and is passed to the client. The file server session against which the file is opened is also shared. This session must have `SharedProtected()` called on it first, to allow the RFs to be used by a process other than the one which

opened it. The file handle is duplicated as part of the sharing process which means that the file-sharing server can call `Close()` on the file if it no longer requires it open. The file-sharing server API that results in the above server-side code being executed is shown below. Note that the file server session is also implicitly shared.

```
TInt RFileHandleSharer::GetFileHandleWrite(TInt &aHandle)
{
    TPckgBuf<TInt> fileHandleBuf;
    TInt fileServerH = SendReceive(EMsgGetFileWrite,
                                   TIpArgs(&fileHandleBuf));
    aHandle = fileHandleBuf();
    return fileServerH;
}
```

The client code to request and adopt the file handle is shown below:

```
TInt RequestFileWrite()
{
    TInt r;
    RFileHandleSharer handsvr;

    r = handsvr.Connect();
    if (r != KErrNone)
        return r;

    TInt fileHandle;
    TInt fileServerHandle = handsvr.GetFileHandleWrite(fileHandle);
    if (fileServerHandle < 0)
        return fileServerHandle;

    handsvr.Close();

    RFile file;
    r = file.AdoptFromServer(fileServerHandle, fileHandle);
    test(r == KErrNone);

    // file may now be used for access
    ...
}
```

As we can see from the example above, the mechanism for sharing file handles involves a client–server relationship between the two processes sharing the file.

RFileBuf

If an application repeatedly requires small amounts of data from a file, it is advisable to read larger quantities from the file to an intermediate buffer, from which the application can retrieve the small sections of

data. This prevents the application from making an excessive number of IPC calls to the file server. Most file systems access data from sectors of the underlying media. Each sector is typically aligned to a 512-byte boundary or some multiple. For an application author, it is more efficient to align your access to a 512-byte boundary where possible and practical, although this is not essential – any arbitrary byte of user data can be read or written to.

In Symbian OS, the `RFileBuf` object allows you to combine access to files with a buffer. `RFileBuf` provides APIs similar to those of `RFile`; many of them are leaving functions. The following example shows how to create and use an `RFileBuf`:

```
LOCAL_C void WriteToFileBufL(RFs& aFs, const TDesC& aFileName,
                             const TDesC8& aData)
{
    RFileBuf buf(KBufSize);

    TInt err = buf.Open(aFs, aFileName, EFileWrite);
    if (err == KErrNotFound) // file does not exist, so create it
    {
        err = buf.Create(aFs, aFileName, EFileWrite);
    }
    User::LeaveIfError(err);
    CleanupStack::CleanupClosePushL(buf);

    TRequestStatus stat;
    buf.WriteL(aData, stat);
    User::WaitForRequest(stat);
    buf.SynchL();
    buf.Close();
}
```

Declaring an `RFileBuf` sets the size of the associated buffer. If no size is supplied, then a default size of 4 KB is used:

```
RFileBuf buf(KBufSize);
```

Next, we open the file we wish to associate with the buffer:

```
TInt err = buf.Open(aFs, aFileName, EFileWrite);
```

Each of the APIs that opens an `RFile` (`Open()`, `Replace()`, `Create()` and `Temp()`) is also supplied with `RFileBuf`. In addition, `RFileBuf` can use a pre-opened `RFile` object and call `RFileBuf::Attach(RFile &aFile, TInt aPos=0)` to associate it with the file buffer. The `RFile` associated with a buffer can be obtained using `RFileBuf::File()` and can be detached (using `RFileBuf::Detach()`) from the `RFileBuf` so that the buffer may be used with another file.

Several read and write functions are supplied to access the data in the file:

```
TRequestStatus stat;  
buf.WriteL(aData, stat);  
User::WaitForRequest(stat);
```

The buffer and file can be synchronized by calling `SyncL()`, which causes the data to be flushed to file:

```
buf.SyncL() ;
```

Finally we close the file buffer, causing the memory resources to be freed and the `RFile` to be closed. Any unwritten data is also written out to file at this point.

```
buf.Close() ;
```

Conclusion

As you can see from this brief discussion, `RFile` provides only the most basic of read and write functions, operating exclusively on 8-bit data. In consequence, `RFile` is not well suited to reading or writing the rich variety of data types that may be found in a Symbian OS application. Far from being an accident, this is a deliberate part of the design of Symbian OS, which uses *streams* to provide the necessary functionality for storing data structures found in many applications.

7.4 Streams

A Symbian OS stream is the external representation of one or more objects. The process of *externalization* involves writing an object's data to a stream; the process of reading from a stream is termed *internalization*. The external representation may reside in one of a variety of media, including stores, files and memory.

The external representation needs to be free of any peculiarities of internal storage – such as byte order and padding associated with data alignment – that are required by the phone's CPU, the C++ compiler or the application's implementation. Clearly, it is meaningless to externalize a pointer; it must be replaced in the external representation by the data to which it points. The external representation of each item of data must have an unambiguously defined length. This means that special care is needed when externalizing data types such as `TInt`, whose internal representation may vary in size between different processors or C++ compilers.

Storing multiple data items (which may be from more than one object) in a single stream implies that they are placed in the stream in a specific order. Internalization code, which restores the objects by reading from the stream, must therefore follow exactly the same order that was used to externalize them. The implementation of internalization and externalization functions define this order.

Base Classes

The concept of a stream is implemented in the two base classes `RReadStream` and `RWriteStream`, with concrete classes derived from them to support streams that reside in specific media. For example, `RFileWriteStream` and `RFileReadStream` implement a stream that resides in a file and `RDesWriteStream` and `RDesReadStream` implement a memory-resident stream whose memory is identified by a descriptor. In this chapter, the examples concentrate on file-based streams but the principles can be applied to streams using other media.

The following code externalizes a `TInt16` to a file, which is assumed not to exist before `WriteToStreamFileL()` is called:

```
void WriteToStreamFileL(RFs& aFs, TDesC& aFileName, TInt16* aInt)
{
    RFileWriteStream writer;
    writer.PushL() ; // writer on cleanup stack
    User::LeaveIfError(writer.Create(aFs, aFileName, EFileWrite));
    writer << *aInt;
    writer.CommitL() ;
    writer.Pop() ;
    writer.Release() ;
}
```

Since the only reference to the stream is on the stack, and the following code can leave, it is necessary to push the stream to the cleanup stack, using the stream's (not the cleanup stack's) `PushL()` function. Once the file has been created, the data is externalized by using operator `<<`. After writing to the store, it is necessary to call the write stream's `CommitL()` to ensure that any buffered data is written to the stream file. If a series of small contiguous operations have been carried out on the stream, they are not actually written to the underlying file; if they were, the implementation would be rather inefficient.

After `CommitL()`, you remove the stream from the cleanup stack, using the stream's `Pop()` function. Finally, you need to call the stream's `Release()` function to close the stream and free the resources it has been using.

As we can see, the Create API takes a file name: the underlying stream class creates a file used to store the data. As much care should be taken with the location of a stream or store file as with a raw `RFile`. The same platform security rules apply to all files owned by an application

regardless of whether they are used as raw files or for store or stream purposes.

Instead of calling `Pop()` and `Release()`, you could make a single call to the cleanup stack's `PopAndDestroy()`, which would achieve the same result. The following example uses that option; otherwise, the code follows a similar pattern to the externalization example:

```
void ReadFromStreamFileL(RFs& aFs, TDesC& aFileName, TInt16* aInt)
{
    RFileReadStream reader;
    reader.PushL() ; // reader on cleanup stack
    User::LeaveIfError(reader.Open(aFs, aFileName, EFileRead));
    reader >> *aInt;
    CleanupStack::PopAndDestroy() ; // reader
}
```

Using << and >> Operators

These examples use the << operator to externalize the data and the >> operator to internalize it. This is a common pattern that you can use for:

- all built-in types except those, such as `TInt`, whose size is unspecified
- descriptors, although special techniques are needed to internalize the data if the length can vary widely
- any class that provides an implementation of `ExternalizeL()` and `InternalizeL()`.

In Symbian OS, a `TInt` is only specified to be at least 32 bits; it may be longer, so externalizing it with << would produce an external representation of undetermined size. If you try to use <<, the compiler reports an error. Instead, use your knowledge of the data it contains to determine the size of the external representation. For example, if you know that the value stored in a `TInt` can never exceed 16 bits, you can use `RWriteStream`'s `WriteInt16L()` to externalize it and `RReadStream`'s `ReadInt16L()` to internalize it:

```
TInt i = 1234;
writer.WriteInt16L(i);
...
TInt j = reader.ReadInt16L() ;
```

If you use << to externalize a descriptor, the external representation contains both the length and character width (8 or 16 bits) so that >> has sufficient information to perform the appropriate internalization:

```

_LIT(KSampleText, "Hello world");
TBuf<16> text = KSampleText;
writer << text;
...
TBuf<16> newText;
reader >> newText;

```

This is fine, provided that the descriptor's content is reasonably small and known not to exceed a certain fixed length. If the descriptor can potentially contain a large amount of data it would be wasteful always to have to allow for the maximum. To cater for this, `HBufC` has a `NewL()` overload that takes a read stream and a maximum length as parameters and constructs a buffer of precisely the correct size to contain the data:

```

const TInt KStreamsMaxReadBufLength = 10000;

_LIT(KSampleText, "Hello world");
TBuf<16> text = KSampleText;
writer << text;
...
HBufC* newText = HBufC::NewL(reader, KStreamsMaxReadBufLength);

```

You should set the maximum length to be at least as large as the greatest amount of data that you can reasonably expect to have to handle. If the reader attempts to read more than the specified maximum, `HBufC::NewL()` leaves with the error `KErrCorrupt`.

Externalizing and Internalizing

You can externalize and internalize any class that implements `ExternalizeL()` and `InternalizeL()`, which are prototyped as:

```

class Foo
{
public:
    ...
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);
    ...
}

```

For such a class, externalization can use either:

```

Foo foo;
writer << foo

```

or:

```

Foo foo;
foo.Externalize(writer);

```

which are functionally equivalent. Similarly, for internalization, you can use either:

```
Foo foo;
reader >> foo;
```

or:

```
Foo foo;
foo.Internalize(reader);
```

The implementation of externalization and internalization functions should handle all data contained within the stream. In the following code from the Noughts and Crosses application, three classes in the project implement `Externalize()` and `Internalize()` functions and take responsibility for data associated with the designated task. The application UI class implementation calls `Externalize()` on the engine class, which is responsible for the state of the game board: it writes the state of each tile to the stream and writes the ID of the tile with the UI focus. The controller class is responsible for the input from the user; this externalize function records which player's turn it is and the state of the turn.

```
void COandXAppUi::ExternalizeL(RWriteStream& aStream) const
/* Write the application UI's state, which includes all of the
   game state, to a writable stream.
   @param aStream is the stream to which the state is written.
*/
{
    iEngine->ExternalizeL(aStream);
    iController->ExternalizeL(aStream);
    aStream.WriteInt8L(iAppView->IdOfFocusControl() );
}

void COandXEngine::ExternalizeL(RWriteStream& aStream) const
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        aStream.WriteInt8L(iTileStates[i]);
    }
}

void COandXController::ExternalizeL(RWriteStream& aStream) const
{
    aStream.WriteUInt8L(iState);
    aStream.WriteInt8L(iCrossTurn);
}
```

Internalization is performed by the same classes. The engine is internalized first, then the controller and finally the application UI class itself.

```

void COandXAppUi::InternalizeL(RReadStream& aStream)
/* Restore the game state from the supplied readable stream.
@param aStream is the stream containing the game state.
*/
{
    iEngine->InternalizeL(aStream);
    iController->InternalizeL(aStream);
    ReportWhoseTurn() ;
    iAppView->MoveFocusTo(aStream.ReadInt8L() );
}

void COandXEngine::InternalizeL(RReadStream& aStream)
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        iTileStates[i] = aStream.ReadInt8L() ;
    }
}

void COandXController::InternalizeL(RReadStream& aStream)
{
    iState = (TState) aStream.ReadUInt8L() ;
    iCrossTurn = aStream.ReadInt8L() ;
}

```

It should be clear from this discussion that both << and >> can leave. You will need to use them with a trap if they occur in a non-leaving function.

We can see from the example stream code that the classes are very precise about data they are internalizing or externalizing. In the controller code, the member variable `iState` takes 4 bytes of memory in stack space although it is stored as an 8-bit unsigned integer and `ReadUInt8L()` and `WriteUInt8L()` are used. This is because the number of states is pre-defined in the enumeration `TState` and can never be negative. `iCrossTurn`, a `TBool`, again uses 4 bytes of stack space but can be stored in a single byte as it can only have two values, 0 and 1.

Functions for Writing and Reading Streams

The stream base classes provide a variety of functions (see Table 7.1) to read and write specific data types, ranging from 8-bit integers to 64-bit real numbers. These functions are called when you use << and >> on built-in types. You should make explicit use of these functions only when you have to, as in the above example to deal with `TInts`, or when you want to be very specific about how your data is externalized.

Table 7.1 Stream functions

Read and Write Stream functions	Type	External Format
<code>ReadL()</code> <code>WriteL()</code>		Data in internal format
<code>ReadL(RwriteStream&)</code> <code>WriteL(RReadStream&)</code>		Transfer from other stream type
<code>ReadInt8L()</code> <code>WriteInt8L()</code>	<code>TInt8</code>	8-bit signed integer
<code>ReadInt16L()</code> <code>WriteInt16L()</code>	<code>TInt16</code>	16-bit signed integer; bytes stored little-endian
<code>ReadInt32L()</code> <code>WriteInt32L()</code>	<code>TInt32</code>	32-bit signed integer; bytes stored little-endian
<code>ReadUInt8L()</code> <code>WriteUInt8L()</code>	<code>TUInt8</code>	8-bit unsigned integer
<code>ReadUInt16L()</code> <code>WriteUInt16L()</code>	<code>TUInt16</code>	16-bit unsigned integer; bytes stored little-endian
<code>ReadUInt32L()</code> <code>WriteUInt32L()</code>	<code>TUInt32</code>	32-bit unsigned integer; bytes stored little-endian
<code>ReadReal32L()</code> <code>WriteReal32L()</code>	<code>TReal32</code>	32-bit IEEE 754 single-precision floating point
<code>ReadReal64L()</code> <code>WriteReal64L()</code>	<code>TReal</code> , <code>TReal64</code>	64-bit IEEE 754 double-precision floating point

The stream base classes also provide a range of `WriteL()` and `ReadL()` functions to handle raw data. You need to be careful when using them because:

- the raw data is written to the stream exactly as it appears in memory, so you must make sure that it is already in an external format before calling `WriteL()`
- `ReadL()` must read exactly the same amount of data that was written by the corresponding `WriteL()`, perhaps by writing the length immediately before the data, or terminating the data with a unique delimiter
- you must implement a way of dealing with the maximum expected length of the data, say, by using the strategy described earlier for `HBufC::NewL()`
- the 16-bit `WriteL()` and `ReadL()` functions don't give you the standard Unicode compression and decompression that you get when you use `<<` and `>>`.

The full range of `WriteL()` functions is:

```
class RWriteStream
{
public:
    ...
    IMPORT_C void WriteL(const TDesC8& aDes);
    IMPORT_C void WriteL(const TDesC8& aDes, TInt aLength);
    IMPORT_C void WriteL(const TUint8* aPtr, TInt aLength);
    ...
    IMPORT_C void WriteL(const TDesC16& aDes);
    IMPORT_C void WriteL(const TDesC16& aDes, TInt aLength);
    IMPORT_C void WriteL(const TUint16* aPtr, TInt aLength);
    ...
}
```

The `WriteL(const TDesC8& aDes)` function writes out all of the data contained in the descriptor. The next two functions write `aLength` bytes, from the beginning of the descriptor and the location indicated by the pointer, respectively. The 16-bit versions behave similarly, except that they write Unicode characters instead of bytes.

The corresponding set of `ReadL()` functions is:

```
class RReadStream
{
public:
    ...
    IMPORT_C void ReadL(TDes8& aDes);
    IMPORT_C void ReadL(TDes8& aDes, TChar aDelim);
    IMPORT_C void ReadL(TDes8& aDes, TInt aLength);
    IMPORT_C void ReadL(TUint8* aPtr, TInt aLength);
    IMPORT_C void ReadL(TInt aLength);
    ...
    IMPORT_C void ReadL(TDes16& aDes);
    IMPORT_C void ReadL(TDes16& aDes, TChar aDelim);
    IMPORT_C void ReadL(TDes16& aDes, TInt aLength);
    IMPORT_C void ReadL(TUint16* aPtr, TInt aLength);
    ...
}
```

The `ReadL(TDes8& aDes)` function reads `aDes.MaxLength()` bytes from the stream, exactly filling the descriptor. The next function reads the number of characters up to, and including, the first occurrence of the specified delimiter character (if the delimiter is not read, it fills the descriptor). The following two functions read `aLength` bytes into the descriptor or the memory location indicated by the pointer. The fifth function reads and discards `aLength` bytes. As with `WriteL()`, the 16-bit versions have similar actions, but read Unicode characters rather than bytes.

So far, we have seen that we can store the state of a game in a stream and that stream may be stored in a file and persist between power cycles of a Symbian OS phone. We can also see that the store could be

Table 7.2 Types of stream

Header file	Class names	Medium
s32file.h	RFileWriteStream, RFileReadStream	A file; constructors specify either an open RFile or a file server session and a file name
s32mem.h	RDesWriteStream, RDesReadStream	Memory, identified by a descriptor
s32mem.h	RMemWriteStream, RMemReadStream	Memory, identified by a pointer and length
s32mem.h	RBufWriteStream, RBufReadStream	Memory, managed by a dynamic buffer derived from CBufBase; as new data is written through an RBufWriteStream, the destination CBufBase is expanded as necessary
s32std.h	RStoreWriteStream, RStoreReadStream	A stream store; constructors specify a stream store and a stream ID
s32stor.h	RDictionaryReadStream, RDictionaryWriteStream	A dictionary store; constructors specify a dictionary store and a UID
s32crypt.h	REncryptStream, RDecryptStream	A stream; constructors specify the host stream, the CSecurityBase algorithm, and a string (a password) to initialize the CSecurityBase

memory based. As a single entity, a stream is useful for storing simple data structures. However, for more complicated data or for a collection of more than one set of data, we need to use a *store*. Table 7.2 shows some other stream types available in Symbian OS.

7.5 Stores

A Symbian OS store is a collection of streams, and is generally used to implement the persistence of objects. The store class hierarchy is illustrated in Figure 7.2, where the concrete class names are in bold text. The base class for all stores is CStreamStore, whose API defines all the functionality needed to create and modify streams. The classes derived from CStreamStore selectively implement the API according to their needs.

As with streams, stores can reside in a variety of different media, such as in memory (CBufStore) or in a stream, perhaps within another store (CEmbeddedStore), but the most commonly used medium is a file.

The two file-based stores are CDirectFileStore and CPermanentFileStore. The main way in which they differ from each other is

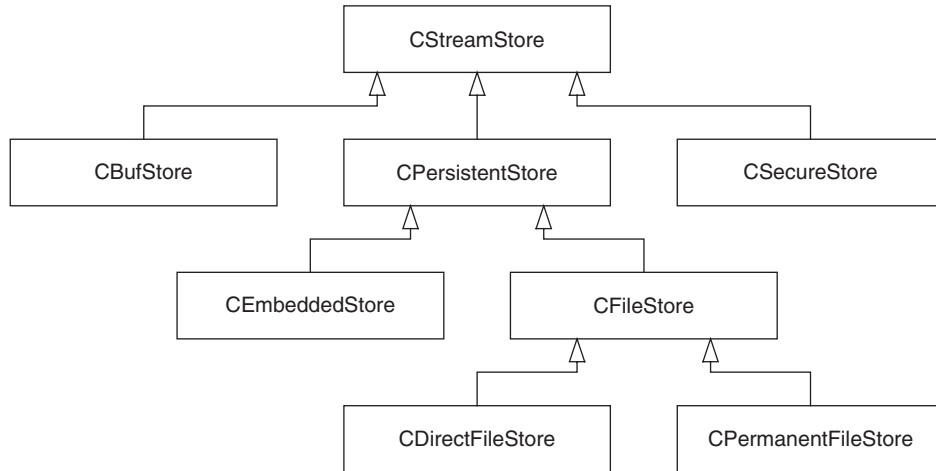


Figure 7.2 Store class hierarchy

that `CPermanentFileStore` allows the modification of streams after they have been written to the store, whereas `CDirectFileStore` does not. This difference results in the stores being used to store persistent data for two different types of application, depending on whether the store or the application itself is considered to contain the primary copy of the application's data.

For an application such as a database, the primary copy of the data is the database file itself. At any one time, the application typically holds in memory only a small number of records from the file, in order to display or modify them. Any modified data is written back to the file, replacing the original version. Such an application would use an instance of `CPermanentFileStore`, with each record being stored in a separate stream.

Other applications hold all their data in memory and, when necessary, load or save the data in its entirety. Such applications can use a `CDirectFileStore` since they never modify the content of the store but simply replace the whole store with an updated version.

Another distinction that can be made between the store types is whether or not they are *persistent*. A *persistent* store can be closed and opened again to read its content. The data in such a store therefore persists after a program has closed it and even after the program itself has terminated. A `CBufStore` is not persistent, since its in-memory data is lost when the store is closed; a file-based store is persistent: the content remains intact for the life of the file.

The persistence of a store is implemented in the `CPersistentStore` abstract class by defining a *root stream*, which is always accessible on opening the store. The root stream, in turn, contains a *stream dictionary* that contains stream IDs – pointers to other streams – as illustrated in

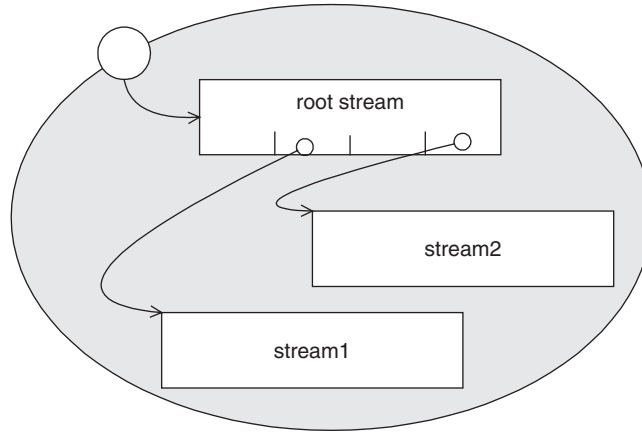


Figure 7.3 Logical view of a persistent store

Figure 7.3. The stream IDs provide access to the rest of the data in the store.

The physical structure of a direct file store consists of:

- a 16-byte header containing three UUIDs and a 4-byte checksum
- a 4-byte stream position, indicating the start of the root stream, which normally contains the stream dictionary
- a sequential list of the data for each stream, usually ending with the root stream.

The stream dictionary starts with an externalized `TCardinality` of the number of associations in the dictionary, followed by the associations themselves. Each association occupies eight bytes and consists of a stream's UUID and a 4-byte stream position, indicating the start of the relevant stream.

`TCardinality` is a useful class that provides a compact externalization of numbers (such as the count of elements in an array) which, though potentially large, normally have small values.

The structure of a permanent file store follows the same logical pattern, but the physical structure is more complex since it has to allow for records to be replaced or deleted.

Creating a Persistent Store

The following code illustrates how to create a persistent store. The example creates a direct file store, but creating a permanent file store follows a similar pattern.

```

void CreateDirectFileStoreL(RFs& aFs, TDesC& aFileName, TUid aAppUid)
{
    CFileStore* store = CDirectFileStore::ReplaceLC(aFs, aFileName,
                                                    EFileWrite);
    store->SetTypeL(TUidType(KDirectFileStoreLayoutUid, KUidAppDllDoc,
                                                                    aAppUid));

    CStreamDictionary* dictionary = CStreamDictionary::NewLC() ;

    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(*store);
    TInt16 i = 0x1234;
    stream << i;
    stream.CommitL() ;
    CleanupStack::PopAndDestroy() ; // stream

    dictionary->AssignL(aAppUid, id);

    RStoreWriteStream rootStream;
    TStreamId rootId = rootStream.CreateLC(*store);
    rootStream << *dictionary;
    rootStream.CommitL() ;
    CleanupStack::PopAndDestroy() ; // rootStream

    CleanupStack::PopAndDestroy() ; // dictionary

    store->SetRootL(rootId);
    store->CommitL() ;

    CleanupStack::PopAndDestroy() ; // store
}

```

To create the store itself we use:

```

CFileStore* store = CDirectFileStore::ReplaceLC(aFs, aFileName,
                                                EFileWrite);

```

The call to `ReplaceLC()` creates the file, if it does not exist, or replaces an existing file. With the direct store type, we cannot modify the individual streams. In a real application, it might be more convenient to store the pointer to the file store object in member data, rather than on the stack.

You need to set the store's type, which we do by calling:

```

store->SetTypeL(TUidType(KDirectFileStoreLayoutUid,
                        KUidAppDllDoc, aAppUid));

```

The three UIDs in the `TUidType` indicate, respectively, that the file contains a direct file store, that the store is a document associated with a Symbian OS application and that the store is associated with the application whose UID is `aAppUid`. For the file to be recognized as

containing a direct file store, it is strictly only necessary to specify the first UID, leaving the other two as `KNullUid`; including the other two allows an application to be certain that it is opening the correct file.

To create a permanent file store you would use:

```
CFileStore* store = CPermanentFileStore::CreateLC(aFs, aFileName,
                                                EFileWrite);
store->SetTypeL(TUidType(KPermanentFileStoreLayoutUid, KUidAppDllDoc,
                        aAppUid));
```

We have used the store's `CreateLC()` function here, because you would not normally want to replace an entire permanent file store. It is more likely that the permanent store, once created, would be opened for edit.

Once you have created a file store, you can find its layout UID by calling `store->Layout()` which, depending on the file store's class, returns either `KDirectFileStoreLayoutUid` or `KPermanentFileStoreLayoutUid`.

We then create a stream dictionary for later use:

```
CStreamDictionary* dictionary = CStreamDictionary::NewLC();
```

Creating, writing and closing a stream follows a similar pattern to the one we saw in Section 7.4:

```
RStoreWriteStream stream;
TStreamId id = stream.CreateLC(*store);
TInt16 i = 0x1234;
stream << i;
stream.CommitL();
CleanupStack::PopAndDestroy(); // stream
```

An important difference is that, to write a stream to a store, you must use an instance of `RStoreWriteStream`, whose `CreateL()` and `CreateLC()` functions return a `TStreamId`. Once writing the stream is complete, you need to add an entry to the stream dictionary, making an association between the stream ID and an externally known UID:

```
dictionary->AssignL(aAppUid, id);
```

If, as in this case, your application writes all the application's document data into a single stream, it is safe to use the application's UID.

Once all the data streams have been written and added to the stream dictionary, you need to create a stream to contain the stream dictionary, and mark it as being the root stream:

```
RStoreWriteStream rootStream;
TStreamId rootId = rootStream.CreateLC(*store);
rootStream << *dictionary;
rootStream.CommitL() ;
CleanupStack::PopAndDestroy() ; // rootStream
...
store->SetRootL(rootId);
```

All that remains to be done is to commit all the changes made to the store and then to free its resources which, in this case, is done by the call to the cleanup stack's `PopAndDestroy()` function.

```
store->CommitL() ;
CleanupStack::PopAndDestroy() ; // store
```

If the store is not on the cleanup stack, you should simply use:

```
delete store;
```

The store's destructor takes care of closing the file and freeing any other resources.

Reading a Persistent Store

The following code opens and reads the direct file store that was created in the previous section.

```
void ReadDirectFileStoreL(RFs& aFs, TDesC& aFileName, TUid aAppUid)
{
    CFileStore* store = CDirectFileStore::OpenLC(aFs, aFileName, EFileRead);

    CStreamDictionary* dictionary = CStreamDictionary::NewLC() ;

    RStoreReadStream rootStream;
    rootStream.OpenLC(*store, store->Root() );
    rootStream >> *dictionary;
    CleanupStack::PopAndDestroy() ; // rootStream

    TStreamId id = dictionary->At(aAppUid);

    CleanupStack::PopAndDestroy() ; // dictionary

    RStoreReadStream stream;
    stream.OpenLC(*store, id);
    TInt16 j;
    stream >> j;
    CleanupStack::PopAndDestroy() ; // stream

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

After opening the file store for reading and creating a stream dictionary, you need to open the root stream, with:

```
rootStream.OpenLC(*store, store->Root() );
```

After that, you can internalize its content to the stream dictionary. We then extract one or more stream IDs, using the dictionary's `At()` function:

```
TStreamId id = dictionary->At(aAppUid);
```

The stream can now be opened using an `RStoreReadStream` and its contents may be internalized as appropriate for the application concerned. Finally, we clean up the store and stream items used.

Permanent Stores

If you create a permanent file store you can, at a later time, reopen it and add new streams, or replace or delete existing streams. To ensure that the modifications are made efficiently, replaced or deleted streams are not physically removed from the store, so the store increases in size with each such change. To counteract this, the stream store API includes functions to compact the store, by physically removing replaced or deleted streams.

`CompactL()` is used to remove free space from the store file, effectively shrinking the size of the file. `ReplaceL()` is used to retrieve unused space within the store file making it available for use by new or modified streams. Both operations can take a prolonged period of time, depending on the complexity and amount of change made to the store. To avoid a long wait for compaction to complete, `RStoreReclaim` can be used to perform the task in smaller steps instead of one single step. `RStoreReclaim` must be open on the store object and released when finished with.

Stores are transactional; that is, each change made by calling `CommitL()` can be considered atomic. Prior to the commit point, if it is no longer required, the change can be reversed by calling `Revert()`. The store reverts to the same state it was in after the most recent `CommitL()` call.

You need to be especially careful that you do not lose a reference to any stream within the store. This is analogous to a memory leak within an application and results in the presence of a stream that can never be accessed or removed. Arguably, losing access to a stream is more serious than a memory leak, since a persistent file store outlives the application that created it. The stream store API contains a tool, whose central class is `CStoreMap`, to assist with stream cleanup.

You can find more information on these topics in the SDK. For more complex data to be saved to file using stores requires more complicated source code. For this reason, Symbian OS provides database APIs in the DBMS component and the SQL RDBMS (see Chapter 22).

Embedded Stores

Not all stores are as simple as the ones we have described so far. As illustrated in Figure 7.4, a store may, in fact, contain an arbitrarily complex network of streams. Any stream may contain another stream – by including its ID – and a stream may itself contain an embedded store.

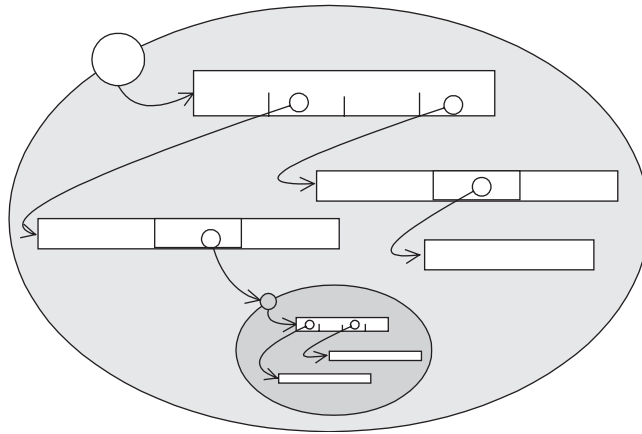


Figure 7.4 A more complex store

It may be useful to store a collection of streams in an embedded store: from the outside, the embedded store appears as a single stream and it can, for example, be copied or deleted as a whole, without the need to consider its internal complexities. An embedded store cannot be modified and thus behaves like a direct file store – which means that you can't embed a permanent file store.

The following code illustrates writing a permanent file store that contains an embedded file store.

```
void COandXAppUi::WriteEmbeddedFileStoreL(RFs& aFs, TDesC& aFileName)
{
    CFileStore* mainStore = CPermanentFileStore::ReplaceLC(aFs,
                                                            aFileName, EFileWrite);
    mainStore->SetTypeL(KPermanentFileStoreLayoutUid);

    RStoreWriteStream hostStream;
    TStreamId embeddedStoreId = hostStream.CreateLC(*mainStore);

    // construct the embedded store
    CPersistentStore* embeddedStore = CEmbeddedStore::NewLC(hostStream);
```



```

RStoreWriteStream subStream;
TStreamId id = subStream.CreateLC(*embeddedStore);
TInt16 i = 0x1234;
subStream << i;
subStream.CommitL() ;
CleanupStack::PopAndDestroy() ; // subStream
embeddedStore->SetRootL(id);
embeddedStore->CommitL() ;
CleanupStack::PopAndDestroy() ; // embeddedStore

hostStream.CommitL() ;
CleanupStack::PopAndDestroy() ; // hostStream

RStoreWriteStream mainStream;
TStreamId mainId = mainStream.CreateLC(*mainStore);
TInt16 j = 0x3456;
mainStream << j;
mainStream.CommitL() ;
CleanupStack::PopAndDestroy() ; //mainStream

RStoreWriteStream rootStream;
TStreamId rootId = rootStream.CreateLC(*mainStore);
TInt16 k = 0x5678;
rootStream << k;
rootStream << mainId;
rootStream << embeddedStoreId;
rootStream.CommitL() ;
CleanupStack::PopAndDestroy() ; //rootStream

mainStore->SetRootL(rootId);
mainStore->CommitL() ;
CleanupStack::PopAndDestroy() ; // mainStore
}

```

To create the embedded store, you first create a stream in the main store as normal:

```

RStoreWriteStream hostStream;
TStreamId embeddedStoreId = hostStream.CreateLC(*mainStore);

```

Then you create the embedded store hosted in that stream:

```

// construct the embedded store
CPersistentStore* embeddedStore = CEmbeddedStore::NewLC(hostStream);

```

You create and write streams within the embedded store, and set the root stream, in exactly the same way as for any other store:

```

RStoreWriteStream subStream;
TStreamId id = subStream.CreateLC(*embeddedStore);
TInt16 i = 0x1234;
subStream << i;
subStream.CommitL() ;

```

```
CleanupStack::PopAndDestroy() ; // subStream
embeddedStore->SetRootL(id);
embeddedStore->CommitL() ;
```

The IDs of the other streams are stored directly in the root stream's data, rather than using a stream dictionary:

```
RStoreWriteStream rootStream;
TStreamId rootId = rootStream.CreateLC(*mainStore);
TInt16 k = 0x5678;
rootStream << k;
rootStream << mainId;
rootStream << embeddedStoreId;
rootStream.CommitL() ;
CleanupStack::PopAndDestroy() ; //rootStream
```

The following code reads the file that was written in the previous example:

```
void COandXAppUi::ReadEmbeddedFileStoreL(RFs& aFs, TDesC& aFileName)
{
    CFileStore* store = CPermanentFileStore::OpenLC(aFs,
                                                    aFileName, EFileRead);

    RStoreReadStream rootStream;
    rootStream.OpenLC(*store, store->Root() );
    TInt16 k;
    TStreamId mainId;
    TStreamId embeddedStoreId;
    rootStream >> k;
    rootStream >> mainId;
    rootStream >> embeddedStoreId;
    CleanupStack::PopAndDestroy() ; // rootStream

    RStoreReadStream mainStream;
    mainStream.OpenLC(*store, mainId);
    TInt16 j;
    mainStream >> j;
    CleanupStack::PopAndDestroy() ; // mainStream

    RStoreReadStream hostStream;
    hostStream.OpenLC(*store, embeddedStoreId);
    CPersistentStore* embeddedStore = CEmbeddedStore::FromLC(hostStream);
    RStoreReadStream subStream;
    subStream.OpenLC(*embeddedStore, embeddedStore->Root() );
    TInt16 i;
    subStream >> i;
    CleanupStack::PopAndDestroy(3); // subStream, embeddedStore
                                   // and hostStream

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

After opening the file, we open the root stream and read its data, including the IDs of the other two streams:

```
RStoreReadStream rootStream;
rootStream.OpenLC(*store, store->Root() );
TInt16 k;
TStreamId mainId;
TStreamId embeddedStoreId;
rootStream >> k;
rootStream >> mainId;
rootStream >> embeddedStoreId;
CleanupStack::PopAndDestroy() ; // rootStream
```

To read the embedded store, open the host stream using the ID from the root stream:

```
RStoreReadStream hostStream;
hostStream.OpenLC(*store, embeddedStoreId);
```

Open the embedded store it contains, using the `FromLC()` function of `CEmbeddedStore`:

```
CPersistentStore* embeddedStore = CEmbeddedStore::FromLC(hostStream);
```

Then we can open the embedded store's root stream and read it like any other stream:

```
RStoreReadStream subStream;
subStream.OpenLC(*embeddedStore, embeddedStore->Root() );
TInt16 i;
subStream >> i;
```

Stores and the Application Architecture

In UIQ, the application architecture provides support for an application to save its data in a direct file store via the base class `CEikDocument`. The majority of S60 applications don't need to access an application document file and this access is disabled. If necessary, you can enable it in an S60 application by supplying an implementation of the application document's `OpenFileL()` function, as shown in the following document class definition:

```
class COandXDocument : public CAknDocument
{
...
public:
...
// override CAknDocument
```

```
virtual CFileStore* OpenFileL(TBool aDoOpen,
                             const TDesC& aFilename, RFs& aFs);
...
};
```

The `OpenFileL()` function simply calls `CEikDocument`'s `OpenFileL()`:

```
CFileStore* COandXDocument::OpenFileL(TBool aDoOpen,
                                       const TDesC& aFilename, RFs& aFs)
{
    return CEikDocument::OpenFileL(aDoOpen, aFilename, aFs);
}
```

When support for the application's document file is enabled, all you need to supply are implementations of the document's `StoreL()` and `RestoreL()` functions, which are prototyped as:

```
public:
    void StoreL(CStreamStore& aStore, CStreamDictionary& aStreamDic) const;
    void RestoreL(const CStreamStore& aStore,
                 const CStreamDictionary& aStreamDic);
```

The default implementations are virtual and empty; an application may implement these functions as:

```
void COandXDocument::StoreL(CStreamStore& aStore,
                           CStreamDictionary& aStreamDic) const
{
    TStreamId id = iAppUi->StoreL(aStore);
    aStreamDic.AssignL(KUidOandXApp, id);
}

void COandXDocument::RestoreL(const CStreamStore& aStore,
                             const CStreamDictionary& aStreamDic)
{
    TStreamId id = aStreamDic.At(KUidOandXApp);
    iAppUi->RestoreL(aStore, id);
}
```

The responsibility for storing and restoring the application's data is transferred to the application UI, which uses code similar to that described in Section 7.4:

```
TStreamId COandXAppUi::StoreL(CStreamStore& aStore) const
{
    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(aStore);
    stream << *this; // alternatively, use ExternalizeL(stream)
    stream.CommitL();
}
```

```

CleanupStack::PopAndDestroy() ;
return id;
}

void COandXAppUi::RestoreL(const CStreamStore& aStore,
                           TStreamId aStreamId)
{
    RStoreReadStream stream;
    stream.OpenLC(aStore, aStreamId);
    stream >> *this; // alternatively use InternalizeL(stream)
    CleanupStack::PopAndDestroy() ;
}

```

These, in turn, use the Application UI's `InternalizeL()` and `ExternalizeL()` functions, which were described in Section 7.4:

```

void COandXAppUi::ExternalizeL(RWriteStream& aStream) const
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        aStream.WriteInt8L(iTileState[i]);
    }
    aStream.WriteInt8L(iGameStatus);
    aStream.WriteInt8L(iCrossTurn);
    aStream.WriteInt8L(iAppView->IdOfFocusControl() );
}

void COandXAppUi::InternalizeL(RReadStream& aStream)
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        iTileState[i] = aStream.ReadInt8L() ;
    }
    iGameStatus = aStream.ReadInt8L() ;
    iCrossTurn = aStream.ReadInt8L() ;
    SetNavigationPanelL(iCrossTurn);
    iAppView->MoveFocusTo(aStream.ReadInt8L() );
}

```

The application architecture takes care of almost everything else, including creating the store when the application is run for the first time and reading the file when the application is run on subsequent occasions.

In addition to creating the root stream and the stream containing the stream dictionary, it also adds an application-identifier stream. This stream contains the application's UID and the application file name (for example, `OANDX.EXE`) and can be used to verify that the file is a valid one for the application to open and read. The application-identifier stream is identified in the stream dictionary by a UID of `KUidAppIdentifierStream` and is created by code of the form:

```

void WriteAppIdentifierL(CFileStore* aStore,
                        CStreamDictionary* aDict, TUid aAppUid, TDesC& aAppName)

```

```

{
    TAppIdentifier appIdent(aAppUid, aAppName);
    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(*aStore);
    stream << appIdent;
    stream.CommitL();
    CleanupStack::PopAndDestroy(); // stream
    aDict->AssignL(KUidAppIdentifierStream, id);
}

```

In S60 applications, you need to ensure that the file is updated when the application is closed. You do this by calling the `SaveL()` function of the application UI in the exit cases of `HandleCommandL()`:

```

void COandXAppUi::HandleCommandL(TInt aCommand)
{
    switch(aCommand)
    {
        case EEikCmdExit:
        case EAknSoftkeyExit:
        {
            SaveL();
            Exit();
        }
        break;
        ...
    }
}

```

UIQ applications don't normally exit, unless they are forced to do so by the memory manager. If the memory manager needs to recover memory, it calls an application document's `MSaveObserver::SaveL()` function, passing it a value of `MSaveObserver::EReleaseRAM`. On receipt of this value, an application is expected to save its state and data and then close down; this is the default action for all applications that derive their document class from `CQikDocument`. Any application that ignores this request – for example, by deriving its document from `CEikDocument` or by providing its own implementation of `MSaveObserver::SaveL()` – is liable to be forced to close, without any further opportunity to save its state and data. System applications and applications that have marked themselves as busy using `CEikonEnv::SetBusy()` are not forced to close.

Dictionary Stores and INI Files

In addition to an application's document file, the application architecture provides support for an application to open, read and modify a second file. By convention, the file has the same name as the application, but has the extension `.ini`; hence such files are known as INI files.

The intention is that an INI file should be used to store global settings and preferences that are independent of the document data that the

application is processing. An application can then, in principle, open a different document without affecting the existing preference settings.

In practice, applications running on mobile phones tend to use only a single document and therefore have no real need to use INI files. In recognition of this fact, the S60 UI disables the application architecture's support for INI files, but it is a simple matter to restore it if necessary, by replacing the `OpenIniFileLC()` function of `CAKnApplication` with a call to the `CEikApplication` function, such as:

```
CDictionaryStore* CMyAppApplication:: OpenIniFileLC(RFs& aFs) const
{
    return CEikApplication::OpenIniFileLC(aFs);
}
```

As well as opening the application's INI file for exclusive read-write access – and pushing it to the cleanup stack – this function creates the INI file if it does not exist and replaces a corrupt file. The function is called by the application architecture, for example to record the last opened file, and can be called by your application code.

The INI file is represented by an instance of `CDictionaryFileStore`, which derives from the `CDictionaryStore` base class. These class names are slightly misleading since they have no connection with stream dictionaries. Also, since they do not derive from `CStreamStore`, they do not represent stream stores. Despite this, there are similarities of usage: the dictionary store contains streams associated with UIDs, and you read and write the streams by means of `RDictionaryReadStream` and `RDictionaryWriteStream` classes, in the same way as you would with a stream store.

However, a significant difference is that a dictionary store never contains more than a simple list of streams, unlike the complex network that is possible in a stream store. Furthermore, you use UIDs to access the streams directly, as illustrated in Figure 7.5, rather than via a stream dictionary.

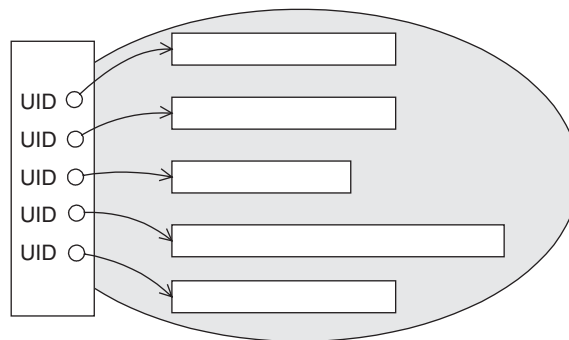


Figure 7.5 A dictionary store

The following simple examples, designed to be used as member functions of an application's application UI, illustrate writing to and reading from the application's INI file.

```
void WriteToIniFileL(RFs& aFs)
{
    CDictionaryStore* iniFile = Application() ->OpenIniFileLC(aFs);
    RDictionaryWriteStream stream;
    // direct access by UID
    stream.AssignLC(*iniFile, TUid::Uid(0x101ffac5));
    TUint16 i = 0x3456;
    stream << i;
    stream.CommitL() ;
    iniFile->CommitL() ;
    CleanupStack::PopAndDestroy(2); // stream and iniFile
}

void ReadIniFileL(RFs& aFs)
{
    CDictionaryStore* iniFile = Application() ->OpenIniFileLC(aFs);
    RDictionaryReadStream readStream;
    // direct access by UID
    readStream.OpenLC(*iniFile, TUid::Uid(0x101ffac5));
    TInt16 i;
    readStream >> i;
    CleanupStack::PopAndDestroy(2); // readStream and iniFile
}
```

Summary

In this chapter, we have discussed the following topics.

- Symbian OS has a system server that provides basic file and directory manipulation.
- Several file systems and media types are typically supported by a Symbian OS phone.
- The introduction of platform security dictates where an application should locate files within a file system. A private directory is useful for private data.
- An `RFile` has limited functionality: basic file reads and writes; richer APIs are required to assist application developers.
- Files located in a process's private directory can be shared with another process by file handle. This allows a server to share its data while storing it in a secure location.
- An `RFileBuf` provides additional buffering to an `RFile`. This is useful for applications that load data repeatedly from a file in small sections.

- A file stream builds on this functionality by allowing specific data structures from a class to be externalized and internalized to and from a file.
- If more than one object must be externalized, separate streams may be desirable. A collection of streams can be saved in a single entity called a store.
- File-based stores are called persistent stores as they persist over a device power cycle, provided that they are stored on non-volatile media.
- Direct stores are written to once and then replaced in their entirety.
- Permanent stores are those in which streams can be modified after creation. Maintenance, such as reclamation, is required to free space previously used by a permanent store.
- A stream can be located by a stream ID, which is stored in a dictionary stream in a known location.
- A dictionary store is useful as each stream can be accessed directly by UID instead of stream ID.
- Store files can grow in complexity. At some point, you must decide whether to use a database.

8

Interprocess Communication Mechanisms

This chapter examines the interprocess communication (IPC) mechanisms available to applications and servers in Symbian OS. Understanding when and how these are required is essential in properly architecting components and making optimum use of the capabilities of the operating system. Each of these mechanisms has its own strengths and limitations, so it is important to choose the right one for a particular task.

Symbian OS offers thread-signaling and synchronization primitives, such as semaphores and mutexes, that can be used to manage and signal ownership of shared memory between threads and processes [Sales 2005], but those are quite low level and basic and are not dealt with here. Symbian OS also offers IPC mechanisms and APIs to exchange data between threads and processes, using higher-level abstractions such as client–server, publish and subscribe and message queues.

8.1 Overview

Client–Server Session IPC

Since the whole of Symbian OS was architected around event-driven, user-initiated interactions, the communication mechanism of choice for most user-space components is the client–server session-based IPC. This is the case because the focus is on providing responsiveness to human users, while enabling many components and applications to share services and resources. These services and resources, as discussed in Chapter 2, are accessed through user-side servers that mediate access to them; hence the ubiquitous use of the client–server IPC mechanism.

Starting in the early 1980s, Symbian OS and its 16-bit EPOC predecessor were designed for people to use and interact with. This is in

contrast to most other embedded and real-time operating systems, which have been designed for use without a user interface, in data planes or control systems. From the outset, devices using Symbian OS had to cope with large amounts of user-initiated I/O, so CPU cycles were precious and reserved for the user. In such an environment, where many applications are using many services simultaneously, all interactions have to be asynchronous and non-blocking, thus ensuring responsiveness for the user.

In client-server IPC, clients connect to servers by name. The first task is to establish a session that then provides the context for all further communication between the client and the server. Such session-based communication comprises client requests and server responses (see Figure 8.1). Within the Symbian OS kernel, which mediates all messages, such requests and responses are associated through session objects. For each session, the kernel delivers messages that the server can use to retrieve data from the relevant client's space (through kernel mediation), process the data and then return. Finally, the server notifies the client that its request is complete. Due to its connection-oriented (albeit message-centric) session-based communication, the client-server mechanism provides a guaranteed request-completion mechanism.

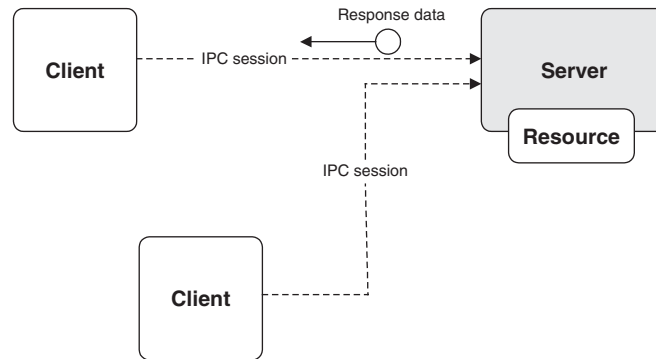


Figure 8.1 Client-server IPC mechanism

For communication to take place in this way, a session object has to be established in the kernel, with a guaranteed number of message slots for the server. This defines the maximum number of parallel outstanding requests which a particular client can issue to the server. Session-based communication ensures that all clients are notified in the case of an error or shutdown of a server and that all resources are cleared when a client disconnects or dies or something else goes wrong.

This type of communication paradigm is good for when many clients need to reliably access a service or shared resource concurrently. In that case the server serializes and mediates access to the service accordingly. Clients must initiate a connection to the server and the server is there to

respond on demand; thus the relationship between clients and servers is one-to-one but not peer-to-peer. For client services and user-initiated I/O, the Symbian OS client–server IPC serves its lightweight micro-kernel-based architecture well.

When deciding whether or not to use client–server IPC, you need to bear the following points in mind:

- clients must know which server provides the service they want
- permanent sessions are maintained between clients and server
- client–server IPC is not really suitable for event multicasting
- although delivery is guaranteed, there is no guarantee of real-time, deterministic delivery.

Publish and Subscribe IPC

Publish and subscribe (P&S) is an IPC mechanism, also known as ‘properties’, that provides a means to define and publish system-wide *global variables*. Such properties are communicated to more than one interested peer asynchronously.

The publish and subscribe API can be used by both user- and kernel-side programs, via similar APIs, and thus can also provide asynchronous communication between user- and kernel-side components. Only user-side usage is discussed in this text.

As shown in Figure 8.2, publish and subscribe threads may have the role of either the publisher or the subscriber, while any thread can be the one which defines the property to be published. Such properties are single data values, uniquely identified with an integral key. Properties have identity and type. The identity and type of a property is the only information that must be shared between a publisher and a subscriber – there is no need to provide interface classes or functions, though that may be desirable in many designs.

There are six basic operations that can be performed on properties:

- define: create a property variable and define its type and identity
- delete: remove a property from the system
- publish: change the value of a property
- retrieve: get the current value of a property
- subscribe: register for notification of changes to a property
- unsubscribe: deregister for notifications of changes.

Of these operations, the first two need to be coupled in the same thread, as it is only permissible for the defining thread to delete a property. Once

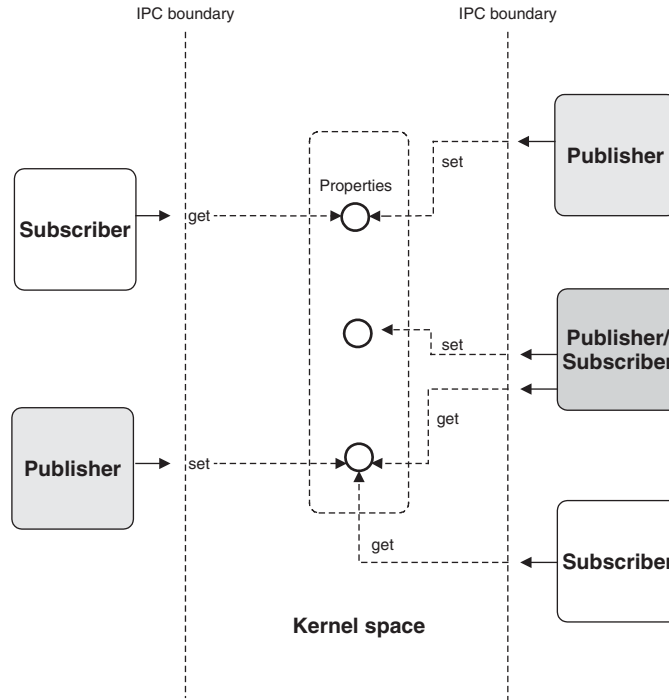


Figure 8.2 Publish and subscribe IPC mechanism

defined, the property persists in the kernel until Symbian OS reboots or the property is deleted. Its lifetime is not tied to that of the thread or process that defined it. When properties are deleted, any outstanding subscriptions for the property are notified of the change, indicating that the properties can no longer be found. An idiosyncrasy worth noting is that the thread defining (and deleting) a property does not necessarily have rights to publish it or subscribe to it.

Since both publishers and subscribers are allowed to define a property, it is not essential for a property to be defined before it is accessed. This paradigm allows for lazy definition of properties and publishing a property that is not defined is not necessarily a programming error.

For some operations, such as publication and retrieval of properties, there are two modes of transaction. Although P&S operations are connectionless, they can be used either transiently or after having set up the association between user and property in the kernel. Thus publishers and subscribers may 'attach' to properties prior to operation.

Subsequently, properties can be published or retrieved, either by using a previously attached handle or by specifying the property category and key associated with the new value. In EKA2, a previously attached handle has the benefit of deterministic, bounded execution time, suitable

for high-priority, real-time tasks. However, an exception to this benefit occurs when publishing a byte-array property that requires allocating a larger space for the new data.

Properties are read and written atomically, so it is not possible for threads reading the property to get an inconsistent value, or for multiple simultaneously published values to be confused.

As far as subscriptions and cancellations are concerned, clients of P&S must register their interest, by attachment, prior to usage. This is due to the asynchronous nature of the notification. For the kernel to recognize deterministically which threads need to be notified of a change in some property, that interest must be associated with the property.

Notification of a property update happens in four stages: a client registers its interest in a property; the property's value is published; the client is notified of the publication (by completion of its subscription request); after notification, the client initiates retrieval of the updated property.

Message Queues IPC

Contrary to the connection-oriented nature of client-server IPC, message queues offer a peer-to-peer, many-to-many communication paradigm. Using the Message Queue API, threads may send messages to interested parties without needing to know if any thread is listening. Moreover, the identity of the recipient is not needed for such communication.

There are five basic operations that can be performed on a message queue:

- creating or opening a message queue
- sending a message
- receiving a message
- waiting for space to become available in the queue
- waiting for data to arrive in the queue.

Message queues can be named and visible to all processes, anonymous but visible to all processes, or local to the current process only. The message queues themselves are managed by the kernel and are sized at the time of their creation, which allows for real-time, deterministic delivery of messages in EKA2, for both sending and receiving of messages.

Threads that use message queues must obtain a valid handle in order to access a particular queue. They may then send (i.e., add) messages to, and receive (i.e., remove) messages from, the queue. Subsequently, a queue may run out of space, in which case senders can either block

on sending or be notified that a queue overflow would occur and thus be allowed to retry later. Similarly, on message reception, readers can either block indefinitely until a message arrives in the queue or be notified asynchronously if there is no message pending for reception and retry later. Software developers therefore have the choice as to which message-sending semantics to employ.

Although it is possible for queues to have multiple readers, in most cases such an arrangement needs to be combined with some sort of out-of-band collaboration between the reader threads.

Message queues effectively allow for fire-and-forget communication semantics between threads. Readers are guaranteed the delivery of messages to the queues, but delivery to final recipients is not guaranteed. Moreover, neither messages nor queues are persistent; they are reference-counted objects that are cleaned up when the last handle to the queue is closed. Both send and receive operations effectively copy the message structures to and from the kernel.

A thread may query a queue to find out if it has space available for messages or if any messages are pending for delivery. Moreover, a thread can interrogate the queue as to its message size. Additionally, a thread may choose to wait for a queue until data or space becomes available.

With the Symbian OS Message Queue API, queues can also be typed so that it is possible to send and receive only messages of a specific type. This paradigm allows for type-safe IPC between participants without the overhead of type checking and the potential errors that may arise.

In general, since the size of the messages that can be exchanged in this API is relatively small (between four and 256 bytes), queues lend themselves nicely to timely event notification from many sources to

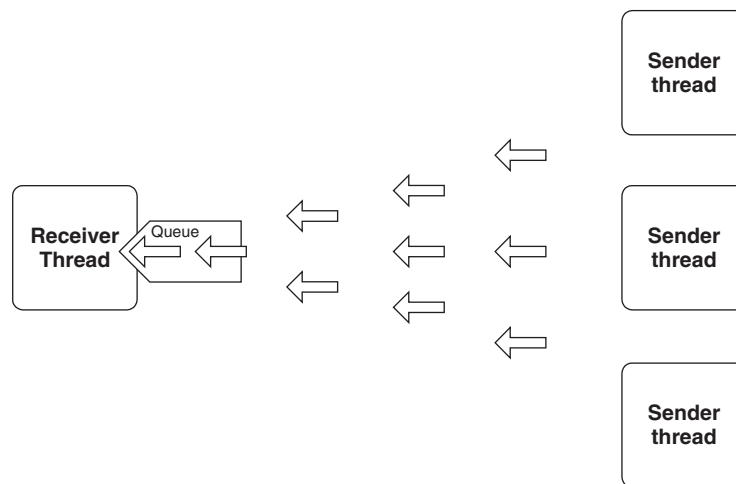


Figure 8.3 Message queue IPC mechanism

one recipient, as well as being an efficient means of passing memory descriptors and pointers between threads in the same process. This latter usage is usually implemented via local queues and can be employed effectively for processor-intensive and multimedia applications.

8.2 Client-server IPC

Client-server communication begins with the client building a session to the corresponding server, which is then used by the client and the server to pass messages and data. There are eight main high-level abstractions that a developer implementing client or server code must be aware of: `RServer2`, `CServer2`, `CSession2`, `RMessage2`, `RMessagePtr2`, `RSessionBase`, `RSubSessionBase` and `TIpArgs`.

To associate a client API with a server (see Figure 8.4), you need to use an `RSessionBase` to connect to an object derived from `CServer2`. This creates an object derived from `CSession2` in order to handle the server's side of the session. When the session is established, messages and data can be communicated to the server and data is written back to the client using an `RMessage2` object.

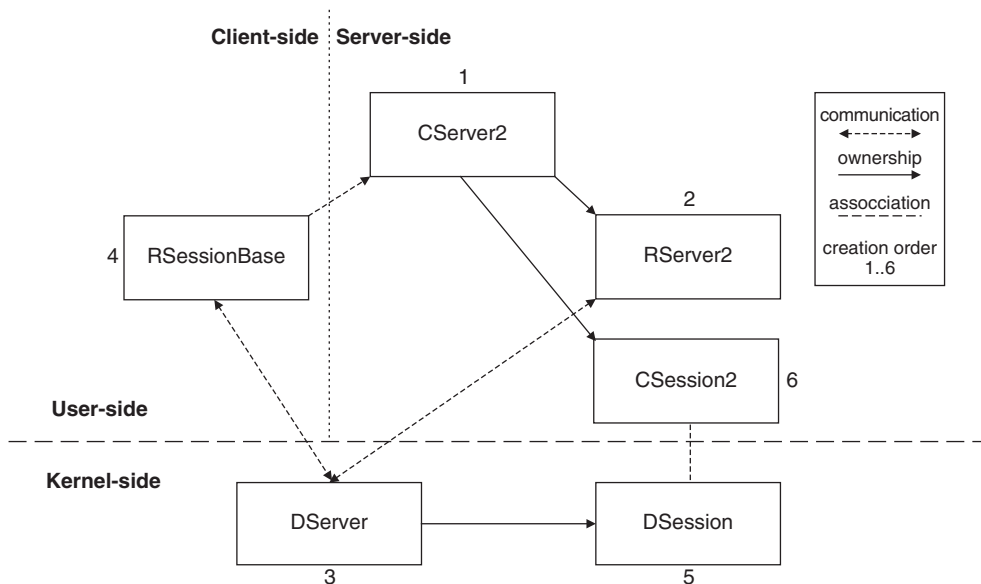


Figure 8.4 Details of client-server IPC mechanism

Setting up a Session

In order for a client thread to use a server, it needs to establish a session with that server. To do so, the client calls the `CreateSession()` method

of the `RSessionBase` class, passing the name of the server to which it wants to connect.

In order to associate a client with a server, the Symbian OS kernel internally creates a new `DSession` kernel object that is then attached to a `DServer` kernel object which represents the server.

```
class CSession2 : public CBase
{
    friend class CServer2;
public:
    IMPORT_C virtual ~CSession2() = 0;
private:
    IMPORT_C virtual void CreateL(); // Default method, does nothing
public:
    inline const CServer2* Server() const;
    IMPORT_C void ResourceCountMarkStart();
    IMPORT_C void ResourceCountMarkEnd(const RMessage2& aMessage);
    IMPORT_C virtual TInt CountResources();

    virtual void ServiceL(const RMessage2& aMessage) = 0;
    IMPORT_C virtual void ServiceError(const RMessage2& aMessage,
                                      TInt aError);
protected:
    IMPORT_C CSession2();
    IMPORT_C virtual void Disconnect(const RMessage2& aMessage);
    IMPORT_C virtual TInt Extension_(TUint aExtensionId, TAny* a0,
                                    TAny* a1);
    ...
private:
    TInt iResourceCountMark;
    TDb1QueLink iLink;
    const CServer2* iServer;
    TAny* iSpare;
};
```

For the attachment to go through and the session to be set up, the server receives a connection message. Consequently, session establishment in EKA2 is, in effect, a two-stage process, allowing for asynchronous operation (should the client choose to do so).

On session connection, the `CServer2`-derived server object calls `NewSessionL()` to create a new `CSession2`-derived object that represents the server-end of the session and to which future messages are routed.

`CServer2` is an active object whose `RunL()` method delivers `RMessage2` message references to the corresponding `CSession2::ServiceL()` methods.

Server Start-up

As mentioned earlier, a server needs a mechanism to receive connections and messages before a client can establish a session with it. Since the only route to communicate with a server is via the kernel, the mechanism

is provided by an RServer2 proxy and its corresponding DServer object (which represents the server resource in the kernel). Among other things, the DServer object hosts the queue of messages intended for its associated server, which retrieves them through the RServer2 proxy.

```
class CServer2 : public CActive
{
public:
...
public:
    IMPORT_C virtual ~CServer2() =0;
    IMPORT_C TInt Start(const TDesC& aName);
    IMPORT_C void StartL(const TDesC& aName);
    IMPORT_C void ReStart();
    inline RServer2 Server() const { return iServer; }
protected:
    inline const RMessage2& Message() const;
    IMPORT_C CServer2(TInt aPriority,
        TServerType aType=EUnsharableSessions);
    IMPORT_C void DoCancel();
    IMPORT_C void RunL();
    IMPORT_C TInt RunError(TInt aError);
    IMPORT_C virtual void DoConnect(const RMessage2& aMessage);
    IMPORT_C virtual TInt Extension_(TUint aExtensionId, TAny* a0,
        TAny* a1);
private:
    IMPORT_C virtual CSession2* NewSessionL(const TVersion& aVersion,
        const RMessage2& aMessage) const = 0;
    void Connect(const RMessage2& aMessage);
    void DoConnectL(const RMessage2& aMessage,
        CSession2* volatile& aSession);
...
    TInt iSessionType;
    RServer2 iServer;
    RMessage2 iMessage;
    TAny* iSpare;
    TDb1Que<CSession2> iSessionQ;
protected:
    TDb1QueIter<CSession2> iSessionIter;
private:
    void Disconnect(const RMessage2& aMessage);
    static void BadMessage(const RMessage2& aMessage);
    static void NotConnected(const RMessage2& aMessage);
    friend class CPolicyServer;
};
```

When developing a CServer2-derived class, you do not need to deal explicitly with the RServer2 kernel-object proxy. The framework builds this proxy during the start-up of a CServer2, which then uses it internally to receive messages from the DServer kernel object.

When a server starts, it needs to declare a name that must be used by all connect messages from clients. System servers use a unique name that starts with '!', so that clients (through the client API) can easily find them and connect to them. Other servers cannot spoof the names of

system servers because they need ProtServ platform security capabilities (see Chapter 9). Private servers should use a name that is private to the instance of the application that started them – for example, by including the application’s main thread name in the server name. This raises the question, how does a client work out the name of server? The answer is that the name is intimate between the client-side API and the server. Subsequently, the client-side API is implemented to use this name (i.e. by hard-coding or by inclusion of a mutual header). What is most important here is to make clear that you should implement the client-side of your server to hide this name from client-side users.

Using a Session

A session is the context within which a client and a server communicate. Clients package requests, which they then send over sessions to a server. This is achieved on the client side by means of the `Send()` and `SendReceive()` methods of `RSessionBase`. For example:

```
TInt RSomeSession::Write(TDes8& aDes, TInt aLength, TInt aMode)
{
    TIpArgs args(&aDes, aLength, aMode);
    return SendReceive(ERequestWrite, args);
}
```

A client’s request contains a function number (sometimes known as a request code) and, optionally, some associated data. The function number is an enumeration, such as `ERequestWrite`, whose value and meaning is known to both the client and the server. As we shall see later, the server uses this value to determine the request type and what function should be used to handle it.

If the client has any additional data to communicate, the data needs to be packaged in a `TIpArgs` structure, which may take up to four arbitrary arguments. The `TIpArgs` class has templated constructors that also store the type of each argument. This information is used by the Symbian OS microkernel to validate a server’s usage of the arguments in the various `Read()` and `Write()` methods of `RMessagePtr2`.

On the server side, the client’s request is delivered in a message to the corresponding session’s `ServiceL()` method. The implementation of this method is expected to extract the request’s function number from the message by means of a call to `RMessage2::Function()`.

Practically, except for connection and disconnection requests, a session’s `ServiceL()` method acts as the dispatcher of all a client’s requests to the parts of the server that can handle them. A typical implementation might look something like:

```
void CSomeSession::ServiceL(const RMessage2& aMessage)
{
```

```

...
switch(aMessage.Function())
{
    case CSomeServer::EDoSomething:
    {
        DoSomething(aMessage);
        aMessage.Complete(KErrNone);
        break;
    }
    case CSomeServer::EDoSomethingElse:
    {
        DoSomethingElseAndComplete(aMessage);
        break;
    }
    default:
        aMessage.Complete(KErrNotSupported);
}
...
}

```

As illustrated in this example, although `ServiceL()` is a potentially leaving function, the dispatch methods, such as `DoSomething()`, usually are not. Allowing these functions to leave makes completing the message much harder and generally needs the use of `TRAP` macros. Note also, that since a server needs to complete a request via the message handle, dispatch methods that perform the completion must keep track of the supplied message.

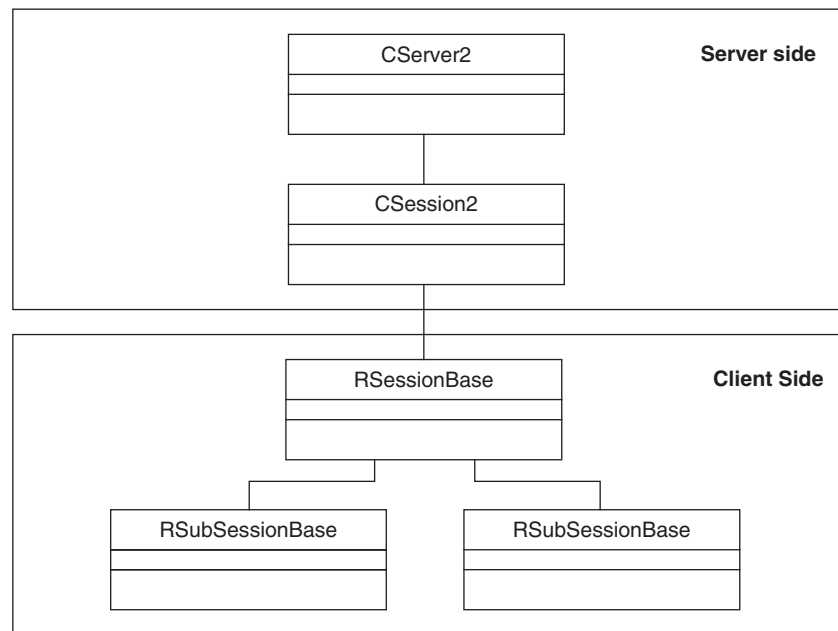


Figure 8.5 Client-server class diagram

It is important to note that once a message has been completed it may no longer be used in any way by the server. Any such attempt results in a KERN-EXEC 44 panic (bad message handle). To avoid this situation, implementations may occasionally need to check the value returned by `RMessagePtr2::IsNull()`. You also need to be aware that if you make a copy of the message, the copy does not have its state updated by a subsequent call to `Complete()`. For this reason, it is best to avoid making copies of `RMessage2` or `RMessagePtr2` objects and to use references in method calls,

If the message contains additional data, this can be read from the message, using one or more of the methods supplied by the `RMessage2` class and its parent, `RMessagePtr2`. For example, a server could access the content of the `aDes` descriptor as follows:

```
TInt length = aMessage.Int1(); // Get length from message param 1
TPtr8 buffer = MyNewBufferL(length);
    // Make a new buffer for the data
aMessage.ReadL(0,buffer);
    // Read data from client descriptor (parameter 0)
```

The code in this example could leave, so it might not be suitable for inclusion in the processing performed by `ServiceL()` or any of its dispatch functions, unless you protect it by means of a TRAP.

Cleaning up

When a client has finished with a client-server session, it should end the session with a call to `RSessionBase::Close()`. This sends a disconnection message to the server, which responds by calling the destructor of the `CSession2`-derived object to destroy its end of the session. This action also causes the kernel to destroy its own representation of the session and sets the handle of the client-side `RSessionBase` class to zero. It is safe for a client to call `Close()` on the `RSessionBase` again; in this case `Close()` does nothing.

If a client thread dies, the kernel performs thread-death cleanup. This involves sending a disconnection message to all the server sessions for which it was a client. Thus, the server-side of the session is safely destroyed, even if the client thread dies unexpectedly.

Servers must perform effective cleanup. When a `CSession2` object is destroyed, any resources associated with it should also be destroyed. Usually, this is just standard C++ destructor processing.

Servers should be written with the greatest care, so they do not terminate prematurely. However, if a server does die, then Symbian OS gives the server's clients the opportunity to recover. All outstanding messages from asynchronous requests are completed with the `KErrDied` return code.

On receiving a notification that a server has died, the client should clean up all `RSessionBase` objects relating to that server. Any `SendReceive()` call issued through an `RSessionBase` to a dead server results in a completion code of `KErrServerTerminated`. This is the case even if a new instance of the server is started: old sessions are not reconnected.

Sessions and Sub-sessions

A new session has a handle that is passed as a parameter in any message. A single client thread may have multiple sessions to any individual server – each session identified by a different handle. However, because sessions (`RSessionBase` objects) are relatively heavyweight in terms of their maintenance in the kernel, there is a mechanism for multiplexing many sub-sessions over a single session, as shown in Figure 8.5.

To create a sub-session, you call `RSubSessionBase::CreateSubSession()`. Use of sub-sessions is private between the sessions and the server; the kernel knows nothing about them. Thus sub-sessions form part of the client–server ‘protocol’ that exploits sessions. Information about sub-sessions is communicated to the server in the fourth IPC argument (which means that when using sub-sessions only the other three arguments are available for use by the client). An `RSubSessionBase` is simply a wrapper around the `RSessionBase` API, using a private `RSessionBase` member.

It is the responsibility and design decision of the client API whether sessions or sub-sessions are to be used. Usually the decision to use sub-sessions is evident, where the client API needs the client to create a master session proxy which then is used by other proxy objects. For example, the file server API has `RFs` as its session object and `RFile` as a file proxy, containing a sub-session that uses the `RFs` session object.

Before the `RFile` sub-session object can be used, it first needs to be initialized with an established `RFs` session. For example, the code of the `RFile::Open()` method looks like:

```
TInt RFile::Open(RFs& aFs, const TDesC& aName, TInt aFileMode)
{
    return (CreateSubSession(aFs, EFsFileOpen, TIpcArgs(&aName, aMode)));
}
```

8.3 Publish and Subscribe IPC

Publish and subscribe is a mechanism by which a process can define an item of data (known as a property) that can be read, or written to, by any number of processes. Neither the reader (the subscriber) nor the writer

(the publisher) needs to know the identity of any of the other processes that access the property.

Users of properties may publish or subscribe to them, using a single common API; kernel-side usage needs other APIs that we do not discuss here. There is one high-level abstraction that a developer implementing property usage should be aware of: `RProperty`.

A property is defined in terms of its category (which denotes a family of properties), a key (which indicates a particular property within a category) and a property type.

When defining a property, you may also define security policies for read and write operations on it. If you do, the property can only be set or read by a process that has been granted those security policies. Once it has been defined, a property's lifetime is not tied to the lifetime of the process (or thread) that defined it.

In order to get or set a property, a process needs to know its type, category and key. Provided all the above criteria are satisfied, any process may set a property.

To support programming patterns where either the publisher or a subscriber is able to define a property, it is not a programming error to attempt to access a property before it has been defined. In such a case, the attempt simply returns an error code.

Owning a Property

Although either a publisher or a subscriber may set a property, it is effectively 'owned' by the executable (not necessarily one process) that defines it.

The main effect of such ownership is that, once defined, the property can only be deleted by an instance of the executable that defined it. In order to make this possible, the category of a property must be the `SECUREID` of the defining process (the value is specified in the MMP file for the process). If an executable's process terminates after defining a property, another instance of the same executable may later delete it. This is why the `RProperty::Define()` and `RProperty::Delete()` methods are static.

```
class RProperty : public RHandleBase
{
public:
    ...
public:
    IMPORT_C static TInt Define(TUId aCategory, TInt aKey,
                               TInt aAttr, TInt aPreallocate=0);
    IMPORT_C static TInt Define(TUId aCategory, TInt aKey,
                               TInt aAttr, const TSecurityPolicy& aReadPolicy,
                                       const TSecurityPolicy& aWritePolicy,
                                       TInt aPreallocated=0);
    IMPORT_C static TInt Define(TUId aKey, TInt aAttr,
                               const TSecurityPolicy& aReadPolicy,
```

```

        const TSecurityPolicy& aWritePolicy,
                TInt aPreallocated=0);
IMPORT_C static TInt Delete(TUId aCategory, TUint aKey);
IMPORT_C static TInt Delete(TUint aKey);
IMPORT_C static TInt Get(TUId aCategory, TUint aKey, TInt& aValue);
IMPORT_C static TInt Get(TUId aCategory, TUint aKey, TDes8& aValue);
IMPORT_C static TInt Get(TUId aCategory, TUint aKey, TDes16& aValue);
IMPORT_C static TInt Set(TUId aCategory, TUint aKey, TInt aValue);
IMPORT_C static TInt Set(TUId aCategory, TUint aKey,
        const TDesC8& aValue);
IMPORT_C static TInt Set(TUId aCategory, TUint aKey,
        const TDesC16& aValue);
IMPORT_C TInt Attach(TUId aCategory, TUint aKey,
        TOwnerType aType = EOwnerProcess);

IMPORT_C void Subscribe(TRequestStatus& aRequest);
IMPORT_C void Cancel();

IMPORT_C TInt Get(TInt& aValue);
IMPORT_C TInt Get(TDes8& aValue);
IMPORT_C TInt Get(TDes16& aValue);
IMPORT_C TInt Set(TInt aValue);
IMPORT_C TInt Set(const TDesC8& aValue);
IMPORT_C TInt Set(const TDesC16& aValue);
};

```

The following code illustrates how to define and delete a property:

```

const TUId KMyPropertyCat = {0x10013456}; // SAME as UID3/SID in MMP

enum TMyPropertyKeys {EMyPropertyInteger};

TInt err = RProperty::Define(KMyPropertyCat, EMyPropertyInteger,
        RProperty::EInt);
...
TInt delErr = RProperty::Delete(KMyPropertyCat, EMyPropertyInteger);

```

Using Properties

Property read and write operations are atomic, therefore it is not possible for threads reading a property to get inconsistent data.

A property value may be retrieved by specifying its identity each time it needs to be retrieved:

```

TInt propInt=-1;
TInt err = RProperty::Get(KMyPropertyCat, EMyPropertyInteger, propInt);

```

Alternatively, it may be retrieved using a handle created by a call to `RProperty::Attach()`:

```

RProperty tmpProperty;
TInt propInt=-1;
TInt err = tmpProperty.Attach(KMyPropertyCat, EMyPropertyInteger);
...
TInt getErr = tmpProperty.get(propInt);

```


The handle is written into the `RProperty` instance that is used to make the call to `Attach()`. Such a reference is commonly used in order to subscribe to a property:

```
...
// using an active object that has a reference to an RProperty handle
iProperty.Subscribe(iStatus);
...
```

`Subscribe()` is an asynchronous function call (with a `TRequest-Status&` parameter) that requests a notification when the property is next set by a publisher. As in this example, it is normally called from an active object that stores the property's handle in an item (of type `RProperty`) in its member data.

When you subscribe in this way, the notification results in a call to the active object's `RunL()` which should read the property's value, using code of the form:

```
...
// later on, in the active object's RunL()
User::LeaveIfError(iProperty.Get(iPropResult));
...
```

The function may then re-subscribe if you need another notification when the property is next set.

As you would expect for any asynchronous service that is accessed from an active object:

- subscription results in a single notification of when a property is next set
- subscribing to a property for a second time without an intervening `Get()` call is a programming error that panics the subscriber
- when you no longer need to be notified of changes to the property, you must cancel any outstanding subscription.

As with most R-classes, you must call `RProperty::Close()` when you are done with an `RProperty` instance.

Determinism

The publish and subscribe mechanism is designed to allow its behavior to be deterministic (that is, with bounded execution times and no possibility of out-of-memory errors). However, in order to achieve deterministic behavior, you have to ensure that your usage conforms with the following conditions:

- when defining a property, always call `Define()` with its `aPreallocate` parameter set to a value of 1

- always attach to properties (this ensures that the associated kernel objects are pre-allocated and makes all operations time-bound)
- if a property stores its data in a byte array, always use an array whose size does not exceed `KMaxPropertySize`
- make sure that the capabilities of a process match the security policies of the property.

8.4 Message Queue IPC

Message queues are sized at the time of creation and provide a convenient way to send and receive messages asynchronously without defining the identity of the recipient or even knowing of its existence.

Operations on message queues are deterministic but are effectively used in a ‘fire and forget’ manner. The main user-side message queue abstractions that are of interest to a developer are: `RMsgQueue` and `RMsgQueueBase`.

A message queue is essentially a block of memory in the Symbian OS microkernel and each message is effectively a slot in that memory, with a size of up to `KMaxLength` (currently 256) bytes. When a message is delivered to a thread it is copied from its slot in the queue into the process space of the receiving thread.

Creating a Message Queue

You must create a message queue before it can be opened by participating threads that wish to read from, or write to, it. A message queue is created with a fixed number of message slots, and a fixed number of bytes for each message. Note that the message size must be a (nonzero) multiple of 4 bytes and must not exceed `KMaxLength`, otherwise the creation operation panics the client.

You can create global or local message queues. A global message queue can be named and is publicly visible from other threads:

```
RMsgQueueBase tmpQ;
TInt err = tmpQ.CreateGlobal(KMyQueueNameLit, KMyQueueSlots, KSlotSize);
```

It can also be nameless (but still accessible by other processes):

```
RMsgQueueBase tmpQ;
TInt err = tmpQ.CreateGlobal(KNullDesC, KMyQueueSlots, KSlotSize);
```

A local message queue is only visible to the current process:

```
RMsgQueueBase tmpQ;
TInt err = tmpQ.CreateLocal(KMyQueueSlots, KSlotSize);
```

All the above operations open a handle (the `RMsgQueueBase` instance) to the created message queue.

You can also create a message queue in the ways described above, but using the templated `RMsgQueue` class instead of `RMsgQueueBase`. `RMsgQueue`'s template parameter defines the message type for the queue. Using `RMsgQueue` is the preferred option since it adds type-safety, as well as enforcing the limit on the size of a message. The following code snippet illustrates the use of `RMsgQueue` to create a global, named message queue.

```
Class TMyMsg
{
public:
    TInt iFirst;
    TInt iSecond;
}
...
RMsgQueue<TMyMsg> tmpQ;
TInt err = tmpQ.CreateGlobal(KMyQueueNameLit, KMyQueueSlots, KSlotSize);
```

Using a Message Queue

You need a handle to a message queue before you can read from, or write to, it. In the case of a global, named message queue, any process can open it by name:

```
TInt openErr = iQueue.OpenGlobal(KMyQueueNameLit);
```

To use the other types of message queue, you need to obtain a handle from the process or thread that created the queue.

Once you have a handle for a queue, preferably as an `RMsgQueue` instance, then you can send (or receive) messages, for example:

```
TInt sendErr = iQueue.Send(iSmallMsgType);
if (KErrOverflow == sendErr)
{
    DecideWhatToDoWhenQueueIsFullL();
}
```

If you do not wish to handle the case where the queue may be full for sending (or empty for receiving) a message, then you could use the blocking family of calls, such as:

```
iQueue.SendBlocking(iSmallMsgType);
```

Bear in mind that this type of call cannot be canceled and no other work can be performed by the calling thread until the call returns. If this would have an impact on your thread's concurrency requirements,

you should use an active object to make calls to the message queue's asynchronous functions `NotifyDataAvailable()` (for receiving messages) and `NotifySpaceAvailable()` (for sending messages), and perform the appropriate `Receive()` or `Send()` in the active object's `RunL()`.

8.5 Which IPC Mechanism Should You Use?

Client–server

When using the client–server mechanism, the relationship between the participants is clear: the client has to initiate requests that the server has to service. Moreover, the server may receive such requests from many clients, in which case requests are serialized and completed one at a time. There are no deterministic guarantees in this scenario and a ‘connection’ has to be set up for every communication channel.

This mechanism is ideal for the case where some kind of resource, with inherently serial access, is offered to many clients simultaneously. Thus the server’s responsibilities are to mediate access to resources through pre-defined service interfaces.

Publish and Subscribe

Publish and subscribe is probably the most important new IPC mechanism and the one with the most impact. It was created to solve the problem of asynchronous multicast event notification and to allow for connectionless communication between threads.

With this new API, interested parties do not need to know who the event provider is. This design is essential in avoiding having many components knowing about many other components and having to connect to them, just to discover and consume events. Thus this API breaks the need to know about, and link to, many client APIs; participants only need to know about the P&S API and the properties they are interested in.

Producers and consumers of events can dynamically join and leave the ‘conversation’ without any prior commitment or connection setup. This paradigm lends itself perfectly to agent-type scenarios, where both asynchronicity and autonomy are essential. Architecturally, for developers, this is very significant, since their components can be designed and deployed without any prior knowledge of who their consumers or producers may be; nor need they supply interfaces to them other than the specification of the properties in question. Effectively, publishers and subscribers are completely shielded from each other. This is the reason why multicasting is so easy and comes for free.

We refer to the publication of events in Publish and subscribe as ‘multicasting’ instead of ‘broadcasting’ because it does not have to

pollute the communication channels of every thread unless we wish it to. Publish and subscribe is the best paradigm to multicast events to interested parties without the need to know explicitly who, where or how these parties receive these events and without having to involve them if they are not interested.

Accounting and resource management is handled by the kernel and thus the complexity of usage is kept really low. Consequently, although delivery to and from the kernel in this mode of communication is guaranteed and deterministic, messages are not guaranteed in terms of their timely reception from remote parties. This is because remote parties may not be interested in the communication or may not even exist – and there is no pre-defined way to know (other than out-of band communication between the parties) of their interest or existence.

It is therefore advised that P&S should be used when a component needs to supply (or consume) timely but transient information to (or from) an unknown number and kind of interested parties, while maintaining maximum decoupling from them.

Typical scenarios would be the dissemination of battery status information across the system and the publication of the status of communication links. In general, whenever such contextual information is to be communicated across the system, P&S is the paradigm to use.

Message Queues

The Message Queue API allows designers to break the connection, while keeping the communication, between two or more threads. In that sense, two threads may communicate with one another without needing to set up connections with each other. In effect, this breaks the synchronicity in the setting up and tearing down of connections as well as removing the need for the rendezvous of these threads for such setup to take place.

Breaking such connection synchronicity has many benefits for scenarios where, otherwise, deadlocks could arise in using the client–server paradigm. Such deadlocks manifest themselves during cancellation or setup of sessions between clients to servers which are themselves clients to other servers, when not properly coded.

This form of IPC is less alien to developers porting from other operating systems to Symbian OS, where similar mechanisms (such as mailboxes) exist.

As outlined above, a queue allows for many senders to communicate messages. The ultimate recipient of messages is not known in a global queue, since any thread may read from it. Consequently, delivery to the final recipient is not guaranteed. For such a guarantee to be possible, participants have to agree on an implicit protocol where a queue may be accessed from only one reader. This is Symbian's recommendation for this IPC for almost all cases. It has to be noted that messages are sent to

queues and not to final recipient threads. Therefore, messages don't have any header payload stating their final destination.

Symbian OS message queues are not pipes. To emulate pipes in Symbian OS using the Message Queue API, one has to build a protocol and abstractions on top of the API. Since messages are anonymous and don't have delivery addresses, queues may be used as half-duplex pipes. It is down to the communicating threads to define this protocol between them and to set up the two queues necessary for peer-to-peer communication between them.

When this is done, two threads may asynchronously communicate with each other without necessarily linking to each other or to client APIs other than the Message Queue API. This is beneficial since (components and) threads become replaceable at run time to be either senders or receivers of messages from a queue. Thus high-availability services can be offered, and be serviced at run time, without disrupting their clients.

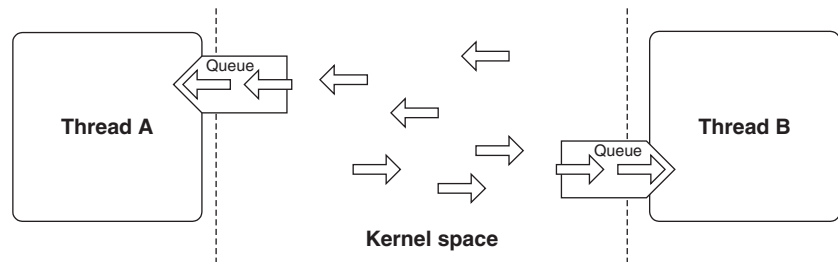


Figure 8.6 Message queues (used as half-duplex pipes)

P&S is excellent for notifications of state changes that are inherently transient but queues are good for communicating information that transcends the sender's state or lifetime. For example, a logging subsystem could utilize a queue to receive messages from many threads that may or may not be running at the point where the messages are read and processed. In that sense, a service provider in this paradigm (as opposed to client-server) is not the one which mediates (and disseminates) data from the resource, but is the one which consumes the data.

Summary

This chapter has examined the interprocess communication (IPC) mechanisms available to applications and servers in Symbian OS:

- client-server IPC
- publish and subscribe IPC
- message queue IPC.

9

Platform Security and Publishing Applications

This chapter deals with preparing your application for distribution. We discuss information related to building an application for, and running it on, a target machine, including giving your application an icon and building a distributable SIS file from a PKG file. There is also an introduction to platform security, so that you can successfully prepare your application for the Symbian Signed program.

9.1 Releasing an Application

In order to release your application, some important steps must be taken. These steps do not require you to add more C++ code, but are considered critical to ensuring the success of an application in the mass market. The extra work involved focuses on things such as preparing an icon for the application, building it as an installable application and authorizing it under the Symbian Signed program. The last step is especially important with the introduction of platform security in Symbian OS v9.

Symbian OS v9 is a major evolution. It is specifically geared to target mid-range phones in tens and hundreds of millions of units – offering even larger opportunities for developers to market their products. As such, it includes a significant number of new features and improvements to enable key emerging technologies required by advanced, open mobile phones for years to come. In order to provide comprehensive support for new functionality such as Digital Rights Management, Device Management and Enterprise-grade data handling, large changes have been made to the

very core of Symbian OS to support vital concepts such as data protection or ‘caging’ and restricting some API usage.

9.2 How Does Platform Security Work?

The platform security enhancements represent an evolution of the perimeter security model of Symbian OS v8 and help ensure the stability of the platform. They provide greater protection against malicious or badly implemented applications, by operating at the software level to:

- detect unauthorized access attempts to hardware, software or data which may lock up the phone or affect other software or the network
- prevent applications from acting in unacceptable ways, irrespective of whether these actions are intentional (i.e. malware) or unintentional. Such actions might include unauthorized access to hardware, attempts to read or write restricted data, etc.

This section gives you an introduction to the way that platform security is implemented, with a particular focus on ensuring that you know how to update your applications to conform to and work within the platform-security-enabled Symbian OS environment. Knowing how to do this will allow you to successfully go through the Symbian Signed process and release your application.

Capabilities

Platform Security provides protection by ensuring that only trusted applications can use certain functionality within Symbian OS. It introduces the idea of *capabilities*, which protect APIs within Symbian OS.

Not all of the APIs in the Symbian OS are capability-protected – in fact, only about 40 % of Symbian OS APIs fall within this category; all other APIs still have unrestricted access. Of those APIs that are protected, many are used to access the low-level aspects on Symbian OS, so it is unlikely that you will need to use them.

If your application needs to use any of the APIs that are protected by a capability, it must request use of the capability. This is called *authorizing* your code. Once the authorization is obtained, the application is *authorized* and can use the protected functionality.

The Symbian Signed programme is your route to authorizing and releasing your applications before the user receives them so we’ll take a quick look at this now, with more detail in Section 9.5.

Authorizing via Symbian Signed

There are a variety of capabilities (see Section 9.7) that are used to protect functionality within Symbian OS. To ease the process of authorizing applications, Symbian Signed groups them together into two sets for verification – basic and extended.

- The basic capabilities protect the APIs that: access confidential user data; provide data about the user's environment; and access network services (both locally and remotely). Being authorized for the basic capabilities ensures that users of your application are authorized for access to services and to data (for example, to send text messages, make Bluetooth connections to other phones and access user-confidential data such as contact information) without the user being asked explicitly for permission.
- The extended capabilities protect the APIs that provide lower-level access to system functions, such as changing the network configuration, in addition to the basic capabilities. Applications requiring authorization for the extended set via Symbian Signed have to undergo additional testing as these APIs require what is known as 'phone-manufacturer approval'.

Within the extended set are some particularly sensitive capabilities, designated 'Phone-Manufacturer Approved', which include DRM, DiskAccess, NetworkControl and AllFiles. These capabilities control access to areas of the phone or content on the device that manufacturers may want to restrict, and therefore Symbian Signed has additional criteria to be satisfied if your application needs any of these capabilities.

If you decide to follow the Symbian Signed route, Section 9.5 contains more details of what you need to do.

Authorizing without using Symbian Signed

Sometimes you don't actually need to authorize your code in advance to use the capability-protected functionality.

The Symbian Signed process does a good job in protecting the phone from errant applications and providing a number of routes to market, but there are times when getting an application signed may not be appropriate. In this situation, you can take advantage of the Unsigned-Sandboxed capabilities set. The unsigned-sandboxed capabilities consist of all functionality that doesn't require any capabilities and all capability-dependent functionality for which access may be granted by the user.

If the application is not signed, Symbian OS can ask the phone user directly when the application is installed if they are prepared to authorize the unsigned-sandboxed capabilities that the application declares in its

project definition (MMP) file (this is called a *blanket grant*). An unsigned-sandboxed application only has access to a subset of APIs. If your application only needs to access functionality in this set there is no need to submit it to Symbian Signed. That said, a phone manufacturer may still require you to ‘sign’ the application with a certificate that you can create using tools in the SDK.

If you prefer your application to be authorized and that the phone 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.

One-shot Grants

It’s not possible for users to blanket-grant all capabilities – only a Symbian Signed application can have this authorization. Instead, if you need limited use of a particular capability (perhaps to send an SMS), you can use selected APIs on a ‘one-shot grant’. To achieve this, because the access is being granted as a one-off, special case, you mustn’t request these capabilities in your project definition file like other capabilities (otherwise installation will fail as they cannot be authorized by the user) – just ensure they are excluded.

If you subsequently want your application to be Symbian Signed, with capabilities you require being given a blanket grant, you only need to add the required capability to your project definition file and your application code can remain unchanged.

9.3 How Do I Support Platform Security?

In order to support the platform security architecture, a number of APIs were added, amended or deleted in Symbian OS v9. Information about these changes is available in the SDKs. If any of the APIs that you use in applications running under versions of Symbian OS before v9 are affected, you need to make the appropriate changes.

Project Definition Files

The capabilities that an application needs are defined at build time in the project’s definition file. The `CAPABILITY` keyword is followed by a list of the capabilities that the executable requires.

```
TARGET      example.dll
CAPABILITY  LocalServices ReadUserData
```

A list of capabilities can be found in Section 9.7.

You can work out what capabilities need to be specified by checking the documentation – this shows which API calls are protected by which capabilities – and specifying all these capabilities in the MMP file. (You can also use the emulator to see if you’ve omitted any required capabilities, see Chapter 10.)

Instead of specifying capabilities in a list, the special capability name `NONE` can be used. This is the default value if the `CAPABILITY` keyword is not found in the MMP file.

There are a number of APIs whose capabilities do not necessarily have to be specified in the MMP file. If the relevant capability is not specified, authorization may still be granted on a one-shot basis. (This applies to both signed and unsigned-sandboxed applications, as discussed in Section 9.2.)

Secure IDs

In Symbian OS v9, each executable has a Secure ID (SID) which uniquely identifies it. SIDs are a specialized use of the `UID3` value, which is specified in your application’s MMP file with the keyword `UID`. The SID value, unless explicitly specified otherwise by the `SECUREID` keyword in the MMP file, defaults to the same value as the application’s `UID3`. To avoid confusion, a `SECUREID` should not be specified in the application’s MMP file. The SID value is not relevant for DLLs since the SID of a process is always that of its EXE.

A SID can be requested in the form of a UID from the Symbian Signed portal, **www.symbiansigned.com**. This is an automated process and the UIDs are issued instantly. Note that the UIDs are divided into two ranges and only values from the protected range are valid SIDs. UIDs from the unprotected range are often used during development and testing. The Symbian OS software installation system checks that a file is using a UID from the protected range. If not, your application won’t install.

Data Caging

Another feature of platform security is data caging. This means that each application, when installed, has its own private directory on the phone that is accessible only by a process with either the same UID as the application or with certain privileged capabilities. For example, your application cannot access configuration files of another application that the user has installed.

Each data-caged directory resides under the `\private` directory on the phone and is named according to the SID of the main application:

```
\private\<application SID>\...
```

The private directory can be created on any writable drive, however only certain drives can be described as secure. A secure drive must have media that is not removable and cannot be accessed by an external 'host' computer using a mass-storage file system connected via USB. For drives that do not satisfy these criteria, the data stored in the secure areas can easily be accessed by the computer system connected to them.

Other directories in the file system are also restricted by Symbian OS, regardless of drive:

- `\resource\...` contains resource files for installed applications. Resource files for an application can only be written to this directory at installation time by the Symbian OS software installer. No capabilities are required to read the resource files stored in this directory.
- `\sys\...` contains system files and installed binaries. Again these binaries can be written to only by the software installer. If an application is installed on media that can be accessed by an external computer then the binary files are protected by a hash checksum. Each time the binaries are loaded, the hash is checked to ensure the binary has not been modified in any way. If tampered with, a binary is not loaded.

Other areas of the file system, from the root directory upwards, are considered public and can be accessed with no capabilities.

9.4 Preparing an Application for Distribution

Having discussed Platform Security, we need to see what effect it has on distributing applications. To do this, we first show you how to build your application so that it can be installed on a phone and then go through the steps required to authorize your application via Symbian Signed.

Building the Application

Up to this point, we've done everything in the emulator. Now it is time to rebuild the application for ARM and install it onto a Symbian OS phone using the appropriate user interface.

Building the application for the target phone is simple. You can build it from the Carbide.c++ IDE, after selecting the ARM UREL target, or you can build it from the command line by typing:

```
abld build armv5 urel
```

This builds a releasable application, which is ready to be copied onto the phone itself. To install the application on to the phone, Symbian OS

provides a powerful, yet straightforward installation system that offers a simple and consistent user interface for installing applications, data or configuration information on to phones.

The basic idea is that end users install all the code, packaged in Symbian OS installation (SIS) files. These can be transferred to the phone from a PC using connectivity software, Bluetooth, infrared or even email and an Internet connection, as illustrated in Figure 9.1.

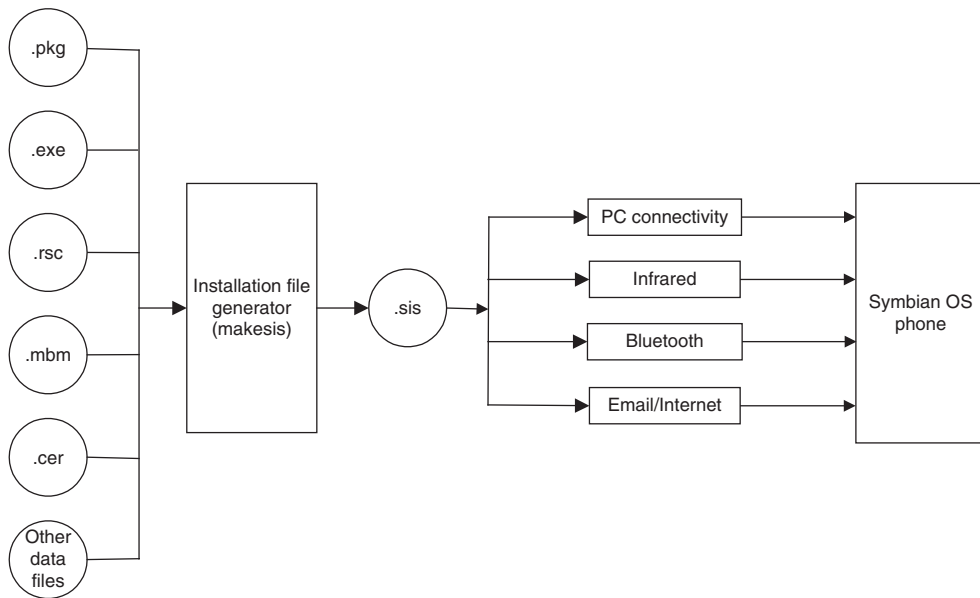


Figure 9.1 Installation methods

The SIS files are generally very small in size (at most a few hundred kilobytes) and are therefore very quick to transfer to the target phone. Three command-line tools are provided to enable you to create SIS files:

- the Installation File Generator, `makesis .exe`, creates the installation files from information specified in the package file
- the Certificate Generator, `makekeys .exe`, creates private–public key pairs and a certificate file; it is used by the Installation File Generator to digitally sign the installation files
- the Installation File Signer, `signsis .exe`, digitally signs the installation files with a Class 3 ID certificate such as an ACS Publisher ID issued by TC TrustCenter.

There are three steps involved in generating the SIS file.

1. Create a PKG package file that specifies all the files that go into the SIS file, and where they are to be installed on the target phone.

2. Run the `makesis` command line tool to generate the SIS file.
3. Sign the application using `signsis.exe`. If the application is to be submitted to Symbian Signed, you sign it with an ACS Publisher ID. If the application does not need to be Symbian Signed, certain phone manufacturers may require you to sign it with a user-generated certificate.

Producing the Package File

Before you create the SIS file, you need to specify information about the application in a package file. This is a plain text file but, because of its flexibility, the format is one of the most obscure of the Symbian OS tool control files.

For our example, we start with the `OandX.pkg` file from the Noughts and Crosses application that is described in Chapter 12. Bear in mind that the resulting SIS file cannot be installed on any phone based on Symbian OS v9, unless it is signed. Here is the content of `OandX.pkg` for an S60 phone:

```
; OandX.pkg
; Language - standard language definitions
&EN
; Vendor
:"Symbian Software Ltd."
%{"Symbian Software Ltd."}
; standard SIS file header
#{"OandX (S60)", (0xE04E4143), 1, 0, 0, TYPE=SA
; Supports S60 v3.0
[0x101F7961], 0, 0, 0, {"Series603rdEditionProductID"}
"\epoc32\data\z\resource\apps\OandX.rsc"-
"!:\resource\apps\OandX.rsc"
"\epoc32\data\z\resource\apps\OandX_loc.rsc"-
"!:\resource\apps\OandX_loc.rsc"
"\epoc32\data\z\resource\apps\OandX.mbm"-
"!:\resource\apps\OandX.mbm"
"\epoc32\data\z\private\10003a3f\import\apps\OandX_reg.rsc"-
"!:\private\10003a3f\import\apps\OandX_reg.rsc"
"\epoc32\release\armv5\urel\OandX.exe"-"!:\sys\bin\OandX.exe"
```

The body of this file should be obvious enough: all source files that will be packaged into the SIS file are listed, along with information about where they should be unpacked upon installation. The `!` drive specifier means ‘the chosen installation drive’. If you require a file to reside on the `C:` drive of the phone, for instance, you can specify that instead, but this is rare, and should only be used if absolutely necessary.

The header is more interesting.

- The `&` line specifies the languages supported by this installation file: in this case, English only.

- The `:` and `%` lines specify the localized and non-localized vendor name.
- The `#` line specifies the application's caption to be used at install time, its final SID and its three-part version number – 1,0,0 is specified here, which will be displayed as 1.0(000), indicating major, minor and build.

This is the general form of a PKG file for a basic, single-language application. In fact, the PKG file format supports more options than this, including nested packages, multi-language installation files and required dependencies. All these are documented in the SDK.

One thing to note is that, because Symbian OS is so flexible and allows many different user interfaces to be developed, you should always ensure that your SIS file can only be installed on the user interface for which it was designed. If you don't, you – or, more importantly, your end-users – may experience problems.

To enable you to do this, each product is assigned a unique ID which is used by the application installation mechanism to ensure that only compatible applications can be installed on the phone. Also, each platform version is assigned a unique ID. It is assumed that a platform version is compatible with all earlier platform versions, but not with any later versions. You can specify the platform information by including in the package file a line of the following form:

```
[0x101F6300], 1, 0, 0, {"UIQ30ProductID"}
```

where:

- `[0x101F6300]` is the product and platform version SID
- `1, 0, 0` is the major version, minor version and build numbers of the platform (not the application)
- `{"UIQ30ProductID"}` is a feature-identification string.

Not all products support this mechanism: for those that do, you can find exact details of the SIDs, feature-identification strings and version numbers to use in the appropriate product-specific SDK. On products that support this mechanism, the installation software refuses to install any SIS file that does not contain the correct platform information.

Generating the Final SIS file

When all the source files and application information have been specified in the PKG file, you can create an installable `OandX.sis` file in your current directory by issuing the command:


```
makesis OandX.pkg OandX.sis
```

You then need to sign the SIS file:

```
signsis OandX.sis OandX_Signed.sis myCert.cer myCert.key password
```

where:

- `OandX_Signed.sis` is the filename for the signed version of the application
- `myCert.cer` is the certificate filename
- `myCert.key` is the certificate's private key
- `password` is the password of the private key.

The SIS file is now ready. If you need to have the application Symbian Signed, read Section 9.5, otherwise go to Section 9.6 to see how to install your application on a mobile phone.

9.5 Overview of Symbian Signed

Symbian Signed was introduced against a background in which, although open phones offer revenue opportunities, there are also associated threats such as fear of viruses, billing storms (where the user is billed illegally or without their knowledge), etc. With an open environment comes responsibility and Symbian Signed is intended to support that responsibility, hence the need for a testing and certification programme that ensures that you know the source and provenance of an application. It also means that the application has been tested to meet certain industry-agreed minimum criteria. Applications that are signed must then be tamper-proof in order to be trusted.

Symbian Signed is a foundation programme on which other licensees, network operators and partners can build if they have a specific need. As a result there are mechanisms to support additional tests where required, for example where:

- access to prototype devices is needed
- the application is going to be a signature application shipped with phones or has operator requirements it must fulfill
- the application makes extensive use of the network and so the network impact of the application needs to be assessed.

To date, over 18 000 SIS files have been signed, with more and more network operators around the world insisting that applications are Symbian Signed before they even consider selling and distributing them via their portals.

Test Criteria

In order to get your application Symbian Signed, it must pass the Symbian Signed test criteria (available at www.symbiansigned.com). We recommend that you make these tests part of your usual development lifecycle for your application to minimize the chances of failure when it comes to testing against the Symbian Signed test criteria.

The test criteria are split into six categories as outlined in Figure 9.2.

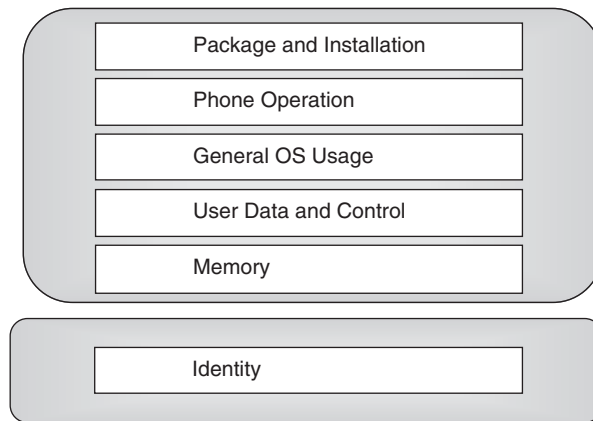


Figure 9.2 Symbian Signed test criteria categories

The Identity category refers to the ACS Publisher ID that you can obtain from TC TrustCenter. The other five categories focus on functionality aspects of the application. For example, Memory tests ensure that your application starts and closes gracefully in low-memory situations. Phone Operation tests ensure that your application does not stop the phone from undertaking its primary function – making phone calls. Various other tests such as installation and uninstallation, verification of the use of correct SIDs and so on are also part of the test criteria. These all combine to help make the use of Symbian OS devices a pleasant experience.

To get your application Symbian Signed, you develop the application, test it and send it to a test house. If it passes the test criteria, your application is Symbian Signed (see Figure 9.3).

Signed or Unsigned?

It is optional for developers to put applications written for versions of Symbian OS before v9 through the Symbian Signed process. The benefits

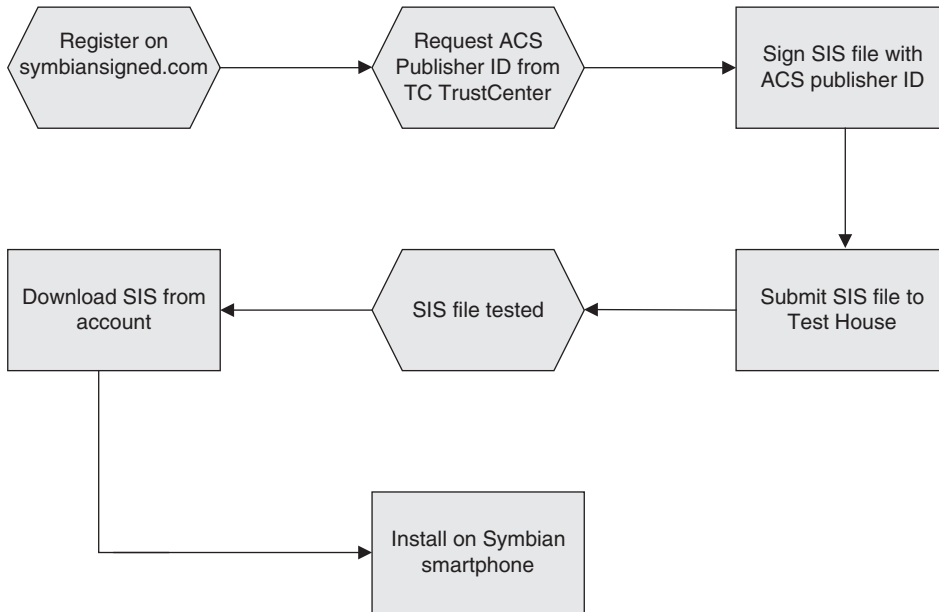


Figure 9.3 Application-signing process

of doing so are driven by market demand. Application robustness, quality and identification of the origin of the application are the primary drivers of this demand. Due to this, some phone manufacturers now insist that all applications delivered in a SIS file are Symbian Signed prior to distribution via their channels. Many network operators are also increasingly aware of the importance of testing and signing applications and require applications to be Symbian Signed prior to distribution via their portals.

If you are developing for Symbian OS v9, Symbian Signed becomes even more significant. This is because, if an application you are developing makes use of capabilities, the application must be Symbian Signed. If it is not Symbian Signed, a phone will not install the application. If an application doesn't use capabilities then it will install without signing – simple! Well actually, not quite that simple.

Some Symbian OS v9 phones may allow unsigned applications that use certain capabilities to be installed with no problems. An unsigned application, however, behaves slightly differently to a signed application when used on the phone. Let's say, for example, you create an application that makes use of the UserEnvironment capability (i.e., it uses Bluetooth APIs) and you do not have to get it signed as you know it will install on the phone you are developing for. When using this unsigned application on the phone, any functionality that utilizes Bluetooth results in a dialog box appearing on the phone's screen asking the user whether to allow the

application to use the Bluetooth functionality. This dialog appears every time the functionality is needed. A Symbian Signed application does not display these dialog boxes at all.

A further consideration is that some phone manufacturers may require these applications to be signed by the developer, sometimes called ‘self-signed’. This means that a signing must take place using a certificate that you have developed using tools on the SDK (i.e. `makekeys.exe`) since, without this, the application will not install. It doesn’t require visiting the Symbian Signed website or obtaining an ACS Publisher ID, so it is a small task in relation to the overall development effort, but it is something to bear in mind.

These policies are defined by the mobile phone manufacturers so it is recommended that you check with them first. For the sake of simplicity in this book, we have assumed that all applications you are developing for Symbian OS v9 phones will be Symbian Signed.

However, there’s just one small catch – an application won’t install on a phone until you’ve had it Symbian Signed, so how do you test it on the phone before submitting it for testing and signing? The solution is to use Developer Certificates.

Developer Certificates

Developer Certificates allow controlled capability access to specific phones. By controlled access we mean that the range of phones that can be used is limited to specific phones according to their IMEI and capabilities access is restricted to only the ones that are needed by the application. The level of access in terms of the number of IMEIs and capabilities required is determined according to identity of the developer (i.e. who you are) and whether the capability requires phone-manufacturer approval.

If you are happy to test your application on just one phone and it only uses the capabilities that are ticked in the first column of Table 9.1, then there is no need for an ACS Publisher ID. If you need to debug on more than one phone then you need an ACS Publisher ID. If you need to debug your application on more than 100 phones and/or you need any of the seven remaining capabilities, then you will need both an ACS Publisher ID and phone-manufacturer approval.

An ACS Publisher ID is a class 3 certificate, obtainable from organizations such as TC TrustCenter and costing US\$200 per annum at the time of writing. It verifies that your organization exists and is bona fide.

All Developer Certificates are issued via the Symbian Signed web portal, once you have registered. If you need a developer certificate that requires phone-manufacturer approval, you simply need to follow the appropriate links on the site.

To request a developer certificate, you use a tool called DevCertRequest which can be downloaded from the Symbian Signed portal. You

Table 9.1 Developer certificate requirements

Number of IMEIs	1	1–100	Phone Manufacturer Approved
Identity Requirements	- Registration - IMEI	- Registration - IMEIs - ACS Publisher ID	- Registration - IMEIs - ACS Publisher ID - Phone Manufacturer Approved
Capability			
Local Services	✓	✓	✓
Location	✓	✓	✓
Network Services	✓	✓	✓
Power Management	✓	✓	✓
ProtServ	✓	✓	✓
Read User Data	✓	✓	✓
Surroundings DD	✓	✓	✓
SW Event	✓	✓	✓
User Environment	✓	✓	✓
Write User Data	✓	✓	✓
Read Device Data	✓	✓	✓
Trusted UI	✓	✓	✓
Write Device Data	✓	✓	✓
All Files			✓
CommDD			✓
Disk Admin			✓
DRM			✓
MultimediaDD			✓

Table 9.1 *(continued)*

Number of IMEIs	1	1–100	Phone Manufacturer Approved
Identity Requirements	- Registration - IMEI	- Registration - IMEIs - ACS Publisher ID	- Registration - IMEIs - ACS Publisher ID - Phone Manufacturer Approved
Capability			
Network Control			✓
TCB			✓

use this tool to enter the ACS Publisher ID, IMEI and capability information and create a Certificate Signing Request (CSR) file. This file is then uploaded to the Symbian Signed portal and the information contained in it is used to generate a developer certificate (CER).

The Signing Process

Now you have developed your application for phones based on Symbian OS v9, you must get it signed. This section describes how.

Preparation

You have written and tested your application successfully on the Symbian OS emulator. You have tested it on a Symbian OS v9 phone after signing it with a developer certificate and you are now confident that not only has it passed your own strenuous black-box and white-box testing, it has also passed all the Symbian Signed test criteria. What's the next stage?

ACS Publisher ID signing

Actually there isn't that much more left to do other than sign your application with your ACS Publisher ID and submit it to the website.

Run the `signsis.exe` tool to sign your SIS file, specifying your public ACS Publisher ID certificate, private key and password:

```
signsis OandX.sis OandX_Signed.sis myACS.cer myACS.key password
```

You should now have a SIS file that is signed and ready to be submitted for testing. You are almost there . . .

Submission

To submit the SIS file, visit the Symbian Signed website at **www.symbian-signed.com** and log in using the username and password you created when you first registered. Choose the Submit option and follow the steps to select the Test House and enter the required information (user details and application details). Finally submit the application and its supporting materials.

The following files are required to be included in the ZIP file you submit.

- the SIS file (signed with your ACS Publisher ID) to be tested by the test house
- the PKG file used to create the SIS file; this will be cross-referenced by the Test House to ensure correct target platform and specification
- a completed `Readme.txt` file, which should include any release notes and quick advice on how to use the application. Depending on the nature of the application, you might choose to supplement – or replace – this file with a user guide in PDF format.

After you have submitted your application, it is sent to the test house you selected. The test house verifies the validity of the ACS Publisher ID and the signature of the SIS file. If verification is successful, the test house examines your application and sends you a quote for the cost of the test run. You receive notification emails throughout this process. If you wish to query the quote you receive, you should contact the test house directly – additional details can be found on the website.

Once your application has successfully passed all of the tests conducted by the test house, the test house uploads your application to the Certificate Authority. The Certificate Authority removes the ACS Publisher ID, stores details of the application in its revocation database, re-signs the application against the Symbian root certificate and sends the signed application back to the test house. The test house then informs you that you can download your Symbian Signed application from the site, along with the test report.

Your application is ready for distribution and can display the Symbian Signed logo as shown in Figure 9.4.



Figure 9.4 Symbian Signed logo

9.5 Installing a SIS File

Once you have a working SIS file, it is easy to install it onto a Symbian OS v9 phone: all you need do is transfer it to the phone and launch the SIS file!

There are several transfer methods available – choose what is supported on your PC or phone and what is most convenient for you at the time:

- serial or USB connection – install the PC Connectivity software accompanying your phone and send files through a serial or a USB cable
- infrared – you need to enable infrared on both your PC and the phone and to run software that allows you to send files between two infrared ports; this is normally part of the PC Connectivity software accompanying your phone
- Bluetooth – you need to enable Bluetooth on both your PC and your phone
- email – you can send the SIS file to your phone as an email attachment if you activate the email account on your phone; as you will use the mobile phone network to send the file, you may incur costs
- Internet – you can download a SIS file straight from a website; you may incur costs when using the mobile phone network to connect to the website.

Once you have transferred the SIS file, it should appear on the phone as a 'received' file – simply select the icon to launch the file and begin the installation process. The program is then visible from the Application Launcher.

One of the benefits of installing applications using the SIS file format, rather than by copying files directly, is that you get an entry in this listing. This means that you can use it to remove the application; just tap on

Uninstall and select your application from the list of uninstallable programs. Then press Uninstall again to confirm. Go back to the Application Launcher and see that it is gone.

9.6 List of Capabilities

The unsigned-sandboxed and basic capabilities are:

- `LocalServices`, which grants access to local network services that usually do not incur a cost, such as Bluetooth
- `NetworkServices`, which grants access to remote network services that may incur a cost, such as text messages; this capability is granted on a one-shot basis if not previously authorized
- `UserEnvironment`, which grants access to live confidential information about the user and their immediate environment (e.g., audio, video or biometric data)
- `ReadUserData`, which grants read access to data that is confidential to the phone user (subject to confirmation)
- `WriteUserData`, which grants write access to data that is confidential to the phone user (subject to confirmation)

The extended capabilities essentially control access to services and data. They are defined at a general level, so that a server can present them to the end-user of a phone when installing an application, allowing the end-user some control over what an application may do.

- `CommDD` grants access to communications device drivers.
- `MultimediaDD` grants access to multimedia device drivers.
- `PowerMgmt` grants the right to power off unused peripherals, switch the phone into and out of standby state, and power the phone down.
- `ReadDeviceData` grants read access to confidential phone settings or data.
- `WriteDeviceData` grants write access to confidential settings that control the phone's behavior.
- `DiskAdmin` grants access to specific disk-administration operations.
- `NetworkControl` grants access or modification rights to network protocol controls.

Summary

In this chapter, we've looked at:

- key concepts in Platform Security
- authorizing applications
- implementing Platform Security in application and project files
- how to build an application for delivering to a phone
- how to get an application Symbian Signed.

10

Debugging and the Emulator

Debugging is an indispensable step in the software development cycle. It is used to find out why an application is not behaving as expected and to correct the behavior. To help with debugging, all Symbian OS SDKs come with an emulator, which allows you to see, on your PC, what will happen when you run the code on a target device – a mobile phone.

The emulator is designed to be the primary development and debugging tool for Symbian OS, alongside any language-specific tools provided by the development environment.

For most development purposes, you can write the same source code to run on both the emulator and Symbian OS phones almost 100 % of the time. The basic emulator is delivered by Symbian, but is then customized by the UI vendors for S60 and UIQ.

This chapter looks at the emulator and the debugging facilities offered by Symbian OS on the emulator, to help you to create robust applications rapidly and efficiently.

10.1 Using the Emulator

Symbian OS phones use an ARM processor but PCs usually use an x86 processor, so C++ programs for the emulator are compiled to native x86 machine instructions and the emulator uses Win32 APIs to emulate machine hardware and Symbian OS services. Due to the hardware difference, there are some restrictions to bear in mind when testing with the emulator.

- The emulator cannot be used to test that code behaves the same on phones of different capabilities (e.g. S60 phones with different processor speeds or specifications), but can be used to test some

differences in behavior such as for different screen resolutions and orientations in later SDK versions.

- The speed at which the emulator runs is dependent on the PC and does not mimic the hardware. Timing differences may be significant, hence testing on hardware is important, even at the early stages, to prove that a concept is possible on the phone and not likely to be too slow to be usable.

The emulator can be slow to start up, which is frustrating, but this can be improved in various ways, depending on the version of the SDK in question. For example, the S60 3rd Edition emulator can use a modified version of the `starter.rsc` file. The simplest existing version is `epoc32\release\winscw\udeb\z\resource\starter_shell.rsc`, which can considerably speed up the emulator start time. The drawback is that it does not start everything you need in the full emulator (for example, you can no longer double click to navigate through the emulator screens, but need to use the four-way navigation buttons, left and right soft keys and menus).

You can give this a try by renaming `epoc32\release\winscw\udeb\z\resource\starter_shell.rsc` to `starter.rsc`. It is advisable to back up the current `starter.rsc` by renaming it before you do this, so you can roll back and use it, if you decide the 'skinny' emulator start up is no good for your purposes.

Drive Mapping

The emulator maps features of the target machine onto features of the PC environment. For software development, it is particularly important to know how the emulator maps drives and directories onto your PC's file system.

On a Symbian OS phone, there are two important drives (see Figure 10.1):

- **z:** is the ROM, which contains a bootstrap loader and all the EXE, DLL and other files required to boot and run Symbian OS and its applications. All files on **z:** are read-only; program files are executed directly from the ROM rather than being loaded into RAM.
- **C:** is the read-write drive that is allocated from internal RAM. **C:** contains application data, application and system INI files and user-installed applications.

On the emulator, these drives are mapped onto subdirectories of the drive on which you installed the SDK (see Figure 10.2 and the following description). This is to separate different emulator builds, and to ensure

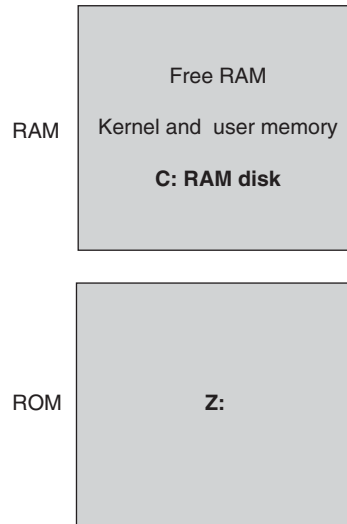


Figure 10.1 Drives on a Symbian OS phone

that applications running on the emulator cannot write to other locations on your PC's C: drive. An emulator configuration directory and startup directory completes the list of directories required by the emulator.

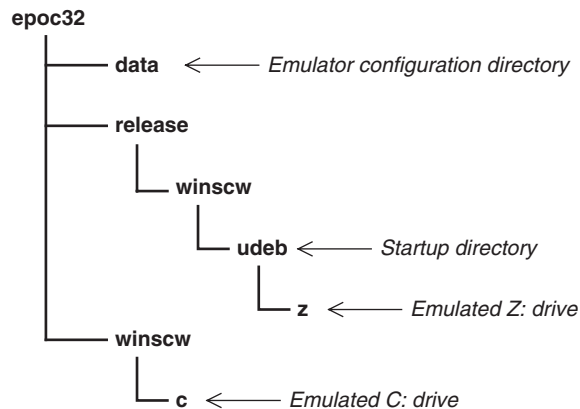


Figure 10.2 Emulator directories

- `\epoc32\data\` is the emulator configuration directory. It contains the initialization parameters for the emulator (`epoc.ini`), the bitmap used as the fascia surround for the screen (`epoc.bmp`), and variants for screens of different sizes.
- `\epoc32\release\wincw\udeb\` is the emulator startup directory. It contains the Windows emulator (`epoc.exe`) and all the shared library DLLs.

- `\epoc32\release\winscw\udeb\z\` is the emulated Z: drive. It contains everything that the EPOC Z: drive should contain, except shared library DLLs, which are in the parent directory.
- `\epoc32\winscw\c\` is the emulated C: drive. It contains data and files. It does not contain compiled C++ programs – those should all be on Z:. In the emulator, all compiled applications become part of the pseudo-ROM that is the emulated Z: drive.

You might wonder why there are so many deeply nested directories. The directories categorize the Symbian OS SDK materials, as follows:

- `epoc32\` sets apart the Symbian OS SDK run-time software from anything else on the same drive
- `release\` sets apart released code from documentation, temporary build files, configuration files, etc.
- `winscw\` sets apart the emulator from target-machine builds; different targets use different directories; for Carbide.c++ and Metrowerks CodeWarrior, it is `winscw`
- `udeb\` sets apart the debug build from other builds; an executable C++ program built for the emulator debug build won't run in any other execution environment, so the debug build is kept distinct from any other build
- `z\` attempts to mirror the structure of Z: on a real Symbian OS mobile phone.

It would be nice if `\epoc32\release\winscw\udeb\z\` could contain the entire emulated Z: drive. Unfortunately, it cannot: EXEs and shared library DLLs cannot use the file structure of Symbian OS, without impractical implications for the path environment in Win32. The only practical thing to do is to place all EXEs and shared library DLLs in the startup directory: `\epoc32\release\winscw\udeb\`.

Finally, data is independent of whether a build is release or debug, so C: is mapped to the `\epoc32\winscw\c\` directory and shared between all builds.

Emulator Keys

The majority of the PC keyboard is mapped in a straightforward way to the Symbian OS keyboard. However, some special keys are available, as shown in Table 10.1.

Table 10.1 PC key mappings to Symbian OS functions

PC key	Symbian OS facility
F1	Menu key
Alt+F2	Help key (this doesn't work in the UIQ emulator as there is no system help file)
Alt+F4	Close the emulator window
F9	Power on
F10	Power off

Different user interfaces have additional keys that are part of the hardware. These are specified in the `epoc.ini` file in `\epoc32\data`. Table 10.2 shows the keys that are available in the UIQ emulator.

Table 10.2 PC key mappings to UIQ emulator keys

PC key	UIQ emulator key
Numeric keypad <minus>	Page up (TwoWayUp)
Numeric keypad +	Page down (TwoWayDown)
Numeric keypad 8	Cursor up (FourWayUp)
Numeric keypad 4	Cursor left (FourWayLeft)
Numeric keypad 6	Cursor right (FourWayRight)
Numeric keypad 2	Cursor down (FourWayDown)
Numeric keypad 5	Enter (FourWayConfirm)
Numeric keypad Enter	Enter (FourWayConfirm)

Communications

The Symbian OS emulator can use a variety of communications methods, such as Bluetooth, infrared or TCP/IP, to access the outside world.

- You can use a supported serial infrared pod for infrared beaming or communication with an infrared-enabled mobile phone.
- You can use a serial Bluetooth device. The S60 v3 FP1 emulator also contains support for USB Bluetooth dongles.
- You can use a LAN for TCP/IP communications.
- You can use a PC-style serial cable, often available for data-capable mobile phones, to connect the Symbian OS emulator to a mobile phone. Note that particular phones may not support all the AT commands required.

Infrared

The easiest way to set up the SDK for IrDA is via the configuration dialogs.

In the S60 3rd Edition emulator, go to Tools, Preferences, then select the PAN tab. Enable IrDA then choose the appropriate COM port number. If your pod doesn't work with the default settings, you'll need to edit the configuration file, as explained below.

For the UIQ 3 SDK, launch the SDKConfig application from the Start menu, select the default device and click on OK, then go to the Communications tab and select the appropriate IrDA port. Again, if you use a pod incompatible with the default, you'll need to edit the configuration file manually.

Symbian OS needs a plain serial port interface to the IR pod – USB pods are unsupported. This means that you must not install any Windows drivers that come with the IR pod you are using. You can see if there are any IR drivers installed by looking in the Windows Device Manager and checking for infrared devices. If you see your IR pod listed there you will need to uninstall the Windows drivers before Symbian OS can use it.

To edit the configuration file for non-default hardware, open the file `\epoc32\release\winscw\udeb\z\private\101f7989\esock\ir.irda.esk`. The two settings you need to change are `irPod` and `irPhysicalCommPort`.

- The `irPhysicalCommPort` should be set to the value of the Symbian OS port to which the IR pod is attached. Note that COM1 on Windows is COMM::0 on Symbian OS, COM2 is COMM::1 etc.
- The `irPod` value should be set to one of following options, depending on which IR pod you are using: `actisys2201`, `actisys2201+`, `actisys2201i`, `tekram`, `jeteye7201`, `jeteye7401`, `paral-lax`, `ifoundry8001a`, `connectTech`.

Make sure that the lines contains precisely one space after the '=' sign:

```
irPod= actisys2201
```

Bluetooth

The easiest way to set up the SDK for Bluetooth is via the configuration dialogs.

In the S60 3rd Edition emulator, go to Tools, Preferences, then select the PAN tab. Enable Bluetooth then choose the appropriate HCI and port number.

For the UIQ 3 SDK, launch the SDKConfig application from the Start menu, select the default device and click on OK, then go to the Communications tab and select the appropriate COM port. If you need to change the HCI in use, you'll need to edit the configuration file manually. Open the file `\epoc32\release\winscw\udeb\z\private\101f7989\`

esock\bt.bt.esk and find the [hci] section. Edit the port= setting to the appropriate Symbian OS COMM port number (remember that COM1 on Windows is COMM::0 on Symbian OS, COM2 is COMM::1 etc.). The hcidllfilename= setting should contain the name of the appropriate DLL. Again, ensure that both lines contain only a single space, after the '=' sign.

IP network

There are various ways to connect the emulator to an IP network – in fact, the S60 3rd Edition emulator is already set up to use WinSock. The UIQ 3 emulator can install WinPCap during installation – if you allowed this, just launch the SDKConfig application from the Start menu, go to the Communications tab and press Apply Ethernet to set it up.

10.2 Emulator Debugging

IDEs such as Carbide.c++ or CodeWarrior allow you to interactively debug programs running in the emulator. They support standard debugging techniques such as setting breakpoints in the source code, stepping through code, and examining memory and variables.

To do this, you must do a debug build (in Carbide.c++ choose an 'emulator debug' option when you select the SDK and Build Configuration initially; in CodeWarrior, select Project, Set Default Target, WINSCW UDEB) to generate the debugging information that the emulator needs.

The following sections outline debugging issues related to Symbian OS on Carbide.c++ and CodeWarrior. They can be used in conjunction with the standard debugging techniques mentioned above, which are discussed in the product documentation.

Carbide.c++ IDE

Before debugging your program, set up the debugger configuration panel, by selecting Run, Debug (see Figure 10.3). Ensure that, when debugging, you use the UDEB versions of both your program and the emulator.

In the Debugging tab, select the View Process output checkbox to direct standard output messages to the console. Selecting the View Windows system messages checkbox logs all Windows' system messages.

Run the debug build of your project by right-clicking the project name in the C/C++ Projects view and selecting Debug As, Debug Symbian OS Application.

Carbide.c++ allows you to step over, step into and step out of code by using breakpoints. You can set or hide breakpoints and examine thread stacks, memory and variables during the debugging process. Symbolics

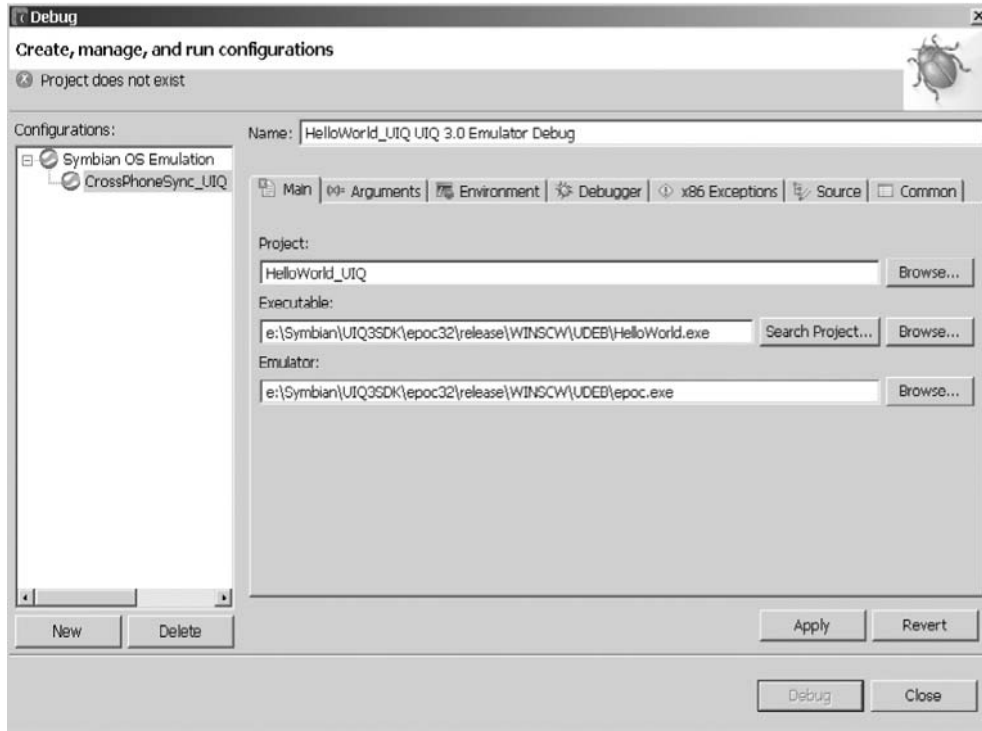


Figure 10.3 Carbide.c++ Debugger Settings

files are loaded by default and symbolics can be viewed. Figure 10.4 shows the debugging console layout.

CodeWarrior IDE

Before starting to debug an application, you should turn on the logging. Select Edit, <selected build> Settings (e.g., WINSCW UDEB Settings). In the Debugger, Debugger Settings panel (see Figure 10.5), select the Log System Messages checkbox. If you select Auto-Targeting in the Library Targeting selection box, the debugger retrieves symbolic data for DLLs as they are loaded.

Once you have started to debug your program, by pressing F5 or by selecting Project, Debug, you may inspect current active processes from View, Processes (see Figure 10.6). This provides information about the running processes and is useful if your program invokes other processes or components, to check whether all those processes are correctly invoked and running or not.

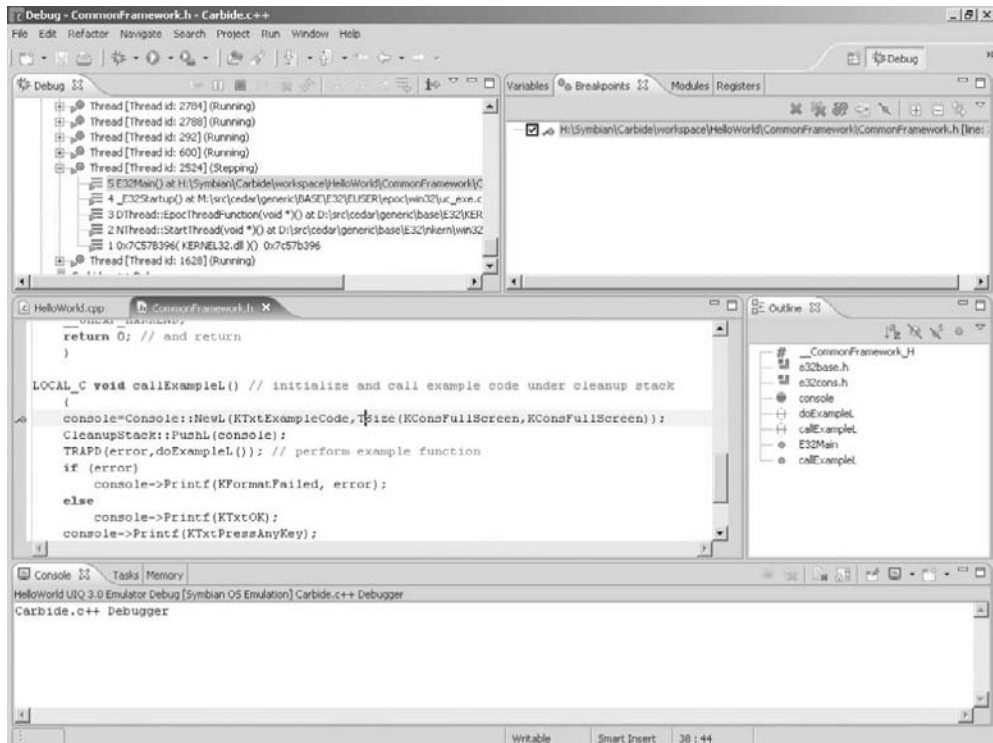


Figure 10.4 Carbide.c++ debugging console layout

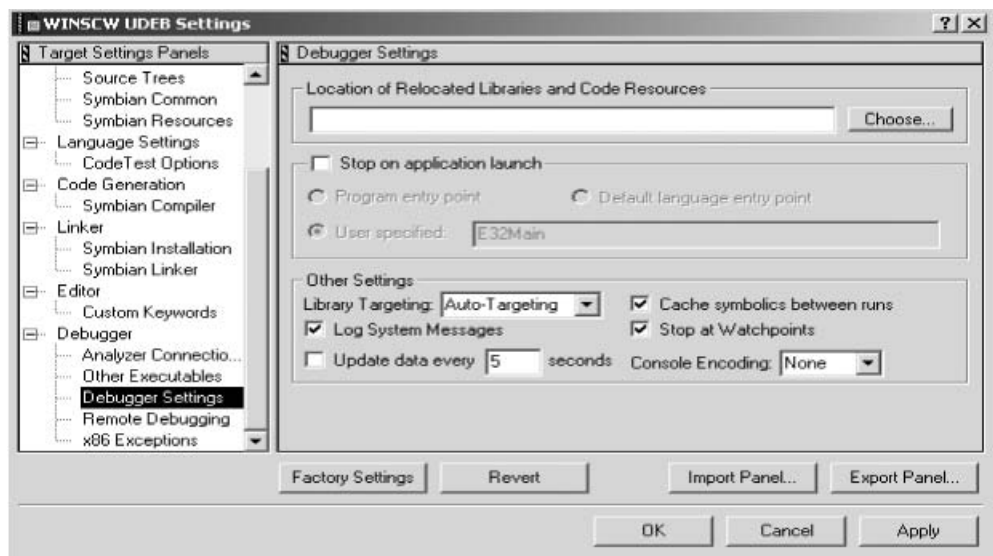


Figure 10.5 CodeWarrior Debugger Settings panel

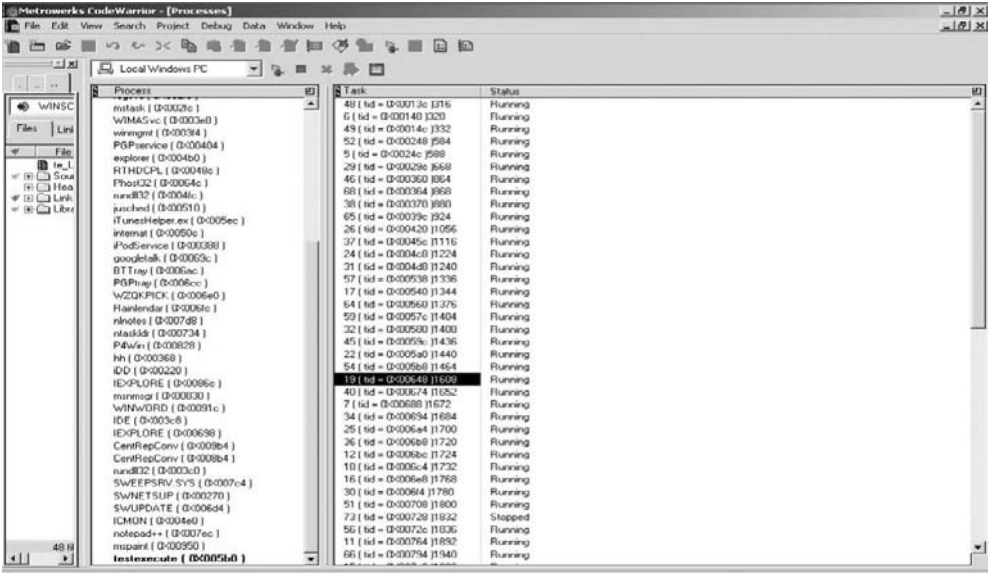


Figure 10.6 CodeWarrior Process View

Furthermore, you can open symbolics files by selecting View, Symbolics (see Figure 10.7). When debugging information is available for loaded DLLs, the function names are displayed, so you can then look into the implementations of problematic functions provided by the relevant DLL.

You can set or hide breakpoints and examine program stack, memory and variables during the debugging process. If the DLL source code is available, you can even step into it and set breakpoints in it.

Emulator Debugging Keys

In debug builds, the emulator GUI provides special key combinations for resource checking and redrawing, trapping memory leaks and checking that drawing code functions correctly. These are available using Ctrl+Alt+Shift. There are also some special key combinations for logging window server events and actions, and a few other miscellaneous keys.

Resource allocation keys

- A shows the number of heap cells allocated on the user heap by the current program.
- B shows the number of file server resources in use by the current program.
- C shows the number of window server resources in use by the current program.

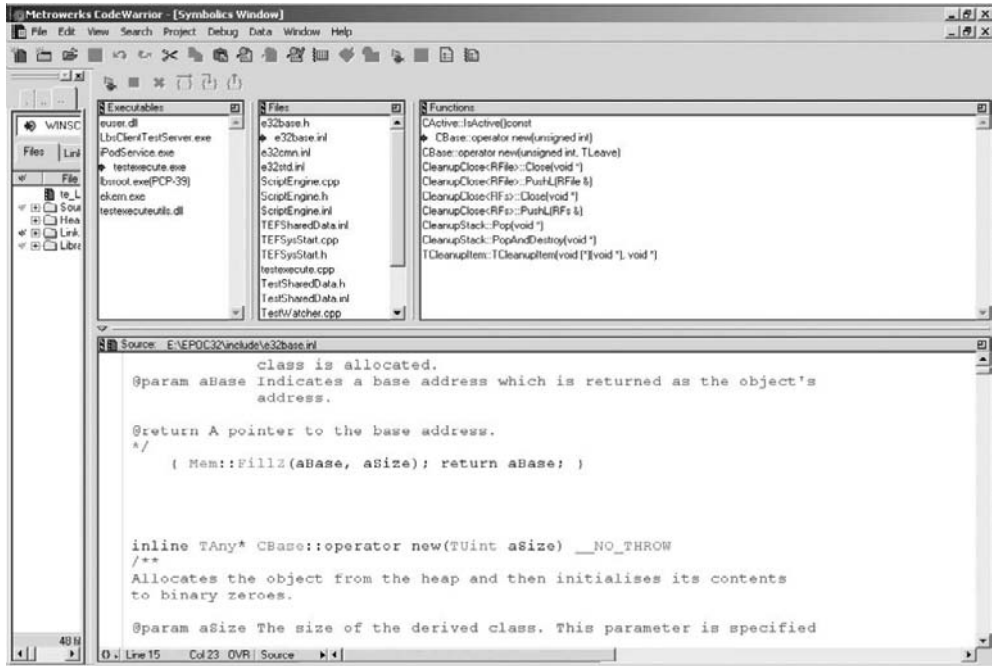


Figure 10.7 CodeWarrior Symbolics View

- P shows the settings dialog for the heap-failure tool.
The heap failure tool artificially generates resource allocation failures. It is used to test the efficiency of an application during such failures. You can set the options as: App heap failure Type (Random or Deterministic), Wserv heap failure Type, and File failure tool.
- Q turns off heap-failure mode.

Drawing keys

- R redraws the whole screen, to test whether the application handles a complete redraw properly.
- F enables window server auto-flush for all programs using the current control environment.
- G disables window server auto-flush for all programs using the current control environment.

Window server logging keys

Note that this group of keys does not work on the UIQ 3 SDK.

- E starts (enables) window server logging.
- D stops (disables) window server logging.

- W dumps the full window tree.
- H dumps the contents of the window server's heap.

Miscellaneous keys

- K kills the application that currently has keyboard focus.
- T brings up a task list in which running applications can be switched to or closed down. Note that this is a debugging aid only.
- V turns on or off verbose information messages in the emulator log file `epocwind.out`.
- Z sends keys A through J in fast sequence to the application, to test its ability to handle fast repeated keys.

Emulator Settings

The emulator can be configured by editing the file `\epoc32\data\epoc.ini`. You should back up this file before making any changes.

For example, the platform security settings can be modified, allowing you to ignore platform security when you are debugging:

- **PlatSecDiagnostics**: when a platform security policy check fails and this setting is ON, a diagnostic message may be written to the emulator log file `epocwind.out`. Diagnostic messages start with the text `'*PlatSec*'` followed by either `'ERROR'` or `'WARNING'`
- **PlatSecEnforcement**: when a capability or other platform security policy check fails and after any diagnostic message has been displayed, if this setting is OFF, the system continues as though the platform security check was passed. If this setting is ON, then the appropriate action for a failed platform security check happens
- **MegabytesOfFreeMemory**: if you are testing memory, you can set the amount of RAM memory (enter a number)
- **LogToDebugger**: if set to 1, system messages are logged to a debugging window that can be viewed from the IDE. In Carbide.c++, select Run, Debug. In the Debugging tab, select the View Process output and View Windows system messages checkboxes. In CodeWarrior, select Edit, Settings for your target. In the Debugger, Debugger Settings panel, select the Log System Messages checkbox.
- **JustInTime**: if set to 1, enables just-in-time debugging for a process. This prevents the emulator closing down when a panic occurs. By

default, just-in-time debugging is enabled. This option can also be set by calling `User::SetJustInTime()` in your code.

The emulator maps the Symbian OS device's file system onto the PC's file system, as described in Section 10.1. Two drives, `C:` (the internal file system) and `Z:` (the device ROM), are created by default. The default location for these drives is:

```
C: = %EPOCROOT%\epoc32\wins\c
Z: = %EPOCROOT%\epoc32\data\z + %EPOCROOT%\epoc32\release\wins\udeb\z
```

Further drives can be configured in `epoc.ini`, for example a `D:` drive can be defined as:

```
_EPOC_DRIVE_D \epoc32\winscw\d
```

There are other ways to configure the Windows emulator supplied with the S60 and UIQ SDKs. More details can be found within the developer documentation.

Certificates for Emulator Testing

In Symbian OS v9, applications that run on phones usually need to be digitally signed. The UIQ 3 SDK supports this by allowing only signed SIS (installation) files to be installed on the emulator. The UIQ 3 SDK includes three sets of certificates and keys to use when preparing applications for testing on the emulator. The certification files are located at `\epoc32\tools\cert` and should be installed into `\epoc32\winscw\c\system\data`.

Note that these certificates only work on the emulator. For applications to run on a phone, the SIS file must be signed via the Symbian Signed process (see Chapter 9).

Which certificate to use depends on whether the application UID is in the protected range and whether system capabilities are required. The certificates provided are:

- `NoCaps.cert` and `NoCaps.key`, which is a dummy certificate to sign SIS files that require only user capabilities or no capabilities at all; it is needed because all SIS files must be signed before they can be installed
- `AllUserCaps.cert` and `AllUserCaps.key`, which blocks the Grant capabilities dialog when one or more user certificates are defined in binaries that are included in the SIS file

- `AllCaps.cert` and `AllCaps.key`, which allows you to install SIS files that include binaries with system capabilities.

One way of finding out which certificate you need is to test them one at a time in the above order.

Making the Most of the Emulator Log File

The emulator writes debugging information to a log file, called `epocwind.out` (typically in the directory specified by the `%temp%` environment variable), when it runs. This log file contains additional information which can prove useful when debugging your code.

Using *RDebug::Print()* statements

One technique for aiding debugging using the emulator is to add `RDebug::Print()` statements to your code (you need to include the `e32debug.h` header file to use `RDebug`). This function is used to print formatted text to the emulator log file, helping you trace what your program is doing. `RDebug::Print()` can be used to print all kinds of data, including memory addresses and values of variables.

You can use `%` followed by a format character in the arguments to `RDebug::Print()` to print out variables. For example, the following code writes `Info, value: 4` to the log file:

```
TInt numberOfWheels = 4;
_LIT(KMsgDebug, "Info, value: %d");
RDebug::Print(KMsgDebug, error);
```

When using `RDebug::Print()` to display the contents of descriptors, the address of the descriptor should be passed instead of the descriptor itself, otherwise a KERN-EXEC 3 panic is raised:

```
TBuf des;
...
RDebug::Print(_L('%S'), des); // Panic
RDebug::Print(_L('%S'), &des); // OK
```

A useful macro, `__LINE__`, evaluates to the current line number in the source code file:

```
RDebug::Print(_L("Debug on line %d"), __LINE__);
```

Using RTest

If you are performing tests within your code, you might find that the `RTest` class provides some useful facilities. `RTest` creates a console window to which test results can be logged. Specifically, different overloads of `operator()` can be used to check the test condition, print out failure messages and panic the test code. `RTest` can also get characters from the keyboard. If the logging flag is set using `RTest::SetLogged()`, the console output is also written to the debug output represented by an `RDebug` object. Details of the `RTest` class APIs can be found in the Symbian Developer Library within SDKs.

Platform Security

As we have seen, security is a major concern in Symbian OS v9, and the log file contains details which may help you track down problems related to Platform Security capabilities.

The start of the log file contains information about the platform security settings:

```
PlatSecEnforcement OFF
PlatSecDiagnostics OFF
PlatSecProcessIsolation ON
PlatSecEnforceSysBin ON
PlatSecDisabledCaps NONE
0.000 Thread 53bf1280 created @ 0x53bf1280 - Win32 Thread ID 0x890
0.120 Thread EFile.exe::Main created @ 0x98fa18 - Win32 Thread ID 0xb3c
1.015 SysStart: using resource file Z:\private\10205C44\
                SSCForStartupModel1088.rsc
13.300 SysStart: starting Z:\sys\bin\HelloWorld.exe
13.305 Thread HelloWorld.exe::Main created @ 0xf92c3c - Win32
                Thread ID 0x5d8
```

These settings are discussed in the Emulator Settings section. They allow you to determine whether you want to see errors related to platform security whilst running your code or in the log file. Capturing the diagnostic messages in the log file is particularly useful, as they are produced when an application attempts to call a function or perform an action for which it does not have the required capabilities: the error message tells you which capabilities are needed.

Below is the next part of the `epocwind.out` file:

```
13.305 Thread HelloWorld.exe::Main created @ 0xf92c3c - Win32
                Thread ID 0x5d8
13.320 Thread HelloWorld.exe::Worker135988996 444077980 created @
                0xf92454 - Win32 Thread ID 0xa00
13.330 *PlatSec* ERROR - Capability check failed - A Message
                (function number=0x00000005) from Thread HelloWorld.exe
                [10282281]0001::Worker135988996 444077980, sent to Server
```

```
!LogServ, was checked by Thread LogServ.EXE[101f401d]0001::  
LogServ and was found to be missing the capability:  
WriteDeviceData.  
13.340 Thread HelloWorld.exe::Worker135988996 444077980  
Panic E32USER-CBase 47
```

Here we can see, at time 13.330, that the running program, HelloWorld.exe, made a call to the LogServer, but the server rejected the call, as HelloWorld didn't have the WriteDeviceData capability. This would cause the program to fail, but can be remedied by granting the program the WriteDeviceData capability, if appropriate.

Panics

The log file also shows details of other run-time failures (panics) in the program. A message is placed in the log whenever a thread panics and is written in the following format:

```
Thread <program>::<thread> <a thread identifier> Panic <panic type>  
                                     <panic number>
```

Referring back to the log file excerpt above, the entry at 13.340 shows that a panic has occurred. The panic is of type E32USER-CBase and is panic number 47. Looking up this information in the SDK, in the System Panic reference section within the Symbian OS Reference, tells us that something caused an active object's RunL() function to leave, resulting in the active scheduler generating the panic E32USER-CBase 47.

Logging

In order to examine the dynamic behavior of your programs, you can add logging in your code. Logging is very useful when debugging active objects and client-server framework applications as well as other inter-process communication mechanisms.

While testing on the emulator, RFileLogger is a client-server API that can be used for logging messages into a file. The logging message contains a time stamp, a thread ID, severity, location (file name and line number) as well as the message. Two formats are supported: an XML-formatted log and a plain-text log. The format is decided by the extension of the filename the user supplied: XML or TXT.

Two types of API usage are shown in the following example code, static and non-static. The difference is in the version of the I/O functions used; in practice, the static version may be slower than the non-static

version since the non-static version is designed for a one-time connection which is continuously used, while the static version implicitly reconnects each time.

```
#include <flogger.h>
...
RFileFlogger logger;
TInt err = logger.Connect();
if(err)
    return EFail; // failed to create a session

_LIT(KLogFilexml, "c:\\TestinXML.xml");
err = logger.CreateLog(KLogFilexml, RFileFlogger::ELogModeAppend);
if(err)
    return EFail; // failed to create a file

TInt n=0;
_LIT(K16BitFormatText, "logger string int = %d, string = %S");
logger.SetLogLevel(RFileFlogger::ESevrWarn);
//As severity is ESevrWarn, warn only; print error to log
_LIT(K16BitString, "The String16");
TBuf<20> buf16(K16BitString);
logger.Log((TText8*)__FILE__, __LINE__, RFileFlogger::ESevrAll,
           K16BitFormatText, n++, &buf16);
logger.Log((TText8*)__FILE__, __LINE__, RFileFlogger::ESevrInfo,
           K16BitFormatText, n++, &buf16);
logger.Log((TText8*)__FILE__, __LINE__, RFileFlogger::ESevrWarn,
           K16BitFormatText, n++, &buf16);
logger.Log((TText8*)__FILE__, __LINE__, RFileFlogger::ESevrErr,
           K16BitFormatText, n++, &buf16);

TInt length = 2;
TExtraLogField logField[2];
logField[0].iLogFieldName.Copy(_L("SUITE_NAME"));
logField[0].iLogFieldValue.Copy(_L("SUITE_VALUE"));
logField[1].iLogFieldName.Copy(_L("TEST_NAME"));
logField[1].iLogFieldValue.Copy(_L("TEST_VALUE"));
logger.Log((TText8*)__FILE__, __LINE__, RFileFlogger::ESevrErr,
           length, logField, K16BitFormatText, n++, &buf16);
logger.Close();

// from now on, static methods are used to create the logs
_LIT(KLogFolder, "Logs");
_LIT(KLogFile, " TestinTXT.txt");
_LIT(KSomeText, "Log me.");
RFileLogger::Write(KLogFolder(), KLogFile(),
                  EFileLoggingModeAppend, KSomeText());
TUint num1 = 100;
_LIT(KWriteNumber, "Writing a number to the log: %d")
RFileLogger::WriteFormat(KTestLogDir2(), KTestLogFileNamel(),
                        EFileLoggingModeAppend, KWriteNumber, num1);
```

RFileLogger should not be used for debugging on devices, as it is not supported on all Symbian OS phones.

Using eshell

eshell is a simple command-line shell, available on both the emulator and on some target hardware, which you may find quite useful. You can run it directly from the UIQ 3 emulator (see Figure 10.8) or change to the emulator's startup directory (`\epoc32\release\wincsw\udeb\`) from the Windows command line and type `eshell`. Information on available commands can be accessed by typing `help` in eshell.



Figure 10.8 eshell on UIQ SDK

The `ps` command provides you with many options to view the current running processes, threads, etc. During debugging, you can use these commands to display the running processes and check whether your program or the applications (servers, libraries) invoked by your programs have started correctly.

- `A` displays all container objects.
- `T` lists threads.
- `S` lists servers.
- `L` lists libraries.
- `E` lists semaphores.

binary dump of the user stack. %d is the id of the thread that has crashed. Figure 10.10 shows an example of the TXT file created by the D_EXC tool.

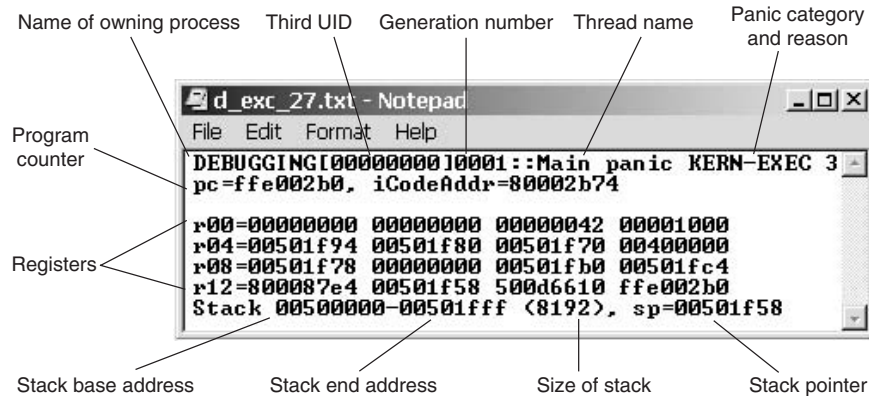


Figure 10.10 D_EXC tool output

Figure 10.11 shows the D_EXC register information and a binary dump of the user stack using Carbide.c++ v1.1 with a UIQ SDK.

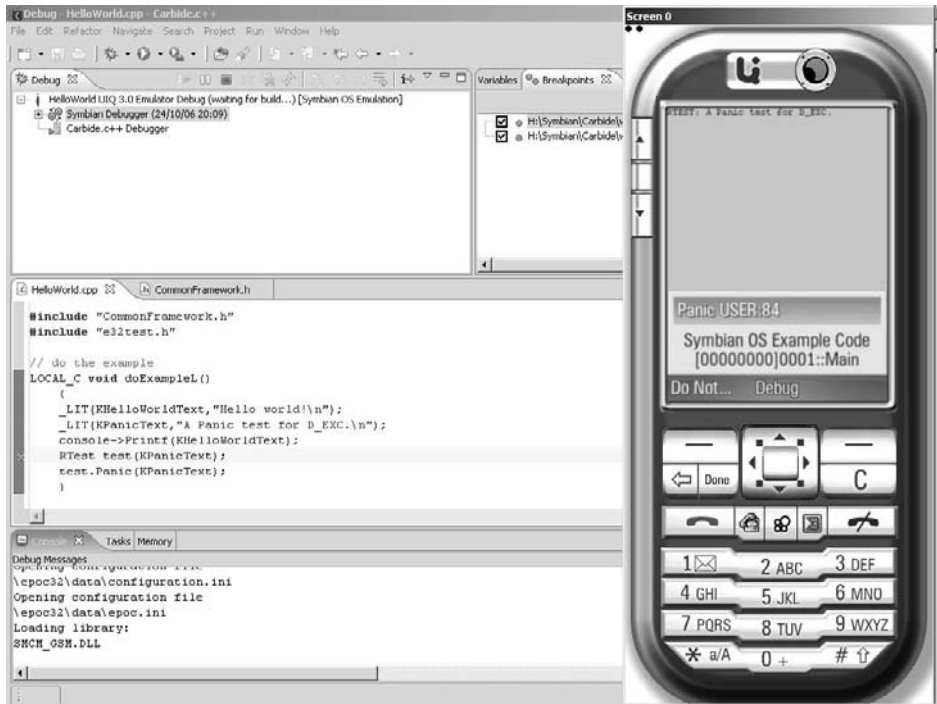


Figure 10.11 D_EXC tool output on UIQ SDK

In this example, a USER 84 panic has been produced via `RTest::Panic()` and the user can choose either to debug on the panic or not. After the panic happens, you can switch to the Task Panel (by pressing `Ctrl+Alt+Shift+T`). Using QFileMan, you can see `d_exc_152.txt` and `d_exc_152.stk` on the C: drive (see Figure 10.12). The generated `d_exc_152.txt` is shown in Figure 10.13.



Figure 10.12 QFileMan showing files produced by `D_EXC` on UIQ SDK



Figure 10.13 An example `d_exc_152.txt`

Memory Tests

When writing code that will run on a mobile phone, it is important to bear in mind the limited resources available on a phone (especially memory) and to ensure that your programs handle any related memory allocation failures correctly.

Due to the limited resources on a phone, requests for memory may occasionally fail. If your code is in this situation, it must inform the user that there is not enough memory to perform the operation or provide the service; certainly the code must not panic without returning meaningful information and should not lose any existing data.

One common error is for programs to allocate memory for objects but to fail to delete the objects, and hence fail to free the memory. This is called ‘leaking’ memory. There are tools available to help catch these errors.

Heap-Management Macros

Each thread has a chunk of memory which contains that thread’s program stack. For the main thread of a process, this chunk also contains the thread’s heap. A program’s requests for memory are allocated from this heap.

A set of heap-management macros are provided in `e32def.h` to help to check for memory leaks in debug builds. `__UHEAP_MARK` is used to mark the start of a new level of heap-cell checking. Every call to `__UHEAP_MARK` should be matched by a later call to `__UHEAP_MARKENDC`, which verifies that the number of heap cells allocated at the current nested level is as expected. If there is a memory leak (because more or fewer heap cells are allocated), a panic is raised with information about the first leaked memory allocation on the heap.

UIQ SDK 3 provides a heap allocator called `RQikAllocator` that can be used by including the following libraries in your MMP file:

```
LIBRARY QikAllocDll.lib  
STATICLIBRARY QikAlloc.lib
```

Memory leaks in code can usually be detected on the emulator by using the above macros and examining the panic code. If it is not obvious where a leak occurs, then you may find `HookLogger` more helpful.

HookLogger

`HookLogger` is a Microsoft Windows GUI-based tool which provides easy journaling facilities for logging memory allocations, processes, thread creation and leaves when executing an application under the Symbian

OS emulator. The main use for most developers is to find the source of a leaked heap cell. The application works by replacing EUSER.DLL from the target Symbian OS SDK with a version that allows the attachment of 'hooks' that are used by HookLogger. After the test execution session is finished, the user can issue the same command to restore the original version of EUSER.DLL.

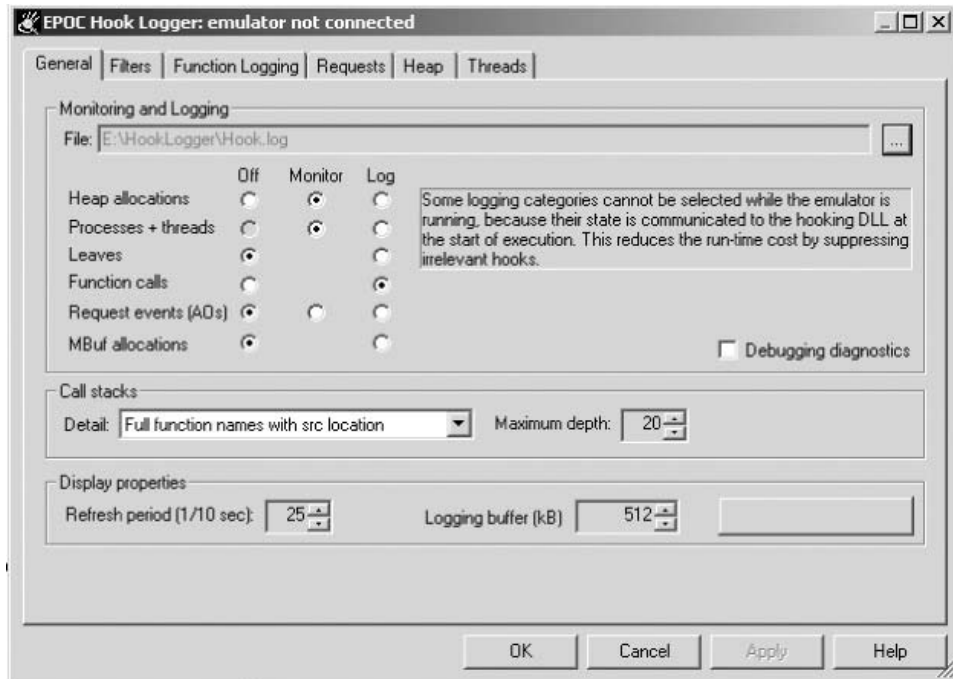


Figure 10.14 HookLogger Configuration

1. Download HookLogger from ***developer.symbian.com\main\downloads\files\HookLogger_Setup.zip*** and install it in a directory on your development drive (e.g. X:\HookLogger\).
2. Attach the hooks to EUSER.DLL by starting a command prompt in the HookLogger directory and typing 'hookeuser winscw'. (Type only 'hookeuser' at the command prompt to get more options.) This replaces the EUSER.DLL of the target Symbian OS SDK with a version used by the HookLogger tool.
3. Run HookLogger .EXE and configure it to monitor heap allocations.
4. To find the memory leak, start the emulator and execute the failing programs to reproduce the memory leak. Go to the Threads tab in HookLogger, find the application that leaks and untick all other threads. (The processes can be filtered by settings on the Filter tab.)

On the Heap tab, list All allocs to display the current memory allocations.

5. Selecting an allocation and click Alloc details to see additional information that should allow you to track down the source file where a leak is occurring.
6. You can use the radio buttons on the General tab to get additional information.
7. Restore the original version of EUSER.DLL by typing 'hookeuser -r winscw'.

10.3 Debugging on a Phone

After debugging on the emulator has been completed, the next stage is to focus on how your application runs on the target device – a mobile phone.

The first stage is to compile your code for the ARM platform by changing the target platform in the IDE to ARM. (You do this in the Build Configuration tab in Carbide.c++ and the Project window in CodeWarrior.) There are two ARM target options: debug (UDEB) and release (UREL). The difference is that debug versions contain unoptimized code, making debugging easier: the release versions are optimized and are usually without debugging code (this can be generated, but the optimizations make debugging more difficult). As a result, the size of the final binary components from release build is generally smaller than debug versions, which is a benefit for such devices.

It's probably best to do your on-target debugging using a release build, as it is possible that some errors may only manifest themselves on release builds, due to compiler optimizations or differences in memory layout or initialization.

On-Target Debugging Agents

On-target debugging requires 'debug agent' software to be running on the device, to communicate with the IDE. For Symbian OS v9, the supported debugger is Metrowerks Target Resident Kernel (MetroTRK).

MetroTRK is a Symbian OS application that runs inside the operating system and provides debug services over serial communications (a cable, Bluetooth or infrared, for example) to the Carbide.c++ Developer and Professional or CodeWarrior Professional debuggers. MetroTRK lets you download an application and its supporting files to the target device and debug it using the IDE. Not all phones support hardware debugging – for a list of those supported and to download MetroTRK itself, see the CodeWarrior product descriptions at [www.forum.nokia.com\codewarrior](http://www.forum.nokia.com/codewarrior).

Emulator versus On-Target Debugging

Debugging can be carried out both on an emulator and on target hardware. Initially you use an emulator, proceeding to target hardware later on. There are a number of differences between emulator debugging and on-target debugging.

- On the emulator, it is possible to write and modify things on the Z: drive; on target hardware, the Z: drive is the ROM drive and cannot be written to or modified in any way.
- On the emulator, stacks can automatically grow to be very large; on target hardware, the stacks are fixed in size and much smaller (typically 8 KB but configurable using the `epocstacksize` keyword in the MMP file). They have to be used very carefully to prevent stack overflow.
- On target hardware, misaligned word access often leads to KERN-EXEC 3 panics (equivalent to an access violation). On ARM processors, you can only access words on aligned boundaries. On the emulator, this is not a problem because it uses x86 architecture processors.
- On target hardware, certain system files are locked – some processes that run on boot can lock files that you may want to overwrite. For example, the way you generate and edit the Comms Database on target hardware can be completely different and is not consistent across different products.
- Floating-point exceptions can occur on target hardware but not when using the emulator. Symbian OS provides a complete software emulation of the IEEE-754 64-bit floating point processor using the `TRea164` class. The IEEE floating point raises an exception when certain things happen, for example division by zero. Applications can elect to handle these exceptions or accept IEEE default behavior. Microsoft Windows programs have the floating-point exceptions disabled, so these exceptions never occur on the emulator; some ARM processors do not have floating-point hardware and so all arithmetic is done using the Symbian OS emulated floating point processor.
- On target hardware, a file may be missing, present (when it should not be there) or wrongly named in the ROM.
- Process errors: On the emulator, it is possible for a process to access the address space of other processes because all Symbian OS threads are part of a single Microsoft Windows process; on target hardware, this is not the case.
- Timing differences may occur between emulator and target hardware, leading to programs succeeding on the emulator but failing on the target hardware.

- The order in which code is called can be different on the emulator than on the target device. For example, the following code might work fine on the emulator but fail on target (or the other way around):

```
TResourceReader reader;  
...  
MyFunction(reader.ReadXxx(), reader.ReadXxx());
```

The result is dependent on the order in which the two `ReadXxx()` calls are made, but the order of evaluation of function arguments (left to right or right to left) may differ from compiler to compiler.

10.4 Miscellaneous Tools

Some additional tools, which can be helpful in debugging, are available from the Symbian. Developer Network website, **developer.symbian.com**. Of particular note is SymScan, the improved successor to LeaveScan.

SymScan scans your source code files to find possible problems related to the naming conventions and the cleanup stack. The output is stored in a TXT file that lists misuses and potential defects in the following categories:

- use of the deprecated literal macro `_L`; you should use the `_LIT` macro, as it is more efficient
- correct use of the cleanup stack: are objects that you create for automatic variables correctly put on the cleanup stack as well as removed and destroyed again?
- the correct opening and closing of R classes (connections to servers)
- functions marked as non-leaving that can leave: you may forget to add a suffix of L to the name of a method that can leave
- correct use of descriptors in function calls.

The source code and binary files can be downloaded from the Symbian website: **developer.symbian.com/main/tools/devtools/code**. The utility is installed into `C:\Program Files\Common Files\Symbian\Tools\`, with documentation in the Start menu under Programs, Symbian OS Tools.

To scan a single file:

```
symscan <filename.cpp>
```

Different options for your scanning target can be specified; the full option list can be viewed by using:

```
symscan - H
```

SymScan can also be run from the command line to scan whole directories and output to a file. Use the Windows 'for' command to scan directories:

```
for /R %i in (*.cpp) do symscan "%i" >> output.txt
```

This scans all C++ files in or below the current directory, placing the result in `output.txt`. Occurrences of the issues listed above are highlighted with their line position within the corresponding source file.

Summary

In this chapter, we have considered how to debug Symbian OS programs using the emulator that is included in SDKs. We've looked at:

- how the emulator maps drives and directories
- general and debugging-specific emulator keys
- emulator settings
- how to debug under Carbide.c++ and CodeWarrior
- logging and log files
- security issues
- other debugging tools
- on-target debugging.

You can now develop a Symbian OS application and debug it if things do not work as you expected.)

11

The Application Framework

In Chapter 1, we built a simple text-mode program that displays ‘Hello World’ on the screen. Such programs are valuable for testing, but production programs almost invariably use a GUI. In the next few chapters, we start looking at real graphics and GUI programming in Symbian OS.

In this chapter we start with an introduction to the Application Framework subsystem, UIKON, and then build a simple GUI application on both the S60 and UIQ platforms.

11.1 Symbian OS Application Framework

UIKON, the Application Framework subsystem, is architecturally central to Symbian OS support for GUI applications. It provides a flexible architecture which allows a variety of mobile phone platforms to run on top of the core operating system. The two most widely used of these are S60 and UIQ, both of which are discussed in this chapter.

UIKON is itself based on two important frameworks (see Figure 11.1):

- CONE, the *control environment*, is the framework for graphical interaction (see Chapters 17 and 18)
- APPARC, the *application architecture*, is the framework for applications and application data.

UIKON provides a library of useful functions in its environment class, `CEikonEnv`,¹ which is in turn derived from `CCoeEnv`, the CONE

¹ If you’re wondering why the classes are prefixed with `CEik` rather than `CUik`, the reason is historical. The original Symbian OS UI framework was called Eikon, in the days

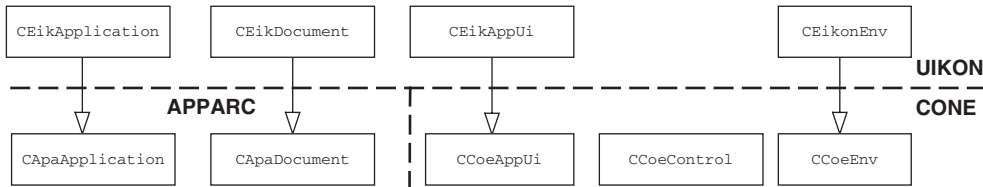


Figure 11.1 Frameworks on which UIKON is built

environment. Your application can use these functions directly without the need to write a derived class.

UIKON provides a framework enabling you to build applications with a graphical user interface. The framework encourages the use of the Model–View–Controller (MVC) design pattern, where the application is split into separate logical parts that encapsulate the different aspects of the whole application. Each part in the application has a specific role.

To support the MVC pattern, the application framework provides separate classes to represent the Model (a document, `CEikDocument`), the View (an application view, derived from `CCoeControl`) and the Controller (an application UI, `CEikAppUi`). These classes provide the basic functionality of an application and an application typically defines its own classes to implement the desired behavior.

- The *application* class is the entry point to the application. It provides a factory for the application document, serves to define the properties of the application and provides an interface to the application's resource file. In the simplest case, the only property that you have to define is the application's unique identifier (UID).
- A *document* represents the data model for the application. If the application is file-based, the document is responsible for storing and restoring the application's data. Even if the application is not file-based, it must have a document class, even though that class doesn't do much apart from creating the application user interface.
- An *application view* is a concrete control whose purpose is to display the application data on screen and allow you to interact with it. In the simplest case, an application view provides only the means of drawing to the screen, but most application views also provide functions for handling input events.
- The *application UI* is entirely invisible. It creates an application view to handle drawing and screen-based interaction. In addition, it

when Symbian OS itself was known as EPOC. The framework was renamed UIKON in a later release of Symbian OS, when Unicode support was introduced, but the class names themselves were not changed.

provides the means of processing commands that may be generated, for example, by menu items.

In addition to the four C++ classes, you have to define elements of your application's GUI—the menu, possibly some shortcut keys, and any string resources you want to use when localizing the application – in a *resource file*. Later in this chapter, we'll briefly describe the simple resource file used for our example code; for further information about resource files, look at Chapter 13.

All versions of Symbian OS support customization of the UI, which is necessary to tailor the look and feel and to support the hardware features of the target machine. These features include the size and aspect ratio of the screen, support for keyboard or pen-based input and the relative significance of voice-centric or data-centric applications. These differences are largely implemented by the creation of additional UI layers above UIKON. This means that every Symbian OS GUI application, regardless of the UI platform on which it is based, uses the UIKON framework architecture.

In particular, the major UIs provide their own customization layers, so let's take a closer look at them.

11.2 S60 and UIQ Platform Application Frameworks

S60 and UIQ platforms extend the framework by adding libraries to provide platform-specific controls. The UIQ-specific library is called *Qikon* and the S60-specific library is called *Avkon*; you can view these as UI layers on top of the core Symbian OS UI. Each contains different components, appropriate to the platform in question, although, because they both have UIKON as a base, their APIs are often similar.

When creating a UI application for either platform, you derive from base classes supplied by the platform-specific libraries (which themselves usually derive from classes in the Symbian OS framework), and call framework functions specific to the platform. To do this, you need to include the appropriate header files and link against libraries accordingly.

For example, for UIQ, you link against `qikctrl.lib`, while for S60 the equivalent is `avkon.lib`. Likewise for header files; the header files to include for each of the application framework classes are shown in Table 11.1.

Note that the `CEik` prefix of the generic Symbian OS classes is replaced with `CQik` for UIQ classes and `CAkn` for S60 classes. This convention is used throughout the UI application framework, for classes, headers and libraries. In general, if a class does not have any prefix, such as `CCoeControl`, it is part of the generic Symbian OS set.

Table 11.1 Header files for UIQ and S60

Class	Generic UIKON class	S60 (Avkon)	S60 header file	UIQ (Qikon)	UIQ header file
Application	CEikApplication (derives from CApaApplication)	CAknApplication	aknapp.h	CQikApplication	qikapplication.h
Document	CEikDocument (derives from CApaDocument)	CAknDocument	akndoc.h	CQikDocument	qikdocument.h
Application UI	CEikAppUi	CAknAppUi	aknappui.h	CQikAppUi	qikappui.h
View	CCoeControl	CCoeControl	coecntrl.h	CQikViewBase (derives from CCoeControl and MCoeView)	qikviewbase.h

11.3 A Graphical Hello World

One of the first questions that many people ask is: ‘Is it possible to write an application that runs on both S60 and UIQ?’ Unfortunately, the answer is ‘No’. As the previous section described, although each of the platforms is based on the generic Symbian OS UIKON framework, S60 and UIQ each extend it differently. When you write an application for UIQ, you must use the UIQ-specific classes to achieve the correct look and feel for the application. The resulting code is different from that of an S60 application, which would use a set of classes specific to S60.

Class Structure

To appreciate the extent of the differences, let’s have a look at the class structure of simple graphical versions of the ‘Hello World!’ application in S60 (HelloS60, in Figure 11.2) and UIQ (HelloUIQ, in Figure 11.3).

The shaded classes are the UI-platform-specific framework classes. As you can see, the structure is fairly similar, although there are some differences; for example the view class of the application on the S60 platform accesses the `CCoeControl` class provided by UIKON directly, where a similar application on UIQ platform chooses to use a specific view base class provided by the UIQ framework that specializes the basic control classes provided by UIKON.

The different UI frameworks have the flexibility of using UIKON in the best way suitable for their specific requirements.

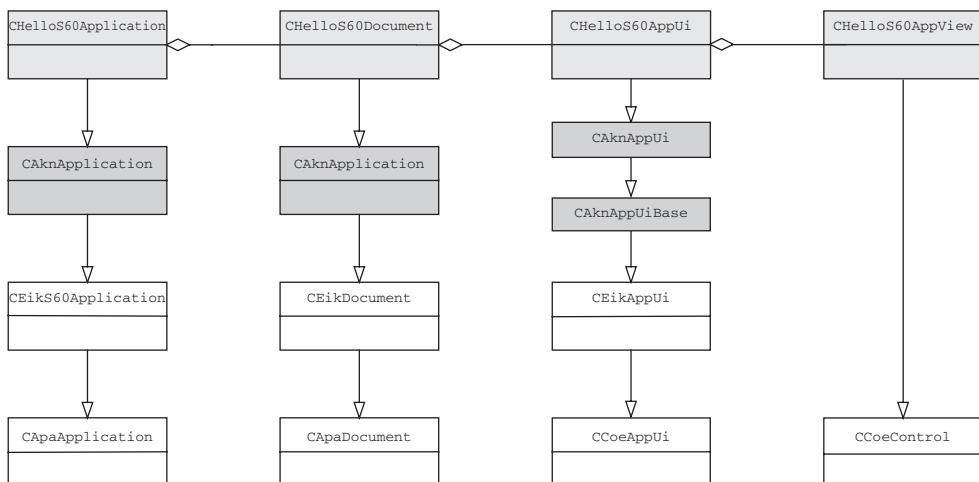


Figure 11.2 Class diagram for HelloS60

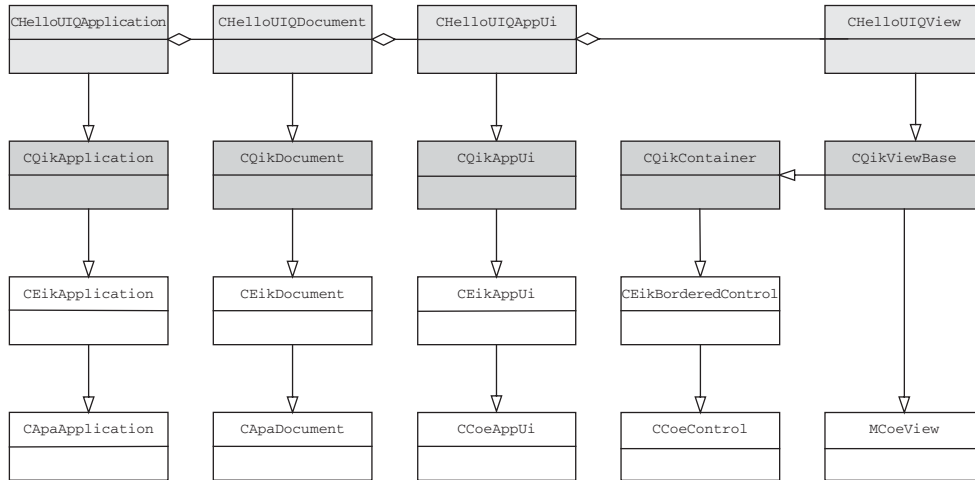


Figure 11.3 Class diagram for HelloUIQ

Generating the Example Code

The HelloS60 and HelloUIQ application example code in this chapter was generated using the Carbide.c++ development environment as follows:

1. Start the Carbide.c++ development environment.
2. Go to File, New, Project.
3. Open the Symbian OS directory and choose C++ application for S60 project.
4. Press Next.
5. Give your project a name, for example, HelloS60, and press Next.
6. Choose S60 3.x GUI application and press Next.
7. The list of S60 SDKs installed on your computer appears.
8. Press Finish to complete the generation of your HelloS60 GUI application.

Repeat Steps 2–8 for the UIQ GUI application: choose C++ application for UIQ project in Step 3 and the name HelloUIQ in Step 5.

In this section, you learn how to build an application for the UIKON GUI on the S60 and UIQ platforms by using our example programs. In reality, it takes much longer to fully learn how a UIKON GUI program works and in this chapter we only briefly comment on the C++ code itself. There's nothing particularly difficult about it, but it is easier to cover

how UIKON uses C++ when we have seen more of how Symbian OS C++ works.

The Project Specification File

As mentioned in Chapter 1, all Symbian OS applications require a project specification file, with an MMP extension. The project specification file for HelloS60 is as follows:

```

TARGET                HelloS60.exe
TARGETTYPE             exe
UID                   0x0100039CE  0xEA7408AF

SOURCEPATH             ..\src
SOURCE                 HelloS60.cpp
SOURCE                 HelloS60Application.cpp
SOURCE                 HelloS60AppView.cpp
SOURCE                 HelloS60AppUi.cpp
SOURCE                 HelloS60Document.cpp

SOURCEPATH             ..\data

START_RESOURCE         HelloS60.rss
HEADER
TARGETPATH             resource\apps
END //RESOURCE

START_RESOURCE         HelloS60_reg.rss
TARGETPATH             \private\10003a3f\apps
END //RESOURCE

USERINCLUDE            ..\inc

SYSTEMINCLUDE          \epoc32\include

LIBRARY                euser.lib
LIBRARY                apparc.lib
LIBRARY                cone.lib
LIBRARY                eikcore.lib
LIBRARY                avkon.lib
LIBRARY                commonengine.lib
LIBRARY                efsrv.lib
LIBRARY                estor.lib

LANG SC

VENDORID               0
SECUREID               0xEA7408AF
CAPABILITY             ReadUserData
// End of File

```

The HelloUIQ project specification file is very similar, with small differences in the name of the target file, the source files and the specific libraries used – for example the UIQ application does not use S60's `avkon.lib`, but lists `qikon.lib`, the equivalent one in UIQ, instead.

The `TARGET`, `TARGETTYPE`, `SOURCE`, `SOURCEPATH`, `USERINCLUDE`, `SYSTEMINCLUDE` and `LIBRARY` entries that are needed in a console application's MMP file are described in Chapter 1. The S60 application has further requirements in the MMP file:

- `UID` – specifies the application's three unique IDs. `UID1` is the same for all binary files and is supplied automatically. `UID2` indicates the kind of executable (0x0100039CE for an application) and `UID3` uniquely identifies our binary
- `START RESOURCE . . . END` blocks – detail the resources required by the application (see Chapter 13 for more details)
- `TARGETPATH` – specifies where project files should be released
- `LANG` – specifies one or more languages by means of two-character codes. The codes can be anything you like, as long as each language has a unique code, but for specific languages you are recommended to use the two-digit codes that are defined in the `TLanguage` enumeration in `e32const.h`, which is included in `e32std.h`. Since the default language code is `SC`, the `LANG` entry could safely be omitted from the above listing. See Chapter 13 for more information on multilingual applications
- `VENDORID` – the vendor's unique identifier (this is optional)
- `SECUREID` – the application's unique identifier; the HelloS60 application needs to use a platform security capability, so it requires a `SECUREID` (see Chapter 9)
- `CAPABILITY` – HelloS60 requires `ReadUserData` capability to enable reading a message from a file.

In a GUI application that only prints 'Hello World!' to the screen, without accessing the file system, there is no need for any capability – this is the case in the HelloUIQ code.

The Application Entry Point

Every Symbian OS application must implement two functions that are called by the framework as it starts the application.

Note that the Carbide.c++ wizard puts the functions discussed in this section into files specific to the GUI – for HelloS60, they are in `HelloS60.cpp`, whereas for HelloUIQ, they are in `HelloUIQApplication.cpp`.

The first is a non-leaving function which creates a new instance of the application class. This function is similar for both the HelloS60 and HelloUIQ applications. For the HelloS60 example, the code is:

```
LOCAL_C CPaApplication* NewApplication()
{
    return new CHelloS60Application;
}
```

The framework expects this factory function to have exactly this prototype. It constructs and returns an instance of the application class, or returns NULL if it cannot be instantiated.

The second function is the application's entry point function, identical for both S60 and UIQ applications, called `E32Main()`:

```
GLDEF_C TInt E32Main()
{
    return EikStart::RunApplication( NewApplication );
}
```

`E32Main()` calls `EikStart::RunApplication()`, passing as an argument a pointer to the factory function which creates an instance of the application class.

The Application Class

The application class represents the properties that are the same for every instance of the application. This includes the information specified in the registration file, for instance, the caption and the capabilities, and the UID. At a minimum, it must also implement two functions: `AppDllUid()` and `CreateDocumentL()`.

`AppDllUid()` returns the application's UID. The implementations for HelloS60 and HelloUIQ are similar.

S60 implementation of `AppDllUid()`:

```
TUid CHelloS60Application::AppDllUid() const
{
    // Return the UID for the HelloS60 application
    return KUidHelloS60App;
}
```

UIQ implementation of `AppDllUid()`:

```
TUid CHelloUIQApplication::AppDllUid() const
{
    // Return the UID for the HelloUIQ application
    return KUidHelloUIQApp;
}
```


The UID itself is usually defined in the HRH file or in a header file that contains the application's global information.

`AppDllUid()` is called by the framework just after calling `NewApplication()`. One of the reasons it needs the UID is to check whether there is a running instance of the application that it can switch to. The value returned must be the same as the UID3 value specified in the MMP file. Once the UID has been verified, the framework calls the `CreateDocumentL()` method.

`CreateDocumentL()` is a factory function for creating an object of the document class. Note that, although the application creates the document, the framework is responsible for destroying it.

S60 implementation of `CreateDocumentL()`:

```
CApaDocument* CHelloS60Application::CreateDocumentL()
{
    return (static_cast<CApaDocument*> (CHelloS60Document::NewL(*this)));
}
```

UIQ implementation of `CreateDocumentL()`:

```
CApaDocument* CHelloUIQApplication::CreateDocumentL()
{
    return CHelloUIQDocument::NewL(*this);
}
```

The Document

The really interesting code starts with the document class. In a file-based application, the document essentially represents the data in the file. You can do several things with that data, each of which makes different demands; printing it, for example, doesn't require an application but editing it does. We'll see more of this in Chapter 17.

If your application handles modifiable data, the application architecture requires your document to create an application user interface that is used to edit the document.

In a non-file-based application, you still have to code a function in the document class to create an application user interface, since it is the application UI that does the real work. Apart from creating the application UI, the document class for such an application is trivial; it implements the `CreateAppUiL()` function, inherited from the document's base class, `CEikDocument`. This function instantiates an object of the application UI class.

S60 implementation of `CreateAppUiL()`:

```
CEikAppUi* CHelloS60Document::CreateAppUiL()
```

```
{
return (static_cast <CEikAppUi*> (new (ELeave) CHelloS60AppUi));
}
```

UIQ implementation of `CreateAppUiL()`:

```
CEikAppUi* CHelloUIQDocument::CreateAppUiL()
{
return new (ELeave) CHelloUIQAppUi;
}
```

This function is called by the framework. Note that `CreateAppUiL()` only carries out first-phase construction. In other words, it does not call the application UI's `ConstructL()` – the framework is responsible for calling that. The framework also takes ownership of the application UI object, so the destructor of the document class does not need to destroy it.

The Application UI

Starting the application UI is the final step in bringing up the application framework. The GUI action proper starts with the application UI, which has two main roles:

- to get commands to the application
- to distribute keystrokes to controls, including the application's main view, which is owned by the application UI.

A command is simply an instruction without any parameters or any information about where it came from, which the program must execute.

The definition is deliberately vague, since commands can originate from a variety of places. In UIQ applications, commands usually originate from an application's menu. In other customized UIs, commands may originate from other sources. In the S60 UI, for example, which shows commonly-used commands on a toolbar, commands may come from a toolbar button.

It doesn't matter where the command comes from: the application UI receives it as a 32-bit command ID in `HandleCommandL()` and simply executes the appropriate code to handle the command.

Some commands can be executed immediately. For example, the Close command exits an application. (We don't ask the user, we simply save their data. If the user didn't want to exit, all they need to do is start the application again.) Other commands need further UI processing. A Find command, for example, needs some text to find, and possibly some indication of where to look. Handling such commands involves *dialogs*, which are the subject of Chapter 16.

Not all input to applications comes from commands. If you type the letter X into a word processor while you're editing, it doesn't generate a command, but a key event. If you tap on an entry in an Agenda, it doesn't generate a command, but a pointer event. Handling pointer and key events is the business of the application view and other controls in the application. Chapter 15 deals with drawing to controls and Chapter 18 covers key and pointer interaction.

The application UI class is constructed and returned by the document's `CreateAppUi()` function, described above. Its second-phase construction method, `ConstructL()`, is called by the framework and should include a call to the second-phase constructor of the base class (`CQikAppUi::ConstructL()` for UIQ and `CAknAppUi::ConstructL()` for S60 – each of which ultimately calls `CEikAppUi::BaseConstructL()`). Among other things, the base-class constructor method reads the application's resource file and creates the visible GUI elements. The `ConstructL()` method should also construct the application view.

S60 implementation of `ConstructL()`:

```
void CHelloS60AppUi::ConstructL()
{
    // Initialise app UI with standard value.
    BaseConstructL();
    iAppView = CHelloS60AppView::NewL( ClientRect() );
    // Create a file to write the text to
    ...
}
```

UIQ implementation of `ConstructL()`:

```
void CHelloUIQAppUi::ConstructL()
{
    // Calls ConstructL that initiates the standard values.
    CQikAppUi::ConstructL();
    // Create the view and add it to the framework
    CHelloUIQView* appView = CHelloUIQView::NewLC(*this);

    AddViewL(*appView);
    CleanupStack::Pop(appView);
}
```

In the UIQ implementation, `AddViewL()` registers the view with the system and adds it implicitly to the control stack, which enables the view to receive key and pointer input. The first view that is added becomes the default view for the application.

The ownership of the view is transferred to the application UI, and it is later unregistered and deleted automatically in the application UI's destructor. It is interesting to note that the derived application UI class in S60 takes ownership of the view object and is responsible for deleting

it, while the base class the UIQ implementation takes this responsibility and the destructor for the application UI class does nothing at all. The destructor for the S60 application UI class is as follows:

```
CHelloS60AppUi::~CHelloS60AppUi()
{
    if ( iAppView )
    {
        delete iAppView;
        iAppView = NULL;
    }
}
```

In S60, the application UI class also handles command events, which may originate from a range of sources such as menu bars, or the soft keys. This is done by implementing the `HandleCommandL()` function:

```
void CHelloS60AppUi::HandleCommandL( TInt aCommand )
{
    switch( aCommand )
    {
        case EEikCmdExit:
        case EAknSoftkeyExit:
            Exit();
            break;
        case ECommand1:
        {
            // Load a string from the resource file and display it
            ...
        }
        break;
        case ECommand2:
        {
            ...
        }
        break;

        default:
            Panic( EHelloS60Ui );
            break;
    }
}
```

Here a command is an instruction, identified by an integer value known as a command ID. Command IDs are enumerated constants defined in header files, which are traditionally given the extension `HRH`, while the other commands (`EEikCmdExit`, `EAknSoftkeyBack`) are defined by the system.

In the S60 implementation, the UI framework calls `HandleCommandL()` when either of the menu options available from the left softkey are selected (see Figure 11.4).

In UIQ, commands are usually handled by application views.



Figure 11.4 S60 emulator with the HelloS60 command menu open

The Application View

The final step of creating an application is creating the application views – the user's entry points into the application.

Anything that can draw to the screen is a *control*. Controls can also (optionally) handle key and pointer events. Note that controls don't have to be able to draw. They can be permanently invisible but that's quite unusual: a permanently invisible control clearly can't handle pointer events, but it can handle keys, as we'll see in Chapter 18.

The application UI is not a control. It owns one or more controls, including such obviously visible controls as the toolbar; we'll see a few others in the next few chapters.

In a typical UIKON application, you write one control yourself and make it the size of the client rectangle, the area of the screen remaining after the toolbar and so on have been taken into account. You then use that control to display your application data, and to handle key and pointer events (which are not commands).

The HelloS60 and HelloUIQ application views are controls whose purpose is to draw the text ‘Hello world!’ on the screen in a reasonably pleasing manner. They do not handle key or pointer events.

UIQ application view

In UIQ, views are the primary means of displaying data and handling commands and other user input events. Views can be configured and their initial commands and controls set in resource files using a number of resource structures, including `QIK_VIEW_CONFIGURATION`, `QIK_VIEW` and `QIK_VIEW_PAGE`. Each view has a class, derived from the framework class `CQikViewBase`, which handles construction and activation of the view, command handling and use of the view’s controls.

A view object is constructed in several stages to minimize the startup time of the application. The first stage constructs as little as possible. The static factory function, `NewL()`, follows the standard Symbian OS two-phase construction rules. The function instantiates the object, which calls `CQikViewBase`’s first-phase constructor and then calls the second-phase `ConstructL()` method, which calls the second-phase constructor `CQikViewBase::BaseConstructL()` to register the view with the view server to allow navigation between applications.

The next stage of construction occurs in the `ViewConstructL()` method which is called the first time the view needs to be activated. For applications which use multiple views, the `ViewConstructL()` method is valuable because it reduces application start-up time by only initializing the views that are used immediately the application starts. It also avoids wasting memory, particularly if some views are not used at all.

`ViewConstructL()` fully constructs the view by reading its configuration resource structure and initializing its controls. In HelloUIQ, it calls the `ViewConstructFromResourceL()` method of `CQikViewBase`.

In UIQ, all views are uniquely identified by a view ID (a `TVwsViewId` object), consisting of the UID of the application and an ID that uniquely identifies the view within the application. The view ID is returned by `CQikViewBase::ViewId()`, a pure virtual method which is called for each view by `CQikAppUi::AddViewL()`. All view classes must implement this method.

In UIQ, a command can be located on a softkey, in a toolbar or in a menu, depending on the interaction style of the phone. To allow for this flexibility, commands are defined in a more abstract way, rather than being explicitly coded as particular GUI elements such as menu items. This is usually done in a resource file using the `QIK_COMMAND_LIST` structure. Commands are associated with views and are handled within view classes. This is done by re-implementing the `HandleCommandL(CQikCommand& aCommand)` method defined by `CQikViewBase`:

```

void CHelloUIQView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        {
            // Just issue simple info messages to show that
            // the commands have been selected
            case EHelloUIQInfoPrint1Cmd:
            {
                // Shows an infoprint
                iEikonEnv->InfoMsg(R_INFOPRINT1_TEXT);
                break;
            }
            case EHelloUIQInfoPrint2Cmd:
            {
                ...
            }
            case EHelloUIQInfoPrint3Cmd:
            {
                ...
            }
            // Go back and exit command will be passed to the CQikViewBase
            // to handle.
            default:
                CQikViewBase::HandleCommandL(aCommand);
                break;
        }
    }
}

```

Note that UIQ applications do not provide an exit command. When you start an application, it remains running and persistent; you do not close it but simply switch away from it to use other applications. However, it is useful to provide an exit command in debug builds, to test that the application exits cleanly without leaking memory.

S60 application view

The S60 application view derives from `CCoeControl`. The application view is fully constructed by the static factory function, `NewL()`, which instantiates the object and then calls the second-phase `ConstructL()` method to perform any initialization which may leave.

```

void CHelloS60AppView::ConstructL( const TRect& aRect )
{
    // Create a window for this application view
    CreateWindowL();
    // Set the windows size
    SetRect( aRect );
    // Activate the window, which makes it ready to be drawn
    ActivateL();
}

```

The `ConstructL()` method calls `CreateWindowL()` to create the view's associated window, then `SetRect()` to set the area on the screen

that the view will occupy and `ActivateL()` to indicate that the view is ready to draw itself.

The Resource File

As regards code, we now have everything we need to start the application framework. We do, however, need an application resource file and registration files.

In this section, we take a brief look at the resource file. This will give us a pretty good idea of how resource files work – good enough to be able to read resource files and guess what they mean. For more details, have a look at Chapter 13.

In the code we've examined so far, we have already seen references to things that must be implemented in the application's resource file:

- strings `R_INFOPRINT1_TEXT`, `R_INFOPRINT2_TEXT`, etc.
- enumerated constants such as `EHelloUIQInfoPrint1Cmd`.

The other things we haven't seen in the code are the definitions necessary to construct some aspects of the application's GUI, such as the menu and any shortcut keys. They are defined in the resource file as well.

Let's look through the resource file to see exactly what is defined and how. First, there's some boilerplate material:

HelloS60 resource file (`HelloS60.rss`):

```
NAME HELL
#include <eikon.rh>
#include <avkon.rh>
#include <avkon.rsg>
#include <appinfo.rh>
#include "HelloS60.rls"
#include "HelloS60.hrh"
```

HelloUIQ resource file (`HelloUIQ.rss`):

```
NAME HELL
#include <eikon.rh>
#include <eikon.rsg>
#include <qikon.rh>
#include <qikon.hrh>
#include <uikon.rh>
#include <uikon.hrh>
#include <QikCommand.rh>
#include "HelloUIQ.rls"
#include "HelloUIQ.hrh"
```

Resource files must contain a `NAME` statement, which comes before any `RESOURCE` statements. It has to be distinct from the names used by UIKON and one or two other system components. Any capitalized

four-letter name will do, but it is common to use an abbreviation of the resource file name.

The `#include` statements load definitions of structures and constants that are used within the resource file. The final `#include` refers to the application's HRH files. This contains the enumerated constants for the application's commands. We need those commands in the C++ file so we can tell which command has been issued; we also need them here so that we can associate the commands with the right menus and shortcut keys.

After the `NAME` and `#include` lines, every UIKON resource file begins with three unnamed resources as follows:

S60 unnamed resources:

```
RESOURCE RSS_SIGNATURE { }

RESOURCE TBUF r_default_document_name
{
    buf="HEWB";
}

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

UIQ unnamed resources:

```
RESOURCE RSS_SIGNATURE { }
RESOURCE TBUF { buf = "HelloUIQ"; }
RESOURCE EIK_APP_INFO { }
```

The `RSS_SIGNATURE` structure allows us to specify version information and the `TBUF` specifies a friendly name for the application's resource file.

Of more interest is the `EIK_APP_INFO` resource. In the S60 file, it identifies the symbolic resource IDs of the HelloS60 menu and softkeys, defined as a Control Button Array (CBA).

The Registration File

The application needs to be registered with the system before it is actually visible to the user. It is then listed by the application launcher in its list and icon views. The registration file is essential for registering the application; however for a full registration the following files (see Figure 11.5) are required:

- one registration (`_reg`), file
- at least one localization (`_loc`) file, one for each language
- at least one multibitmap file (`.mbm`) containing several bitmaps.

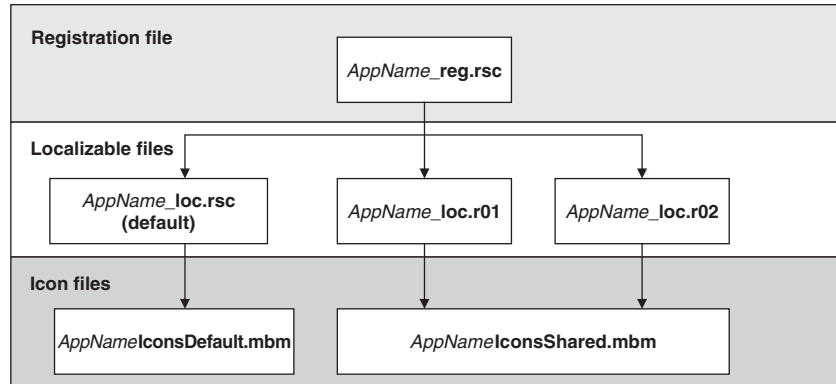


Figure 11.5 Registration files

HelloUIQ_reg.rss:

```

#include <AppInfo.rh>
UID2 KUidAppRegistrationResourceFile
UID3 0xE65A7653 // application UID
RESOURCE APP_REGISTRATION_INFO
{
    app_file = "HelloUIQ"; // filename of application binary
                        // (minus extension)
    // Specify the location of the localisable icon/caption
    // definition file
    localisable_resource_file = "\\Resource\\Apps\\HelloUIQ_loc";
}
  
```

HelloS60_reg.rss:

```

#include "HelloS60.hrh"
#include "HelloS60.rls"
#include <appinfo.rh>
#include <HelloS60.rsg>
UID2 KUidAppRegistrationResourceFile
UID3 _UID3
RESOURCE APP_REGISTRATION_INFO
{
    app_file="HelloS60";
    localisable_resource_file = qtn_loc_resource_file_1;
    localisable_resource_id = R_LOCALISABLE_APP_INFO;
    embeddability = KAppNotEmbeddable;
    newfile = KAppDoesNotSupportNewFile;
}
  
```

The `AppInfo.rh` file needs to be included as it contains the structure definition of the resource file. The `UID2` should always be `KUidAppRegistrationResourceFile` and `UID3` must correspond to the unique application UID.

The `APP_REGISTRATION_INFO` structure is the resource structure that defines how the application looks and behaves and describes its services. For both applications, we create a link between the registration file and the application executable, using the `app_file` structure; we also create links to the localizable application information files using the `localisable_resource_file` structure. This is described further in Chapter 13.

Summary

In this chapter, we've looked at:

- the Symbian OS Application framework
- an overview of the S60 and UIQ application frameworks; how S60 and UIQ applications are put together and how they interact with UIKON framework provided by Symbian
- the basic code you need to start writing graphical applications for Symbian OS.

12

A Simple Graphical Application

After a series of chapters describing individual aspects of Symbian OS, it is time to pull a few threads together. In Chapter 11, we described a minimal GUI application that simply showed how the UI framework and the application UI fit together. Now we look at a complete, simple, but non-trivial example application that forms a basis for the examples used in several of the following chapters.

In this chapter, we cover some aspects of application design and introduce the idea of how to program a user interface that does not assume a particular screen size. This latter point is important if you want to target your application at multiple user interfaces, as it helps speed up your porting efforts. In addition, we show how an application can use the principles described in Chapter 7 to save and restore its persistent data.

Before looking at the application's structure and behavior in more detail, we start with a general introduction to the Noughts and Crosses (otherwise known as Tic-Tac-Toe) application. Figure 12.1 shows the application running on a phone using the S60 user interface.

This application runs a game of noughts and crosses in which two players take turns to make their moves. The object of the game is to place three noughts (or crosses) in any row, column or diagonal before your opponent manages to do so.

We've implemented the game in a way that minimizes the differences needed to run on phones using the S60 and UIQ user interfaces. Depending on the nature of the phone it is running on, you can make a move either by using a combination of the cursor and pushbutton keys or by tapping on the screen.

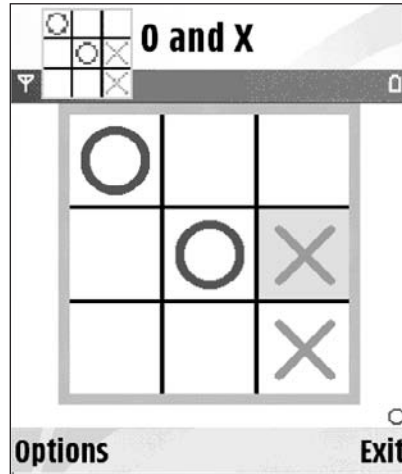


Figure 12.1 The Noughts and Crosses application

We've also kept the logic relating to the game itself very simple so that the bulk of the code illustrates issues of a general nature, relevant to most applications. In particular, we've designed the layout of the display in a way that can easily be adapted to show a rectangular grid with any number of tiles and to display on phones with a variety of screen formats. The grid layout is defined by a set of constants in `oandxdefs.h`:

```
static const TInt KTilesPerRow   = 3;
static const TInt KTilesPerCol   = 3;
static const TInt KNumberOfTiles = (KTilesPerRow*KTilesPerCol);
```

All the application's layout and drawing code is based on these values. By changing them, the layout can be adapted for games using differently sized and shaped boards.

The same header file also defines values that are specific to the Noughts and Crosses game, including the constant `KTilesPerSide` (which, unlike the earlier values, assumes that the board layout is square) and the states that can be associated with each tile:

```
static const TInt KTilesPerSide = 3;

enum TTileState
{
    ETileBlank   = 0,
    ETileNought  = 1,
    ETileCross   = 4
};
```

The state values are chosen so that `ETileCross` is greater than the product of `ETileNought` with either `KTilesPerRow` or `KTilesPerCol`, which helps with the calculation of a winner.

This file also defines the height of the application's status window, used to show whose turn it is to play:

```
static const TInt KStatusWinHeight = 20; // in pixels
```

In Chapter 11 you saw that the simplest Symbian OS graphical application needs instances of four classes: the application, the document, the application UI and a view. For this game, we've added two more classes: a controller and a model, or engine. Since the game control logic is so simple, we could have implemented it entirely within the application UI class, rather than defining separate controller and engine classes. However, we've chosen to add these classes from the start, in preparation for the more complex form of the game that is developed in Chapter 20.

The overall structure of the game is shown in Figure 12.2.

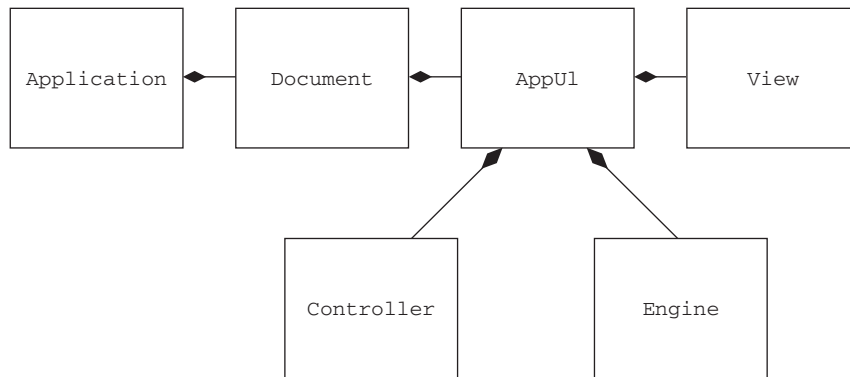


Figure 12.2 Structure of the Noughts and Crosses application

We cover the S60 version of the game first and then look at the changes that are needed to create the UIQ version.

12.1 Implementing the Game on S60

The Application Class

The S60 class definition for the Noughts and Crosses application class is listed below. As you can see, it is very similar to the corresponding class in Chapter 11's basic application: it implements `AppDllUid()`, to report the application's UID to the application framework, and `CreateDocumentL()`.

```

class COandXApplication : public CAknApplication
{
public:
    // From CAppApplication
    TUid AppDllUid() const;

protected:
    // From CEikApplication
    CAppDocument* CreateDocumentL();
};

```

This is fairly typical for applications. It is rare for an application class to override any of the other application class member functions.

The Document Class

The class definition for the Noughts and Crosses document class, listed below, provides the implementation of `CreateAppUiL()`, which is always required. In addition, it implements persistent storage of the game's state in the `StoreL()` and `RestoreL()` functions, using the principles described in Chapter 7.

```

class COandXDocument : public CAknDocument
{
public:
    static COandXDocument* NewL(CEikApplication& aApp);
    virtual ~COandXDocument();

    // From CEikDocument
    CEikAppUi* CreateAppUiL();
    void StoreL(CStreamStore& aStore, CStreamDictionary& aStreamDic) const;
    void RestoreL(const CStreamStore& aStore,
        const CStreamDictionary& aStreamDic);
    CFileStore* OpenFileL(TBool aDoOpen, const TDesC& aFilename, RFs& aFs);
private:
    COandXDocument(CEikApplication& aApp);
private:
    COandXAppUi* iAppUi;
};

```

The implementations of `StoreL()` and `RestoreL()` delegate to the application UI class the actual task of storing and restoring the application's data, but they retain the responsibility for associating the application's UID with the ID of the stream in which the data is stored.

```

void COandXDocument::StoreL(CStreamStore& aStore,
    CStreamDictionary& aStreamDic) const
{
    TStreamId id = iAppUi->StoreL(aStore);
}

```

```

        aStreamDic.AssignL(KUidOandXApp, id);
    }

void COandXDocument::RestoreL(const CStreamStore& aStore,
                             const CStreamDictionary& aStreamDic)
{
    TStreamId id = aStreamDic.At(KUidOandXApp);
    iAppUi->RestoreL(aStore, id);
}

```

The application framework provides support for an application to save its document data in an application-specific direct file store and, in UIQ, it is sufficient just to implement the two functions listed above.

In contrast, S60 applications are assumed not to be document-based and the default behavior is that the application framework never calls these two functions. In order to restore the support, you need to override the `OpenFileL()` function to call `CEikDocument`'s implementation of `OpenFileL()`:

```

CFileStore* COandXDocument::OpenFileL(TBool aDoOpen,
                                       const TDesC& aFilename, RFs& aFs)
{
    return CEikDocument::OpenFileL(aDoOpen, aFilename, aFs);
}

```

The Application UI Class

The application UI class normally forms the core of an application, and the `COandXAppUi` class is no exception. In addition to its standard role of handling much of the interaction with the user, `COandXAppUi` owns the application's view, controller and engine instances and supplies the central logic for saving and restoring the application's persistent data.

```

class COandXAppUi : public CAknAppUi
{
public:
    COandXAppUi();
    virtual ~COandXAppUi();

    // New functions
    void ReportWhoseTurn();
    void ReportWinnerL(TInt aWinner);

    // From CEikAppUi, for persistent data
    TStreamId StoreL(CStreamStore& aStore) const;
    void RestoreL(const CStreamStore& aStore, TStreamId aStreamId);
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);

private:
    // From MEikMenuObserver

```



```

void DynInitMenuPanel(TInt aResourceId, CEikMenuPane* aMenuPane);
// From CEikAppUi
void HandleCommandL(TInt aCommand);
void ConstructL();

public:
    // AppUi owns the engine, controller and application view.
    COandXEngine* iEngine;
    COandXController* iController;
private:
    COandXAppView* iAppView;
    // Set if the application view was added to the control stack.
    TBool iStacked;
};

```

As you can see from the above class definition, part of the user interaction consists of reporting significant events such as whose turn it is and, at the end of a game, who (if anyone) has won. In the case of the S60 version, the application UI class also handles all aspects of using menu commands, including dynamically setting the menu content according to the current state of the game. We have more to say about using menus later in this chapter.

The second-phase constructor and the destructor make it clear that the application UI class owns the engine, the controller and the view, which are all created in a leave-safe way:

```

void COandXAppUi::ConstructL()
{
    BaseConstructL(EAknEnableSkin);
    iEngine = COandXEngine::NewL();
    iController = COandXController::NewL();
    iAppView = COandXAppView::NewL(ClientRect());
    AddToStackL(iAppView); // Enable keypresses to the view
    iStacked = ETrue;
    ReportWhoseTurn();
}

COandXAppUi::~COandXAppUi()
{
    if (iStacked)
    {
        RemoveFromStack(iAppView);
    }
    delete iAppView;
    delete iController;
    delete iEngine;
}

```

It is worth noting that, it is the responsibility of the application UI class to ensure that the view is added to the control stack, via a call to `AddToStackL()`, to ensure that it has the opportunity to receive and process keypresses. Since this call can fail, it is necessary to set a flag if

it succeeds, so that the flag can be tested (in this case, since there is only one view, in the destructor) before the view is removed from the control stack.

This application uses two different ways to report events to the user. Reporting whose turn it is to play, which happens between every move in the game, is delegated to the view:

```
void COandXAppUi::ReportWhoseTurn()
{
    iAppView->ShowTurn();
}
```

If either player wins, which obviously happens no more than once per game, the application UI class reports the winner by means of a UI-specific dialog that displays the appropriate text, read from the application's resource file. The S60 version uses an information note:

```
void COandXAppUi::ReportWinnerL(TInt aWinner)
{
    TBuf<MaxInfoNoteTextLen> text;
    iEikonEnv->ReadResource(text, aWinner==ETileCross
        ? R_OANDX_X_WINS : R_OANDX_O_WINS);
    CAknInformationNote* infoNote = new (ELeave) CAknInformationNote;
    infoNote->ExecuteLD(text);
}
```

The application's persistent data is saved and restored via the `StoreL()` and `RestoreL()` functions of the application UI class which, you may recall, are called from the application's document class.

```
TStreamId COandXAppUi::StoreL(CStreamStore& aStore) const
{
    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(aStore);
    stream << *this; // alternatively, use ExternalizeL(stream)
    stream.CommitL();
    CleanupStack::PopAndDestroy();
    return id;
}

void COandXAppUi::RestoreL(const CStreamStore& aStore,
    TStreamId aStreamId)
{
    RStoreReadStream stream;
    stream.OpenLC(aStore, aStreamId);
    stream >> *this; // alternatively use InternalizeL(stream)
    CleanupStack::PopAndDestroy();
}
```

`Store()` and `Restore()`, respectively, call the `ExternalizeL()` and `InternalizeL()` functions of the application UI class via the

<< and >> operators. As you can see from the following listing, the application UI class itself has no persistent data. The functions deal directly with the view's data, which is simply the ID of the control that currently has focus, and leave the engine and the controller to take care of their own data.

```
void COandXAppUi::ExternalizeL(RWriteStream& aStream) const
{
    iEngine->ExternalizeL(aStream);
    iController->ExternalizeL(aStream);
    aStream.WriteInt8L(iAppView->IdOfFocusControl());
}

void COandXAppUi::InternalizeL(RReadStream& aStream)
{
    iEngine->InternalizeL(aStream);
    iController->InternalizeL(aStream);
    ReportWhoseTurn();
    iAppView->MoveFocusTo(aStream.ReadInt8L());
}
```

The S60 application UI class is also responsible for handling menu commands, via the `DynInitMenuPaneL()` and `HandleCommandL()` functions. We discuss them – and the differences between the ways they are handled in S60 and UIQ – later in this chapter.

Finally, it is worth pointing out that we provide access to the controller and engine via global static functions, declared (in `oandxappui.h`) as:

```
GLREF_C COandXAppUi* OandXAppUi();
GLREF_C COandXController& Controller();
GLREF_C COandXEngine& Engine();
```

The implementations of these functions are:

```
GLDEF_C COandXAppUi* OandXAppUi()
{
    return static_cast<COandXAppUi*>(CEikonEnv::Static()->AppUi());
}

GLDEF_C COandXController& Controller()
{
    return *OandXAppUi()->iController;
}

GLDEF_C COandXEngine& Engine()
{
    return *OandXAppUi()->iEngine;
}
```

We have chosen to use global functions in this example because they accurately represent the intention that these functions should be

accessible from anywhere in the program. Furthermore, they do not need verbose source text to call them, which is useful in example code. We would not use them in production code, since there is no control of when and from where they are called. Additionally, calling a global function involves accessing thread-local storage, which is slow compared with other function calls. We could improve the efficiency by implementing them as static members of a class, but this relies on the availability of writable static data and so would not work on all phones. The best solution would be to implement them as normal member functions of, for example, the application UI class and pass a (non-owning) reference to the constructor of each class that needs to access them. We have not done that in this example because it would add a significant amount of incidental complexity to each of the constructors concerned.

The Controller Class

The controller class, whose definition is listed below, is responsible for managing the state of the game and its logic, including updating the data in the engine class.

```
class COandXController : public CBase
{
public:
    static COandXController* NewL();
    virtual ~COandXController();
enum TState
{
    ENewGame, EPlaying, EFinished
};
// state
inline TBool IsNewGame() const;
// game control
void Reset();
// stream persistence
void ExternalizeL(RWriteStream& aStream) const;
void InternalizeL(RReadStream& aStream);

TBool HitSquareL(TInt aIndex);
TBool IsCrossTurn() const;
void SwitchTurn();
private:
    void ConstructL();
private: // private persistent state
    TState iState;
    TBool iCrossTurn;
};
```

In this stand-alone, non-communicating, version, the game state is simply one of the three values `ENewGame`, `EPlaying` or `EFinished`, together with a flag that indicates which player is next to play. The only externally accessible state-query functions needed are `IsCrossTurn()`,

which simply returns the value of `iCrossTurn`, and `IsNewGame()`, coded as:

```
inline TBool COandXController::IsNewGame() const
{ return iState==ENewGame; }
```

The central logic of the controller is contained in the `HitSquareL()` function:

```
TBool COandXController::HitSquareL(TInt aIndex)
{
    if (iState == EFinished)
    {
        return EFalse;
    }
    if (iState == ENewGame)
    {
        iState = EPlaying;
    }
    if (Engine().TryMakeMove(aIndex, IsCrossTurn()))
    {
        SwitchTurn();
        TInt winner = Engine().GameWonBy();
        if (winner)
        {
            iState = EFinished;
            OandXAppUi()->ReportWinnerL(winner);
        }
        return ETrue;
    }
    return EFalse;
}
```

This function is called from the application's board view whenever a player attempts to make a move. It disallows the move if the game state is `EFinished`; if this is the first move in the game (i.e. the game state is `ENewGame`), it sets the state to `EPlaying`. If the game is in play, it calls the `TryMakeMove()` function of the engine class to check if the move is valid; if so, it records the move and switches whose turn it is. A call to the `GameWonBy()` function of the engine class checks if the last move resulted in a win by either player and, if it did, the game state is set to `EFinished` and a call to the `ReportWinner()` function of the application UI class displays the result.

In its current form, `HitSquareL()` does not check for, or report, a drawn game. If there is no winner, the game continues until all tiles contain a nought or a cross (or a player uses a menu item to start a new game). The functionality for a drawn game is added in the communicating version of the game, described in Chapter 20.

The controller is reset for a new game by calling `Reset()`, which cancels the current game, clears the board (by calling the `Reset()` function of the engine class) and sets Noughts as the current player:

```

void COandXController::Reset()
{
    Engine().Reset();
    iState = ENewGame;
    if (IsCrossTurn())
    {
        SwitchTurn();
    }
}

```

The `SwitchTurn()` function simply negates the flag held in `iCrossTurn` and calls the `ReportWhoseTurn()` function of the application UI class:

```

void COandXController::SwitchTurn()
{
    iCrossTurn = !iCrossTurn;
    OandXAppUi() -> ReportWhoseTurn();
}

```

Persistence of the state of the controller class is handled by the `ExternalizeL()` and `InternalizeL()` functions, which are called from the application UI class. Their actions are, respectively, to write and read the controller's member data to or from the passed stream:

```

void COandXController::ExternalizeL(RWriteStream& aStream) const
{
    aStream.WriteUInt8L(iState);
    aStream.WriteInt8L(iCrossTurn);
}

void COandXController::InternalizeL(RReadStream& aStream)
{
    iState = static_cast<TState>(aStream.ReadUInt8L());
    iCrossTurn = static_cast<TBool>(aStream.ReadInt8L());
}

```

The controller is created via a call to its static `NewL()` function, which is coded as:

```

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

```

The second-phase constructor is simply:

```
void COandXController::ConstructL()
{
    Reset();
}
```

We have coded it in this way to preserve the standard form of class construction, as described in Chapter 4. However, since – in this case – `ConstructL()` does not leave, it is worth pointing out that we could have simplified matters by not defining a `ConstructL()` function and doing everything in the `NewL()` function.

The Engine Class

The engine class, whose class definition is listed below, represents the game data and contains the functions that operate on that data. Since Noughts and Crosses is a simple game, the engine class is correspondingly straightforward. The class itself has no knowledge of the overall state of the game, nor of whose turn it is to make a move. As you saw in the previous section, both of these are the responsibility of the controller.

```
class COandXEngine : public CBase
{
public:
    static COandXEngine* NewL();
    virtual ~COandXEngine();

    void Reset();
    TInt TileStatus(TInt aIndex) const;
    TBool TryMakeMove(TInt aIndex, TBool aCrossTurn);
    TTileState GameWonBy() const;
    // persistence
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);
private:
    COandXEngine();
    TInt TileState(TInt aX, TInt aY) const;
private:
    TFixedArray<TTileState, KNumberOfTiles> iTileStates;
};
```

The current state of the board is represented by a simple array of tile states, one for each component tile. Each tile state contains one of the three values `ETileBlank`, `ETileNought` or `ETileCross`.

The key functions are `TryMakeMove()` and `GameWonBy()`, both of which are called from the controller. `TryMakeMove()` only allows a move to be made if the corresponding tile is initially blank. If the move is allowed, the function sets the tile-state variable corresponding to the

tile to either `ETileCross` or `ETileNought`, depending on the value passed in `aCrossTurn`.

```
TBool COandXEngine::TryMakeMove(TInt aIndex, TBool aCrossTurn)
{
    if (iTileStates[aIndex] == ETileBlank)
    {
        iTileStates[aIndex] = aCrossTurn ? ETileCross : ETileNought;
        return ETrue;
    }
    return EFalse;
}
```

`GameWonBy()` checks if there is a winner and, if so, whether the winner is the Nought player or the Cross player. It does so by determining if there is a horizontal, vertical or diagonal line of three noughts or three crosses. If there is such a line, it returns either `ETileNought` or `ETileCross`; otherwise it returns zero, to indicate that, as yet, there is no winner.

```
TTileState COandXEngine::GameWonBy() const
{
    const TInt KNoughtWinSum = KTilesPerSide * ETileNought;
    const TInt KCrossWinSum = KTilesPerSide * ETileCross;

    // is there a row or column of matching tiles?
    for (TInt i = 0; i < KTilesPerSide; ++i)
    {
        TInt rowSum = 0;
        TInt colSum = 0;
        for (TInt j = 0; j < KTilesPerSide; ++j)
        {
            rowSum += TileState(j, i);
            colSum += TileState(i, j);
        }
        if (rowSum == KNoughtWinSum || colSum == KNoughtWinSum)
            return ETileNought;
        if (rowSum == KCrossWinSum || colSum == KCrossWinSum)
            return ETileCross;
    }

    // is there a diagonal of matching tiles?
    TInt blTrSum = 0; // bottom left to top right
    TInt tlBrSum = 0; // top left to bottom right
    for (TInt i = 0; i < KTilesPerSide; ++i)
    {
        tlBrSum += TileState(i, i);
        blTrSum += TileState(i, KTilesPerSide - 1 - i);
    }
    if (blTrSum == KNoughtWinSum || tlBrSum == KNoughtWinSum)
        return ETileNought;
    if (blTrSum == KCrossWinSum || tlBrSum == KCrossWinSum)
        return ETileCross;
    return ETileBlank; // No winner
}
```


As with the controller class, the `ExternalizeL()` and `InternalizeL()` functions of the engine class, called from the application UI class, simply write and read the state – in this case, the values of each of the board’s tiles – to and from the appropriate stream:

```
void COandXEngine::ExternalizeL(RWriteStream& aStream) const
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        aStream.WriteInt8L(iTileStates[i]);
    }
}

void COandXEngine::InternalizeL(RReadStream& aStream)
{
    for (TInt i = 0; i<KNumberOfTiles; i++)
    {
        iTileStates[i] = static_cast<TTileState>(aStream.ReadInt8L());
    }
}
```

Like the controller class, the engine class is constructed by means of a call to its static `NewL()` function and, again, there are no potentially leaving calls to be made from a second-phase constructor. For the engine class, we’ve chosen to illustrate an alternative way of implementing construction in such a case, where the non-leaving calls are made from the constructor rather than from the `NewL()` function:

```
COandXEngine* COandXEngine::NewL()
{
    return new(ELeave) COandXEngine;
}

COandXEngine::COandXEngine()
{
    Reset();
}
```

The View Class

Views and the view architecture are discussed in Chapter 14 and the detailed behavior of controls is described in Chapters 15, 17 and 18. In this chapter, therefore, we only briefly describe the general aspects of controls and views, concentrating on those features that are specific to the Noughts and Crosses application.

An application’s view is an instance of a window-owning control (as defined in Chapter 15) and derives, directly or indirectly, from the `CCoeControl` class. For the Noughts and Crosses application, the view is a compound control, with the view’s area being tiled with a number of subsidiary controls.

We've chosen to implement the S60 view class by deriving directly from `CCoeControl`. S60 applications can derive from more specific classes, such as `CAknView`, but we decided not to for this application in order to reduce the differences between the S60 and UIQ versions, and because many of the S60-specific views are more suited to text displays. The view also inherits from an `MViewCmdHandler` interface class, which we describe later.

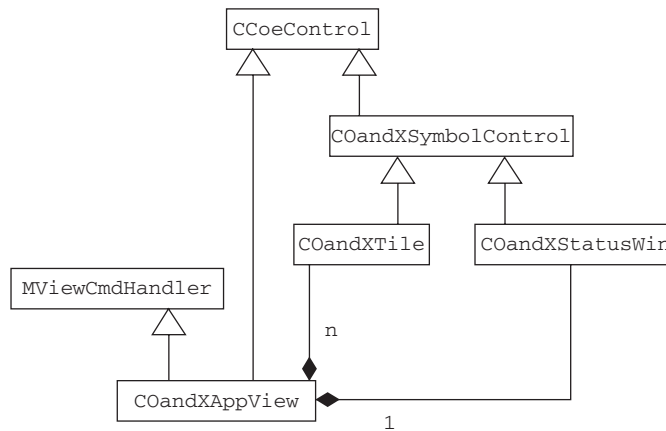


Figure 12.3 The View class of the Noughts and Crosses application

Figure 12.3 shows that all the components of the view are themselves also controls, in the sense that they all derive from `CCoeControl`. The view owns a single status-display control and a series of tiles.

Both the status display control and the tiles need to display either a nought or a cross, so we've chosen to implement them as subclasses of a generic symbol-drawing control, which knows how to draw those symbols. The two subclasses simply add the specification of the size and location, within their own area, of the symbol.

As you can see from the class definition of the symbol-drawing base class, its only member function is to draw a nought or a cross symbol:

```

class COandXSymbolControl : public CCoeControl
{
protected:
    void DrawSymbol(CWindowGc& aGc, const TRect& aRect,
                    TBool aDrawCross) const;
};
  
```

The implementation of this function is:

```

void COandXSymbolControl::DrawSymbol(CWindowGc& aGc,
    const TRect& aRect, TBool aDrawCross) const
  
```

```

{
    TRect drawRect(aRect);

    // Shrink by about 15%
    drawRect.Shrink(aRect.Width()/6, aRect.Height()/6);

    // Pen size set to just over 10% of the shape's size
    TSize penSize(aRect.Width()/9, aRect.Height()/9);
    aGc.SetPenSize(penSize);
    aGc.SetPenStyle(CGraphicsContext::ESolidPen);

    if (aDrawCross)
    {
        aGc.SetPenColor(KRgbGreen);

        // Cosmetic reduction of cross size by half the line width
        drawRect.Shrink(penSize.iWidth/2, penSize.iHeight/2);

        aGc.DrawLine(drawRect.iTl, drawRect.iBr);
        TInt temp;
        temp = drawRect.iTl.iX;
        drawRect.iTl.iX = drawRect.iBr.iX;
        drawRect.iBr.iX = temp;
        aGc.DrawLine(drawRect.iTl, drawRect.iBr);
    }
    else // draw a circle
    {
        aGc.SetPenColor(KRgbRed);
        aGc.SetBrushStyle(CGraphicsContext::ESolidBrush);
        aGc.DrawEllipse(drawRect);
    }
};

```

You see this code again in Chapter 15, where the more detailed aspects of drawing this control are discussed; here, it is sufficient to note that it is written to be as independent as possible of the size or position of the symbol to be drawn. The fact that the same code successfully draws both the large symbols in the tiles and the much smaller symbols in the status area shows that this approach is both feasible and sensible.

The symbol is centered in the rectangle (which is always a square) passed to the function in `aRect`, but shrunk to leave a small border, and the pen width is set to match the size of the symbol. Note that the cross symbol is shrunk by a further small amount since, if the circles and crosses were both drawn to exactly the same size, the cross would look larger to the eye. This kind of attention to detail can make a large difference to the appearance of any application.

The tile and status window control classes both derive from `COandX-SymbolControl`:

```

class COandXTile : public COandXSymbolControl
{
public:
    COandXTile();

```

```

~COandXTile();
void ConstructL(RWindow& aWindow);
// New function
void SetOwnerAndObserver(COandXAppView* aControl);
// From CCoeControl
TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent,
                             TEventCode aType);
TCoeInputCapabilities InputCapabilities() const;
protected:
    void FocusChanged(TDrawNow aDrawNow);
    void HandlePointerEventL(const TPointerEvent& aPointerEvent);
private:
    void Draw(const TRect& aRect) const;
    // New function
    void TryHitL();
private:
    MViewCmdHandler* iCmdHandler;
};

class COandXStatusWin : public COandXSymbolControl
{
public:
    static COandXStatusWin* NewL(RWindow& aWindow);
    ~COandXStatusWin();
private:
    COandXStatusWin();
    void ConstructL(RWindow& aWindow);
    void Draw(const TRect& aRect) const;
}

```

Note that, even though S60 devices do not normally use touch screens, we include code that handles input from keys – via `OfferKeyEventL()` – and any pointer device – via `HandlePointerEventL()`. It is quite safe to do this: if a particular device does not support one or the other input device, the relevant code simply is not called. Including both options makes it easier to port the code to different devices.

The status window's `Draw()` function simply calculates the square in which to draw the symbol and then calls `DrawSymbol()`, obtaining the flag indicating which symbol to draw by calling the `IsCrossTurn()` function of the controller class:

```

void COandXStatusWin::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc();
    TRect boxRect = Rect();
    gc.Clear(boxRect);
    TInt boxHeight = boxRect.iBr.iY - boxRect.iTl.iY;
    boxRect.iTl.iX = boxRect.iBr.iX - boxHeight;
    DrawSymbol(gc, boxRect, Controller().IsCrossTurn());
}

```

The `Draw()` function for each tile works in a similar way:

```
void COandXTile::Draw(const TRect& /*aRect*/) const
{
    TTileState tileType;
    tileType = iCmdHandler->TileStatus(this);

    CWindowGc& gc = SystemGc();
    TRect rect = Rect();

    if (IsFocused())
    {
        gc.SetBrushColor(KRgbYellow);
    }
    gc.Clear(rect);
    if (tileType!=ETileBlank)
    {
        DrawSymbol(gc, rect, tileType==ETileCross);
    }
}
```

In this case, the symbol to be drawn is determined by a call to the `TileStatus()` function of the view class, by a mechanism that is described later. The tile which currently has focus is drawn with a yellow background; the others are drawn with the default brush color, which is white.

During the game, a player places a nought or a cross in a particular tile either by tapping on it or by using cursor keys to move the highlight to that tile and pressing the phone's Selection or Action key. A pointer event is handled by the tile's `HandlePointerEventL()` function:

```
void COandXTile::HandlePointerEventL (const TPointerEvent& aPointerEvent)
{
    if (aPointerEvent.iType == TPointerEvent::EButton1Down)
    {
        TryHitL();
    }
}
```

A key event is handled by `OfferKeyEventL()`:

```
TKeyResponse COandXTile::OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                         TEventCode aType)
{
    TKeyResponse keyResponse = EKeyWasNotConsumed;
    if (aType!=EEEventKey)
    {
        return keyResponse;
    }
    switch (aKeyEvent.iCode)
    {
        case EKeyOK:
```

```

    TryHitL();
    keyResponse = EKeyWasConsumed;
    break;
default:
    keyResponse = EKeyWasNotConsumed;
    break;
}
return keyResponse;
}

```

In both cases, this results in a call to the tile's `TryHitL()` function, which is implemented as:

```

void COandXTile::TryHitL()
{
    if (iCmdHandler->TryHitSquareL(this))
    {
        DrawDeferred();
    }
}

```

`TryHitL()` calls the `TryHitSquareL()` function of the view class, using the same mechanism that is also used to call `TileStatus()`.

The `COandXAppView` class definition has 19 member functions, not counting those that it inherits from its base classes, but only the two mentioned above are called from a tile. As an encapsulation aid, we have chosen to declare these two functions, as pure virtual functions, in an `MViewCmdHandler` interface class:

```

class MViewCmdHandler
{
public:
    virtual TBool TryHitSquareL(const COandXTile* aControl) = 0;
    virtual TTileState TileStatus(const COandXTile* aControl) const = 0;
};

```

The view class inherits from this interface and provides the concrete implementations, as indicated by the bold portions of the `COandXAppView` class definition:

```

class COandXAppView : public CCoeControl, public MCoeControlObserver,
                    public MViewCmdHandler
{
public:
    static COandXAppView* NewL(const TRect& aRect);
    virtual ~COandXAppView();

    // From CCoeControl
    TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                TEventCode aType);

```

```

// new functions
void MoveFocusTo(const TInt aIndex);
TInt IdOfFocusControl();
void ShowTurn();
void ResetView();

private:
    COandXAppView();
    void ConstructL(const TRect& aRect);

    void SwitchFocus(TInt aFromIndex, CCoeControl* aToControl);
    void DrawComps(TRect& aRect) const;
    COandXTile * CreateTileL();

    // From CCoeControl
    void Draw(const TRect& aRect) const;
    void SizeChanged();
    TInt CountComponentControls() const;
    CCoeControl* ComponentControl(TInt aIndex) const;

    // From MCoeControlObserver
    void HandleControlEventL(CCoeControl* aControl, TCoeEvent aEventType);

    // From MViewCmdHandler
    TBool TryHitSquareL(const COandXTile* aControl);
    TTileState TileStatus(const COandXTile* aControl) const;

private:
    RPointerArray<COandXTile> iTiles; // View owns the tiles
    COandXStatusWin* iStatusWin;      // and its own status window.
    TRect iBoardRect; // Board area
    TRect iBorderRect; // Bounding rectangle for border
    TInt iTileSide;   // Tile dimension, allowing for line widths
                    // and border
};

```

Note that the functions are declared public in the interface class, but private within `COandXAppView`. By providing each tile with an `MViewCmdHandler*` pointer to the enclosing view, we expose only those two functions (they are not even available via a `COandXAppView*` pointer).

The interface class is a useful way of encapsulating the interaction between the control and the view or, indeed, the rest of the program. The control does not care about any of the view's internal details, provided that it correctly handles the request to play in a particular square and the querying of a square's content. Similarly, the view does not care whether the command originated with a key event, a pointer event, or some other kind of event – it just requires these functions to be called at the right time with the right parameters.

This is a useful technique, which is used frequently within a variety of Symbian OS APIs, and it is a good one to adopt for your programs, especially if they are more complicated than the Noughts and Crosses application. As a further illustration, think about the way commands

reach `HandleCommandL()`. The menu bar uses an `MEikMenuObserver` for this and does not otherwise care about the large API of the `CEikAppUi` class. The button bar uses the `MEikCommandObserver` interface. Similarly, command buttons use an `MCoeControlObserver` interface, which the button bar implements by converting button events into application commands.

`TryHitSquareL()` simply calls the controller's `HitSquareL()` function, to determine whether the attempt is successful:

```
TBool COandXAppView::TryHitSquareL(const COandXTile* aControl)
{
    return Controller().HitSquareL(Index(aControl));
}
```

It returns `ETrue` only if the square was previously empty and now contains a symbol, otherwise it returns `EFalse`.

As was mentioned earlier, `TileStatus()` is called from each tile when it needs to draw itself, in order to determine which, if any, symbol should be drawn in the tile. The function's action is simply to request the information from the engine:

```
TTileState COandXAppView::TileStatus (const COandXTile* aControl) const
{
    return Engine().TileStatus(Index(aControl));
}
```

The majority of the view's functions are described in Chapter 15, where they are used to illustrate the general behavior of controls. Here, we mention only `ShowTurn()`, which is called from the application UI class and the controller whenever there is a change in whose turn it is to play. It simply requests the status window to redraw itself:

```
void COandXAppView::ShowTurn()
{
    iStatusWin->DrawDeferred();
}
```

How `DrawDeferred()` works is explained in Chapter 17; here it is sufficient to know that it results in a call to the status window's `Draw()` function. As we saw earlier, there is no need for `ShowTurn()` to pass a parameter indicating whose turn it is, since the status window's `Draw()` function queries the controller directly for the symbol it needs to draw.

Command Menus

The application uses only two commands: one to specify whether the Nought player or the Cross player should make the first move, and one

to abandon the current game. The first of these is only relevant before a game starts and the second is only used while a game is in progress. In consequence, we've chosen to use a dynamic command menu, which shows only those commands that can sensibly be used, given the current state of the game. We've implemented the menu in a way that illustrates some of the options that are available for setting up and changing the content of a command menu. Although at least one of the commands, that to select the first player, could reasonably be implemented via a dialog, we've chosen not to do so, leaving the discussion of dialog use to Chapter 16.

For S60, the specification of the command menu starts in the resource script, with the definition of an `EIK_APP_INFO` resource:

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

This resource contains a pointer to the resource that contains the application's menu bar and a Control Button Array that defines the softkey labels – in this case, Options and Exit – that are to appear in the control pane.

The menu bar resource, in turn, specifies one or more `MENU_PANE` resources that list the menu items they contain. As in this case, an S60 menu bar normally contains only one menu pane. If you specify more than one menu pane, the S60 UI concatenates their contents into a single list.

```
RESOURCE MENU_BAR r_oandx_menubar
{
    titles =
    {
        MENU_TITLE
        {
            menu_pane = r_oandx_menu;
        }
    };
}
```

For this application, the menu pane specifies only one menu item; the second menu item is constructed dynamically.

```
RESOURCE MENU_PANE r_oandx_menu
{
    items =
    {
        MENU_ITEM
```

```

    {
        command = EOandXNewGame;
        txt = "New game";
    }
};
}

```

Each menu item specifies the text to be displayed and a non-zero value, unique within the application, used to identify the command. The command IDs are normally defined via an enumeration in the application's HRH file; for this application, they are defined, in `oandx.hrh`, as:

```

enum TOandXIds
{
    EOandXNewGame = 0x1000, // start value must not be 0
    EOandXSwitchTurn
};

```

The resource file specifies only the command ID and text for the New game command. The other command is added dynamically according to context, using one or other of the following two resources for its text:

```

RESOURCE TBUF r_oandx_o_moves_first
{
    buf = "Noughts move first";
}

RESOURCE TBUF r_oandx_x_moves_first
{
    buf = "Crosses move first";
}

```

It is worth noting here that a truly commercial application would read the text for these resources – and most other text within the resource script – from a separate file, in order to facilitate translation into other languages. We have not done so in order to make the meaning of these resources clearer in the text of this book.

S60 applications that need to modify the content of their command menus should supply an implementation of the `DynInitMenuPaneL()` function of the application UI class, which is called each time a menu is about to be displayed. The `aResourceId` parameter identifies the menu and `aMenuPane` points to a menu pane that has already been constructed from the relevant resource. For this application, the implementation is:

```

void COandXAppUi::DynInitMenuPaneL(TInt aResourceId,
                                    CEikMenuPane* aMenuPane)
{
    if (aResourceId == R_OANDX_MENU)

```

```

{
    if (iController->IsNewGame())
    {
        CEikMenuItem::SData item;
        iCoeEnv->ReadResource(item.iText,
            iController->IsCrossTurn() ? R_OANDX_O_MOVES_FIRST :
                                      R_OANDX_X_MOVES_FIRST);

        item.iCommandId = EOandXSwitchTurn;
        item.iFlags = 0;
        item.iCascadeId = 0;
        aMenuPane->AddMenuItemL(item);

        aMenuPane->DeleteMenuItem(EOandXNewGame);
    }
}
}

```

The function first checks that the resource ID is for the relevant menu, but does not modify the menu pane (which therefore displays the New game menu item) if a game is in progress. Otherwise, a new menu item is constructed, in a `CEikMenuItem::SData` data structure, using the relevant resource text, and added to the menu pane, and the New game item is deleted.

When the user selects a command, the S60 UI responds by calling the `HandleCommandL()` function of the application UI class. This is normally implemented via a switch statement, with each case corresponding to one of the possible command IDs:

```

void COandXAppUi::HandleCommandL(TInt aCommand)
{
    switch(aCommand)
    {
        {
            case EEikCmdExit:
            case EAknSoftkeyExit:
            {
                SaveL();
                Exit();
            }
            break;
            case EOandXNewGame:
            {
                iController->Reset();
                iAppView->ResetView();
            }
            break;
            case EOandXSwitchTurn:
            {
                iController->SwitchTurn();
            }
            break;
            default:
                break;
        }
    }
}

```

In the Noughts and Crosses application, each case, with the exception of the Exit command, corresponds to one of the command IDs specified in `oandx.hrh` and is handled simply by making the appropriate changes to the controller's state.

Depending on how the Exit command is initiated, it may have either of the standard command IDs shown in the above code. The application's response must always be to call `Exit()` but, in addition, it calls `SaveL()` to save the current game state in the application's persistent data.

12.2 Differences for UIQ 3

One of the major influences in the development of UIQ 3 was the need to make it easier for developers to support multiple form factors (both in terms of screen dimensions and use of portrait or landscape alignments) and a variety of input methods (touch screen or keyboard; menu-driven or softkey-driven interaction) from a single code base.

In order to facilitate such flexibility, all UIQ 3 applications need to derive their views from the new `CQikViewBase` or `CQikMultiPageViewBase` classes. The UIQ version of the Noughts and Crosses application derives its main view from the `CQikViewBase` view class (which is a subclass of `CCoeControl`) and not directly from `CCoeControl` itself. However, since it only has one view, it makes no specific use of the view architecture that is described in Chapter 14.

The Application UI Class

There is a small difference in the way that the application UI class creates and initializes the View. Remember that the `S60 ConstructL()` function of the application UI class is implemented as:

```
void COandXAppUi::ConstructL()
{
    BaseConstructL(EAknEnableSkin);
    iEngine = COandXEngine::NewL();
    iController = COandXController::NewL();
    iAppView = COandXAppView::NewL(ClientRect());
    AddToStackL(iAppView); // Enable keypresses to the view
    iStacked = ETrue;
    ReportWhoseTurn();
}
```

The destructor needed to take note of `iStacked` in order to determine whether or not it should remove the view from the control stack. For UIQ, the corresponding code is:

```
void COandXAppUi::ConstructL()
{
```

```

CQikAppUi::ConstructL(); // initiate the standard values
iEngine = COandXEngine::NewL();
iController = COandXController::NewL();
// Create the view and add it to the framework
iAppView = COandXView::NewL(*this);
AddViewL(*iAppView);

ReportWhoseTurn();
}

```

The most significant difference is that the UIQ version registers the view by calling `AddViewL()`, which also takes care of adding it to the control stack. After this call, the application UI class has full ownership of the view, so you do not have to worry about either deregistering the view or removing it from the control stack when the application UI class is destroyed.

The View Class

The majority of the remaining source code differences that are needed in a UIQ version of the application stem from the differences between `CQikViewBase` and `CCoeControl`. So, while both forms of the view own their component controls, the mechanisms of ownership and access are different. A view that derives from `CCoeControl`, as in the S60 version, creates its component controls (normally in its `ConstructL()` function) and stores pointers to them in specific items of member data. In the S60 version, the relevant `ConstructL()` code is:

```

...
for (TInt i = 0; i < KNumberOfTiles; i++)
{
    User::LeaveIfError(iTiles.Append(CreateTileL()));
}
...
iStatusWin = COandXStatusWin::NewL(Window());

```

The controls are normally explicitly deleted in the destructor:

```

COandXAppView::~COandXAppView()
{
    for (TInt i=0; i<KNumberOfTiles; i++)
    {
        delete iTiles[i];
    }
    iTiles.Close();
    delete iStatusWin;
}

```

As described more fully in Chapter 17, access to each child control is via the `CountComponentControls()` and `ComponentControl()` functions that each component-owning control must implement:

```

TInt COandXAppView::CountComponentControls() const
{
    return KNumberOfTiles + 1;
}

CCoeControl* COandXAppView::ComponentControl(TInt aIndex) const
{
    if (aIndex==KNumberOfTiles)
    {
        return iStatusWin;
    }
    else
    {
        return const_cast<COandXTile*>(iTiles[aIndex]);
    }
}

```

CQikViewBase provides its own mechanism for component control ownership, so views that derive from this class do not need to provide explicit data members to store pointers to the components, nor do they need to provide code in the destructor to delete them. In addition, such views do not need to (and should not) provide implementations of `CountComponentControls()` and `ComponentControl()`.

Instead, UIQ views add component controls – normally from within the view's implementation of `ViewConstructL()` – to a built-in array that is initialized by a call to `InitComponentArrayL()` and then accessed via a `Components()` function, for example:

```

void COandXView::ViewConstructL()
{
    ViewConstructFromResourceL(R_OANDX_UI_CONFIGURATIONS);
    InitComponentArrayL();
    for (TInt i = 0; i < KNumberOfTiles; i++)
    {
        COandXTile * tile = new(ELeave) COandXTile;
        AddControlLC(tile, i);
        tile->ConstructL(this);
        tile->SetFocusing(ETrue);
        CleanupStack::Pop(tile);
    }
    // Status window added, but not created here
    AddControlLC(iStatusWin, KNumberOfTiles);
    CleanupStack::Pop(iStatusWin);
    iStatWinNotAppended = EFalse;
    // Status window now owned by the view

    UpdateCommandsL(Controller().IsNewGame(), Controller().IsCrossTurn());
    SetFocusByIdL(0);
}

```

In the UIQ version of the application, as you can see from the above code, each tile is created and then appended to the array with a call to `AppendLC()` which, as its name implies, leaves a pointer to the

tile on the cleanup stack. The tile is removed from the cleanup stack once any potentially leaving initialization is complete. The view takes ownership of all component controls that are added in this way and they are automatically deleted when the view is destroyed.

If you think carefully about the code sequence for adding a component into a view, you might begin to wonder if it is truly safe. For example, what if the actions performed by `AddControlLC()` cause a leave before the control is added to the cleanup stack? Furthermore, if the view takes ownership of the control, should `AddControlLC()` leave the control on the cleanup stack at all? Could a leave before the control is popped from the cleanup stack result in the control being deleted twice – once from the cleanup stack and once by the owning view? In Symbian OS, it is worth asking questions like this, since leave-safeness is paramount.

In fact, this sequence is safe. `AddControlLC()` adds the control to the cleanup stack before performing any potentially leaving actions, and the item that is added to the cleanup stack is one that is specifically designed to transfer ownership of the view (in a leave-safe way) as it is popped.

The creation of the status window follows a slightly different pattern. The reason for this is that the status window is accessed from the controller, via the `ReportWhoseTurn()` function of the application UI class, and the first access occurs before the view's `ViewConstructL()` has been called. (To optimize start-up time, particularly for applications with multiple views, `ViewConstructL()` is only called at the time that the view first needs to be made visible.) To prevent this attempted access from causing a panic, we need to create the status window in the view's `ConstructL()` and add it in `ViewConstructL()`.

However, there is then a risk that the status window will not be deleted if the code leaves between its creation and it being successfully added. We solved this problem by including the `iStatWinNotAppended` flag in the class data, setting it when the status window is created and clearing it when the window has successfully been added to the array. The view's destructor has to delete the status window only if the flag is set:

```
COandXView::~COandXView()
{
    if (iStatWinNotAppended)
        // Once appended, system code takes care of a component's deletion
        {
            delete iStatusWin;
        }
}
```

It is also worth pointing out that the initial construction of the view, from `ViewConstructL()`, is by means of a call to `ViewConstructFromResourceL()`. In suitable cases, an entire multi-page view can be constructed from the resource accessed by this function, in the same way that a dialog can be constructed from a resource, as described in Chapter 16. In the present case, the resource specifies a single page, without supplying any content:

```
RESOURCE QIK_VIEW_PAGES r_oandx_layout_pages
{
    pages =
    {
        QIK_VIEW_PAGE
        {
            page_id = EOandXViewPage;
        }
    };
}
```

Since the UIQ version of the view class does not have accessible implementations of `ComponentControl()` or `CountComponentControls()`, functions that, in an S60 application, call one or both of these functions need a different implementation. In the Noughts and Crosses application, the functions that are affected are those associated with which tile has focus. In the S60 version, these are `IdOfFocusControl()` and `MoveFocusTo()`:

```
TInt COandXAppView::IdOfFocusControl()
{
    TInt ret = -1;
    for (TInt i=0; i<KNumberOfTiles; i++)
    {
        if (ComponentControl(i)->IsFocused())
        {
            ret = i;
            break;
        }
    }
    ASSERT_ALWAYS(ret>=0, Panic(EOandXNoTileWithFocus));
    return ret;
}

void COandXAppView::MoveFocusTo(const TInt index)
{
    TInt oldIndex = IdOfFocusControl();
    if (index != oldIndex)
    {
        SwitchFocus(oldIndex, ComponentControl(index));
    }
}
```


`SwitchFocus()` is implemented as:

```
void COandXAppView::SwitchFocus(TInt aFromIndex, CCoeControl* aToControl)
{
    ComponentControl(aFromIndex)->SetFocus(EFalse, EDrawNow);
    aToControl->SetFocus(ETrue, EDrawNow);
}
```

In the UIQ version, the two corresponding functions are `IdOfFocusedControl()` and `SetFocusByIdL()`, whose basic implementations are:

```
TInt COandXView::IdOfFocusedControl()
{
    TInt i;
    for (i=0; i<KNumberOfTiles; i++)
    {
        CCoeControl *control=ControlById<CCoeControl>(i);
        if (control->IsFocused())
            break;
    }
    return i;
}

void COandXView::SetFocusByIdL(TInt aControlId)
{
    RequestFocusL(ControlById<CCoeControl>(aControlId));
}
```

In the UIQ version, we changed the names of both functions to emphasize the differences in implementation. Note, in particular, that `SetFocusByIdL()` is a leaving function, whereas `MoveFocusTo()` is not. Fortunately this does not give rise to any other complications in the current application, since they are called only from within leaving functions.

While on the subject of focus, there are two further differences that need mentioning. First, the default status of newly created controls in S60 is for them to be capable of accepting focus, which is not the case in UIQ. In UIQ you need to call `SetFocusing(ETrue)` on each control that needs to accept focus (in this case, on each of the board's tiles). In contrast, in S60, you need to call `SetFocusing(EFalse)` on each control, such as the game's status window, that you do not want to be able to accept focus. Secondly, UIQ 3 views provide automatic, intelligent key- or button-based navigation between any of the component controls that can accept focus – that is, those controls for which `SetFocusing(ETrue)` has been called. In consequence, the UIQ version has no need to implement the view's `OfferKeyEventL()` function.

Commands

UIQ 3 views are primarily designed for use in applications with multiple views, each with its own set of commands, so they are not seen at their

best in an application such as the Noughts and Crosses application, which only has a single view. The application does, however, allow the basic differences to be illustrated.

A major difference in UIQ is that commands are treated in a more abstract way than in S60. A UIQ device may be set up to access its commands by means of either a pointer device or softkeys, and the commands themselves may or may not be displayed in a command menu. In addition, commands may be provided from a variety of sources within the application, such as one or more individual controls – or even from outside the application itself, for example, from a front-end processor.

The UIQ application framework contains a command model that deals with all these variations. In consequence, the application programmer need only be concerned with which commands should be available and not with how they are presented or selected.

The first change is to the resource script. As we saw earlier, an S60 application uses an `EIK_APP_INFO` resource, to specify the menu bar and the menu items it contains. In contrast, a UIQ 3 application must specify an empty `EIK_APP_INFO` resource, plus a `QIK_VIEW_CONFIGURATIONS` resource that lists the UI configurations it supports. In the Noughts and Crosses application, these resources are:

```
RESOURCE EIK_APP_INFO { }

RESOURCE QIK_VIEW_CONFIGURATIONS r_oandx_ui_configurations
{
    configurations =
    {
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            command_list = r_oandx_portrait_commands;
            view = r_oandx_layout;
        },
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikSoftkeyStylePortrait;
            command_list = r_oandx_portrait_commands;
            view = r_oandx_layout;
        }
    };
}
```

Each configuration specifies its list of available commands via a `QIK_COMMAND_LIST` resource:

```
RESOURCE QIK_COMMAND_LIST r_oandx_portrait_commands
{
    items =
    {
```

```

QIK_COMMAND
{
    id = EEikCmdExit;
    type = EQikCommandTypeScreen; // Only visible in debug
    stateFlags = EQikCmdFlagDebugOnly;
    text = "Close (debug)";
},
QIK_COMMAND
{
    id = EOandXNewGame;
    type = EQikCommandTypeScreen;
    text = "New game";
},
QIK_COMMAND
{
    id = EOandXFirstPlayer;
    type = EQikCommandTypeScreen;
    text = "Noughts move first";
}
};
}

```

As in the S60 version, the commands are updated dynamically, but the mechanism is different. S60 applications update their menus via the `DynInitMenuPanel()` function, which is called immediately before a menu is made visible.

In UIQ, since a command may be selected by means of a softkey, or from a toolbar, there is no guarantee that a menu pane is displayed before the selection takes place. In consequence, you cannot use `DynInitMenuPanel()` to update the available commands. Indeed, since some commands may be permanently visible, UIQ applications need to perform such updating each time the view's state changes in a way that affects the commands.

In many cases, it would be appropriate to implement the view's `HandleControlEventL()` function, which is called (with an `EEEventStateChanged` event code) each time one of the view's controls changes its state. In the Noughts and Crosses application, all significant state changes take place in the Controller, so we've taken the more direct option of adding a `SetStateL()` function to the Controller, implemented as:

```

void COandXController::SetStateL(TState aState)
{
    iState = aState;
    OandXAppUi()->UpdateCommandsL(IsNewGame(), IsCrossTurn());
}

```

This calls an `UpdateCommandsL()` function in the application UI class, which simply calls a similarly named function in the view:

```

void COandXView::UpdateCommandsL(TBool aIsNew, TBool aIsCrossTurn)
{

```

```
iCommandManager.SetAvailable(*this, EOandXNewGame, !aIsNew);
iCommandManager.SetAvailable(*this, EOandXFirstPlayer, aIsNew);
iCommandManager.SetTextL(*this, EOandXFirstPlayer,
    aIsCrossTurn?R_OANDX_O_FIRST:R_OANDX_X_FIRST);
}
```

The following points are relevant when updating the menus:

- While a game is in progress, there is no need to worry about whether the text for the ‘Who moves first’ command is updated correctly or not, because that command is only available (and visible) before a new game is started.
- When executing the ‘Who moves first’ command, the game is always new, so the only interest is in setting the correct text for the next potential use of that command.

In addition, the controller’s `Reset()` function is renamed to `ResetL()` because it can now leave:

```
void COandXController::ResetL()
{
    Engine().Reset();
    if (IsCrossTurn())
    {
        SwitchTurn();
    }
    SetStateL(ENewGame);
}
```

`ResetL()` cannot be used to initialize the controller (and engine) because `SetStateL()` calls the view’s `UpdateCommandsL()` function, and the view does not exist at the time the controller is constructed. Instead, in the controller’s `ConstructL()`, we explicitly set the relevant data:

```
...
Engine().Reset();
iState = ENewGame;
iCrossTurn = EFalse;
...
```

Since UIQ expects an application to have multiple views and each view may have a different set of commands, command processing is implemented within each individual view. A UIQ application, therefore, does not supply a `HandleCommandL()` function in the application UI class, but implements this function in each of its views. Note that the argument to this function is a reference to an instance of `CQikCommand`, rather than the command ID that is passed to this function in the

application UI class. Otherwise, the implementation is broadly similar to that in the S60 version:

```
void COandXView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        case EOandXNewGame:
            Controller().ResetL();
            SetFocusByIdL(0);
            DrawNow();
            break;
        case EOandXFirstPlayer:
            Controller().SwitchTurn();
            UpdateCommandsL(Controller().IsNewGame(),
                           Controller().IsCrossTurn());
            break;
        default:
#ifdef _DEBUG
            OandXAppUi()->SaveGameStateL();
#endif
        // The Go back and Exit commands are handled by CQikViewBase.
        CQikViewBase::HandleCommandL(aCommand);
        break;
    }
}
```

Persistence

In UIQ, the default behavior is to allow applications to use persistent data, so, unlike in S60, there is no need to supply an implementation of the document's `OpenFileL()` function.

UIQ applications do not normally have a Close command (except for testing purposes, in debug builds). Instead, system code may instruct an application to close at any time that memory needs to be freed, first giving it an opportunity to save its persistent data. In order to access the functionality more easily, we have added – but only in debug builds – a call to the `SaveGameStateL()` function of the application UI class in the default case of the view's `HandleCommandL()`. The implementation is very simple:

```
void COandXAppUi::SaveGameStateL()
{
    SaveL();
}
```

We have to supply this new function because the `SaveL()` function of the application UI class is protected, and so cannot be called directly from the view. With this call in place, the persistent data is saved each time the Close (Debug) command is selected.

Since the application's persistent data contains one item – the ID of the tile that currently has focus – from the view, there is one final complication that we have to deal with. It is caused by the view's controls being created and initialized in the `ViewConstructL()` function which, as mentioned earlier, is called as late as possible, just before the view first becomes visible. This means that the document's `RestoreL()` function (and, at least when the application is run for the first time, its `StoreL()` function) are called before the view is fully initialized.

We resolve this problem by adding an `iInitialFocusId` data member to the `COandXView` class and coding the view's `SetFocusByIdL()` function as:

```
void COandXView::SetFocusByIdL(TInt aControlId)
{
    CCoeControl * control = ControlById<CCoeControl>(aControlId);
    if (control)
    {
        RequestFocusL(control);
    }
    else // Control does not yet exist
    {
        iInitialFocusId = aControlId;
    }
}
```

If the control does not yet exist, the value is copied into `iInitialFocusId` and used, within `ViewConstructL()`, to move focus to the correct control after it has been created.

We also need to deal with the case where the persistent data needs to be written before the tiles exist (this only happens once, the first time the application is run, when the application's INI file is being created). The relevant data is obtained by a call to the view's `IdOfFocusedControl()`, whose implementation is:

```
TInt COandXView::IdOfFocusedControl()
{
    TInt i;
    for (i=0; i<KNumberOfTiles; i++)
    {
        CCoeControl *control=ControlById<CCoeControl>(i);
        if (!control) // Not yet created
        {
            return 0;
        }
        if (control->IsFocused())
        {
            break;
        }
    }
    return i;
}
```

If a control does not yet exist, the function simply returns zero, which is the correct default value.

Summary

This chapter has described a simple but non-trivial application to play a game of Noughts and Crosses.

After a brief introduction to the game, the chapter explained the overall architecture of the application, before describing each of the S60 version's major component classes in more detail.

The chapter concluded with a discussion of the main differences between the S60 and UIQ versions of the game.

13

Resource Files

In Chapters 2 and 11, resource files are used to define the main elements required by a Symbian OS application UI. In Chapter 16 we also use resource files to specify dialogs.

In this chapter we review resource files and the resource compiler more closely, to understand their role in development for Symbian OS. This chapter provides a quick tour – for a fuller reference, see the SDK.

13.1 Why a Symbian-Specific Resource Compiler?

The Symbian OS resource compiler starts with a text source file and produces a binary data file in parallel with the application's executable. In contrast, Windows uses a resource compiler which supports icons and graphics as well as text-based resources, and which builds the resources right into the application executable so that an application can be built as a single package. Many Windows programmers never see the text resource script nowadays, because their development environment includes powerful and convenient GUI-based editors.

So, why does Symbian OS have its own resource compiler, and how can an ordinary programmer survive without the graphical resource editing supported by modern Windows development environments?

Unlike Windows developers, Symbian OS developers target a wide range of hardware platforms, each of which may require a different executable format. Keeping the resources separate introduces a layer of abstraction that simplifies the development of Symbian OS, and the efforts required by independent developers when moving applications between different hardware platforms. Furthermore, and perhaps more importantly, it provides good support for localization. In addition to facilitating the

process of translation by confining the items to be translated into separate files, it allows a multilingual application to be supplied as a single executable together with a number of language-specific resource files.

An application based on the GUI application templates generated by the Carbide.c++ wizard (see Chapter 11) uses a resource file to contain GUI element definitions (menus, dialogs, etc.) and strings that are needed by the program at run time. The run-time resource file has the extension `.rsc`, and post-Symbian OS v9 it resides in the applications resource directory rather than in the same directory as the application.

13.2 Source File Syntax

Because processing starts with the C preprocessor, a resource file has the same lexical conventions as a C program, including source-file comments and C preprocessor directives.

The built-in data types in Table 13.1 are used to specify the data members of a resource, as described later. A resource file may contain statements of the types shown in Table 13.2.

Table 13.1 Built-in Data Types

Data Type	Description
BYTE	A single byte, which may be interpreted as a signed or unsigned integer value.
WORD	Two bytes, which may be interpreted as a signed or unsigned integer value.
LONG	Four bytes, which may be interpreted as a signed or unsigned integer value.
DOUBLE	Eight-byte real, for double precision floating point numbers.
TEXT	A string terminated by a null. This is deprecated: use LTEXT instead.
LTEXT	A Unicode string with a leading length byte but no terminating null.
BUF	A Unicode string with no leading byte or terminating null.
BUF8	A string of 8-bit characters with no leading byte or terminating null.
BUF<n>	A Unicode string of up to n characters with no leading byte or terminating null.

Table 13.1 *(continued)*

Data Type	Description
LINK	The ID of another resource (16 bits).
LLINK	The ID of another resource (32 bits).
SRLINK	A 32-bit self-referencing link that contains the resource ID of the resource in which it is defined. Its value is assigned automatically by the resource compiler.

Table 13.2 Resource File Statement Types

Statement type	Description
NAME	Defines the leading 20 bits of any resource ID. Must be specified prior to any RESOURCE statement.
STRUCT	Defines a named structure for use in building aggregate resources.
RESOURCE	Defines a resource, which may optionally be given a name.
ENUM/enum	Defines an enumeration and supports a syntax similar to that of C.
CHARACTER_SET	Defines the character set for strings in the generated resource file. If not specified, cp1252 is the default.

STRUCT Statements

A STRUCT statement takes the following form:

```
STRUCT struct-name [ BYTE | WORD ] { struct-member-list }
```

`struct_name` specifies a name for the struct. The name must start with a letter and be in upper-case characters. It may contain letters, digits and underscores, but not spaces. The optional `BYTE` and `WORD` keywords are intended for use with structs that have a variable length. They have no effect unless the struct is used as a member of another struct, when they cause the data of the struct to be preceded by a length `BYTE` or length `WORD`, respectively.

`struct_member_list` is a list of member initializers, terminated by semicolons, and enclosed in braces `{ }`. A member may be one of the built-in types, a previously defined struct or an array. An array member

does not specify either the number or the type of its elements. It is common to supply members with default values, frequently either 0 or an empty string. The following example, taken from `eikon.rh`, illustrates many of the features of a `STRUCT` statement.

```
STRUCT DIALOG
{
    LONG flags=0;
    LTEXT title="";
    LLINK pages=0;
    LLINK buttons=0;
    STRUCT items[];    // an array
    LLINK form=0;
}
```

`STRUCT` statements are conventionally placed in separate files with a `.rh` (resource header) extension and included in the resource script. You may find it instructive to review the contents of the various RH files that you will find in the `\epoc32\include` directory of an installed SDK.

RESOURCE Statements

The `RESOURCE` statement is probably the most important, and certainly the most frequently used, statement. It takes the form:

```
RESOURCE struct_name [ id ] { member_initializer_list }
```

`struct_name` refers to a previously encountered `STRUCT` statement. In most application resource scripts, this means a struct defined in an included RH file. `id` is an optional symbolic resource ID. If specified, it must be in lower-case, starting with a letter, and containing letters, digits and underscores, but not spaces. `member_initializer_list` consists of a list of member initializers, separated by semicolons, and enclosed in braces `{ }`.

The following example shows a resource constructed for S60, using the `DIALOG` struct defined above.

```
RESOURCE DIALOG r messages_dialog
{
    title = "Title text";
    flags = EAKnDialogSelectionList;
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
    items =
    {
        DLG_LINE
        {
            type = EAKnCtSingleListBox;
            id = EListControl;
            control = LISTBOX
        }
    }
}
```

```

    {
        flags = EAknListBoxSelectionList;
        array_id = r_message_list;
    };
}
};
}

```

In this case the `items` array contains only one item, which is itself a `DLG_LINE` struct. It is not necessary for the resource to declare the initializers in the same order as they were specified in the corresponding struct. The resource in this example does not provide initializers for the `DIALOG` struct's `pages` and `form` items, so their values will take the default values specified in the struct.

The punctuation rules for a `RESOURCE` statement are quite simple:

- assignment statements, of any type, are terminated by a semicolon
- items in a list are separated by commas
- at the end of a structure, no punctuation is required.

For example:

```

RESOURCE ARRAY r_message_list
{
    items =
    {
        LBUF
        {
            txt= "\tMessage 1";      // Rule 1
        },                        // Rule 2
        LBUF
        {
            txt= "\tMessage from file"; // Rule 1
        }                        // Rule 2
    };                          // Rule 1
}                               // Rule 3

```

The rules apply regardless of whether or not the punctuation occurs after a closing brace.

ENUM Statements

To ensure that your resource script and C++ program use the same values for symbolic constants, the resource compiler supports both `enum` and `#define` definitions of constants, with a syntax similar that of C++. By convention, these definitions are contained in `HRH` include files. The `.hrh` extension is intended to convey that the file is suitable for inclusion either as a `.h` file in C++ source, or as a `.rh` file in resource scripts.

The **NAME** Statement

A resource script must contain a single **NAME** statement, which must appear before the first **RESOURCE** statement. The **NAME** keyword must be followed by a name, in upper case, containing a maximum of four characters, for example:

```
NAME HELO
```

The **NAME** is used in generating symbolic IDs for the resources, as discussed in Section 13.7.

13.3 Bitmaps and Icons

When your application is installed on a phone, you want it to have an easily recognizable icon for the user to select. Such icons are created by the bitmap conversion process that is illustrated in Figure 13.1.

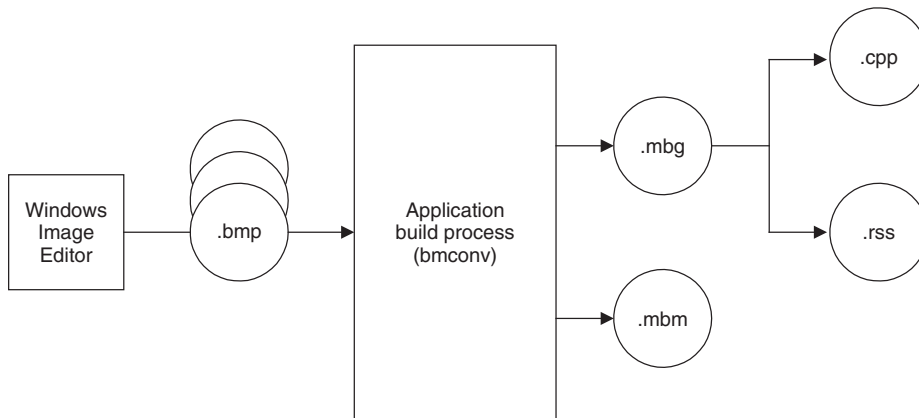


Figure 13.1 Bitmap conversion

The first step of the process is to create Windows bitmaps and convert them into the specific file format used by Symbian OS, called a ‘multi-bitmap’ file or MBM. The MBM file, together with an associated MBG file which contains an ID for each bitmap in the corresponding MBM file, is constructed from one or more Windows BMP files using the `bmconv` (bitmap converter) tool, which is called during the main application build process.

The MBM format is also used when icons are required elsewhere in your application – for example, a splash screen on startup.

Windows programmers may find it easier to think of a MBM file as an addition to the application’s resource file, and a MBG as an addition to the RSG generated header, containing symbolic IDs for the resources.

Under Windows, bitmaps and other resources are incorporated directly into resource files. Under Symbian OS, they are treated separately because the tools used to create MBMs and resource files are different. This also permits finer control over the content of these files – for example, some Symbian OS phones may compress resource file text strings to save space, but leave bitmaps uncompressed to avoid performance overheads at display time.

As illustrated in Figure 13.2, the application icon should be created from two Windows bitmaps: the icon itself, and a mask which is black in the area of the icon.

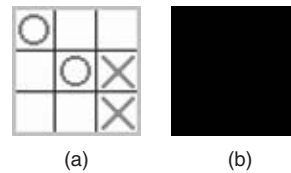


Figure 13.2 Icon and mask bitmaps

Only the regions of the bitmap corresponding to the black regions of the mask are copied to the screen. Everything else is ignored – the white areas of the mask are effectively transparent, regardless of the color in the corresponding parts of the data bitmap.

The application icon is used in a number of places by the UI. For example, on the UIQ main application launcher screen, the size of the icon is increased when it is selected, as shown in Figure 13.3.



Figure 13.3 Icon size increases when selected

It is necessary to supply the application icon in a number of different sizes, so that the UI can select the best one to display. A good initial selection of sizes is shown in Table 13.3.

UIs support different color depths for items shown on screen. Symbian OS provides support from 1-bit up to 24-bit color, and individual phones vary in terms of the color depth supported by their hardware.

Bitmap icons are easy to produce, and at this stage you can simply create them using a suitable graphics package.

To show how to add the icons to our application, we assume three icons, with corresponding icon masks, have been created and placed in

Table 13.3 Icon Sizes

	User Interface	
icon size	UIQ	S60
Small	18x18	32x32
Medium	40x40	40x40
Large	64x64	64x64

the \images folder within the application directory. These are named:

OandX_xLarge.bmp and OandX_xLarge_mask.bmp,
 OandX_Large.bmp and OandX_Large_mask.bmp,
 OandX_Small.bmp and OandX_Small_mask.bmp.

These six bitmaps must be turned into a single MBM file, along with its associated MBG.

Converting the Bitmaps

The next step is to add the bitmap conversion information to the application build process for the Noughts and Crosses example. You do this by including the following text in the project's MMP file:

```
START BITMAP OandX.mbm
HEADER
  TARGETPATH \Resource\Apps
  SOURCEPATH ..\images
  SOURCE c24 OandX_Small.bmp
  SOURCE 8 OandX_Small_mask.bmp
  SOURCE c24 OandX_Large.bmp
  SOURCE 8 OandX_Large_mask.bmp
  SOURCE c24 OandX_xLarge.bmp
  SOURCE 8 OandX_xLarge_mask.bmp
END // BITMAP
```

The statements have the following meanings:

- **START BITMAP:** marks the start of the bitmap conversion data and specifies the MBM multi-bitmap filename.
- **HEADER:** specifies that a symbolic ID file, OandX.mbg, is to be created (in the \epoc32\include folder).
- **SOURCEPATH:** specifies the location of the original Windows bitmaps.
- **TARGETPATH:** specifies the location where the MBM file is to be generated.

- **SOURCE:** specifies the color depth and the names of one or more Windows bitmaps to be included in the MBM.
- **END:** marks the end of the bitmap conversion data.

The MBM is generated in the `\Resource\Apps` directory. It's standard practice to specify the MBM name to be the same as that of the EXE file.

Symbian OS developers conventionally specify only one bitmap file per **SOURCE** statement, and specify each mask file immediately after its corresponding bitmap file. The ordering of the statements does not really matter, but the color-depth value does. A value of `c24` specifies 24-bit color (the default for UIQ 3).

The recommended value to be specified for each mask is 1, the lowest color depth. Since masks contain only black, they need not be stored with a high color depth, and specifying this value may save valuable storage space for the resulting MBM, as well as RAM usage when the MBM is loaded at run time.

Depending on circumstances, and particularly if bitmaps are stored in compressed form, using a color depth of 1 may not save on memory usage. Also, if your icon and mask are to be used in a speed-critical operation (an animation, for example) you may want to include both the icon and mask at the same color depth as used by the phone itself. Since the Window Server will not need to convert the images between the display mode of the phone and the color-depth in which they are stored, you will sacrifice some storage space and RAM for a bit of extra speed. For general use, however, the first approach is recommended.

The format of the generated `OandX.mbg` file is as follows:

```
// oandx.mbg
// Generated by BitmapCompiler
// Copyright (c) 1998-2005 Symbian Software Ltd. All rights
// reserved.
//

enum TMbmOandx
{
    EMbmOandxOandx_small,
    EMbmOandxOandx_small_mask,
    EMbmOandxOandx_large,
    EMbmOandxOandx_large_mask,
    EMbmOandxOandx_xlarge,
    EMbmOandxOandx_xlarge_mask
};
```

The file contains an enumeration whose type name includes the MBM filename, and several enumerated constants whose names are made up from the MBM filename and the source bitmap filename.

You need to include the enumerations generated in this file in your application's resource file, as follows:

```
#include <oandx.mbg>
```

The **bmconv** Tool

The bitmap converter tool, **bmconv**, can also be used as a stand-alone application, either to package bitmaps into a single MBM file, or to extract bitmaps from a multi-bitmap file.

To convert bitmaps into a single MBM file, use a command such as the following:

```
bmconv /h file.mbg file.mbm icon1.bmp icon2.bmp icon1mask.bmp  
icon2mask.bmp
```

This gives a verbose log (which can be suppressed using the **/q** switch):

```
BMCONV version 114.  
Compiling...  
Multiple bitmap store type: File store  
Epoc file: file.mbm  
  
Bitmap file 1   : icon.bmp  
Bitmap file 2   : icon2.bmp  
Bitmap file 3   : iconmask.bmp  
Bitmap file 4   : icon2mask.bmp  
Success.
```

To extract BMP files from a multi-bitmap file, specify the **/u** flag after the **bmconv** command:

```
bmconv /u file.mbm icon.bmp icon2.bmp iconmask.bmp icon2mask.bmp
```

One useful application of this facility is to capture a screen from the emulator using Ctrl, Alt, Shift, S. This results in an MBM file containing a single entry. Once you have extracted the bitmap, you can display and manipulate it using a graphics editor.

You can also view the contents of an MBM file by specifying the **/v** flag after the command:

```
bmconv /v file.mbm
```

To see the full set of supported options, just type **bmconv** on the command line.

Scalable Vector Graphics

Recent SDKs have introduced support for a new format for icons, known as Scalable Vector Graphics-Tiny (SVG-T). This format can be used for icons and themes instead of bitmaps, and removes the need to supply your icons in different sizes as discussed above. Using SVG-T alleviates one of the main disadvantages of bitmapped graphics formats, namely that image quality is often lost when the image is scaled up.

To support this format, the S60 SDK contains two additional tools, the converter tool and `mifconv`. The SVG to SVG-T converter tool (installed from the directory `\S60Tools\svg2svgt`) takes a standard SVG file as input and generates an equivalent SVG-T file. `mifconv` is then used to package the SVG-T icons into a Multiple Images (or Multi-Icon) File (MIF); a corresponding MBM file is generated only if there are bitmap icons amongst the source icons.

Further details are available in the S60 SDK or on Nokia's website.

13.4 Updating the Resource Files

Before you can rebuild your application with its new icons, you need to update some of the resource files to ensure that the icons are included in the build.

Each application requires a localizable application information file, usually named `<application-name>_loc.rss`. This is specified in the application registration file (see Section 13.5).

The application information file contains a resource, `LOCALISABLE_APP_INFO`, specifying a caption for your application to be displayed in the application launcher, the number of icons in the icon file, and the location of the icon file. For example:

```
// OandX_loc.rss
//
//   Symbian Software Ltd 2005. All rights reserved.
//

#include <AppInfo.rh>

RESOURCE LOCALISABLE_APP_INFO
{
    caption_and_icon =
    {
        {
            CAPTION_AND_ICON_INFO
            {
                caption = STRING_r_oandx_caption;
                number_of_icons = 3; // each icon must be bitmap/mask pair
                icon_file = "\\Resource\\Apps\\oandx.mbm";
            }
        }
    };
}
```

The string resource is a text resource defined in the resource file `OandX.rss`.

13.5 Application Registration Files

One of the major changes in Symbian OS v9.1 is the introduction of the application registration file. Since GUI applications are now EXEs (rather than APPs as in pre-v9 versions), the application registration file is used to tell the application architecture that this particular EXE is a GUI application. Application registration files contain information about an application's name, UID and properties that is required by the application launcher or system shell. All applications must provide a registration file.

A registration file is a standard Symbian OS compiled resource file (RSC). The RSS file used to build the registration file should be given the same filename as the application, but with a `_reg` suffix; this replaces the Application Information File (AIF) supported on earlier versions. The registration file must build into a private data-caged directory belonging to the application architecture server, since the application architecture is ultimately responsible for launching all applications. Putting the RSS file in a private directory prevents a malicious application replacing an existing application's RSS file with a spoof version.

As an example, here's `OandX_reg.rss`:

```
// OandX_reg.rss
//
// Copyright (c) 2006 Symbian Software Ltd. All rights reserved.

#include <appinfo.rh>
#include <OandX.rsg>

//specify the UID2 and UID3 values
UID2 KUidAppRegistrationResourceFile
UID3 0xE04E4143

RESOURCE APP_REGISTRATION_INFO
{
    app_file = "OandX";
    localisable_resource_file = "\\resource\\apps\\OandX_loc";
}
```

This example specifies localizable application information as a resource within the application UI resource file. Localizable application information can also be provided in a separate resource file, as discussed previously. The `APP_REGISTRATION_INFO` structure minimally needs to provide the name (but not extension) of the application binary (using the `app_file` statement). If a localizable application information file is provided, as in this example, its full path and filename must be specified, excluding the drive letter and file extension. This file defines

the application's captions and the name of the icon file. By convention it has the same filename as the application, but with a `_loc` suffix.

To build the registration file and the localizable application file, add the following lines to the application's MMP file:

```
// V9 application registration file
START RESOURCE \OandX_reg.rss
#ifdef WINSCW
    TARGETPATH    \private\10003a3f\apps
#else
    TARGETPATH    \private\10003a3f\import\apps
#endif
END

START RESOURCE \OandX_loc.rss
    TARGETPATH    \resource\apps
    LANG          SC
END
```

The registration file is used by the application architecture server, which has a SID of `0x10003a3f`. The `import` subdirectory allows an application to write files into a second application's private data directory, providing that it knows the second application's SID.

In our MMP file, the `TARGETPATH` value ensures that registration files are correctly deployed to the application architecture's private data-caged area. On the emulator (WINSCW), all registration files are located in `\private\10003a3f\apps`. This is also true on real hardware for registration files built into the ROM, but for applications installed onto a phone using the standard software installation method, data caging prevents writing to this directory. The registration files are therefore installed into `\private\10003a3f\import\apps`, and the application architecture retrieves them from there. In all cases, the registration file must be located on the same drive as the application.

13.6 Localizable Strings

If you intend to translate your application into one or more different languages, there are some additional things to bear in mind. A translator will not necessarily be a programmer and will probably find it difficult to preserve the general structure of a resource script through the translation process. For this reason alone, it is good practice to store all resource text strings in a separate resource localizable string (RLS) file, with a `.rls` extension. An RLS file defines symbolic identifiers for strings, to which the resource file refers when it needs the associated string.

All you have to do is to replace each text string in the resource script with a symbolic identifier and, in a separate file, associate the identifier with the original string.

As an example, consider the following two resources:

```
RESOURCE MENU_PANE r_edit_menu
{
    items=
    {
        MENU_ITEM
        {
            command = ECmd1;
            txt = "Item 1";
        },
        MENU_ITEM
        {
            command = ECmd2;
            txt = "Item 2";
        }
    };
}

...

RESOURCE TBUF r_text_info
{
    buf = "Information text message!";
}
```

The three text strings in these resources would appear in an RLS file as follows:

```
rls_string STRING_r_edit_menu_first_item "Item 1"
rls_string STRING_r_edit_menu_second_item "Item 2"
rls_string STRING_r_message_info "Information text message!"
```

The keyword `rls_string` appears before each string definition, followed by a symbolic identifier, and then the string itself in quotes.

The resource file itself is then modified to include the RLS file (in this case, assumed to have the name `application.rls`) and to refer to the strings via their symbolic names:

```
#include "application.rls"
...
RESOURCE MENU_PANE r_edit_menu
{
    items=
    {
        MENU_ITEM
        {
            command = ECmd1;
            txt = STRING_r_edit_menu_first_item;
        },
        MENU_ITEM
        {
            command = ECmd2;
```

```

        txt = STRING_r_edit_menu_second_item;
    }
};
}
...
RESOURCE TBUF r text_info
{
    buf = STRING_r_message_info;
}

```

You can include C- or C++-style comments in the text localization file to inform the translator (and yourself!) of the context in which each text item appears, and to give information about any constraints, such as the maximum length permitted for a string. There is a double advantage: the translator sees only the text and the comments, and the programmer sees a resource script free from comments that could disguise the structure of a resource.

There is no specific convention about the file name extension to be used for text localization files. At Symbian, developers tend to use a `.rls` extension but S60 suggests the use of a `.loc` extension.

Multilingual Applications

As described in Chapter 11, you specify multiple language options by means of a `LANG` statement in the application's MMP file. The `LANG` keyword is followed by a list of two-character codes, one for each language, for example:

```
LANG 01 02
```

The codes can be anything you like, but are normally taken from the two-digit values specified in the `TLanguage` enumeration in `e32const.h`.

The Symbian OS build tools use the `LANG` statement to build the application's resource files once for each code in the list. For each build:

- a `LANGUAGE_XX` symbol is defined
- the built file is given a `.rXX` extension

where `XX` represents the language code for that build.

You should include the appropriate RLS files in each of the application's localizable resource scripts, for example:

```

// application_loc.rss

#include <AppInfo.rh>

```

```

#ifdef LANGUAGE_01
    #include "appenglish.rls"
#elif defined LANGUAGE_02
    #include "appfrench.rls"
#endif

RESOURCE LOCALISABLE_APP_INFO
...

```

In this case, the result of the build is to generate two resource files, named `application_loc.r01` and `application_loc.r02`, respectively containing English and French text. The SDK documentation explains how to use the application's PKG file to build a number of language-specific resource files into a multilingual application's releasable SIS file.

If an application's MMP file does not contain a `LANG` statement, the resource script is built only once and the resulting resource file is given a `.rsc` extension. Effectively, this means that the default `LANG` statement is:

```
LANG SC
```

For simplicity, all later discussions of building resource files consider only the default case.

13.7 Multiple Resource Files

A single resource file supports up to 4095 resources, but a Symbian OS application may use multiple resource files, each containing this number of resources. The application identifies each resource by a symbolic ID comprising two parts:

- a leading 20 bits (five hex digits) that identify the resource file
- a trailing 12 bits (three hex digits) that identify the resource (hence the 4095-resource limit).

The leading 20 bits are generated from the four-character name specified in the `NAME` statement in the resource file. You can tell what the 20 bits are by looking at the RSG file. Here, for example, is `OandX.rsg`, which has the name `OANX`:

```

#define R_DEFAULT_DOCUMENT_NAME          0x485b8002
#define R_OANDX_MENUBAR                  0x485b8004
#define R_OANDX_MENU                      0x485b8005
#define R_OANDX_O_WINS                    0x485b8006

```

```
#define R_OANDX_X_WINS                0x485b8007
#define R_OANDX_WHO_STARTS            0x485b8008
#define R_OANDX_O_MOVES_FIRST         0x485b8009
#define R_OANDX_X_MOVES_FIRST         0x485b800a
```

Uikon's resource file name is `EIK`, and its resource IDs begin with `0x00f3b`.

You don't have to choose a name that's distinct from all other Symbian OS applications on your system – with only four letters, that could be tricky. The resource files available to your application are likely to be only those from Uikon, Qikon or Avkon and your application, so you simply have to avoid names starting with `EIK`, `Q` (for `UIQ`) and `AKN` (for `S60`). Avoid `CONE`, `BAFL`, and other Symbian component names as well, and you should be safe.

13.8 Compiling a Resource File

The resource compiler is invoked as part of the application build process, either from within the IDE, or from the command line with, for example:

```
abld build winscw udeb
```

As we saw in Chapter 1, this command runs the build in six stages, one of which is resource compilation.

From the command line, you can invoke the resource compiler alone with a command of the form:

```
abld resource winscw udeb
```

but you must first have run the `abld makefile` command to ensure that the appropriate makefiles exist (type `abld` on its own to get help on the available options).

Building for the `winscw` target by any of these methods causes the RSC file to be generated in the `\epoc32\release\winscw\<variant>\z\resource\apps` directory (where `<variant>` is either `udeb` or `urel`). Building for any ARM target causes the RSC file to be generated in the `\epoc32\data\z\resource\apps\` directory.

To use a resource at run time, a program must specify the resource ID. The resource compiler therefore generates not only the resource file, but also a header file, with a `.rsg` extension, containing symbolic IDs for every resource contained in the file. This header file is written to the `\epoc32\include\` directory, and is included in the application's source code. That's why the resource compiler is run before you run the C++ compiler when building a Symbian OS program. The RSG file is always generated to `\epoc32\include\`, but if the generated file

is identical to one already in `\epoc32\include\`, the existing one isn't updated.

Uikon has a vast treasury of specific resources (especially string resources) accessible via the resource IDs listed in `eikon.rsg`.

Figure 13.4 shows the overall build process, indicating the relationship between the various file types involved.

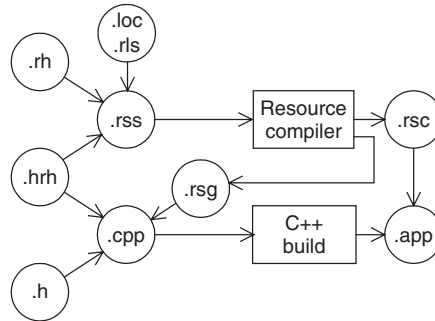


Figure 13.4 Build process

Application-specific input files to the resource compilation process for a typical application include those listed in Table 13.4.

You may also need to include system RH files, such as `eikon.rh`, and other UI-specific RH files.

In addition to `AppName.hrh` and the generated `AppName.rsg` file, C++ files might need to include system files such as `eikon.rsg` and `eikon.hrh`, together with one or more UI-specific RSG and HRH files.

Conservative RSG Update

When you compile `OandX.rss`, either from the CodeWarrior IDE or from the command line, it generates both the binary resource file `OandX.rsc` and the generated header file `OandX.rsg`.

The project correctly includes, for example, a dependency of `OandX.obj` on `OandX.rsg` (because `OandXAppUI.cpp` includes `OandX.rsg`). So, if `OandX.rsg` is updated, the `OandX.exe` project needs rebuilding. Scale this up to a large application, and a change in one resource file, and hence in the generated headers, could cause lots of rebuilding.

The resource compiler avoids this potential problem by updating the RSG file only when necessary – and it isn't necessary unless resource IDs have changed. Merely changing the text of a string or the placement of a GUI element won't cause the application's executable to go out of date.

This is why, when you run the resource compiler, you are notified if the RSG file has been changed.

Table 13.4 Application-specific Input files

Filename	Description
AppName.rss	The application's resource script.
AppName_reg.rss	The application's registration file.
AppName.rls or AppName.loc	The application's localizable strings.
AppName.rsc	The generated resource file.
AppName.rsg	Generated header containing symbolic resource IDs that are included in the C++ program at build time.
AppName.hrh	An application-specific header containing symbolic constants, for example the command IDs which are embedded into resources such as the menus, button bars and, if relevant, shortcut keys. Such header files are used by both resource scripts and C++ source files, in places such as <code>HandleCommandL()</code> .
AppName.loc	The application's localizable strings.
Eikon.rh	Header files that define Uikon's standard STRUCTs for resources.
Eikon.hrh	Header files that define Uikon's standard symbolic IDs, such as the command ID for <code>EEikCmdExit</code> , and the flags used in various resource STRUCTs.
Eikon.rsg	Resource IDs for Uikon's own resource files, which contain many useful resources. Many of these resources are for internal use by Symbian OS, although some are also available for application programs to use.

13.9 The Content of a Compiled Resource File

The resource compiler builds the run-time resource file sequentially, starting with header information that identifies the file as being a resource file. It then appends successive resources to the end of the file, in the order of definition. The resource compiler will not have built a complete index until the end of the source file is reached; hence the index is the last thing to be built and appears at the end of the file.

Each index entry is a word that contains the offset in bytes from the beginning of the file to the start of the appropriate resource. The index

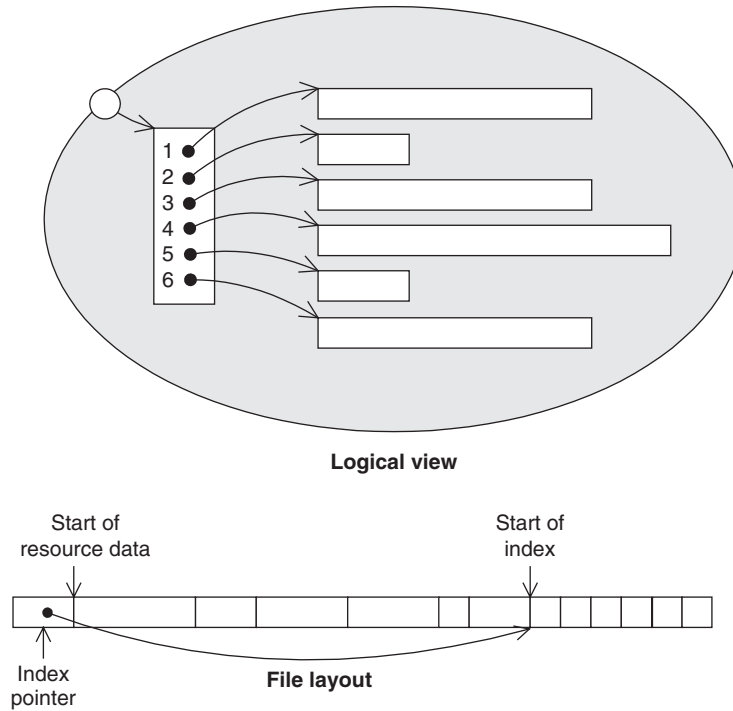


Figure 13.5 Compiled resource file structure

contains, at the end, an additional entry that contains the offset to the byte immediately following the last resource, which is also the start of the index. The structure is illustrated in Figure 13.5.

Each resource is simply binary data, whose length may be found by subtracting its own index entry from that of the following resource. To see what a compiled resource looks like, let's take a look at a resource from the Noughts and Crosses example.

The menu pane and menu item resource `STRUCTs` are declared in `uikon.rh` as follows:

```
STRUCT MENU_PANE
{
    STRUCT items[]; // MENU_ITEMS
    LLINK extension=0;
}
STRUCT MENU_ITEM
{
    LONG command=0;
    LLINK cascade=0;
    LONG flags=0;
    LTEXT txt;
    LTEXT extratxt=" ";
    LTEXT bmpfile=" ";
```

```
WORD bmpid=0xffff;
WORD bmpmask=0xffff;
LLINK extension=0;
}
```

and the menu pane resource itself is:

```
RESOURCE MENU_PANE r_oandx_menu
{
    items =
    {
        MENU_ITEM
        {
            command = EOandXNewGame;
            txt = "Item 0";
        },
        MENU_ITEM
        {
            command = EEikCmdExit;
            txt = "Close (debug)";
        }
    };
}
```

As you can see, each of the two menu items only specify two of the nine elements; the remaining seven take the default values as specified in the declaration.

If you were to analyze a hex dump of a resource file, you would clearly see the text of the various strings that are defined. However, you might be slightly puzzled: we have said that Symbian OS applications use a Unicode build, whereas these strings appear to be plain ASCII text. Furthermore, if you count through the resource, identifying items on the way, you will find that there is an occasional extra byte here and there – for example, each of the strings appears to be preceded by two identical count bytes, instead of the expected single byte.

This occurs because the compiled resource is compressed, in order to save space in Unicode string data. The content of the resource is divided into a sequence of runs, alternating between compressed and uncompressed data. Each run is preceded by a count of the characters in that run, with the count being held in a single byte, provided that the run is no more than 255 bytes in length.

Apart from the effects of data compression, the compiled resource just contains the individual elements, listed sequentially. A `WORD`, for example, occupies two bytes, and an `LTEXT` is represented by a byte specifying the length, followed by the text. Where there are embedded `STRUCTS`, the effect is to flatten the structure. The run-time interpretation of the data is the responsibility of the class or function that uses it.

13.10 Reading Resource Files

Via CCoeEnv

The application framework opens the application's resource file during the startup of the application. The most straightforward way to read one of its resources is to call CCoeEnv's `AllocReadResourceL()` or `AllocReadResourceLC()` functions, which take a resource ID as a parameter. Both functions allocate an `HBufC` big enough for the uncompressed resource and read it in. CCoeEnv also supplies variants of these two functions to read a resource as explicit 8-bit or 16-bit descriptors, or descriptor arrays.

```
text = iEikonEnv->AllocReadResourceL(R_MSG_1);
```

Alternatively, you can use CCoeEnv's `ReadResource()` function, which simply reads a resource into an existing descriptor, and panics if the read fails.

```
TBuf<16> ECommandText;
iEikonEnv->ReadResource(ECommandText, R_MSG_2);
```

Via BAFL

BAFL provides the basic APIs for reading resources:

- the oddly-named `RResourceFile` class, declared in `barsc.h`, is used for opening resource files, finding a numbered resource, and reading its data. (`RResourceFile` behaves more like a C class than an R class.)
- `TResourceReader`, declared in `barsread.h`, is a kind of stream-oriented reader for data in an individual resource. `TResourceReader` functions are provided that correspond to each of the resource compiler's built-in data types.

As an example of the use of `TResourceReader` functions, here's the code to read in a `MENU_ITEM` resource:

```
EXPORT_C void CEikMenuPaneItem::ConstructFromResourceL
(TResourceReader& aReader)
{
    iData.iCommandId=aReader.ReadInt32();
    iData.iCascadeId=aReader.ReadInt32();
    iData.iFlags=aReader.ReadInt32();
    iData.iText=aReader.ReadTPtrC();
}
```

```

iData.iExtraText=aReader.ReadTPtrC();
TPtrC bitmapFile=aReader.ReadTPtrC();
TInt bitmapId=aReader.ReadInt16();
TInt maskId=aReader.ReadInt16();
aReader.ReadInt32(); // extension link
if (bitmapId != -1)
{
    SetIcon(CEikonEnv::Static()->CreateIconL(bitmapFile, bitmapId,
                                              maskId));
}
}

```

In this code, a `TResourceReader` pointing to the correct (uncompressed) resource has already been created by the framework, so all this code has to do is actually read the resource.

The code exactly mirrors the resource `STRUCT` definition: the `MENU_ITEM` definition starts with `LONG` command, so the resource-reading code starts with:

```
iData.iCommandId=aReader.ReadInt32()
```

The next defined item is `LLINK` cascade, which is read by

```
iData.iCascadeId=aReader.ReadInt32()
```

and so on.

All resource reading functions will uncompress any compressed data.

Summary

In this chapter, we've seen:

- why a Symbian OS-specific resource compiler is needed
- a brief explanation of the syntax
- using bitmaps for the application icon
- the new registration files required by Symbian OS v9
- how to organize your resource text to ease the task of localization
- how to use multiple resource files
- how to read data from a resource file.

14

Views and the View Architecture

In general terms, a view can be defined as any class that enables you to display some or all of an application's data. All applications that display data in more than one way are described as having multiple views. A typical example is a calendar application, which would normally have at least a month view, a week view and a day view.

In Symbian OS, a view is defined more specifically as a class that implements an interface to the view architecture. The view architecture is the part of the control framework that provides the functionality for applications to make and receive requests to show a particular representation of their data. In addition to the view architecture providing the mechanism for an application to switch from one of its views to another, it also allows the operating system and other applications to communicate directly with an application's views.

As we have seen from many of the examples in earlier chapters, it is not necessary for applications to use the view architecture in order to provide displays of their data. However, UIQ 3 integrates the concept of a view so strongly into its UI that it rarely makes sense not to use views in UIQ applications. Furthermore, UIQ uses the view architecture to integrate the views from all the applications on a mobile phone, a feature that enables the user to switch between applications on the basis of the task being performed. An example of this would be to find a person's details in the contacts list and then to open a new text or email message, with that person's details automatically entered in the 'To:' field.

S60 also provides classes that support the view architecture interface, but the implementation is somewhat different. S60 uses the view architecture primarily for manipulating views within applications, rather than for integrating views from multiple applications.

14.1 The View Architecture

Symbian OS provides a view server with which each application may register its views. The relationship between the view server and registered application views is illustrated in Figure 14.1.

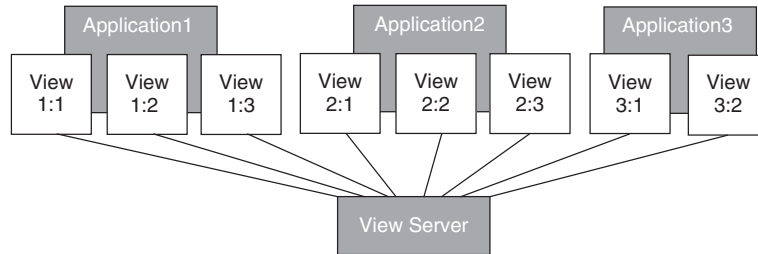


Figure 14.1 The view server and application views

In addition to enabling switching from one view to another, registering a view with the view server gives the following advantages:

- support for saving data – if a view is registered with the view server, the data for that view is always saved before the view is deactivated
- support for sending data – by packaging messages as descriptors identified by their UIDs, data can be sent from one view to another (within the same or different applications), via the view server.

The view server interface is entirely encapsulated by application UI functions and the Symbian OS view interface, `MCoeView`. This means that you, as a developer, never need to interact with it directly.

The only requirement on a Symbian OS view class is that it must be a C++ class that implements the `MCoeView` interface, as illustrated in Figure 14.2.

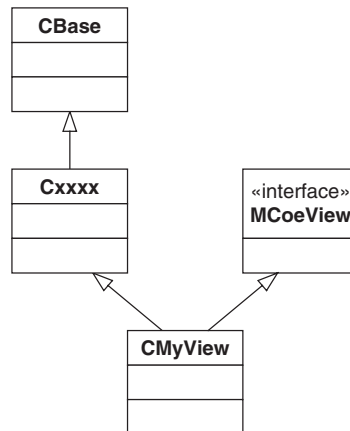


Figure 14.2 Generic derivation of a view

The view has to display application data, and may need to allow the user to interact with the data, so it has to provide at least some of the features of a control. However, for the purposes of defining a view, it doesn't matter how these features are supplied. This somewhat abstract specification allows views to be implemented in a variety of ways. It therefore isn't surprising that, although the underlying architecture is the same, the UIQ and S60 implementations differ.

UIQ Views

UIQ provides a base class for views, derived from `CCoeControl`. So, in UIQ, each view 'is a' control and, as shown in Figure 14.3, the architecture of a view-based application follows the underlying Symbian model very closely. Each view is owned by the application UI and the only additional programming required is to implement the view-related functions in the application UI and the pure virtual functions defined in `MCoeView`.

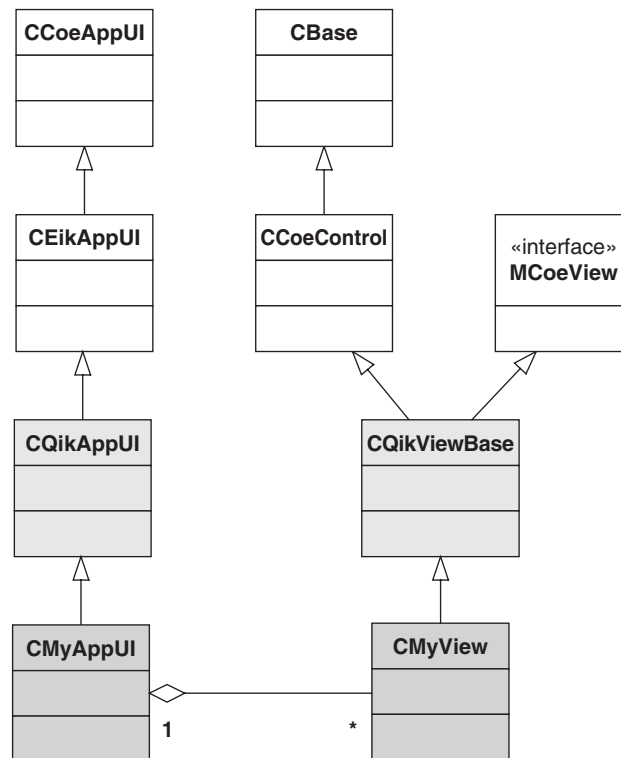


Figure 14.3 Derivation of a UIQ view

S60 Views

As illustrated in Figure 14.4, the S60 implementation is more specialized.

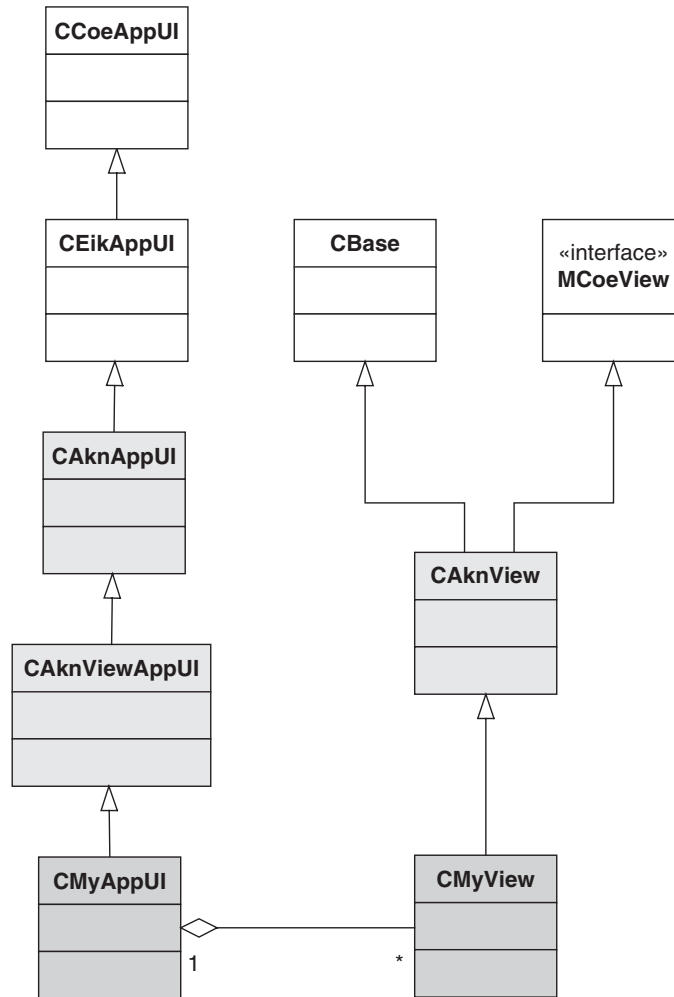


Figure 14.4 Derivation of an S60 view

In S60, the application UI is derived from the `CAknViewAppUi` class, rather than from the standard `CAknAppUi`. Also, the base class for views, `CAknView`, is not derived from `CCoeControl`. Unlike in UIQ, where a view is a control, an S60 view is normally implemented as a class that owns one or more `CCoeControl`-derived objects.

Portability

Creating portable views is likely to prove difficult for several reasons. In addition to the difference between the S60 and UIQ view base classes, each UI supports a variety of screen sizes, orientations and input methods. Each provides flexibility within its own view implementation to accommodate its own permutations, but each is dependent upon its own specific events and, crucially, its own UI-specific controls. Realistically, a portable view would have to be based either on Uikon or on the use of custom controls.

The Role of the Application UI

The view server interface itself is entirely encapsulated within the application UI, which provides mechanisms to create, register and activate views. Although the relevant functions are available in the `CCoeAppUi` class, UIQ and S60 applications need to use the versions provided in `CQikAppUi` and `CAknViewAppUi` respectively.

Views are normally constructed by the application UI. Both UIQ and S60 provide mechanisms for simplifying the process of constructing views, using resource files to specify some or all of their content. Once created, a view must be added to the view framework by means of a call to `AddViewL()` which, in S60, is prototyped as:

```
void AddViewL(CAknView *aView);
```

This call transfers ownership of the view to the view framework and registers the view with the view server, using the `RegisterViewL()` function, which is prototyped as:

```
void RegisterViewL(TVwsViewId);
```

The parameter to this call is a view ID, which is described in Section 14.2.

Once a view has been added to the view framework, you don't normally need to be concerned with its later removal, deregistration or destruction, as these processes are performed automatically by the application UI's destructor. If, during the lifetime of the application, you want to dynamically delete a view (other than the currently visible view) you must first call `RemoveView()`. This deregisters the view with the view server and removes it from the view framework. You are then free to call the view's destructor.

The process of displaying a selected view is known as activation. Conventionally, S60 applications activate only their own views but, in principle, provided that a view's unique ID is known (see Section 14.2), any application can activate any view.

In both S60 and UIQ, a view is activated by calling the application UI's `ActivateViewL()` function. Two overloads of this function are available, prototyped as follows:

```
void ActivateViewL(const TVwsViewId& aViewId);  
void ActivateViewL(const TVwsViewId& aViewId, TUid aCustomMessageId,  
                  const TDesc8& aCustomMessage);
```

The second of these allows you to send an additional message ID and an associated item of data to a view. There is no restriction on the content of the message, which can therefore be tailored to the requirements of a particular application. If you are activating a view in another application, you need to know about the messages and data that the target view can receive. There is no means of finding out this information at run time, so you have to rely on either the target application's exported header files or its documentation.

Another problem might occur in UIQ if the target application is not running when you attempt to activate one of its views. If this is likely to be the case, you can configure the view server to start the target application if it is not already running. See the UIQ SDK for further details.

Default Views

Each application has a default view. Typically, the default view is displayed when the application is first started and when it is brought to the foreground without a view being specified. This view must be constructed, registered and set as the default as early as possible during construction of the application. Depending on the nature of the application, construction and registration of non-default views could be deferred to a later time.

You specify the default view by means of a call to `SetDefaultViewL()`, and an application may use the same call to change its default view while it is running. If you do not specify a default view, the framework automatically chooses the one stored at index zero in its array of views.

Observers

Symbian OS provides a number of view observer interfaces. Of particular interest is the `MCoeViewDeactivationObserver` interface. You might implement this in floating windows, such as dialogs, which need to know when the view underneath is deactivated.

14.2 The MCoeView Interface

The view server communicates with a view through the MCoeView interface, whose class definition is shown below. It follows that every view must implement this interface.

```
class MCoeView
{
public:
    virtual TVwsViewId ViewId() const=0;
private:
    virtual void ViewActivatedL(const TVwsViewId& aPrevViewId,
        TUid aCustomMessageId, const TDesC8& aCustomMessage)=0;
    virtual void ViewDeactivatedL()=0;
protected:
    IMPORT_C virtual TVwsViewIdAndMessage ViewScreenDeviceChangedL();
    IMPORT_C virtual TBool ViewScreenModeCompatible(TInt aScreenMode);
private:
    IMPORT_C virtual void ViewConstructL();
    IMPORT_C virtual void PrepareForViewActivation();
};
```

All views have to implement the pure virtual functions, `ViewId()`, `ViewActivatedL()` and `ViewDeactivatedL()`; they normally also implement `ViewConstructL()`. The remaining functions have default implementations, described in the SDK, that the majority of applications do not need to override.

The View ID

The view's unique ID is an instance of the `TVwsViewId` class, which is composed of two parts. The first is the application's globally unique, Symbian-allocated UID, for example:

```
const TUid KAppUid = { 0x0257696A } ;    // Symbian allocated UID
```

The second is a number, also in the form of a `TUid`, that needs to be unique only within the application itself, such as:

```
const TUid KViewNumber = { 0x00000001 } ;    // View number
```

The two values are combined into a `TVwsViewId` instance as follows:

```
const TVwsViewId KViewId { KAppUid, KViewNumber } ;    // Unique view ID
```

This value must be returned by the view's `ViewId()` function. It is used both to register the view and to activate it. When a view is activated, it is passed the view ID of the previously active view, which is useful if the application needs to return from the current view to the one that was previously visible.

Construction

The construction of a view can be divided between the view's `ConstructL()` function and the `MCoeView` interface function, `ViewConstructL()`. Views supply a default `ViewConstructL()` function that does nothing, so you are free to choose whether or not to provide your own implementation.

The framework calls a view's `ViewConstructL()` immediately before the view is activated for the first time, so putting the bulk of the construction into `ViewConstructL()` can considerably improve an application's start-up response time, especially if the application has many views. However, you should avoid putting large amounts of time-consuming code into `ViewConstructL()`, since this may cause the view server to time out.

Activation and Deactivation

When a view is activated, the view server calls the view's `ViewActivatedL()` function, passing any custom message ID and message data that was specified in the call to the application UI's `ActivateViewL()`. In S60, `CAknView` implements `ViewActivatedL()` which, in turn, calls another pure virtual function, `DoActivateL()`. Depending on the UI, your derived view class must implement one of these two functions.

What your implementation does depends on your design. Typically, you process any passed message ID and data and make the view and its contents visible. You might also need to do some additional construction and layout.

The UIQ view server maintains only one system-wide active view so when a view is activated the view server calls `ViewDeactivated()` on any previously active view, in any application. The S60 view server maintains one active view for each running application, so the deactivation message is only sent when the new view is in the same application. As with `ViewActivatedL()`, S60's `CAknView` implements `ViewDeactivated()` and provides `DoDeactivateL()`. Again depending on the UI, your derived view class must implement one function or the other.

On deactivation, a view typically makes itself invisible. Any other actions depend on the nature and design of the application.

14.3 Introduction to the Example Application

Now that the basic principles have been described, the remainder of this chapter illustrates the use of views by extending the Noughts and Crosses example that was first introduced in Chapter 12. This extension adds a

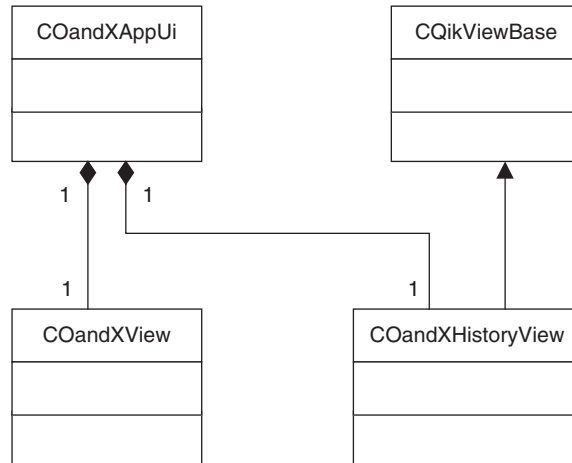


Figure 14.5 Multiple view ownership

second view, as illustrated in Figure 14.5, to display information about past games.

You can choose to display either a game history, which lists the results of the last six games, or the game statistics, which include the total number of games played and the number of wins for each player. Figure 14.6 shows the statistics displayed on a UIQ phone.

As you can see from this illustration, the view has its own menu of commands, which is the case for most applications with multiple views.



Figure 14.6 Viewing the game statistics

The data for the additional view, together with the code that generates it, is implemented within the game's `COandXController` class. The extension to this class is not described here, as it is not directly relevant to the workings of views; if you are interested in the implementation, you can look at the example code that is available for downloading from developer.symbian.com/main/academy/press.

One word of warning – the additional data is saved in the application's INI file. This means that the INI file is larger than it was for Chapter 12's version of the game. If you run this version after having run the earlier one, you must either first delete the old file (in the emulator it is located in `epoc32\winscw\c\Private\00404143`) or accept that the view's data is incorrect until you first select the Reset history menu item.

The History View

In order to keep the code simple, and to reduce the code differences between the S60 and UIQ versions, we've chosen to implement the new view as a collection of label controls. Figure 14.7 shows the class diagram for the UIQ version of the view.

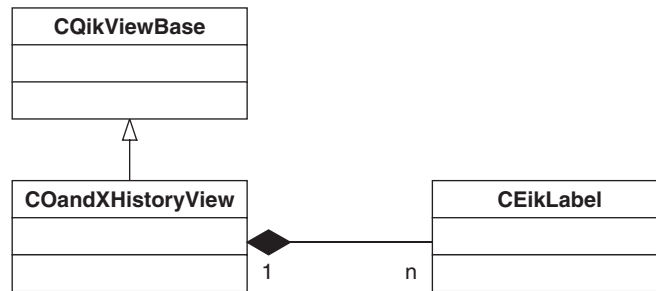


Figure 14.7 History view for UIQ

The relevant data members of the view class are defined in `oandx-histview.h` as follows:

```

const TInt KNumDataLines = KNumHistoryRecords+1;

class COandXHistoryView : public CQikViewBase
{
...
private:
...
    CEikLabel* iTitle;
    TFixedArray<CEikLabel*, KNumDataLines> iDataLines;
...
};
  
```

The value of `KNumHistoryRecords` is defined in `oandxdefs.h` as:

```
static const TInt KNumHistoryRecords = 6;
```

The component labels are logically divided into one title line and an array of seven data lines, the last of which is not used for displaying data.

In the UIQ version, the view's structure and layout can be specified in the application's resource script, `oandx.rss`, by a `QIK_VIEW_CONFIGURATIONS` item.

```
RESOURCE QIK_VIEW_CONFIGURATIONS r_history_view_ui_configurations
{
    configurations =
    {
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            view = r_history_view_layout;
            command_list = r_history_view_commands;
        },
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikSoftkeyStylePortrait;
            view = r_history_view_layout;
            command_list = r_history_view_commands;
        }
    };
}
```

The corresponding `QIK_VIEW` and `QIK_VIEW_PAGES` items are as follows:

```
RESOURCE QIK_VIEW r_history_view_layout
{
    pages = r_history_view_layout_pages;
}

RESOURCE QIK_VIEW_PAGES r_history_view_layout_pages
{
    pages =
    {
        QIK_VIEW_PAGE
        {
            page_id = EOandXHistoryViewPage;
            page_content = r_history_view_page_control;
        }
    };
}
```

The content and layout of the view's control are specified by the `QIK_CONTAINER_SETTINGS` item.

```

RESOURCE QIK_CONTAINER_SETTINGS r_history_view_page_control
{
    layout_manager_type = EQikRowLayoutManager;
    layout_manager = r_history_view_row_layout_default;
    controls =
    {
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewLabelCtrl;
            type = EEikCtLabel;
            control = r_history_view_title_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData1Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData2Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData3Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData4Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData5Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData6Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
        },
        QIK_CONTAINER_ITEM_CI_LI
        {
            unique_handle = EHistoryViewData7Ctrl;
            type = EEikCtLabel;
            control = r_history_view_data_label;
            // The following makes the Label fill the entire remaining
            // application space.
            layout_data = r_history_view_row_layout_data_fill;
        }
    };
}

```

Again, this is just an array of labels, the first of which is used for a title.

```
RESOURCE LABEL r_history_view_title_label
{
    horiz_align=EEikLabelAlignHCenter;
    standard_font=EEikLabelFontLegend;
    txt = ""; // Text supplied dynamically
}
```

The remaining labels are used to display data.

```
RESOURCE LABEL r_history_view_data_label
{
    horiz_align=EEikLabelAlignHLeft;
    txt = ""; // Text supplied dynamically
}
```

The control uses a default layout, which arranges the labels vertically, one above the other, each with its natural, default height. The last, unused label is set to occupy all the remaining space in the screen area allocated to the control.

The S60 version of the History view, whose class diagram is shown in Figure 14.8, uses the same collection of controls, but the implementation is slightly different.

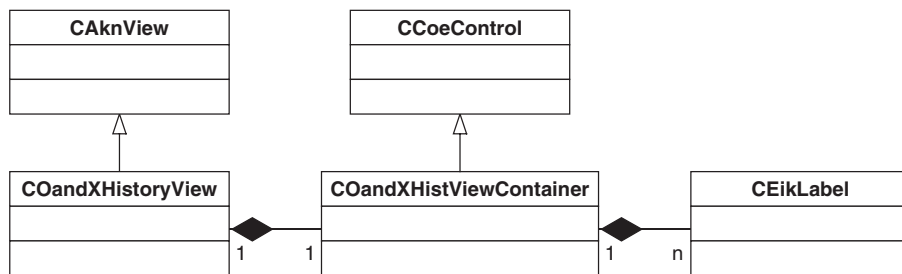


Figure 14.8 History view for S60

The view itself, defined in the S60 version of `oandxhistview.h`, owns a single container control, `COandXHistViewContainer`, which is implemented as a normal compound control.

```
class COandXHistoryView : public CAknView
{
    ...
private:
    COandXHistViewContainer* iContainer;
    ...
};
```

```

const TInt KNumDataLines = KNumHistoryRecords+1;

class COandXHistViewContainer : public CCoeControl
{
    ...
private:
    ...
    void ConstructL(const TRect& aRect);
    void Draw(const TRect& aRect) const;
    void SizeChanged();
    TInt CountComponentControls() const;
    CCoeControl* ComponentControl(TInt aIndex) const;
    ...

private:
    CEikLabel* iTitle;
    TFixedArray<CEikLabel*, KNumDataLines> iDataLines;
    ...
};

```

The container, in turn, owns the same collection of labels as in the UIQ version, but this time they are created in code from a single LABEL resource.

```

void COandXHistViewContainer::ConstructL(const TRect& aRect)
{
    // Create a window for this application view
    CreateWindowL();

    // Create components
    TResourceReader reader;
    iTitle = new (ELeave) CEikLabel();
    iTitle->SetContainerWindowL(*this);
    iEikonEnv->CreateResourceReaderLC(reader, R_HISTORY_VIEW_LABEL);
    iTitle->ConstructFromResourceL(reader);
    CleanupStack::PopAndDestroy(); // reader

    for (TInt i=0; i< KNumDataLines; i++)
    {
        iDataLines[i] = new (ELeave) CEikLabel();
        iDataLines[i]->SetContainerWindowL(*this);
        iEikonEnv->CreateResourceReaderLC(reader, R_HISTORY_VIEW_LABEL);
        iDataLines[i]->ConstructFromResourceL(reader);
        CleanupStack::PopAndDestroy(); // reader
        iDataLines[i]->SetLabelAlignment(ELayoutAlignLeft);
    }

    ...

    // Set the window's size
    SetRect(aRect); // needs to be after component creation - see
                    // SizeChanged()

    // Activate the window, which makes it ready to be drawn
    ActivateL();
}

```

The control's `SizeChanged()` function performs the layout of the labels as follows:

```
void COandXHistViewContainer::SizeChanged()
{
    TRect rect = Rect();

    TInt labelHeight = (rect.iBr.iY - rect.iTl.iY)/(KNumDataLines+1);
    rect.iBr.iY = labelHeight;
    iTitle->SetRect(rect);
    for (TInt i=0; i<KNumDataLines; i++)
    {
        rect.iTl.iY += labelHeight;
        rect.iBr.iY += labelHeight;
        iDataLines[i]->SetRect(rect);
    }
}
```

In this case, for simplicity, the available vertical space is divided equally between all the labels, including the last, unused one.

The control's `ComponentControl()` and `CountComponentControls()` functions are defined as follows:

```
CCoeControl* COandXHistViewContainer::ComponentControl(TInt aIndex) const
{
    if (aIndex==0)
    {
        return iTitle;
    }
    else
    {
        return iDataLines[aIndex-1];
    }
}

TInt COandXHistViewContainer::CountComponentControls() const
{
    return KNumDataLines+1;
}
```

Since the labels fill the entire area of the control, the `Draw()` function has nothing to do other than to clear the entire rectangle.

```
void COandXHistViewContainer::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc();
    gc.Clear();
}
```

Additional S60 Considerations

Unlike the corresponding UIQ version, the S60 version that was described in Chapter 12 did not use a view for its display of the game. Therefore the current S60 version of `oandxappview.h` needs to define an additional `CAknView`-derived class, `COandXGameView`. Following the pattern shown in Figure 14.7, this view class owns the game's main control (which has been renamed to `COandXAppViewContainer`).

When reading about the code required to support the History view, you will need to bear in mind that corresponding changes have to be made to the code of the Game view.

14.4 Creating and Managing the Views

As is generally the case, the Noughts and Crosses application's views are created and managed from the application UI (see Figures 14.3 and 14.4). For convenience, the application UI stores pointers to the views, as shown in the following extract from the UIQ class definition. They are non-owning pointers, since ownership of the views is transferred to the view framework as soon as they are created.

```
class COandXAppUi : public CQikAppUi
{
    ...
private:
    COandXView* iAppView;           // Non-owning pointers
    COandXHistoryView* iHistoryView; // to the two views
};
```

Construction and Registration

The application creates its views in the application UI's second-phase constructor, as illustrated below for the UIQ application.

```
void COandXAppUi::ConstructL()
{
    // Calls ConstructL that initiate the standard values.
    BaseConstructL();
    iEngine = COandXEngine::NewL();
    iController = COandXController::NewL();

    // Create the view and add it to the framework
    iAppView = COandXView::NewLC(*this);
    AddViewL(*iAppView);
    CleanupStack::Pop(); // game view

    // and the history view
    iHistoryView = COandXHistoryView::NewLC(*this);
    AddViewL(*iHistoryView);
}
```

```

CleanupStack::Pop(); // history view

SetDefaultViewL(*iAppView);

ReportWhoseTurn();
}

```

The code follows the same overall pattern for both S60 and UIQ. Once a view has been created, it is added to the framework by calling `AddViewL()`. At that point the framework takes over ownership of the view and calls `RegisterViewL()` to register the view with the view server. In UIQ, where the view is derived from `CCoeControl`, calling `AddViewL()` also adds the view to the control stack. In S60, where `CAknView` has no knowledge of the class used for the control, we have to include explicit code to add a view to the control stack.

The final task in creating the application's views is to specify the default view, by means of a call to the application UI's `SetDefaultViewL()`.

Activation and Deactivation

As is common in many applications, the Noughts and Crosses example calls `ActivateViewL()` in response to the user selecting a menu option. From the History view, selecting the Return to game menu option results in a call to the simpler of the two overloads.

```

void COandXHistoryView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        ...
    case EOandXDisplayGame:
        OandXAppUi()->ActivateViewL(TVwsViewId(KUidOandXApp,KUidOandXView));
        break;
        ...
    }
}

```

From the Game view, selecting either Game statistics or Game history results in a call to the second overload of `ActivateViewL()`.

```

void COandXView::HandleCommandL(CQikCommand& aCommand)
{
    switch(aCommand.Id())
    {
        ...
    case EOandXDisplayHistory :
        _LIT8(KDummy, "");
        OandXAppUi()->ActivateViewL(
            TVwsViewId(KUidOandXApp,KUidOandXHistoryView),
            TUid::Uid(EHistoryViewDisplayHistory),KDummy);
    }
}

```



```

        break ;
case EOandXDisplayStats :
    OandXAppUi()->ActivateViewL(
        TVwsViewId(KUIdOandXApp,KUIdOandXHistoryView),
        TUid::Uid(EHistoryViewDisplayStats),KDummy);

        break ;
    ...
}
}

```

In this case, the message ID specifies whether the view is to display either the history or the statistical data, and the message data is not used.

When the view is activated, it receives an event in the form of a call to its `ViewActivatedL()` function, which will include any message ID and data that was passed to the application UI's `ActivateViewL()` function. If there was a previously active view, it is automatically deactivated and it also receives an event to inform it of the deactivation.

14.5 Implementing the **MCoeView** Interface

As described in Section 14.2, the functions to be implemented are the following:

```

class MCoeView
{
public:
    virtual TVwsViewId ViewId() const=0;
private:
    virtual void ViewActivatedL(const TVwsViewId& aPrevViewId,
                               TUid aCustomMessageId,
                               const TDesC8& aCustomMessage)=0;
    virtual void ViewDeactivatedL()=0;
    ...
    IMPORT_C virtual void ViewConstructL();
    ...
};

```

Although the classes in which these functions are defined are differently derived in S60 and UIQ, the implementations are broadly similar.

The View IDs

The Noughts and Crosses application's two view IDs are constructed in a standard way from the application's globally unique UID, defined in `oandxapplication.h` as:

```

const TUid KUIdOandXApp = {0xE04E4143}; // 0e 'N' 'A' 'C'

```

and two values that are unique within the application. These values are defined in `oandxdefs.h` as:

```
const TUid KUidOandXView      = {0x00000001};
const TUid KUidOandXHistoryView = {0x00000002};
```

Each view's `ViewId()` function constructs and returns the appropriate view ID, for example:

```
TVwsViewId COandXHistoryView::ViewId() const
{
    return TVwsViewId(KUidOandXApp, KUidOandXHistoryView);
}
```

View Activation

In the UIQ version of Noughts and Crosses, the implementation of the Game view's `ViewActivatedL()` function is particularly simple:

```
void COandXView::ViewActivatedL(const TVwsViewId& /*aPrevViewId*/,
                                TUid /*aCustomMessageId*/, const TDesC8& /*aCustomMessage*/)
{
    MakeVisible(ETrue);
    DrawNow();
}
```

On activation, this view is not supplied with any message data and the view is already fully constructed. The ID of the previously activated view can be ignored, so all we need to do is to ensure that the view is made visible, and force it to be drawn.

The S60 implementation of `DoActivateL()` follows a similar pattern, with two minor differences.

```
void COandXGameView::DoActivateL(const TVwsViewId& /*aPrevViewId*/,
                                  TUid /*aCustomMessageId*/, const TDesC8& /*aCustomMessage*/)
{
    __ASSERT_ALWAYS(!iGameViewStacked, Panic(EOandXControlAlreadyStacked));
    AppUi()->AddToStackL(*this, iContainer);
    iGameViewStacked = ETrue;

    iContainer->MakeVisible(ETrue);
    iContainer->DrawNow();
}
```

First, an S60 view owns a control, rather than being derived from `CCoeControl`, so the calls to `MakeVisible()` and `DrawNow()` have to be made to the owned control, pointed to by `iContainer`.

The second difference is that, unlike in UIQ, adding an S60 view by means of the application UI's `AddViewL()` function does not add the view's control to the control stack. There are several ways of implementing the addition to, and subsequent removal from, the control stack. In this case, we've chosen to call the application UI's `AddToStackL()` each time a view is activated (and to remove it from the control stack when the view is deactivated). The code ensures that the addition is leave-safe (as described in Chapter 12) and that the control isn't added to the stack twice.

The History view activation, shown here for the UIQ version, needs a bit more processing.

```
void COandXHistoryView::ViewActivatedL(const TVwsViewId& /*aPrevViewId*/,
                                       TUid aCustomMessageId, const TDesC8& /*aCustomMessage*/)
{
    switch (aCustomMessageId.iUid)
    {
        {
            // switch using the message ID
            case EHistoryViewDisplayHistory:
                ChangeDisplayL(ETrue);
                break;
            case EHistoryViewDisplayStats:
                ChangeDisplayL(EFalse);
                break;
            default:
                // display as last time.
                ChangeDisplayL(iDisplayingHistory);
                break ;
        }
    }
}
```

This view needs to take note of the message ID and set itself to display either historical or statistical data, using a `ChangeDisplay()` function.

```
void COandXHistoryView::ChangeDisplayL(TBool aDisplayHistory)
{
    iDisplayingHistory = aDisplayHistory;
    UpdateCommandsL();
    CreateNewItemL(); // Called unconditionally, to update data every time
    MakeVisible(ETrue);
    DrawNow();
}
```

Apart from the differences mentioned earlier in this section, the S60 implementation of `DoActivateL()` in the History view is very similar. Its version of `ChangeDisplayL()` doesn't call `UpdateCommandsL()`, because of the different way that S60 applications modify their command menus. We'll come back to this, and also describe the action of `CreateNewItemL()`, in the History View Content section below.

View Deactivation

When a view is activated, the view server calls the `ViewDeactivated()` function of the view that was previously active. As with `ViewActivatedL()`, the functionality depends on the nature of the application. The simplest action is just to make the view invisible, as is done in the UIQ version of the History view.

```
void COandXHistoryView::ViewDeactivated()
{
    MakeVisible(EFalse);
}
```

The debug build of the Game view additionally saves the game state (including data used to construct the History view).

```
void COandXView::ViewDeactivated()
{
#ifdef _DEBUG
    OandXAppUi() -> SaveGameStateL();
#endif
    MakeVisible(EFalse);
}
```

This action is not normally necessary, at least in release builds of UIQ, since the framework will request the application to save its data as and when necessary, including when the application closes down. Including this action in the example illustrates a difference between the Game and History views: the game data can change while the Game view is active, but never changes while the History view is active.

In the S60 version, the implementations of `DoDeactivate()` are similar to those of UIQ's `ViewDeactivated()`. The only differences are the absence of debug code to save the game state and the addition of code to remove the view's control from the control stack. The control-stack error checking in the S60 code assumes that each call to `DoActivateL()` is paired with a later call to `DoDeactivate()`, and will therefore fail if the application activates a window in another application.

View Construction

The UIQ version of the History view performs the least amount of work possible in its `ConstructL()`.

```
void COandXHistoryView::ConstructL()
{
    // This should always be called in the concrete view implementations.
    BaseConstructL();
}
```

The bulk of the construction takes place in `ViewConstructL()`.

```
void COandXHistoryView::ViewConstructL()
{
    ViewConstructFromResourceL(R_HISTORY_VIEW_UI_CONFIGURATIONS);
    iTitle = LocateControlByUniqueHandle<CEikLabel>(EHistoryViewLabelCtrl);
    for (TInt i=0; i<KNumDataLines; i++)
    {
        iDataLines[i] = LocateControlByUniqueHandle<CEikLabel>
            (EHistoryViewData1Ctrl+i);
    }
    iEikonEnv->ReadResourceL(iNumGamesText, R_HISTORY_VIEW_GAMES_TEXT);
    iEikonEnv->ReadResourceL(iNumOWinsText, R_HISTORY_VIEW_OWINS_TEXT);
    iEikonEnv->ReadResourceL(iNumXWinsText, R_HISTORY_VIEW_XWINS_TEXT);
    iEikonEnv->ReadResourceL(iNumDrawsText, R_HISTORY_VIEW_DRAWN_TEXT);
    iEikonEnv->ReadResourceL(iStatOWonText, R_HISTORY_VIEW_O_WINNER_TEXT);
    iEikonEnv->ReadResourceL(iStatXWonText, R_HISTORY_VIEW_X_WINNER_TEXT);
    iEikonEnv->ReadResourceL(iStatDrawText, R_HISTORY_VIEW_NO_WINNER_TEXT);
    iEikonEnv->ReadResourceL(iHistoryTitle, R_HISTORY_VIEW_HISTORY_TITLE);
    iEikonEnv->ReadResourceL(iStatsTitle, R_HISTORY_VIEW_STATS_TITLE);
}
```

Since the UIQ History view's structure is entirely specified in the application's resource file, the bulk of the construction is done by a single call to `ViewConstructFromResourceL()`. The remaining code obtains (non-owning) copies of the pointers to the component label controls and then reads in, from the resource file, copies of the text items that are used to generate the view's content. These take the form of formattable text, such as:

```
RESOURCE TBUF r_history_view_games_text
{
    buf = "%d games played";
}
```

S60 views do not normally use `ViewConstructL()`. Construction of an S60 view follows a slightly different course, since it owns a control rather than being one. The History view's `ConstructL()` has to create the control.

```
void COandXHistoryView::ConstructL()
{
    BaseConstructL(R_OANDX_HISTORY_VIEW);
    iContainer = COandXHistViewContainer::NewL(ClientRect());
}
```

The data used to generate the view's content is owned by the control, rather than by the view, and is initialized in the control's `ConstructL()`.

```
void COandXHistViewContainer::ConstructL(const TRect& aRect)
{
```

```

...
iEikonEnv->ReadResourceL(iNumGamesText, R_HISTORY_VIEW_GAMES_TEXT);
iEikonEnv->ReadResourceL(iNumOWinsText, R_HISTORY_VIEW_OWINS_TEXT);
iEikonEnv->ReadResourceL(iNumXWinsText, R_HISTORY_VIEW_XWINS_TEXT);
iEikonEnv->ReadResourceL(iNumDrawsText, R_HISTORY_VIEW_DRAWN_TEXT);
iEikonEnv->ReadResourceL(iStatOWonText, R_HISTORY_VIEW_O_WINNER_TEXT);
iEikonEnv->ReadResourceL(iStatXWonText, R_HISTORY_VIEW_X_WINNER_TEXT);
iEikonEnv->ReadResourceL(iStatDrawText, R_HISTORY_VIEW_NO_WINNER_TEXT);
iEikonEnv->ReadResourceL(iHistoryTitle, R_HISTORY_VIEW_HISTORY_TITLE);
iEikonEnv->ReadResourceL(iStatsTitle, R_HISTORY_VIEW_STATS_TITLE);
...
}

```

The remainder of the container's `ConstructL()` code was described in Section 14.3.

History View Content

Although the topic is not directly related to the `MCoeView` interface functions, this is a convenient place to discuss the generation of the content for the Noughts and Crosses application's History view. The key function is `CreateNewItemsL()`, whose UIQ implementation is as follows:

```

void COandXHistoryView::CreateNewItemsL()
{
    // Clear all existing data
    TBuf<KFormatBufSize> itemName;
    iEikonEnv->ReadResourceL(itemName, R_HISTORY_VIEW_NO_DATA_TEXT);
    for (TInt i=0; i<KNumDataLines; i++)
    {
        iDataLines[i]->SetTextL(itemName);
    }

    // Insert the data to be displayed
    TInt played = Controller().GamesPlayed();
    if (iDisplayingHistory)
    {
        // Set the title in the label
        iTitle->SetTextL(iHistoryTitle);

        // Add all available hstory data to the listbox
        for (TInt i=0; i<KNumHistoryRecords; i++)
        {
            switch (Controller().GameRecord(i))
            {
                case ETileNought:
                    itemName.Format(iStatOWonText, played - i);
                    break;
                case ETileCross:
                    itemName.Format(iStatXWonText, played - i);
                    break;
                case ETileDraw:
                    itemName.Format(iStatDrawText, played - i);
                    break;
            }
        }
    }
}

```

```

        default:
            iEikonEnv->ReadResourceL(itemName, R_HISTORY_VIEW_NO_DATA_TEXT);
            break;
        }
        iDataLines[i]->SetTextL(itemName);
    }
}
else // Displaying statistics
{
    // Set the title in the label
    iTitle->SetTextL(iStatsTitle);

    // Total games played
    itemName.Format(iNumGamesText, played);
    iDataLines[0]->SetTextL(itemName);

    // Wins by Noughts
    TUInt oWins = Controller().WonByO();
    itemName.Format(iNumOWinsText, oWins);
    iDataLines[1]->SetTextL(itemName);

    // Wins by Crosses
    TUInt xWins = Controller().WonByX();
    itemName.Format(iNumXWinsText, xWins);
    iDataLines[2]->SetTextL(itemName);

    // Drawn (or abandoned) games
    itemName.Format(iNumDrawsText, played - oWins - xWins);
    iDataLines[3]->SetTextL(itemName);
}
}

```

The function's purpose is to set the text in each of the label controls used to display the History view's content. For efficiency, the labels are accessed by the pointers that were copied into member data by `ConstructL()`, rather than having to look them up each time they are needed.

Since the history data and statistical data will, in general, occupy different numbers of the labels, the first action is to clear them of all content.

If the view is displaying historical data, each available past result is read from the Controller and used to determine which text item to display. The text is formatted to contain the corresponding game number and then set into the appropriate label. A similar process is used to construct the text to be displayed if the view is to display statistical data.

The various text items are supplied from member data, rather than having to be repeatedly read from the resource file.

14.6 Command Menus

We've already seen, in Chapter 12, the basics of how command menus behave in both S60 and UIQ. From the point of view of implementation,

the main differences between the two approaches for an application with a single view are that:

- in S60, menu content is updated when the framework calls the application UI's `DynInitMenuPanel()` function, immediately before a menu is made visible, whereas UIQ requires the application to update the menu content itself at the time that the application's state changes
- UIQ provides a `HandleCommandL()` function in the view, whereas S60 implements it in the application UI.

The second of these differences largely disappears in applications that make use of the view architecture, since the S60 `CAknView` class provides a `HandleCommandL()` function. The default implementation does nothing and should be overridden in each of your application's views.

Changes for UIQ

In the example application, the only difference in the UIQ version of the Game view is that we have added two more commands, as shown below in the command resource, to select the additional displays.

```
RESOURCE QIK_COMMAND_LIST r_oandx_portrait_commands
{
    items =
    {
        // The first command is only visible in debug mode.
        QIK_COMMAND
        {
            id = EEikCmdExit;
            type = EQikCommandTypeScreen;
            // Indicate that this command will only be visible in debug
            stateFlags = EQikCmdFlagDebugOnly;
            text = STRING_r_oandx_close_debug_cmd;
        },
        QIK_COMMAND
        {
            id = EOandXNewGame;
            type = EQikCommandTypeScreen;
            text = "New game";
        },
        QIK_COMMAND
        {
            id = EOandXFirstPlayer;
            type = EQikCommandTypeScreen;
            text = "Noughts move first";
        },
        QIK_COMMAND
        {
            id = EOandXDisplayStats;
            type = EQikCommandTypeScreen;
            text = "Game statistics";
        },
    },
}
```



```

QIK_COMMAND
{
    id = EOandXDisplayHistory;
    type = EQikCommandTypeScreen;
    text = "Game history";
}
};
}

```

The History view also has its own set of commands.

```

RESOURCE QIK_COMMAND_LIST r_history_view_commands
{
    items=
    {
        // The first command is only visible in debug mode.
        QIK_COMMAND
        {
            id = EEikCmdExit;
            type = EQikCommandTypeScreen;
            // Indicate that this command will only be visible in debug
            stateFlags = EQikCmdFlagDebugOnly;
            text = STRING_r_oandx_close_debug_cmd;
        },
        QIK_COMMAND
        {
            id = EOandXDisplayStats;
            type = EQikCommandTypeScreen;
            text = "Game statistics";
        },
        QIK_COMMAND
        {
            id = EOandXDisplayHistory;
            type = EQikCommandTypeScreen;
            text = "Game history";
        },
        QIK_COMMAND
        {
            id = EOandXResetHistory;
            type = EQikCommandTypeScreen;
            text = "Reset history";
        },
        QIK_COMMAND
        {
            id = EOandXDisplayGame;
            type = EQikCommandTypeScreen;
            text = "Return to game";
        }
    };
}

```

The first three of these commands are ones that also appear in the Game view's command menu. Although a view-based application will normally have a different set of menu options in each of its views, the menus will frequently contain common items. In such a case it is perfectly

acceptable for the common commands to have not only the same text, but also the same ID.

We have already discussed the additional code to handle the new commands by activating the appropriate view. The updating of the available commands needs to be changed to take the additional commands into account. In the History view, for example, the updating code is:

```
void COandXHistoryView::UpdateCommandsL()
{
    iCommandManager.SetAvailable(*this, EOandXDisplayHistory,
                                !iDisplayingHistory);
    iCommandManager.SetAvailable(*this, EOandXDisplayStats,
                                iDisplayingHistory);
}
```

Since the UIQ version of the application in Chapter 12 already used a view, there are no other significant changes to be described.

Changes for S60

In Chapter 12, the S60 version of the Noughts and Crosses application did not use a view for its display, so there are more command-menu changes to describe.

As in the UIQ version, the Game view has two additional menu items.

```
RESOURCE MENU_PANE r_oandx_game_menu
{
    items =
    {
        MENU_ITEM
        {
            command = EOandXNewGame;
            txt = "New game";
        },
        MENU_ITEM
        {
            command = EOandXFirstPlayer;
            txt = "Noughts move first";
        },
        MENU_ITEM
        {
            command = EOandXDisplayStats;
            txt = "Game statistics";
        },
        MENU_ITEM
        {
            command = EOandXDisplayHistory;
            txt = "Game history";
        }
    };
}
```

The new menu resource for the History view is as follows:

```
RESOURCE MENU_PANE r_oandx_history_menu
{
    items =
    {
        MENU_ITEM
        {
            command = EOandXDisplayStats;
            txt = "Game statistics";
        },
        MENU_ITEM
        {
            command = EOandXDisplayHistory;
            txt = "Game history";
        },
        MENU_ITEM
        {
            command = EOandXDisplayGame;
            txt = "Show game";
        },
        MENU_ITEM
        {
            command = EOandXResetHistory;
            txt = "Reset history";
        }
    };
}
```

As with the UIQ version, the updating of the available commands needs to be modified. In the History view, the new `DynInitMenuPaneL()` is as follows:

```
void COandXHistoryView::DynInitMenuPaneL(TInt aResourceId,
                                          CEikMenuPane* aMenuPane)
{
    if (aResourceId == R_OANDX_HISTORY_MENU)
    {
        if (IsDisplayingHistory())
        {
            aMenuPane->DeleteMenuItem(EOandXDisplayHistory);
        }
        else
        {
            aMenuPane->DeleteMenuItem(EOandXDisplayStats);
        }
    }
}
```

The most significant change in the handling of commands is the need to implement `HandleCommandL()` in each view. In the Game view the implementation is as follows:

```
void COandXGameView::HandleCommandL(TInt aCommand)
{
    {
```

```

switch (aCommand)
{
    case EAknSoftkeyBack:
    {
        AppUi()->HandleCommandL(EEikCmdExit);
        break;
    }
    default:
    {
        AppUi()->HandleCommandL(aCommand);
        break;
    }
}
}

```

The implementation in the History view follows the same pattern.

Although it would be possible to handle the bulk of the commands in the views, as in UIQ, we've taken the decision in this case to delegate all responsibility for processing the commands to the application UI's `HandleCommandL()` function. Doing so reduces the differences between the old and new S60 examples, but at the cost of having more differences between the S60 and UIQ view-based examples than strictly necessary.

The application UI's implementation of `HandleCommandL()` becomes:

```

void COandXAppUi::HandleCommandL(TInt aCommand)
{
    _LIT8(KDummy, "");
    switch(aCommand)
    {
        case EEikCmdExit:
        case EAknSoftkeyExit:
        {
            SaveL();
            Exit();
        }
        break;
        case EOandXNewGame:
        {
            iController->Reset();
            iAppView->Container()->ResetView();
        }
        break;
        case EOandXFirstPlayer:
        {
            iController->SwitchTurn();
        }
        break;
        case EOandXDisplayStats:
        {
            if (iHistView->IsActivated())
            {
                iHistView->ChangeDisplayL(EOandXSetStats);
            }
        }
    }
}

```

```

else
{
    ActivateViewL(TVwsViewId(KUIdOandXApp, KUIdOandXHistoryView),
                  TUid::Uid(EHistoryViewDisplayStats), KDummy);
}
break;
}
case EOandXDisplayHistory:
{
    if (iHistView->IsActivated())
    {
        iHistView->ChangeDisplayL(EOandXSetHistory);
    }
    else
    {
        ActivateViewL(TVwsViewId(KUIdOandXApp, KUIdOandXHistoryView),
                      TUid::Uid(EHistoryViewDisplayHistory), KDummy);
    }
    break;
}
case EOandXDisplayGame:
{
    ActivateViewL(TVwsViewId(KUIdOandXApp, KUIdOandXView));
    break;
}
case EOandXResetHistory:
{
    Controller().ResetStats();
    iHistView->ChangeDisplayL(iHistView->IsDisplayingHistory());
    break;
}
default:
    break;
}
}

```

The decision to handle all commands in the application UI has made the processing of some of the commands a little more complicated. For example, the `EOandXDisplayHistory` command might have come from either the History view (when it is currently displaying statistical data) or from the Game view. In the first case, all that is needed is to call the view's `ChangeDisplayL()` function, while the second is asking for a true switch to the History view and needs a call to `ActivateViewL()`.

The processing would be simpler if it were clear from where the commands originated. One solution would be to transfer the handling of such commands into the originating view's `HandleCommandL()` function, as in the UIQ version. A less elegant solution would be to assign different command IDs to the two versions of each such command, depending on the view to which it belongs.

Summary

The Symbian OS view architecture supports the notion of task-based usage across multiple applications.

Though the concepts are similar, the implementations of the view architecture on different platforms are significantly different.

Your applications are not obliged to participate in the view architecture or to register any views. The view architecture will treat such applications as having a single, default view.

15

Controls

Controls are the principal means of interaction between an application and the user. This chapter covers the basic principles of writing and using controls, explaining what a control is and its place in UI development.

The examples provided in this chapter have deliberately been kept general but, where applicable, the differences between the S60 UI and the UIQ UI are explained. You still need to consult the Style Guide and other documentation in the appropriate SDK for information on topics specific to a particular phone, such as preferred layout and color schemes.

15.1 What Is a Control?

In Symbian OS, controls provide the principal means of interaction between an application and the user. Applications make extensive use of controls: each application view is a control, and controls form the basis of all dialogs and menu panes.

Every control is derived from the abstract class `CCoeControl` (defined in `coecntrl.h`), which gives fundamental features and capabilities to a control. A control occupies a rectangular area on the screen and, in addition to responding to user-, application- and system-generated events, may display any combination of text and image.

Depending upon the particular user interface, the user-generated events may include:

- key presses, either alphanumeric or from device-specific buttons
- pointer events, generated by the user touching the screen with a pen.

Drawing a control's content can be initiated in two ways. The application itself can initiate the drawing, for example when the control's displayable data changes. Alternatively, the system may initiate drawing, for example when all or part of the control is exposed by the disappearance of an overlying control belonging to the same or another application.

Symbian OS is a full multitasking system, in which multiple applications may run concurrently, and the screen is a single resource that must be shared among all these applications. Symbian OS implements this sharing by associating one or more *windows* with each application, to handle the interaction between the controls and the screen. The windows are managed by the *window server*, which ensures that the correct window(s) are displayed, managing overlaps and exposing and hiding windows as necessary.

In order to gain access to the screen, each control must be associated with a window. However, there is not necessarily a separate window for each control. Some controls, known as *window-owning controls*, use an entire window, but many others, known as non-window-owning controls or *lodger controls*, share a window owned by another control.

You can test whether a control owns a window by checking whether `OwnsWindow()` returns `ETrue` or `EFalse`, although its main use is within `CCoeControl`'s framework code.

For a system such as Symbian OS, which runs on devices with limited power and resources, windows are expensive. Each window uses resources in its associated application, and also needs corresponding resources in the window server. Furthermore, each window results in additional client-server communication between the application and the window server. It therefore makes sense to use lodger controls as much as possible. Fortunately, there isn't a great deal of difference in programming terms between the two types.

Any application with a graphical user interface requires access to the screen, and so must contain at least one window-owning control. Other controls might need to own a window if they require the properties of a window, such as the ability to overlap another window.

15.2 Control Types

Controls may be either simple or compound. A compound control contains one or more component controls; a simple control does not.

Simple Controls

Perhaps the simplest of simple controls is one that draws a blank rectangle and does not respond to any user- or system-generated events. As an example, we'll look at the application view used in `HelloBlank`, a 'Hello World' application that uses such a blank control for its view. The class definition for the application view is as follows:

```
class CBlankAppView : public CCoeControl
{
public:
    void ConstructL(const TRect& aRect);
};
```

We don't need the class constructor or destructor to do anything special, so we'll simply rely on the default implementations. For this control, the only member function we need to write is `ConstructL()`, the second-stage constructor.

```
void CBlankAppView::ConstructL(const TRect& aRect)
{
    CreateWindowL(); // Create a window for this control
    SetRect(aRect);  // Set the control's size
    SetBlank();      // Make it a Blank control
    ActivateL();     // Activate control, making it ready to be drawn
}
```

The four functions that `ConstructL()` calls are all, in one way or another, associated with enabling the control to display itself.

Since all controls must be associated with a window, and this is the only control in the application, it follows that it has to be a window-owning control. The `CreateWindowL()` function both creates the window and sets the control as its owner.

`SetRect()` sets the size and position, in pixel units, of the control's rectangle on the screen, relative to its associated window. Since this control owns a window, the window's size is also adjusted to that of the control. A control is responsible for ensuring that it is capable of drawing every pixel within the specified area and that its content is correctly positioned within this rectangle.

The call to `SetBlank()` is what makes this a blank control. All it does is enable the control subsequently to fill its rectangle with a plain background color. How it does this is explained in Section 15.7.

Before the final call to `ActivateL()`, drawing of the control remains disabled. For drawing to be enabled, the control must be both visible and activated. Controls are visible by default but need explicit activation by means of a call to `ActivateL()`, because controls are not always in a fit state to draw their content immediately after construction. For example, the data that a control is to display may not be available at the time of its construction, and this may also affect the control's required size and position. In such a case, the call to `ActivateL()` would not be made from within `ConstructL()`, but would be called from elsewhere, once the necessary additional initialization is complete.

The blank control class is instantiated as an application view from the application UI class.

```
void CMyApplicationAppUi::ConstructL()
{
    BaseConstructL(EAknEnableSkin);
    iAppView = new(ELeave) CBlankAppView;
    iAppView->ConstructL(ClientRect());
}
```

Compound Controls

To illustrate some of the differences between simple and compound controls, we use the Noughts and Crosses example application that was introduced in Chapter 12. As a reminder, the appearance of the application is shown in Figure 15.1.

The implementation uses an array of tiles, derived from `CCoeControl`, as component controls in the application view. The tiles supply all the mechanisms for displaying their content and focusing them, using either the cursor keys or the pointer. In this chapter we concentrate on those features that are necessary to allow a control to include component controls, rather than on the internal implementation of the tiles.

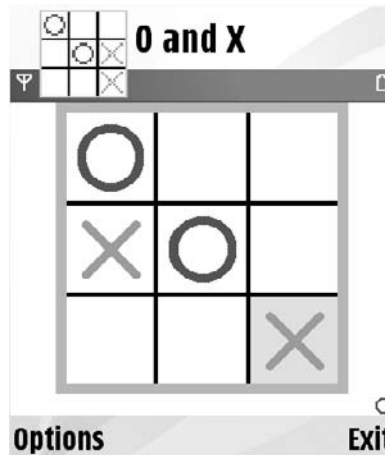


Figure 15.1 The Noughts and Crosses application

As mentioned in Chapter 12, the view needs to be added to the control stack so that it has the opportunity to process key presses. This is done by calling `AddToStackL(iAppView)`, normally from the `ConstructL()` function of the application UI class. We see the consequences later, during the discussion of the view itself.

The control stack maintains a prioritized list of all controls that have an interest in processing key events and ensures that key events are

offered to the controls in the order that they were added. As well as controls from GUI applications, the list contains controls associated with any front-end processor (FEP), with the processing of debug keys (Ctrl, Alt, Shift, key combinations) and with active dialogs.

If any view has been placed on the control stack, you must ensure its later removal, using code of the form:

```
iEikonEnv->RemoveFromStack(iAppView);
```

The application's view class is defined as follows:

```
class COandXAppView : public CCoeControl, public MCoeControlObserver,
                    public MViewCmdHandler
{
public:
    static COandXAppView* NewL(const TRect& aRect);
    virtual ~COandXAppView();

    // From CCoeControl
    TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                TEventCode aType);

    // new functions
    void MoveFocusTo(const TInt aIndex);
    TInt IdOfFocusControl();
    void ShowTurn();
    void ResetView();

private:
    COandXAppView();
    void ConstructL(const TRect& aRect);

    void SwitchFocus(TInt aFromIndex, CCoeControl* aToControl);
    void DrawComps(TRect& aRect) const;
    COandXTile * CreateTileL();

    // From CCoeControl
    void Draw(const TRect& aRect) const;
    void SizeChanged();
    TInt CountComponentControls() const;
    CCoeControl* ComponentControl(TInt aIndex) const;

    // From MCoeControlObserver
    void HandleControlEventL(CCoeControl* aControl, TCoeEvent aEventType);

    // From MViewCmdHandler
    TBool TryHitSquareL(const COandXTile* aControl);
    TTileState TileStatus(const COandXTile* aControl) const;

private:
    RPointerArray<COandXTile> iTiles; // View owns the tiles
    COandXStatusWin* iStatusWin;      // and its own status window.
    TRect iBoardRect; // Board area
```

```

TRect iBorderRect; // Bounding rectangle for border
TInt iTileSide;    // Tile dimension, allowing for
                  // line widths and border
};

```

The second-stage constructor is implemented as follows:

```

void COandXAppView::ConstructL(const TRect& aRect)
{
    // Create a window for this application view
    CreateWindowL();

    for (TInt i = 0; i < KNumberOfTiles; i++)
    {
        User::LeaveIfError(iTiles.Append(CreateTileL()));
    }
    ComponentControl(0) -> SetFocus(ETrue);
    iStatusWin = COandXStatusWin::NewL(Window());

    // Set the window's size
    SetRect(aRect); // needs to be after component creation - see
                  // SizeChanged()
    // Activate the window, which makes it ready to be drawn
    ActivateL();
}

```

Apart from the regular tasks of creating a window, setting its size and activating it, the majority of the code is concerned with creating and initializing the tile control and status window instances.

Setting a component control to have *keyboard focus* normally has an effect on the appearance of the component. It usually highlights itself in some way and, if the component accepts text, may cause a text cursor to become visible. Setting and unsetting keyboard focus also has other implications, as discussed in the next section.

Support for the component controls is supplied by implementations of two `CCoeControl` functions: `CountComponentControls()` and `ComponentControl()`. Since the view needs to respond to key presses, we also need an implementation of `CCoeControl`'s `OfferKeyEvent()` function.

Supplying implementations for `CountComponentControls()` and `ComponentControl()` is mandatory for any compound control. The two functions work as a pair and should always be coded to be consistent with each other. The control framework calls both functions in order to access the component controls, using code of the form:

```

...
for (TInt i=0; i<CountComponentControls(); i++)
{
    CCoeControl* component = ComponentControl(i);
    ...
}
...

```

Chapter 17 provides more information about these functions; the implementations for the Noughts and Crosses game are fairly straightforward.

```
TInt COandXAppView::CountComponentControls() const
{
    return KNumberOfTiles + 1;
}

CCoeControl* COandXAppView::ComponentControl(TInt aIndex) const
{
    if (aIndex==KNumberOfTiles)
    {
        return iStatusWin;
    }
    else
    {
        return const_cast<COandXTile*>{iTiles[aIndex]};
    }
}
```

15.3 Control Layout

A control is responsible for the layout of its content. If it is a compound control, it is also responsible for setting the position and size of its component controls. If a control's size is set when it is created and does not subsequently change, it can usually set the position and size of any components within its `ConstructL()` function. If a control's size is likely to change during its lifetime, then this approach will not work and the control will need to rework its content layout accordingly. The control framework provides a `SizeChanged()` function specifically to meet this need. Its default implementation is empty.

Although the view of the Noughts and Crosses application does not change size after its creation, we've used `SizeChanged()` in this application view to calculate and set the size and position of the border surrounding the playing area and the component tiles. It's coded to cope with different screen sizes and alternative board layouts, containing different numbers of tiles.

Although it looks a little daunting at first sight, the `SizeChanged()` function consists of a small handful of sections, each of which performs relatively simple geometrical calculations.

```
void COandXAppView::SizeChanged()
{
    // all component tiles must already exist
    __ASSERT_DEBUG(iTiles[KNumberOfTiles-1], Panic(EOandXNoTiles));
    TRect rect = Rect();
    rect.iTl.iY = rect.iBr.iY - KStatusWinHeight;
```

```

iStatusWin->SetRect(rect);
rect = Rect();
rect.iBr.iY -= KStatusWinHeight;
TSize controlSize = rect.Size();
TSize tileSize;
tileSize.iWidth=2*((controlSize.iWidth-2*KBorderWidth
    -(KTilesPerRow-1)*KLineWidth)/(2*KTilesPerRow));
tileSize.iHeight=2*((controlSize.iHeight-2*KBorderWidth
    -(KTilesPerCol-1)*KLineWidth)/(2*KTilesPerCol));
iTileSide = tileSize.iWidth < tileSize.iHeight ?
    tileSize.iWidth :tileSize.iHeight;
TSize boardSize;
boardSize.iWidth = KTilesPerRow*iTileSide +
    (KTilesPerRow-1)*KLineWidth;
boardSize.iHeight = KTilesPerCol*iTileSide +
    (KTilesPerCol-1)*KLineWidth;
iBoardRect.iTl.iX = (controlSize.iWidth - boardSize.iWidth)/2;
iBoardRect.iTl.iY = (controlSize.iHeight - boardSize.iHeight)/2;
iBoardRect.iBr.iX = iBoardRect.iTl.iX + boardSize.iWidth;
iBoardRect.iBr.iY = iBoardRect.iTl.iY + boardSize.iHeight;
iBorderRect = iBoardRect;
iBorderRect.Grow(KBorderWidth,KBorderWidth);

for (TInt i=0; i<KNumberOfTiles; i++)
{
    TInt row = i / KTilesPerRow;
    TInt col = i % KTilesPerRow;
    TRect tileRect;
    tileRect.iTl.iX = iBoardRect.iTl.iX + col * (iTileSide + KLineWidth);
    tileRect.iTl.iY = iBoardRect.iTl.iY + row * (iTileSide + KLineWidth);
    tileRect.iBr.iX = tileRect.iTl.iX + iTileSide;
    tileRect.iBr.iY = tileRect.iTl.iY + iTileSide;
    ComponentControl(i)->SetRect(tileRect);
}
}

```

The code first performs independent calculations of the tile width and height that will enable the required number of tiles, and their borders, to fit in the available area, making sure that the tile dimensions are always an even number of pixels. It takes the smaller of the two dimensions and uses it for the side of the tile squares.

It is then a fairly simple matter to calculate overall board and border positions, which are stored in member data for later use when drawing the board. Finally, it calculates the tile positions and sets them by calling each tile's `SetRect()` function.

In terms of designing an application to work with a variety of display sizes, it is interesting to note the division of effort used in calculating the positions and sizes of the various elements within the view. As we've just seen, the majority of the work is done in the view's `SizeChanged()` function. This is called relatively infrequently, when the view is initialized or when its size changes, and so it makes sense to use it to do as much of the work as possible. In consequence, the view's drawing function has

relatively little to do, other than actually drawing the various features of the board.

`SizeChanged()` is called whenever one of the control's size-changing functions is called on the control. These include:

- `SetExtent()`
- `SetSize()`
- `SetRect()`
- `SetCornerAndSize()`
- `SetExtentToWholeScreen()`.

In Noughts and Crosses, the call is triggered by the call to `SetRect()` in the view's `ConstructL()` function:

```
void COandXAppView::ConstructL(const TRect& aRect)
{
    // Create a window for this application view
    CreateWindowL();

    for (TInt i = 0; i < KNumberOfTiles; i++)
    {
        User::LeaveIfError(iTiles.Append(CreateTileL()));
    }
    ComponentControl(0)->SetFocus(ETrue);
    iStatusWin = COandXStatusWin::NewL(Window());

    // Set the window's size
    SetRect(aRect); // needs to be after component creation - see
                    // SizeChanged()
    // Activate the window, which makes it ready to be drawn
    ActivateL();
}
```

All we have to do is to ensure that `SetRect()` is not called until after the tiles have been created.

There is no system-wide mechanism for notifying controls of size changes, and `SizeChanged()` is called only as a result of calling one of the five functions listed above. In consequence, if resizing one control affects the size of other controls, it is the application's responsibility to ensure that it handles the resizing of all affected controls.

There are three other functions that are associated with changes in the size or position of a control:

- `SetSizeWithoutNotification()` sets a control's size without calling `SizeChanged()`.
- `SetPosition()` sets the pixel position of the top-left corner of the control. If the control owns its containing window, the function

achieves this by setting the position of the window; otherwise the position of the control is set relative to its containing window. The positions of the control's components are adjusted accordingly, and `PositionChanged()` is called.

- `PositionChanged()` responds to changes in the position of a control and is called whenever the application calls `SetPosition()` on the control. It has an empty default implementation.

15.4 Handling Key and Pointer Events

Whether an application needs to handle both types of event depends in part on the target phone's UI. One of the design considerations for the S60 UI is that it should support one-handed operation. In consequence, S60 phones do not have touch-sensitive screens, which require holding the phone in one hand and the stylus in the other, so applications written specifically for this UI do not need to handle pointer events.

Applications running on pointer-based phone UIs, such as UIQ, generally have to handle both types of event. Such phones generally have some keys, and also use front-end processors (FEPs) associated with, say, handwriting recognition, or a virtual (on-screen) keyboard, to generate key events.

Whether you choose to handle one or both types of event may also depend on the nature of your application. In general, if there are no specific reasons to the contrary, handling both types of event will make it easier to convert your application to run on a phone that uses a different UI. Conforming to the general interaction style of an existing phone is generally a good idea.

The two types of event have some fundamental differences. A pointer event occurs at a specific position on the screen, which usually corresponds to one particular application's window and, more precisely, one particular control associated with that window. A key event has no such intrinsic connection to a particular control: it is the internal logic of an application that determines this connection, and it is often the case that different types of key event are processed by different controls within the application.

This fundamental difference is reflected in the way that Symbian OS handles the two types of event.

Key Events

Key events are system events that usually originate from the keyboard driver. The keyboard driver passes such events to the window server which, in turn, offers them to one or more of the foreground application's controls by a mechanism that is described later in this section. Key events

from other sources (such as a FEP) may follow a different route, but are received and handled by a control in exactly the same way as those that originate from the keyboard driver.

In UIQ, the primary method of text input is the front-end processor. UIQ provides two FEPs: handwriting recognition and an on-screen virtual keyboard. (Other FEPs may be written by third parties.) Both FEPs receive pointer events and translate them into key events, which are passed to the application. Whether the key event is generated by a FEP or a physical keyboard shouldn't make any difference to your application. The FEP architecture was designed so that applications don't need to be FEP-aware.

In principle, all Symbian OS applications are expected to respond to key events.

Representation of key events

A key event is represented by an instance of `TKeyEvent` and a key-event type, which may be `EEventKeyDown`, `EEventKey` or `EEventKeyUp`. Every time a key is pressed, all three key-event types are generated. Unless your application is particularly interested in detecting when a key is pressed or released, you can safely ignore key events of types other than `EEventKey`.

The `TKeyEvent` class has four data members.

- `iCode` contains the key's character code, and is usually of most significance to an application.
- `iModifiers` contains the state of any modifier keys, such as Control or Shift, as defined in the `TEventModifier` class.
- `iRepeats` contains the number of auto-repeat events, 0 signifying an event without repeats. It is normal to ignore this value.
- `iScanCode` contains the scan code of the key that caused the event. Standard scan codes are defined in `TStdScanCode`. In general, this value is useful only to game applications for which the position of the key on the keyboard is more important than the character it represents.

The legal values for `iCode`, `iScanCode`, and `iModifiers` are defined in `e32keys.h`. UIQ defines some additional key codes in `quartzkeys.h`, to represent the events generated by the two or four direction keys on the keypad, and the Confirm key.

Values from the `TKeyCode` enumeration are used to interpret `iCode`. The values are `<0x20` for non-printing Unicode characters,

0x20 – 0xf7ff for printing Unicode characters, and $\geq 0xf800$ for the usual function, arrow, menu, etc. keys found on PC keyboards. Although Symbian OS phones may have no keyboard, many of these key events may still be generated using a FEP.

It is very rare for scan codes to be of interest. The scan codes are defined by the historical evolution of IBM PC keyboards since 1981: they originally represented the physical position of a key on the 81-key IBM PC keyboard, and have evolved since then to support new keyboards and preserve compatibility with older ones. Since keyboards and keypads used on Symbian OS phones are rather different, scan codes have limited value.

Distributing key events

The control stack is used to pass key events to an application's controls. To register its interest in processing key events, a control must be added to the control stack by means of a call to the `AddToStackL()` function of the application UI class. A control should also override the `OfferKeyEventL()` method and provide its own key-handling behavior.

All key events that may potentially be handled by controls, whether they are generated from a real key press or from a FEP, are processed by the control stack's `OfferKeyL()` function. This calls the `OfferKeyEventL()` function of each object on the control stack until either:

- all the controls have been offered the key event and have indicated, by returning `EKeyWasNotConsumed`, that they cannot process the event
- a control can process the key event, and indicates that it has done so by returning the value `EKeyWasConsumed`.

It follows that a control's implementation of `OfferKeyEventL()` must ensure that it returns `EKeyWasNotConsumed` if it does not do anything in response to a key event, otherwise other controls or dialogs may be prevented from receiving the key event. The default action of `CCoeControl`'s `OfferKeyEventL()` function is just to return `EKeyWasNotConsumed`.

Processing key events

A compound control may process and consume keys and/or may offer key events to its component controls. A typical pattern is for the container control to process cursor key events, using them to navigate between its component controls. Other key events may be passed to one or other of the components.

The Noughts and Crosses application demonstrates the typical response to key events. It doesn't have to check whether it should have been offered

the key event: the framework will only offer such an event if the control is supposed to have it.

The application doesn't have to consume all key events that it is offered: the view's `OfferKeyEventL()` function takes no action for events other than `EEventKey` and consumes only the up, down, left and right cursor key events when they can successfully be used to navigate between the tiles.

```

TKeyResponse COandXAppView::OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                             TEventCode aType)
{
    TKeyResponse keyResponse = EKeyWasNotConsumed;
    if (aType!=EEventKey)
    {
        return keyResponse;
    }
    TInt index = IdOfFocusControl();
    switch (aKeyEvent.iCode)
    {
        case EKeyLeftArrow: // check not in first column
            if (index % KTilesPerRow)
            {
                MoveFocusTo(index-1);
                keyResponse = EKeyWasConsumed;
            }
            break;
        case EKeyRightArrow: // check not in last column
            if ((index % KTilesPerRow) < KTilesPerRow - 1)
            {
                MoveFocusTo(index+1);
                keyResponse = EKeyWasConsumed;
            }
            break;
        case EKeyUpArrow: // check not on top row
            if (index >= KTilesPerRow)
            {
                MoveFocusTo(index-KTilesPerRow);
                keyResponse = EKeyWasConsumed;
            }
            break;
        case EKeyDownArrow: // check not in bottom row
            if (index < KNumberOfTiles - KTilesPerRow)
            {
                MoveFocusTo(index+KTilesPerRow);
                keyResponse = EKeyWasConsumed;
            }
            break;
        default:
            keyResponse = ComponentControl(index)->
                OfferKeyEventL(aKeyEvent, aType);
            break;
    }
    return keyResponse;
}

```

The view finds the current tile (the one which has the focus) with a call to `IdOfControlWithFocus()`.

```

TInt COandXAppView::IdOfControlWithFocus()
{
    TInt ret = -1;
    for (TInt i=0; i<CountComponentControls(); i++)
    {
        if (ComponentControl(i)->IsFocused())
        {
            ret = i;
            break;
        }
    }
    __ASSERT_ALWAYS(ret>=0, Panic(EOandXNoCurrentTile));
    return ret;
}

```

If the requested move is allowed – for example, an `EKeyUp` event is allowed only if the current tile is not in the top row – the required change in the tile index is determined and focus is transferred to the new current tile by calling `SetFocus()`.

If the key event is not a cursor key, the view just passes it to the current tile by calling its `OfferKeyEventL()`.

```

TKeyResponse COandXTile::OfferKeyEventL(const TKeyEvent& aKeyEvent,
                                         TEventCode aType)
{
    if (aType!=EEventKey)
    {
        return EKeyWasNotConsumed;
    }
    TKeyResponse keyResponse;
    switch (aKeyEvent.iCode)
    {
        case EKeyOK:
            TryHitL();
            keyResponse = EKeyWasConsumed;
            break;
        default:
            keyResponse = EKeyWasNotConsumed;
            break;
    }
    return keyResponse;
}

```

The tile is only interested in the key event with key code `EKeyOK`, corresponding to an ‘enter’ press on an S60 phone’s joystick control. The call to `TryHitL()` does nothing if the current tile already contains a nought or a cross, otherwise it sets the tile to contain a nought or a cross, depending on whose turn it is.

The above code usefully demonstrates the three main ways in which a key event may be handled:

- ignore it, if it is a key that is either not recognized or not needed at that time

- handle it directly in the control
- use it to generate another event or command (such as `TryHitL()`) that may be processed elsewhere.

Key events are only offered to controls that have been added to the control stack. If such a control is a compound control, it must explicitly offer the event to any of its component controls that may be capable of processing the event.

Input capabilities

In general, a control that supplies its own `OfferKeyEventL()` should also provide a matching implementation of `InputCapabilities()`. FEPs call this function, via the application UI and the control framework, to determine what classes of key event are appropriate to send to the currently focused control. In the case of a compound control, the FEP receives an *or ed* combination of the input capabilities of the control and all focused component controls. In consequence, it is not always necessary for a container control to override `CCoeControl`'s default implementation (which returns `TCoeInputCapabilities::ENone`). In the Noughts and Crosses example, only the `COandXTile` class overrides this function:

```
TCoeInputCapabilities COandXTile::InputCapabilities() const
{
    return TCoeInputCapabilities::ENavigation;
}
```

This information is supplied to the FEP to help it to optimize its performance. The nature of the optimization is left to the FEP itself, so you should not assume that a control will never receive key events other than those specified in its own `InputCapabilities()` function and those of its component controls.

Focus

The basic concept of *focus* – sometimes referred to as *keyboard focus* – is fairly straightforward. In previous discussions the ‘current’ tile was equated with the one that had ‘focus’, and that is essentially the case. In any compound control there is generally one component that is the current control of interest, the one with which the user is interacting, for example:

- typing text into an edit box
- selecting one item from those in a list box
- changing a date in a date editor.

You might be tempted to define the control with focus as being the one to which key events are offered, but that is not entirely true. Certainly, some key events are directed to the control with focus, but others are not. Key events might be offered to, and consumed by, other types of control before there is any chance of their being offered to the control with focus. Such potential consumers of keys include FEPs, dialogs and the application's menu bar.

Some FEPs consume key events as well as generate them. An example is one that consumes a key-event sequence to generate a single Chinese character. Such FEPs own a control that is placed on the control stack at a higher priority than visible controls, giving it first refusal of all key events.

All controls are capable of having focus by default. You can set a control to be incapable of receiving focus by calling `SetFocusing(ETFalse)`, and check if a control is in this state by calling `IsNonFocusing()`. A non-focusing control can be made capable of receiving focus by calling `SetFocusing(ETTrue)`.

Only one component control should have focus at any one time, and it is the application's responsibility to ensure that this is obeyed by its views. A container can find out if a control has focus by calling its `IsFocused()` function, and can set or remove focus on one of its component controls with that control's `SetFocus()` function.

In addition to setting or removing focus on a control, `SetFocus()` also calls the control's `FocusChanged()` function. The parameter passed to this latter function is `EDrawNow` if the control is visible and activated, otherwise `ENoDrawNow`. The purpose of this function is to give the control an opportunity to change its appearance in response to the change in focus. The default implementation is empty, so controls should implement it as and when necessary. One simple implementation of this function can be the following:

```
void COandXTile::FocusChanged(TDrawNow aDrawNow)
{
    if (aDrawNow == EDrawNow)
    {
        DrawNow();
    }
}
```

There are two other focus-related functions, `PrepareForFocusGainL()` and `PrepareForFocusLossL()`. The control framework does not call them, and their default implementations are empty.

Pointer Events

Pointer events are system events that originate in the digitizer driver, which passes the events to the window server. The window server associates the pointer event with a particular window and sends the event to the application that owns the window group containing that window.

As discussed earlier, S60-specific controls have no need to respond to pointer events, but all the general-purpose controls available in Symbian OS, and all UIQ-specific controls, respond to pointer as well as key events.

Representation of pointer events

A pointer event is represented by an instance of a `TKeyEvent`, defined in `w32std.h`.

```
struct TPointerEvent
{
    enum TType
    {
        EButton1Down,
        EButton1Up,
        EButton2Down,
        EButton2Up,
        EButton3Down,
        EButton3Up,
        EDrag,
        EMove,
        EButtonRepeat,
        ESwitchOn,
    };
    TType iType;           // Type of pointer event
    TUint iModifiers;      // State of pointing device & assoc. buttons
    TPoint iPosition;      // Window co-ordinates of mouse event
    TPoint iParentPosition; // Position relative to parent window
};
```

The following event types are defined:

- `EButton1Down` (pen down)
- `EButton1Up` (pen up)
- other buttons: you don't get these on pen-based devices
- `EButtonRepeat`, which is generated when a button is held down continuously in one place, just like keyboard repeat – most useful for controls such as scroll arrows or emulated on-screen keyboards
- `EDrag`, which is rarely used in Symbian OS, but can be useful
- `ESwitchOn`, as some phones can be switched on by a tap on the screen

- `EMove`, which you never get on pen-based devices. Devices supporting move also have to support a mouse cursor (which the window server provides functionality for), control entry/exit notification, and changing the cursor to indicate what will happen if you press button 1. That's all hard work and adds little value to Symbian OS phones.

The `iModifiers` member contains any pointer-event modifiers, including Shift, Ctrl, and Alt keys (where available), using the same values as for key-event modifiers. For instance, in a text editor, Shift and tap might be used to make or extend a selection.

Distributing pointer events

Unlike key events, the Symbian OS framework code directs pointer events to the appropriate control without any explicit assistance from the application. Normally the window server associates a pointer event with the foremost window whose rectangle encloses the event's position.

Pointer grab is one exception to this rule. It is used to ensure that the control that received a pointer-down event receives all subsequent pointer events until the next pointer-up event, even if the pointer is dragged so that these later events occur outside the original control. Implementing pointer grab needs cooperation between the window server, which has to ensure that the subsequent events are directed to the same window, and the control framework, which has to ensure that they go to the same control within that window.

The other exception is *pointer capture*, which is initiated and terminated by calling a control's `SetPointerCapture()` function. While set, pointer capture prevents any other control from receiving pointer-down events. The most common uses of pointer capture are:

- in dialogs, to throw away pointer events that occur outside the dialog
- in menus, to dismiss the menu in response to a pointer down event outside its window.

Processing pointer events

Within the application, the event is passed to the `HandleWSEventL()` function of the application UI class, which is the same function that handles key events. This function recognizes the event as a pointer event associated with a particular window and calls the `ProcessPointerEventL()` of the control that owns the window. This in turn calls the control's `HandlePointerEventL()` function. If the control has components, the default implementation of `HandlePointerEventL()` scans the visible non-window-owning components to locate one that contains the event's position. If one is found, `HandlePointerEventL()` calls its `ProcessPointerEventL()` function.

To customize the response to pointer events in a simple control, you should override its `HandlePointerEventL()` function. You wouldn't normally override this function in a compound control, but if you do, make sure that you don't prevent pointer events from being passed to a component control.

In the Noughts and Crosses example, the tile's `HandlePointerEventL()` function is implemented as follows:

```
void COandXTile::HandlePointerEventL(const TPointerEvent&aPointerEvent)
{
    if (aPointerEvent.iType == TPointerEvent::EButton1Down)
    {
        TryHitL();
    }
}
```

The `TryHitL()` function uses controller code to attempt to add a nought or a cross to the relevant tile and, if successful, redraws the tile.

The way a control handles pointer events is, in many ways, similar to handling key events:

- it does not need to check if it is supposed to receive a pointer event because it does not receive one otherwise
- it can ignore the event
- it can use the event to generate a command that is processed elsewhere.

However, a pointer event is delivered only to the particular control for which it is intended, which means:

- there is normally no need to pass the event to another control
- the concept of not consuming a pointer event has no meaning, which is reflected in both the function's name (`HandlePointerEventL()`) and its return type (`void`).

In addition to calling `HandlePointerEventL()`, the `ProcessPointerEventL()` function also performs some useful preprocessing of pointer events. These include:

- discarding any pointer-down events on invisible controls
- implementing pointer grab
- ignoring pointer events until the next pointer-up event (which is initiated by calling the control's `IgnoreEventsUntilNextPointerUp()` function, normally after handling a pointer-down event)

- reporting, but then discarding, any pointer-down events on dimmed controls
- reporting any pointer-down event on an unfocused control that is capable of taking focus
- reporting, if necessary, and after calling `HandlePointerEventL()`, the need to transfer focus to this control.

The three possible reporting options are discussed in Section 15.5.

15.5 Observing a Control

It is frequently useful for a control to be able to notify some other class of significant events. The standard mechanism for doing this is to use an observer interface.

The control framework defines the `MCoeControlObserver` interface class with a single member function, `HandleControlEventsL()`, to provide the mechanism for receiving and processing notifications of a range of common control events. Any class that derives from this interface is known as a *control observer*. A control's observer is frequently, but not necessarily, the control's container.

A control's observer can be set using the control's `SetObserver()` function and subsequently referenced via the `Observer()` function. A control wishing to report an event to its observer should call `ReportEventL()`, passing one of the enumerated event types listed below. `ReportEventL()` will report the event only if the control's observer has previously been set by means of a call to `SetObserver()`.

There are three general-purpose events that your control may report.

- `EEventStateChanged`: a control should use this event to report that some significant piece of internal data has changed. The container control of a dialog responds to a report of this event from one of its component controls by calling the dialog's `HandleControl-StateChangeL()`. You should implement this function to make the appropriate changes to any other controls that are affected by the change of state. The event is not used elsewhere.
- `EEventRequestExit`: the intended use of this event is to indicate the successful completion of an operation, for example selecting an item from a choice list. The control framework does not use this event.
- `EEventRequestCancel`: this event should be used to indicate that an operation has been cancelled, for example backing out of a choice list, leaving the original choice unchanged. The control framework does not use this event.

The control framework uses a further three events to handle the three reports made by `ProcessPointerEventL()`.

- `EEventInteractionRefused`: the control framework notifies this event when a pointer-down event occurs on a dimmed control. A dialog responds to this event by calling `HandleInteractionRefused()`, whose default behavior is to display a message informing the user that the control is not available.
- `EEventPrepareFocusTransition`: the control framework notifies this event when a pointer-down event occurs on a control that does not yet have, but could get, focus. An appropriate response, which is used in dialogs, would be to call the currently focused control's `PrepareForFocusLossL()` function. This gives the control an opportunity to ensure that it is in a suitable state to lose focus. If the control is not in a suitable state, and cannot change itself into such a state, its implementation of `PrepareForFocusLossL()` should leave, thereby aborting the attempted change in focus. In some cases, it might also be appropriate to call `PrepareForFocusGainL()` in the control that is about to gain focus. This function should also leave if the control is not prepared to gain focus.
- `EEventRequestFocus`: the control framework also notifies this event when a pointer-down event occurs on a control that does not yet have, but could get, focus. The event is notified after notification of the `EEventPrepareFocusTransition` event, and after calling the control's `HandlePointerEventL()` function. The appropriate response is to transfer focus to the control in which the pointer-down event occurred.

In the Noughts and Crosses application, the tiles have the application's view as their observer. The view implements `HandleControlEventL()` as follows:

```
void COandXAppView::HandleControlEventL(CCoeControl* aControl,
                                         TCoeEvent aEventType)
{
    switch (aEventType)
    {
    {
    case EEventRequestFocus:
        SwitchFocus(IdOfFocusControl(), aControl);
        break;
    default:
        break;
    }
    }
}
```

None of the tiles is ever dimmed, and the tiles are never in a state that would prevent them losing focus. In consequence, of the three pointer-related events that the application's view could receive, it needs to handle

only `EEventRequestFocus`. When `EEventRequestFocus` event is received, the corresponding tile is focused via the `SwitchFocus()` function.

```
void COandXAppView::SwitchFocus(TInt aFromIndex, CCoeControl* aToControl)
{
    ComponentControl(aFromIndex)->SetFocus(EFalse, EDrawNow);
    aToControl->SetFocus(ETrue, EDrawNow);
}
```

15.6 Drawing a Control

The `HelloBlank` example application introduced in Section 15.2 just uses a blank control for its view, but most applications use customized controls.

The standard way of customizing how a control draws its content is to implement a `Draw()` function, so the class definition of the simple blank control is changed as follows:

```
class CBlankAppView : public CCoeControl
{
public:
    CBlankAppView();
    ~CBlankAppView();
    void ConstructL(const TRect& aRect);
private:
    void Draw(const TRect& aRect) const;
};
```

As we want to draw the control explicitly, we remove the call to `SetBlank()` from `ConstructL()`.

```
void CHelloBlankAppView::ConstructL(const TRect& aRect)
{
    CreateWindowL(); // Create a window for this control
    SetRect(aRect);  // Set the control's size
    ActivateL();     // Activate control, making it ready to be drawn
}
```

The `Draw()` function must be capable of drawing to every pixel within the rectangular area occupied by the control. This is a simple implementation for a blank window-owning control:

```
void CHelloBlankAppView::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc(); // Get the standard graphics context
    gc.Clear();                 // Clear the whole window
}
```

Drawing requires a graphics context; the appropriate one for drawing to a window, an instance of `CWindowGc`, is found by calling `CCoeControl::SystemGc()`.

A `Draw()` function is free to ignore the rectangular region passed as a parameter, provided it does not draw outside its own rectangle. An alternative version of `Draw()` that respects the passed parameter would use a different overload of the graphics context's `Clear()` function.

```
void CHelloBlankAppView::Draw(const TRect& aRect) const
{
    CWindowGc& gc = SystemGc(); // Get the standard graphics context
    gc.Clear(aRect);           // Clear only the specified rectangle
}
```

This is effectively the code that the default implementation of `Draw()` uses to draw a blank window.

Whether or not to ignore the passed rectangle depends on what the control needs to draw, and considerations such as whether it is more efficient in terms of time and/or memory usage to do one rather than the other. In the current example, as is commonly the case, drawing is simple and fast, so there is no great advantage in restricting the drawing to the passed rectangle.

The `Draw()` function is discussed in more detail below.

Drawing and the Window Server

This section introduces some of the concepts involved in Symbian OS graphics, to aid in understanding the practical issues of drawing a control. The mechanisms that Symbian OS uses to support drawing are covered in detail in Chapter 17.

It is virtually axiomatic that all window-owning controls must create the window they own, and this is normally done from the control's `ConstructL()` function. Most controls use a standard window, of type `RWindow`, created with the `CreateWindowL()` function, as in the earlier example. In addition to creating the client-side `RWindow` resource, the function also creates a corresponding resource in the window server. This records little more than the window's position and size, together with the region of the window that is visible and any region that needs to be redrawn (the *invalid* region). Once the window is created, the control can access it via the `Window()` function.

Drawing a control may be initiated either by the control itself, for example if the application modifies the data displayed by the control, or by the window server, for example when a previously hidden part of a window is exposed. Drawing initiated by the window server is known as a *redraw* operation, to distinguish it from application-initiated drawing, but the mechanisms are similar.

To draw to its window, a control uses a window-specific graphics context, an instance of `CWindowGc`, derived from the generic `CGraphicsContext`. The control's drawing code makes calls to `CWindowGc` functions to set the required values for attributes such as pen color and line width or font style and size, and to draw lines, shapes or text. The graphics context issues a corresponding sequence of commands to a buffer, whose content is subsequently transferred to the window server. Buffering the commands greatly reduces the number of inter-process communications between the application and the window server, thereby improving performance.

Before such a drawing sequence, the control must perform a little housekeeping to ensure that both the application and the window server are in the correct state for drawing to start. This includes informing the window server, by means of a call to its `Invalidate()` function, which region of the window has content that is no longer valid, and activating the standard graphics context (owned by the UI control framework). This activation prepares the graphics context for drawing to the control's window and ensures that all its settings have their default values. At this point, the application's drawing code can be called, with the graphics context in a known state, independent of any previous drawing operation.

On completion of the application's drawing code, further housekeeping is required, to inform the window server that the drawing sequence is complete and to free the graphics context so that it is available to be used with another window.

Redrawing

When the window server knows that a region of a window is invalid, for example when an overlying window is removed, it sends a redraw event to the application owning the window. The application framework calls the window server to determine which window needs to be redrawn and the bounding rectangle of the invalid region, and then calls the appropriate window-owning control's `HandleRedrawEvent()` function, which you should never need to override.

`HandleRedrawEvent()` makes a call to the control's `Draw()` function and, if necessary, the `Draw()` functions of any component controls that intersect with the rectangular region to be redrawn. In addition, it performs all the housekeeping mentioned in the previous section.

Application-initiated drawing

As mentioned earlier, an application does not make direct calls to `Draw()`. If the application needs to initiate drawing of a control, it should call either `DrawNow()` or `DrawDeferred()`.

The application may call `DrawNow()` on a control after it has been created and is ready for drawing, or if a change in application data or the

control's internal state means that entire control's appearance is no longer up to date. If the control is not ready to be drawn, that is, if it is either invisible or not yet activated, `DrawNow()` does nothing. Otherwise, for a normal window-owning control, `DrawNow()` surrounds a call to the control's `Draw()` function with a call to the window's `Invalidate()` function and the other window-server-related housekeeping that was mentioned earlier. For a compound control, it also generates calls to the `DrawNow()` function of each of the component controls.

For a complex control, partial redrawing of the control may sometimes be more appropriate than drawing the entire control. One example is where a user adds a character in a word processor. Instead of drawing the whole control, redrawing only the area of the screen occupied by the new character reduces processing and can improve performance. In such a case you should not use `DrawNow()` but will need to write your own customized drawing code. See Chapter 17 for more information on this topic.

As an alternative to calling `DrawNow()`, you may choose to call a control's `DrawDeferred()` function, which simply causes the control's area to be marked as invalid, eventually causing the window server to initiate a redraw. The control framework handles redraw events at a lower priority than user-input events, which means that any pending user-input events will be processed before the redraw event. `DrawDeferred()` therefore allows a control to do drawing at a lower priority than drawing performed by `DrawNow()`.

An advantage of using `DrawDeferred()` is that if you make multiple calls to `DrawDeferred()` on the same area of a control, the window server will not generate a redraw event to do drawing that has already been superseded. If you make multiple calls to `DrawNow()`, however, all of them get processed, even if they have already been superseded by the time they are processed.

In the Noughts and Crosses application, a tile's `TryHitL()` function calls `DrawDeferred()` following a successful attempt to set the tile to contain a nought or a cross.

```
Void COandXTile::TryHitL()
{
    if (iAppView->TryHitSquareL(this))
    {
        DrawDeferred();
    }
}
```

If `DrawNow()` were used here, the control would be drawn twice in the case where a pointer event switched focus to this control as well as setting it to contain a nought or a cross.

Both `DrawNow()` and `DrawDeferred()` are implemented by `CCoeControl` and may not be overridden.

The Draw() Function

The `Draw()` function needs to be implemented by all non-blank controls, but is intended to be called only from the control's own member functions, hence `Draw()` is declared as a private member function of `CCoeControl`.

The function is passed a reference to a `TRect`, indicating the region of the control that needs to be redrawn, and the function must guarantee to draw to every pixel within the rectangle. It may, subject to considerations of efficiency, ignore the rectangle and draw the whole of the control. However, drawing outside the specified rectangle may slow down drawing. Depending on how drawing is implemented and the complexity of what is being drawn, this could potentially result in visible flicker in the display.

The `Draw()` function should not draw outside the control's area. For window-owning controls, drawing is clipped to the window boundary, so drawing outside the control will only reduce efficiency. Drawing is not clipped to the boundary of a non-window-owning control, so in this case it is essential that the control does not draw outside its boundary.

You may safely assume that:

- before any `Draw()` function is called, the graphics context has been activated and its properties set to their defaults
- the graphics context is deactivated for you after the return from `Draw()`.

Your implementation of `Draw()` should gain access to a `CWindowGc` graphics context by means of a call to `SystemGc()`, and all drawing should be done using this graphics context.

In the Noughts and Crosses example, responsibility for drawing is divided between the view and the component tiles. The view draws only the outer regions and the lines between the tiles.

```
void COandXAppView::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc();
    TRect rect = Rect();

    // Draw outside the border
    gc.SetPenStyle(CGraphicsContext::ENullPen);
    gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
    gc.SetBrushColor(KRgbWhite);
    DrawUtils::DrawBetweenRects(gc, rect, iBorderRect);

    // Draw a border around the board
    gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
    gc.SetBrushColor(KRgbGray);
    DrawUtils::DrawBetweenRects(gc, iBorderRect, iBoardRect);

    // Draw the first vertical line
    gc.SetBrushColor(KRgbBlack);
```

```

TRect line;
line.iTl.iX = iBoardRect.iTl.iX + iTileSide;
line.iTl.iY = iBoardRect.iTl.iY;
line.iBr.iX = line.iTl.iX + KLineWidth;
line.iBr.iY = iBoardRect.iBr.iY;
gc.DrawRect(line);
TInt i;
// Draw the remaining (KTilesPerRow-2) vertical lines
for (i = 0; i < KTilesPerRow - 2; i++)
{
    line.iTl.iX += iTileSide + KLineWidth;
    line.iBr.iX += iTileSide + KLineWidth;
    gc.DrawRect(line);
}
// Draw the first horizontal line
line.iTl.iX = iBoardRect.iTl.iX;
line.iTl.iY = iBoardRect.iTl.iY + iTileSide;
line.iBr.iX = iBoardRect.iBr.iX;
line.iBr.iY = line.iTl.iY + KLineWidth;
gc.DrawRect(line);
// Draw the remaining (KTilesPerCol-2) horizontal lines
for (i = 0; i < KTilesPerCol - 2; i++)
{
    line.iTl.iY += iTileSide + KLineWidth;
    line.iBr.iY += iTileSide + KLineWidth;
    gc.DrawRect(line);
}
}

```

The lines are drawn as thin rectangles rather than using `gc.DrawLine()`. This ensures that the lines cover the precise region we want, without the need for messy calculations involving the line width, and we avoid potential problems related to the rounded ends of lines.

Each tile is responsible for drawing itself.

```

void COandXTile::Draw(const TRect& /*aRect*/) const
{
    TInt tileType;
    tileType = iAppView->SquareStatus(this);

    CWindowGc& gc = SystemGc();
    TRect rect = Rect();

    if (IsFocused())
    {
        gc.SetBrushColor(KRgbYellow);
    }
    gc.Clear(rect);
    if (tileType != ETileBlank)
    {
        DrawSymbol(gc, rect, tileType == ETileCross);
    }
}

```

First we check if the tile is empty or not with a call to `SquareStatus()`.

Then the tile background is drawn. Note that the background color is set to yellow only for the tile that currently has focus. White is the default brush color, and we can safely assume that the graphics context defaults have been set before `Draw()` was called.

Highlighting the control with focus isn't always necessary. The Noughts and Crosses application running on, say, the Sony Ericsson P990 (using UIQ) doesn't receive key events. You don't need a tile to be highlighted in order to know whether you can tap on it to enter a nought or a cross.

Having drawn the background, we use the status of the tile again to determine whether we need to draw a red circle or a green cross. The relevant symbol is drawn by means of a call to `DrawSymbol()`, which is implemented in the `COandXSymbolControl` superclass (see Chapter 12).

```
void COandXSymbolControl::DrawSymbol(CWindowGc& aGc, TRect& aRect,
                                     TBool aDrawCross) const
{
    TSize size;
    size.SetSize(aRect.iBr.iX - aRect.iTl.iX, aRect.iBr.iY - aRect.iTl.iY);
    aRect.Shrink(size.iWidth/6, size.iHeight/6); // Shrink by about 15%
    aGc.SetPenStyle(CGraphicsContext::ESolidPen);

    size.iWidth /= 9; // Pen size set to just over 10% of shape's size
    size.iHeight /= 9;
    aGc.SetPenSize(size);
    if (aDrawCross)
    {
        aGc.SetPenColor(KRgbGreen);
        // Cosmetic reduction of cross size by half the line width
        aRect.Shrink(size.iWidth/2, size.iHeight/2);
        aGc.DrawLine(aRect.iTl, aRect.iBr);
        TInt temp;
        temp = aRect.iTl.iX;
        aRect.iTl.iX = aRect.iBr.iX;
        aRect.iBr.iX = temp;
        aGc.DrawLine(aRect.iTl, aRect.iBr);
    }
    else // draw a circle
    {
        aGc.SetPenColor(KRgbRed);
        aGc.SetBrushStyle(CGraphicsContext::ESolidBrush);
        aGc.DrawEllipse(aRect);
    }
};
```

This isn't ideal drawing code, as we're drawing over an area that we have previously blanked. That means there is some potential for the display to flicker; ideally, each pixel should only be drawn once.

However, the redrawing is confined to a small region and is done immediately, so in practice the display behavior is acceptable.

Before drawing the cross, we shrink the bounding rectangle by half the line width: without this, a cross would look slightly larger than a nought. It's always worth paying attention to such details in an application, to create the best possible impression on the user.

In conclusion, these two functions do indeed satisfy the basic requirements for the effective drawing of a control:

- the combined drawing code of the view and the tiles covers every pixel of the required area
- neither function draws outside the relevant control
- apart from the redrawing of the nought and cross symbols, each pixel is drawn only once.

15.7 Backed-up Windows

In the window server, a standard window is represented by information about its position, size, visible region, and invalid region – and that's about all. Traditionally, no memory is set aside for the drawn content of the window, which is why the window server has to ask the application to redraw when a region is invalid. (An exception to this is redraw storing, described in Chapter 17.)

However, in some cases it is impractical for the application to redraw the window, for instance if it is:

- a program that is not structured for event handling and so cannot redraw
- a program that is not structured in an MVC manner (see Chapter 17), has no model, and so cannot redraw, even if it can handle events
- a program that takes so long to redraw that it's desirable to avoid redraws if at all possible.

A program in an old-style interpreted language, such as OPL, is likely to suffer from all of these problems.

In these cases, you can ask the window server to create a *backed-up window* of class `RBackedUpWindow`. The window server creates a backup bitmap for each such window, and uses application-initiated drawing to update the bitmap as well as to draw to the screen. The advantage of this type of window is that the window server can handle redraws by drawing from the bitmap, without having to send a redraw event to the application.

A control that uses a backed-up window should create it by using `CreateBackedUpWindowL()` rather than `CreateWindowL()`, and access it via `BackedUpWindow()` rather than `Window()`. Both window classes are derived from `RDrawableWindow`, so if you are not sure which type of window the control owns, you should access the window via `DrawableWindow()`.

An obvious disadvantage is that a backed-up window uses much more memory than a standard window, and memory is a scarce resource on mobile phones. If there is insufficient memory for a control to create a backed-up window, this will usually have severe consequences for the whole application. It follows that you should only make the decision to use a backed-up window if there is no realistic alternative.

Code designed for drawing to backed-up windows usually doesn't work with standard windows, because standard windows require redraws, which code written for a backed-up window won't be able to handle.

On the other hand, code that is good for writing to a standard window is usually good for writing to a backed-up window: although the backed-up window won't call for redraws, there's no difference to the application-initiated draw code. The only technique that won't work for backed-up windows is to invalidate a window region in the hope of receiving a later redraw.

Standard controls are lodger controls that are designed to work in standard windows. Such controls will, in general, also work properly in backed-up windows. All Uikon stock controls, for example, are designed to work in windows of both types.

On a standard window, `CCoeControl`'s `DrawDeferred()` function works by invalidating the window region corresponding to the control, causing a later redraw event. This won't work on a backed-up window, so in that case `DrawDeferred()` simply calls `DrawNow()`:

```
void CCoeControl::DrawDeferred() const
{
    ...
    if(IsBackedUp())
        DrawNow();
    else
        Window().Invalidate(Rect());
    ...
}
```

`DrawNow()` itself is, of course, coded to behave correctly for a control associated with a backed-up window.

15.8 Backed-up-Behind Windows

Backed-up-behind windows provide an optimization that is particularly useful for controls that are displayed for only a short time, such as menu

panes, pop-ups and dialogs. As with backed-up windows, the window server creates a bitmap, but the image that it contains is of the area behind the window, not the content of the window itself. When the backed-up window disappears, the window server can use the bitmap to restore the underlying image, without needing to generate redraw events.

A backed-up-behind window is not a separate window class. Either an `RWindow` or an `RBackedUpWindow` may be converted to a backed-up-behind window by calling the window's `EnableBackup()` function. This function is declared in the class definition of `RDrawableWindow`, in `w32std.h`, as follows:

```
IMPORT_C void EnableBackup(TUint aBackupType=EWindowBackupAreaBehind);
```

There are two values associated with the `TUint` parameter:

- `EWindowBackupAreaBehind` creates a backed-up-behind bitmap to store the image for just the area behind the window
- `EWindowBackupFullScreen` creates a backed-up-behind bitmap for the whole underlying screen image.

The first of these is frequently used for an overlying window that can be moved around the screen. As the window moves, the exposed underlying area is restored from the bitmap, and the bitmap content is updated to the image behind the window's new position.

The second value is typically used in cases where the underlying screen image is faded – that is, drawn in a less bright range of colors – when the overlying window appears. In this case, the bitmap stores the unfaded image.

A window-owning control may set its window to be one of these two types, for example:

```
DrawableWindow() ->EnableBackup(EWindowBackupAreaBehind);
```

Alternatively, it may set its window to own both types of bitmap simultaneously, as follows:

```
DrawableWindow() ->EnableBackup(EWindowBackupAreaBehind |  
                                EWindowBackupFullScreen);
```

This option is useful if you need to display a moveable window on a faded background. In this case, the full screen bitmap stores the unfaded image, but the bitmap of the area behind the window stores the faded image.

The bitmap is created at the time the window becomes visible, not when the call to `EnableBackup()` is made. When backing up the

whole screen, you need to call `EnableBackup()` before the window is activated, and before you fade the background by calling `RWindowBase::FadeBehind()`.

Backed-up-behind windows are supported for purposes of optimization, and any call to `EnableBackup()` is not guaranteed to result in the creation of the requested bitmap. It may, for example, fail if there is insufficient memory available to create the necessary backup bitmap. Unlike in the case of an `RBackedUpWindow`, such a failure is not catastrophic for the application; the only consequence is that the application will be sent redraw events that would not otherwise have been received.

In addition, the usage of such windows is subject to a number of limitations, designed to minimize the amount of memory consumed.

- A window isn't given the backup when it requests it, but only when it becomes visible – and it will only keep the backup for the period that it is visible. A window that is repeatedly made visible and invisible may receive and lose the backup many times.
- An application may own only one window of each type at any one time. The application may therefore have a maximum of two backed-up-behind windows, one of each type, or it may have one window set to own both types of bitmap, as described earlier.
- If a window-owning control requests a backup of type `EWindowBackupAreaBehind`, any window that has already claimed a backup of this type will lose it, even if that window is in another application.
- If a window-owning control requests a backup of type `EWindowBackupFullScreen`, this request will fail if another window has already claimed a backup of this type, even if that window is in another application.

15.9 Dimmed and Invisible Controls

The control framework provides support for dimming controls or making them invisible, as a means of indicating the controls with which the user is or is not allowed to interact.

A control is set to be visible or invisible by calling `MakeVisible(ETrue)` or `MakeVisible(EFalse)`, and its visibility is tested by calling `IsVisible()`. By default, a control's components do not inherit the containing control's visibility, but this behavior can be changed by calling the container's `SetComponentsToInheritVisibility()` function, which also takes an `ETrue` or `EFalse` parameter. The effect of this call only extends to the immediate component controls, so this function must also be called on any components which are themselves compound controls.

A control's `Draw()` function is only called while the control is visible. A compound control must take the responsibility for drawing any area that is occupied by any invisible component control.

A dimmed control is conventionally displayed in a subdued palette of colors. Dimming can be added by calling `SetDimmed(ETrue)`, removed by calling `SetDimmed(EFalse)`, and checked by calling `IsDimmed()`. A control should be dimmed if you want to indicate that it is temporarily unavailable, but still want the user to be able to see its content.

There is no explicit support for displaying a dimmed control. The control's `Draw()` function is responsible for testing, by means of `IsDimmed()`, the state of the control and then drawing the content in the appropriate colors. The standard way of dimming a control varies from UI to UI, so you need to follow the examples and style guidelines provided in the appropriate SDK. However, the `CWindowGC` class provides the following two functions that may be useful when you need to draw in a dimmed state:

```
void SetFaded(TBool aFaded);  
void SetFadingParameters(TUint8 aBlackMap, TUint8 aWhiteMap);
```

The first function instructs the window server to map all the colors it uses for drawing to a less colorful range, so other windows can stand out. For example, a window is less colorful when a dialog is displayed in front of it. You can set the range into which colors get mapped by means of a call to `SetFadingParameters()`. If you don't make a call to `SetFadingParameters()`, then a UI-specific default range is used.

A compound control should never set focus to a dimmed or invisible control.

Summary

Controls are the principal means of interaction between an application and the user. This chapter started by explaining what a control is and what its place is in UI development, and presented different aspects of control development.

We showed how to create controls by extending `CCoeControl`, and the difference between simple controls and compound controls. We showed how to manage the control layout and how controls should respond to key and pointer events.

In the section on drawing a control we demonstrated how to customize a control by implementing the `Draw()` method and showed how to use the window server to handle complex drawing situations.

The topics, along with the examples provided, should help you to understand basic UI development and design user interfaces for particular applications.

16

Dialogs

This chapter explains the basic principles of programming and using dialogs. The examples provided in this chapter have deliberately been kept general. Where applicable, the differences between UIQ and S60 are explained.

16.1 What Is a Dialog?

A dialog is simply a specialized window-owning compound control. The dialog framework manages the behavior and many other aspects of dialogs, including layout, drawing and user interaction with component controls. A dialog can be *waiting* or *non-waiting*, and it can be *modeless* or *modal*. A dialog is a pop-up window with a title, one or more buttons, and one or more lines which contain controls that display information or let the user input information.

By default, dialogs are modal. Once a modal dialog is running, the user can interact only with the dialog until the dialog is dismissed. The user is therefore forced to respond, and cannot interact with the underlying application until they have done so. A modeless dialog allows the user to interact with other parts of the application's user interface while it is active.

The component controls that make up a specific dialog are generally defined in the application's resource file. The dialog framework code uses the definitions to construct the appropriate controls and incorporate them into the dialog. The layout and positioning of the elements of the dialog are handled automatically by the dialog framework, but fully dynamic construction of a dialog is possible and we can influence the process.

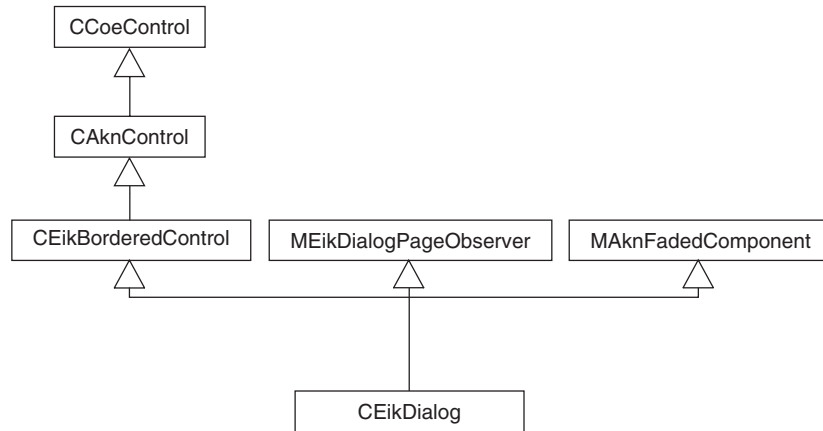


Figure 16.1 Generic derivation of CEikDialog

On the S60 platform all dialog classes are derived directly or indirectly from CEikDialog, which is derived from CCoeControl, as shown in Figure 16.1. In UIQ 3, CEikDialog is deprecated in favor of CQikSimpleDialog and CQikViewDialog, both directly derived from CCoeControl, as shown in Figures 16.2 and 16.3.

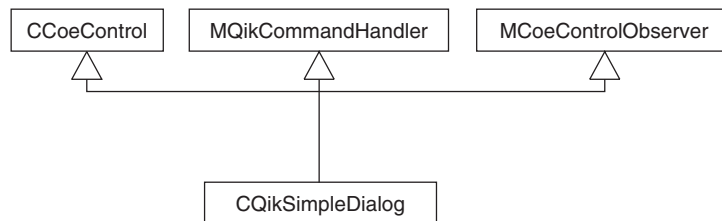


Figure 16.2 Generic derivation of CQikSimpleDialog

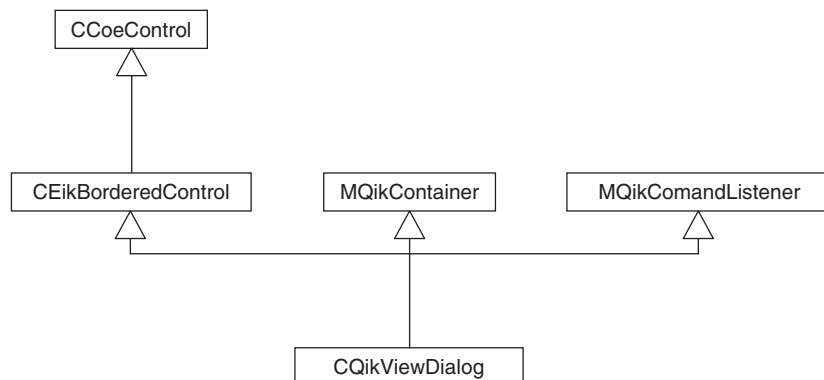


Figure 16.3 Generic derivation of CQikViewDialog

As shown in the above figures, dialogs also inherit from a number of interface classes. The purpose of these is discussed below.

16.2 Simple Dialogs

Simple Single-Page Dialog

Creating a simple dialog requires a few straightforward steps. The first is to create a `DIALOG` resource in our resource file to define the dialog title and the set of dialog lines.

The second is to create a class derived from `CQikSimpleDialog` if we are developing for UIQ, or from `CEikDialog` if we are developing for S60. The minimum code we have to provide in this class is the code to initialize the component controls. Another piece of code that we may want to implement is the code to save the control values when the dialog is dismissed, if our dialog requests input from the user.

The last step is to launch the dialog. This is done by invoking the dialog's `ExecuteLD()` method (implemented in `CEikDialog` and `CQikSimpleDialog`).

A simple single-page UIQ dialog is presented as an example in Figure 16.4.



Figure 16.4 UIQ simple dialog

Dialog controls are displayed in lines, in a vertical list. The way controls are arranged in a dialog is a key design decision, making many things easier for both the user and the developer.

- Navigation from field to field is easy: use the pointer, or the up and down navigation keys.
- Dialog layout is easy: the only difficulty is establishing the width of the controls.

- The width of the dialog limits the width of a control. If a dialog line contains text longer than the control's width, the text will be truncated by replacing the last three displayable characters with an ellipsis (...).
- Because layout is handled by the dialog framework, you do not have to specify the pixel coordinates of each field in the dialog definition resource, so there is no need for a GUI-based resource builder.

The example dialog has only two controls. Dialogs can have more, but you need to consider what will happen if they overflow the screen vertically. In UIQ, for example, if the controls overflow the screen, the dialog becomes scrollable, with scroll bars automatically added on the dialog's right hand side by the framework. However, as a design principle, it is not good style to use scrollable dialogs. If you need more lines, you can use multi-page dialogs (see below).

Do not create monster dialogs that offer a bewildering set of choices to the user. Keep in mind the dialog's usability when you have to create huge dialogs.

In UIQ, focus is indicated only in text and numeric fields. For text fields the flashing cursor indicates the location of focus, and for numeric fields a highlighted background is used. In S60, focus is indicated in all controls that can receive focus.

Standard Dialogs

Both UIQ and S60 provide a number of convenient standard dialogs. When using standard dialogs, we do not have to create a derived class; we only have to define the `DIALOG` resource in our resource file and launch the dialog.



Figure 16.5 UIQ alert dialog

Alerts

An *alert* dialog displays the title 'Information', one or two lines of text, and a button labeled 'Continue' in UIQ (see Figure 16.5) or 'Ok' in S60 (see Figure 16.6).



Figure 16.6 S60 alert dialog

The UI environment constructs a ready-made alert dialog invoked with `iEikonEnv>AlertWin()`, specifying either one or two string parameters. An alert dialog is an example of a pre-constructed, or sleeping dialog, which does not need to allocate any resources at the time it is displayed. As a consequence, you can never run out of memory when using `AlertWin()`.

Queries

A *query* dialog, as illustrated in Figure 16.7, enables a minimal form of interaction.



Figure 16.7 UIQ query dialog

A query dialog displays a title, a line of text and two buttons. You can use it to ask a simple yes/no question, as follows:

```
if (iEikonEnv->QueryWinL(R_EIK_TBUF_CONTACT_DELETE_CONFIRM))
    User::LeaveIfError(cntModel->DeleteContactL(iContactId));
else
    ; //do nothing
```

The call to `iEikonEnv->QueryWinL()` specifies a string from a resource file to be used as a question. The query dialog has a Yes button and a No button; `iEikonEnv->QueryWinL()` returns `ETrue` if Yes is pressed, otherwise `EFalse`.

Unlike alert dialogs, query dialogs are not sleeping dialogs, so the process of constructing and executing a query can leave.

Other standard dialogs

UIQ and S60 provide several other standard dialogs. UIQ, for example, includes dialogs:

- to enter and change a password
- to set the current date and time
- to set the options for formatting dates and times.

Many of the more sophisticated controls include dialogs of their own. EDWINs (text editors), for example, include dialogs for Find, Replace and Options for replace.

16.3 Complex Dialogs

Unlike the previous examples, most dialogs contain more than one line and some very complex dialogs can contain multiple pages. However, this does not mean that such dialogs are significantly more complicated to write. If the various lines are truly independent of each other, the only added complexity is the need to initialize the extra components appropriately, and to process the results on completion of the dialog.

UIQ and S60 use different classes to handle complex dialogs, `CQikSimpleDialog` and `CQikViewDialog` in UIQ and `CEikDialog` in S60.

Complex dialogs introduce issues that did not arise in the earlier example, such as:

- focus can change from one control to another, and controls may not be in a fit state to lose or gain focus
- controls may be dependent on one another, so that modifying the state of one control requires the state of one or more other controls to change.

Change of Focus

In UIQ, application code can obtain a pointer to the focused control using `CQikViewDialog::FocusedControl()`. In S60, it can find the ID of the currently focused control using `CEikDialog::IdOfFocusControl()`, and then obtain a pointer to the control itself by calling `Control()`, which takes a control ID as its parameter.

The dialog framework extends the `MCoeControlObserver` interface. Because of this, whenever focus is about to be lost by a control, the dialog framework will handle an `EEventPrepareFocusTransition` event. The framework's default action is to identify the control in the currently focused line and call its `PrepareForFocusLossL()` function. The control must either ensure that it is in a fit state to relinquish focus or leave. If the function leaves, focus will not be removed from the control.

The focused control can be changed in a number of ways. A general technique, usable in both UIs, is to call the `SetFocus()` method on a control in the dialog. In UIQ, a change of focus can be initiated by calling `RequestFocusL()`, passing the control which is to receive the focus. In S60, a change of focus can be initiated by calling `TryChangeFocusToL()`, passing the ID of the control which is to receive the focus.

In S60, any attempt to transfer focus to a dimmed dialog item results in a call to the dialog's `HandleInteractionRefused()` function, whose default action is to display an information message. You may override the behavior if you need to take a more specific action. The parameter passed to the function is normally the ID of the relevant control, but may be zero if the user is attempting to transfer to a disabled page in a multi-page dialog.

Change of State

Controls may notify their observers of significant state changes by reporting an `EEventStateChanged` event. In UIQ, the `HandleControlEventsL()` method handles the control events. In S60, the framework calls the dialog's `HandleControlStateChangeL()` function, passing the ID of the relevant control. The default implementation of this function is empty, but may be overridden to perform any required action.

16.4 Single-Page Dialogs

Most simple dialogs are single-page dialogs. In order to create a dialog, we have to follow some simple steps. First, we create the `DIALOG` resource in the resource file. For UIQ, the dialog resource is named `QIK_DIALOG`. Then, we extend our dialog class, from `CEikDialog` if we are developing for S60 and from `CQikSimpleDialog` if we are developing for UIQ. The last step is to launch the dialog.

As the Noughts and Crosses application does not have simple page dialogs, we use the username and password dialog from the `QDialogs` example in the UIQ version 3 SDK, as illustrated in Figure 16.8.



Figure 16.8 UIQ single-page dialog

The first step is to define the dialog resource:

```
RESOURCE QIK_DIALOG r_dialogs_user_password_dialog
{
    title = STRING_r_dialogs_user_password_dialog_title;
    configurations =
    {
        QIK_DIALOG_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            container = r_dialogs_user_password_dialog_container;
            command_list = r_dialogs_user_password_dialog_commands;
        }
    };
    controls = r_dialogs_user_password_dialog_controls;
}
```

We define the title, a configuration and a controls collection to be part of our dialog. One important aspect to notice at this point is the title definition; it is defined as an `rls_string`. This technique is very useful when you want to develop applications that support internationalization.

The controls collection is a `QIK_CONTROL_COLLECTION` resource defined as follows:

```
RESOURCE QIK_CONTROL_COLLECTION r_dialogs_user_password_dialog_controls
{
    items =
    {
        QIK_CONTROL
        {
            unique_handle = EDialogsUserPasswordDialogEdwin;
            type = EEikCtEdwin;
            control = r_dialogs_user_password_dialog_edwin;
        },
        QIK_CONTROL
        {
            unique_handle = EDialogsUserPasswordDialogSecretEditor;
            type = EEikCtSecretEd;
            control = r_dialogs_user_password_dialog_secret_editor;
        }
    };
}
```

As you can see, we have a list of control items. For each control that is part of our dialog, we specify the control type and the control definition. The first control is an Edwin resource and the second control is a SecretEd resource

We then extend our class from `CQikSimpleDialog`, to handle our dialog:

```
class CDialogsUserPasswordDialog : public CQikSimpleDialog
{
public:
    static TInt RunDlgLD(TDes& aUsername, TDes& aPassword);
    ~CDialogsUserPasswordDialog();
private:
    CDialogsUserPasswordDialog(TDes& aUsername, TDes& aPassword);
    void ConstructL();
};
```

The minimum amount of code we have to write is for dialog initialization. As our dialog doesn't have to initialize anything, we do not have to write any code for this, so our constructor is empty. Additional code can be added, for example, code to handle dialog commands or code to handle control events.

Our dialog is ready to be used. We create a `CDialogUserPasswordDialog` object and call its `ExecuteLD()` method:

```
userPasswordDialog->ExecuteLD(R_DIALOGS_USER_PASSWORD_DIALOG);
```

16.5 Multi-Page Dialogs

If you need to use a dialog that has more lines than can comfortably fit the screen of a Symbian OS phone, you can split the dialog into a number of pages and display it as a multi-page dialog. It obviously makes sense to ensure, if at all possible, that the items on each page are more closely related to each other than they are to the items on other pages. Each page of a multi-page dialog has a label tab, used to navigate from page to page.

UIQ Variant

Figure 16.9 shows an example from a synchronization application running on a UIQ phone.



Figure 16.9 A multi-page dialog on UIQ

To create a dialog from a resource file, we use the `ViewConstructFromResourceL()` method of `CQikViewDialog`. The method takes two parameters: the ID of a `QIK_VIEW_CONFIGURATION` resource and the ID for a `QIK_CONTROL_COLLECTION` resource.

The configuration resource looks like this:

```
RESOURCE QIK_VIEW_CONFIGURATIONS
    r_dialogs_multi_page_dialog_ui_configurations
{
    configurations =
    {
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchPortrait;
            command_list = r_dialogs_multi_page_dialog_commands;
            view = r_dialogs_multi_page_dialog_layout;
        },
        QIK_VIEW_CONFIGURATION
        {
            ui_config_mode = KQikPenStyleTouchLandscape;
            command_list = r_dialogs_multi_page_dialog_commands;
            view = r_dialogs_multi_page_dialog_layout;
        }
    }
}
```

```

    },
    ...
    QIK_VIEW_CONFIGURATION
    {
        ...
    },
};
}

```

The most important element in the `QIK_VIEW_CONFIGURATION` is the view element, which defines a `QIK_VIEW`. In this case, it is another resource that links to the dialog pages:

```

RESOURCE QIK_VIEW r_dialogs_multi_page_dialog_layout
{
    pages = r_dialogs_multi_page_dialog_layout_pages;
}

```

The dialog pages look like this:

```

RESOURCE QIK_VIEW_PAGES r_dialogs_multi_page_dialog_layout_pages
{
    pages =
    {
        QIK_VIEW_PAGE
        {
            page_id = EDialogsMultiPageDialogPage1;
            tab_caption = STRING_r_dialogs_multi_page_dialog_page1;
            container_unique_handle =
                EDialogsMultiPageDialogPage1ScrollableContainerCtrl;
            page_content = r_dialogs_multi_page_dialog_page_control;
        },
        QIK_VIEW_PAGE
        {
            page_id = EDialogsMultiPageDialogPage2;
            tab_caption = STRING_r_dialogs_multi_page_dialog_page2;
            container_unique_handle =
                EDialogsMultiPageDialogPage2ScrollableContainerCtrl;
            page_content = r_dialogs_multi_page_dialog_page_control;
        }
    };
}

```

Each page has a content specified by the `page_content` element. For each page, the content is an array of controls, so the `page_content` element is linked to a scrollable container:

```

RESOURCE QIK_SCROLLABLE_CONTAINER_SETTINGS
    r_dialogs_multi_page_dialog_page_control
{
    controls =
    {

```

```

QIK_CONTAINER_ITEM_CI_LI
{
    type = ...;
    control = ...;
},
QIK_CONTAINER_ITEM_CI_LI
{
    type = ...;
    control = ...;
},
...
};
}

```

The collection resource used to create the dialog is defined by the following code:

```

RESOURCE QIK_CONTROL_COLLECTION r_dialogs_multi_page_dialog_controls
{
    items =
    {
        QIK_CONTROL
        {
            unique_handle =
                EDialogsMultiPageDialogPage1ScrollableContainerCtrl;
            type = EQikCtScrollableContainer;
            control = r_dialogs_multi_page_dialog_scrollable_container;
        },
        QIK_CONTROL
        {
            unique_handle =
                EDialogsMultiPageDialogPage2ScrollableContainerCtrl;
            type = EQikCtScrollableContainer;
            control = r_dialogs_multi_page_dialog_scrollable_container;
        },
        QIK_CONTROL
        {
            unique_handle = EDialogsMultiPageDialogLabelCtrl;
            type = EEikCtLabel;
            control = r_dialogs_multi_page_dialog_label;
        }
    };
}

```

S60 Variant

Figure 16.10 shows a multi-page dialog running on an S60 phone.

From a programming point of view, most of the differences between single- and multi-page dialogs lie in the dialog's resource file definitions. A multi-page dialog can be created by using the following resource:

```

RESOURCE DIALOG r_aknexform_double_and_single_line_form
{

```

```

flags = EEikDialogFlagNoDrag | EEikDialogFlagFillAppClientRect |
      EEikDialogFlagNoTitleBar | EEikDialogFlagNoBorder |
      EEikDialogFlagCbaButtons;
buttons = R_AVKON_SOFTKEYS_OPTIONS_BACK;
pages = r_aknexform_double_and_single_line_form_pages;
}

```



Figure 16.10 A multi-page dialog on S60

This is a fairly standard, but modeless (no `EEikDialogFlagWait` flag) S60 `DIALOG` resource. It has an associated menu and so uses `Options` and `Back` buttons, rather than `OK` and `Cancel`, but that does not affect the description of the dialog itself.

The most important change is that the resource includes a `pages` element rather than the `items` array that is used for single-page dialogs. The `pages` item specifies the identifier of a second resource:

```

RESOURCE ARRAY r_aknexform_double_and_single_line_form_pages
{
  items =
  {
    PAGE
    {
      id = EAknExPagePhone;
      text = "Phone";
      form = r_aknexform_double_line_text_number_field_form;
    },
    PAGE
    {
      id = EAknExPageEmail;
      text = "Email";
      form = r_aknexform_text_number_field_form;
    },
    PAGE
    {
      id = EAknExFPPageConnection;

```

```

        text = "Connection";
        form = r_aknexform_double_line_icon_form;
    },
    PAGE
    {
        id = EAknExPageSecurity;
        text = "Security";
        form = r_aknexform_double_line_text_number_with_icon_form;
    }
};
}

```

This is an ARRAY resource, whose elements are PAGE resource structs. The content of this struct varies with the UI, but the first two elements, a page ID and the text to be displayed in the tab for that page, are common to all UIs. Like the IDs for individual dialog lines, page IDs should not be zero and should be unique within that dialog. The IDs used in this example are defined, in the application's HRH file, as follows:

```

enum TAknExFormPageControlIds
{
    EAknExPagePhone = 1,
    EAknExPageEmail,
    EAknExPageConnection,
    EAknExPageSecurity,
    ...
};

```

All PAGE structs contain at least one item that specifies a further resource and you may specify either a `lines` or a `form` item. As a quick reference, the STRUCT PAGE is as follows:

```

STRUCT PAGE // S60
{
    WORD id = 0;
    LTEXT text;
    LTEXT bmpfile = "";
    WORD bmpid = 0xffff;
    WORD bmpmask;
    LLINK lines = 0;
    LLINK form = 0;
    WORD flags = 0;
}

```

Use the `lines` element if you want to create a standard multi-line dialog. The resource it specifies is simply an array of `DLG_LINE` items, exactly like the array used to specify the lines of a single-page dialog:

```

RESOURCE ARRAY r_dialog_page_lines_array
{

```

```

items =
{
    DLG_LINE
    {
        ...
    },
    ...
    DLG_LINE
    {
        ...
    }
};
}

```

We also have the option to use the `form` element to specify that the page is to be handled as a form. In this case, the referenced resource must be a `FORM` struct which specifies, in addition to an array of `DLG_LINE` structs, a form-specific `flags` item. The resource used for the first page of the multi-page dialog in the form example application (slightly truncated to improve clarity) is:

```

RESOURCE FORM r_aknexform_double_line_text_number_field_form
{
    flags = EEikFormUseDoubleSpacedFormat;
    items =
    {
        DLG_LINE
        {
            type = EEikCtEdwin;
            prompt = qtn_aknexform_form_label_edwin;
            id = EAknExFormDlgCtrlIdAccessPoint;
            itemflags = EEikDlgItemTakesEnterKey | EEikDlgItemOfferAllHotKeys;
            control = EDWIN
            {
                flags = EEikEdwinNoHorizScrolling | EEikEdwinResizable;
                width = AKNEXFORM_EDWIN_WIDTH;
                lines = AKNEXFORM_EDWIN_LINES;
                maxlength = EAknExFormEdwinMaxLength;
                ...
            };
            tooltip = qtn_aknexform_hint_text_edwin;
        },
        DLG_LINE
        {
            type = EEikCtNumberEditor;
            prompt = qtn_aknexform_form_label_number;
            id = EAknExFormDlgCtrlIdPacketData;
            itemflags = EEikDlgItemTakesEnterKey | EEikDlgItemOfferAllHotKeys;
            control = NUMBER_EDITOR
            {
                min = AKNEXFORM_NUMBER_EDITOR_MIN_VALUE01;
                max = AKNEXFORM_NUMBER_EDITOR_MAX_VALUE01;
            };
        }
    }
}

```



```

        tooltip = qtn_aknexform_hint_text_number;
    }
};
}

```

The control IDs must be unique across all pages of the dialog and must be non-zero. The form example application defines them, in its HRH file, in the following way:

```

enum TAknExFormDialogControlIds
{
    EAknExFormDlgCtrlIdAccessPoint = 0x100,
    ...
    EAknExFormDlgCtrlIdPacketData,
    ...
};

```

Apart from avoiding the possibility of confusion in the mind of the programmer, there is no particular reason to keep page IDs and control IDs distinct from each other.

Once you have mastered the construction of resources for a multi-page dialog, there is very little else you need to know in order to use them. There are a few additional dialog functions to be aware of, such as `ActivePageId()` and `SetPageDimmedNow()`. In addition, `PageChanged()` is called immediately after focus is transferred to a control on a different page. Its default implementation is empty and you should override it to perform any processing that may be necessary as the result of the page change.

16.6 Dialog APIs

These simple dialogs are enough to show us that `CEikDialog`, for S60, and `CQikSimpleDialog` and `CQikViewDialog`, for UIQ, offer a large range of framework functions, which you can override, and library functions, which you do not override but use to provide specific dialog processing. In this section, our quick tour of dialogs continues with an overview of these APIs.

Resource Specifications

Let's start with a closer look at the resource specification for dialogs. For S60, you define a dialog using the `DIALOG` resource `STRUCT`. A dialog resource might look like:

```

STRUCT DIALOG
{

```

```

LONG flags = 0;
LTEXT title = "";
LLINK pages = 0;
LLINK buttons = 0;
STRUCT items[];
LLINK form = 0;
}

```

The flags for the dialog as a whole are specified in the `flags` member of the `DIALOG` structure. Flag bit values are defined by `EEikDialogFlag` constants in `eikdialog.hrh`. Typical dialogs specify `EEikDialogFlagWait`; other flags control such things as button positioning (right or bottom) and whether there is a title.

The flags for a dialog line are specified in the `flags` member of the `DLG_ITEM` structure. Bit values for these flags are defined by `EEikDlgItem` constants in `eikdialog.hrh`; they allow you to specify that there should be a separator after the item, that the control doesn't take focus, that it has no border, etc.

When you develop a single-page dialog, you do not have to specify a `pages` member. For multi-page dialogs, `pages` members contain the resource definition for multiple pages.

For UIQ, you have two options. If you use the `CQikSimpleDialog` class, you define a dialog using the `QIK_DIALOG` resource `STRUCT`:

```

STRUCT QIK_DIALOG
{
    BYTE    version = 0;
    LONG    flags = 0;
    LTEXT    title = "";
    LLINK    icon = 0;
    STRUCT    configurations[];
    LLINK    controls = 0;
}

```

If you use `CQikViewDialog`, you define a dialog using `QIK_VIEW_CONFIGURATION` and `QIK_CONTROL_COLLECTION` resources:

```

STRUCT QIK_VIEW_CONFIGURATION
{
    BYTE    version = 0;
    LONG    ui_config_mode = 0;
    LLINK    command_list = 0;
    LLINK    view = 0;
}
STRUCT QIK_CONTROL_COLLECTION
{
    STRUCT    items[];
}

```

Framework Functions

`CEikDialog`, `CQikSimpleDialog` and `CQikViewDialog` provide a rich API for derived dialog classes. Many of the functions in the dialog API are either convenience functions that add to the API's size without adding to its complexity or else they are for unusual requirements such as constructing dialogs dynamically without using a resource file.

S60 variant

The framework functions are a set of virtual functions that are called when various events happen during dialog processing. If we want to handle those events, we have to override the default function provided by `CEikDialog`. In each case, the default function does nothing.

To set the control's values and change its size and layout, we override the `PreLayoutDynInitL()` method, which is called before layout. If we want to set the control values but not change the size and layout, we override the `PostLayoutDynInitL()` method, which is called after layout. The methods are defined as:

```
virtual void PreLayoutDynInitL();
virtual void PostLayoutDynInitL();
```

We can control the way dialog's component controls are constructed by overriding the `CreateCustomControlL()` method, which is called when the control type indicated in the resource file is not recognized by the control factory:

```
virtual SEikControlInfo CreateCustomControlL(TInt aControlType);
```

The `OkToExitL()` method has to be overridden if we want to change the dialog behavior when the OK button is pressed.

Sometimes, we may wish to do some processing when the user has finished working with one of the dialog's lines. Changes in time and date editors are not reported using `HandleControlStateChangeL()`, so we need to intercept the event when the user changes the focused line.

The solution is to provide an implementation for `LineChangedL()`. The dialog automatically calls this function when the focused line changes and passes the ID of the control to which the focus is moving. We can find out the ID of the control that currently has the focus by calling `IdOfFocusControl()`:

```
CExampleMyDialog::LineChangedL(TInt aControlId)
{
    TInt id = IdOfFocusControl();
```

```
// Now do my stuff
...
}
```

An older technique is to override the virtual `PrepareForFocusTransitionL()` method. It has a non-trivial default implementation, but it does not include a parameter specifying which line is currently focused. We override the function with:

```
CExampleMyDialog::PrepareForFocusTransitionL()
{
    CEikDialog::PrepareForFocusTransitionL();
    TInt id = IdOfFocusControl();
    // Now do my stuff
    ...
}
```

We call the base-class implementation first; this leaves if the control with the current focus is not valid and is unable to lose focus. Then we call `IdOfFocusControl()` to get the control ID that is currently focused.

UIQ variant

For UIQ, we have to take into consideration the base class that we are using to create dialogs: `CQikSimpleDialog` for simple dialogs and `CQikViewDialog` for complex dialogs. A big difference from the deprecated `CEikDialog` is that the layout in `CQikSimpleDialog` doesn't have to be row-based and building blocks can be used as a layout. In `CQikSimpleDialog`, you still find the `PreLayoutDynInitL()` and `PostLayoutDynInitL()` methods, but you do not find methods dealing with lines because of the differences in the layout. `CQikViewDialog` should be treated more like a view.

You can call the library functions in Table 16.1 from the framework functions.

16.7 Stock Controls for Dialogs

A basic part of dialog programming is using stock controls in dialogs. The general techniques for using stock controls are:

- specify a control type in the `DLG_LINE` struct, using `type =`
- specify initialization data for the control in the `DLG_LINE` struct, using `control=` and an appropriate resource `STRUCT`
- further initialize the control from `PreLayoutDynInitL()` or `PostLayoutDynInitL()`

- extract values from the control when needed, in `OkToExitL()` or other dialog processing functions
- do other things, such as controlling the control's visibility, using dialog library functions.

Table 16.1 Library Functions

Function	Description
<code>CCoeControl* Control(TInt aControlId) const;</code>	Gets a pointer to the control whose ID is specified; panics if the ID does not exist
<code>CCoeControl* ControlOrNull(TInt aControlId) const;</code>	Gets a pointer to the control whose ID is specified; returns 0 if the ID does not exist
<code>CEikLabel* ControlCaption(TInt aControlId) const;</code>	Gets the caption associated with the control whose ID is specified
<code>void SetLineDimmedNow(TInt aControlId, TBool aDimmed);</code>	Dims or un-dims a line; lines should be dimmed if it is not currently meaningful to select them
<code>void MakeLineVisible(TInt aControlId, TBool aVisible);</code>	Makes the control on a line visible or invisible (but does not change the visibility of its caption)
<code>void MakeWholeLineVisible(TInt aControlId, TBool aVisible);</code>	Makes a whole line visible or invisible, both the caption and control
<code>TInt IdOfFocusControl() const;</code>	Gets the ID of the control with focus—that is, the control currently being used
<code>void TryChangeFocusToL(TInt aControlId);</code>	Calls <code>PrepareForFocusLossL()</code> on the currently focused control and, if it does not leave, transfers focus to the control whose ID is specified; this is the way to change focus: the control with the focus should only refuse the request if its state is invalid

UIQ provides 42 stock controls that you can use in dialogs. Other user interfaces may have more or fewer, depending on the user interface design. In this section, we give a fast tour of those controls, including the resource `STRUCT`s you use to initialize them, and their C++ classes.

Table 16.2 Stock Control Classes in UIQ

CEikHorOptionButtonList	CQikTTimeEditor
	CQikDateEditor
CEikLabeledButton	CQikDurationEditor
	CQikTimeEditor
CQikSlider	CQikTimeAndDateEditor
CQikVertOptionButtonList	CEikButtonBase
	CEikCheckBox
CQikSoundSelector	CEikOptionButton
	CEikCommandButtonBase
CQikTabScreen	CEikTwoPictureCommandButton
	CEikTextButton
CQikTabScreenPage	CEikBitmapButton
	CEikCommandButton
CQikNumericEditor	CEikMenuButton
CQikFloatingPointEditor	
CQikNumberEditor	CEikChoiceListBase
	CQikColorSelector
CEikAlignedControl	CEikChoiceList
CEikImage	
CEikLabel	CEikEdwin
	CEikGlobalTextEditor
CEikBorderedControl	CEikRichTextEditor
CEikCalendar	
CEikClock	CEikListBox
CEikComboBox	CEikHierarchicalListBox
CEikProgressInfo	CEikTextListBox
CEikSecretEditor	CEikColumnListBox
CEikWorldSelector	
CQikIpEditor	
CEikScrollBar	
CEikArrowHeadScrollBar	
CQikToolbar	
CQikScrollableContainer	

All the classes in Table 16.2 are ultimately derived from `CCoeControl`. For the classes in the left-hand column the additional derivation is direct, while for those in the right-hand column it is indirect, via `CEikBorderedControl`.

16.8 Custom Controls in Dialogs

In addition to the stock controls that are available, we may need to include a control of our own. To do so, carry out the following steps.

1. Write the control and test it outside a dialog to make sure that it works correctly.

2. If necessary, in an RH file, define a resource file struct associated with the control, specifying the member names and types for the resource initialization data.
3. Choose a value for the control type that is not used by any other control and add its definition to an HRH file – a value of 0x1000 or more is safe.
4. In the resource file for the dialog, include a `DLG_LINE` struct that specifies the new control type value and includes the control's resource struct, if it has one.
5. If your control has a resource struct, implement the control's `ConstructFromResourceL()` function to read initialization data in from the struct.
6. Implement the dialog's `CreateCustomControlL()` function to test for the relevant control type, and construct and return an `SEik-ControlInfo` struct appropriate for your control.

The custom dialog's resource struct, in an RH file, normally provides default values for its members, as in the following example:

```
STRUCT MYCUSTOMCONTROL
{
    WORD width = 100;
    WORD height = 50;
}
```

The control type needs to be defined in an HRH file since it needs to be `#included` in both the resource file and one or more C++ source files. It is typically defined as an enumeration:

```
enum
{
    EMyCustomControl = 0x1000
}

enum
{
    EMyControlId
}
```

An enumeration also defines the control's ID, needed for any control within a dialog.

The following dialog resource is for an `S60` dialog, so we omit a specification of the dialog's title and use one of the standard Avkon button combinations. We also replace the default value for the control's width:

```

RESOURCE_DIALOG r_mycustomcontrol_dialog
{
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
    items =
    {
        DLG_LINE
        {
            type = EMyCustomControl;
            id = EMyControlId;
            control = MYCUSTOMCONTROL
            {
                width = 200;
            };
        }
    };
}

```

The `ConstructFromResourceL()` function of the custom control is similar to the `ConstructL()` function of a normal control, except that the data is read from the resource file. A control needs both a `ConstructFromResourceL()` and a `ConstructL()` if it is to be used both inside and outside a dialog. On entry to `ConstructFromResourceL()`, the passed resource reader is positioned at the start of the control's data and the function consumes all available data for the control.

```

void CMyCustomControl::ConstructFromResourceL(TResourceReader &aReader)
{
    // Read the width and height from the resource file.
    TInt width = aReader.ReadInt16();
    TInt height = aReader.ReadInt16();
    TSize controlSize (width, height);

    SetSize(controlSize);

    ActivateL();
}

```

The default implementation of `ConstructFromResourceL()` is empty, so we do not need to implement it if our custom control has no associated resource struct.

We must, however, implement the dialog's `CreateCustomControlL()` function in each dialog that contains one or more custom controls and it must be capable of creating every custom control that appears in the dialog. The function is called from the system's control factory whenever it is asked to construct a control of an unknown type. It is unusual in that it returns an `SEikControlInfo` structure, defined in `eikfctry.h` as:

```

struct SEikControlInfo

```



```
{
    CCoeControl* iControl;
    TInt iTrailerTextId;
    TInt iFlags;
};
```

In most circumstances, it is sufficient to set the `iControl` element to point to the newly constructed control and set the remaining elements to zero, as in the following example:

```
SEikControlInfo CMyCustomControlDialog::CreateCustomControlL(
                                                    TInt aControlType)
{
    SEikControlInfo controlInfo;
    controlInfo.iControl = NULL;
    controlInfo.iTrailerTextId = 0;
    controlInfo.iFlags = 0;
    switch (aControlType)
    {
        case EMyCustomControl:
            controlInfo.iControl = new(ELeave) CMyCustomControl;
            break;
        default:
            break;
    }
    return controlInfo;
}
```

Summary

This chapter has introduced and explained the basics of dialogs. There are many differences between developing dialogs on S60 and on UIQ. In S60, all dialogs are ultimately derived from `CEikDialog`. UIQ has deprecated `CEikDialog` and replace it with two new classes: `CQikSimpleDialog` and `CQikViewDialog`.

The chapter started by explaining what a dialog is and presented some examples of simple and complex dialogs. The dialog API was also presented and the differences between S60 and UIQ were explained. We ended the chapter by showing how to customize a dialog and giving some basic examples of standard dialogs.

17

Graphics for Display

In the previous few chapters, we have begun to get familiar with the GUI, but we have got about as far as we can without a deeper understanding of Symbian OS graphics. Although we saw some specific examples of drawing to an application's views in Chapter 15, most of the other drawing – for example, in menus, dialogs and standard controls – has been done by the Uikon, UIQ or S60 frameworks.

Now it is time to look at graphics in more detail. In this chapter, we cover the things you need to know for more sophisticated and effective on-screen drawing:

- how to get graphics on screen
- how to use the `CGraphicsContext` API
- the model–view–controller (MVC) paradigm
- how to update the screen without producing visible flicker
- how to share the screen using windows (`RWindow`) and controls (`CCoeControl`)
- the special effects supported by the Symbian OS graphics system
- drawing-related features in the window server
- size-independent drawing, including zooming
- target-independent drawing and drawing to more than one output device
- device characteristics.

In Chapter 18, we cover support for user interaction based on keyboard and pointer devices.

17.1 Drawing Basics

GUIs present many more opportunities for displaying data than a console program. Even in a program that does nothing more than display ‘Hello World’, you face these issues:

- What font should you use?
- What colors should you use for the foreground and background?
- Where should you put the text?
- Should you set the text off in some kind of border or frame?
- How big is your screen and on how much of it do you draw the text?

Whichever way you look at it, you have to make these decisions, so the part of your program that says ‘Hello world!’ is bigger than the corresponding part of a text-mode program. Here is the `Draw()` function of a typical graphical ‘Hello World’ program:

```
void CHelloWorldAppView::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc();
    gc.Clear();
    TRect rect = Rect();
    rect.Shrink(10, 10);
    gc.DrawRect(rect);
    rect.Shrink(1, 1);
    const CFont* font = iEikonEnv->TitleFont();
    gc.UseFont(font);
    TInt baseline = rect.Height() / 2 + font->AscentInPixels() / 2;
    gc.DrawText(*iHelloWorld, rect, baseline, CGraphicsContext::ECenter);
    gc.DiscardFont();
}
```

That’s eleven lines of code, where one would have been enough in a console-based program. The good news, though, is that you can make decisions you need and it is relatively easy to write the code to implement whatever you decide. The example above illustrates the essentials of drawing:

- draw your graphics to a control – `CHelloWorldAppView` is derived from `CCoeControl`
- use the `CGraphicsContext` API to draw the graphics.



Figure 17.1 HelloWorld application screen

Controls

In Chapter 15 we explained that all drawing is done to a control – a rectangular area of screen that occupies all or part of a window. The base class for all controls is `CCoeControl`, which is defined by the CONE component in Symbian OS.

The screen in Figure 17.1 includes two windows: the application view and the button bar. The application view is a single control and the button bar comprises several controls: a container for the whole button bar, which is a compound control and component controls, including the four buttons.

This application already allows us to make some generalizations about controls:

- for an application, a control is the basic unit of GUI interaction: a control can do any sensible combination of drawing, pointer handling and key handling
- a window is the basic unit of interaction for the system: controls always use all or part of a window
- controls can be compound: that is, they can contain component controls; a compound control is sometimes known as a container.

Getting the Graphics Context

In Symbian OS, all drawing is done through a graphics context. The `CHelloWorldAppView::Draw()` function uses `SystemGc()`, a function in `CCoeControl`, to get hold of a graphics context:

```
CWindowGc& gc = SystemGc();
```

All graphics-context classes are derived from `CGraphicsContext`. Each derived class – such as `CWindowGc` – is used to draw on a particular graphics device (in our example, a window). It implements all the functionality specified by the base class and may also implement extra functionality appropriate for the device in question. For example, we can clear the screen through the graphics context:

```
gc.Clear();
```

A ‘graphics context’ is a common notion in the world of computer graphics. Windows uses a ‘device context’; Java uses a `Graphics` object.

Drawing a Rectangle

The next three lines of code draw a rectangular border ten pixels in from the edge of the application view’s area on the screen:

```
TRect rect = Rect();  
rect.Shrink(10, 10);  
gc.DrawRect(rect);
```

`CCoeControl::Rect()` gives the coordinates of the rectangle occupied by the control from within which `Rect()` is called, in this case the application view. The coordinates are given relative to the window that the control uses. The coordinates used by `CWindowGc` drawing functions must also be relative to the window, so this is convenient.

`Shrink()` makes the rectangle 10 pixels smaller than the control’s rectangle on every side – top, right, bottom and left. `TRect` contains many utility functions like this.

`DrawRect()` draws a rectangle using the default graphics context settings. These settings specify:

- that the pen creates a black, one-pixel wide, solid line: this causes the boundary of `rect` to be drawn in black
- that the brush is null, which means that the rectangle is not filled.

You can rely on the default graphics context configuration being set up prior to your `Draw()` function. Do not waste your time setting things that are the default.

Drawing the Text

Now, we draw the text, centered in the rectangle. For good measure, we start by shrinking the rectangle by one pixel on each side, so that

we can afford to white it out without affecting the border we have just drawn:

```
rect.Shrink(1, 1);
```

Then, we get a font from the UI environment:

```
const CFont* font = iEikonEnv->TitleFont();
```

This is our first encounter with a `CFont*`. Later in this chapter, we look at how to get a font of a desired typeface and size, with bold or italic attributes, and so on. To avoid these issues right now, we use a title font from the environment – it is the font used on the title bar of dialog boxes, and it is suitably bold and large.

It is not enough just to have a pointer to the font; we must also tell the graphics context to use it:

```
gc.UseFont(font);
```

This call to `UseFont()` sets the font for all subsequent text-drawing functions – until another `UseFont()` is issued or until `DiscardFont()` is called.

Now we need to draw the text, centered in the `rect` rectangle:

```
TInt baseline = rect.Height() / 2 + font->AscentInPixels() / 2;
gc.DrawText(*iHelloWorld, rect, baseline, CGraphicsContext::ECenter);
```

where `iHelloWorld` is a pointer to a descriptor containing the text to be drawn.

The `DrawText()` function conveniently both draws text – with the graphics context’s pen and font settings – and clears the entire rectangle area using the current brush settings. Horizontal justification is specified by the final parameter (here, `CGraphicsContext::ECenter`, to indicate that the text should be horizontally centered).

Vertical Justification

`DrawText()` does not handle vertical justification because the algorithm is simple and (unlike horizontal justification) vertical justification does not depend on the text of the string. You have to specify the baseline in pixels down from the top of the rectangle, so, as illustrated in Figure 17.2, start with half the height of the rectangle (measured from its top downwards) and then add half the font’s ascent.

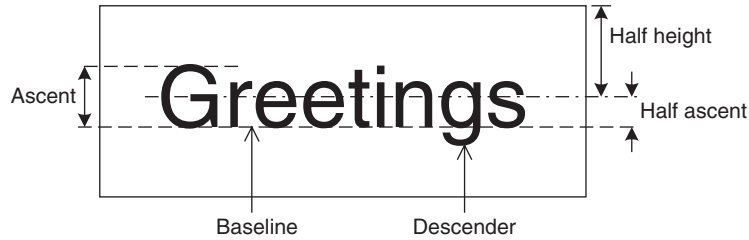


Figure 17.2 Calculating the vertical justification

At this point, we have drawn the text, so we are finished with the font and must discard it:

```
gc.DiscardFont();
```

It is important to discard the font at this stage, in order to avoid a memory leak.

17.2 The **CGraphicsContext** API

All concrete graphics-context classes are derived from **CGraphicsContext**, which offers a rich API for device-independent drawing. Its main features are shown, in UML, in Figure 17.3.

CGraphicsContext contains the main drawing functions and is defined in `gdi.h`. All drawing is done using the current pen, brush and font settings, and is clipped to the current clipping region. The pen, brush, font and clipping region settings provide context for the graphics functions – hence the name of the class.

You can only set graphics context settings. There is no class for pen, brush and so on, and you cannot interrogate a graphics context to find out its current settings. You can keep a graphics context if you need to keep its settings and you can reset a graphics context with a single function call if you need to throw all the settings away.

The drawing functions are illustrated by many examples throughout this book. **CGraphicsContext** is thoroughly documented in the S60 and UIQ SDKs. They also contain an example program, `FontsShell`, that illustrates all the functions covered in this section.

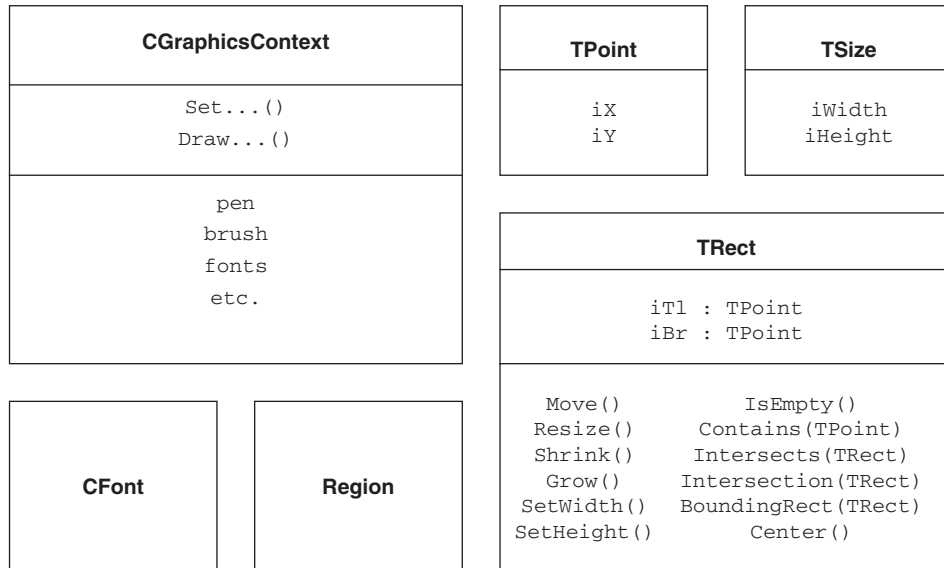


Figure 17.3 Graphics context API classes

Coordinate Classes

Graphics are drawn to a device whose coordinate system is defined in pixels. Each point on the device has an (x, y) coordinate, measured from an origin at the top left of the device, with x coordinates increasing toward the right and y coordinates increasing downwards.

Later in this chapter, we see how pixel coordinates are related to real-world units such as inches or centimeters. For now, we concentrate on pixels and screen-oriented graphics.

Supporting classes for points, rectangles, sizes and regions are defined in `e32std.h`:

- `TPoint` contains `iX` and `iY` coordinates
- `TRect` contains two points, `iTl` for top left and `iBr` for bottom right
- `TSize` contains `iWidth` and `iHeight` dimensions.

These classes are equipped with a large range of constructors, operators and functions to manipulate and combine them, but they make no attempt to encapsulate their members. You do not have to use get and set functions

to access the coordinates of a point, and so on. In truth, there would be very little point in doing so: the representations of these objects are genuinely public.

The two points that define `TRect` can be interpreted by specific graphics implementations in different ways. One common interpretation is to place the top-left point inside the rectangle and the bottom-right point just outside it, as shown in Figure 17.4.

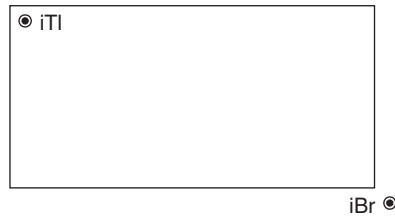


Figure 17.4 An interpretation of `TRect`

This definition makes some things easier, such as calculating the size, because you simply subtract the *x* and *y* coordinates of the top-left from the bottom-right point. It makes other things harder, such as calculations for interactively drawing a rectangle, because you have to add (1, 1) to the pointer coordinates to include the bottom-right corner correctly. If `TRect` is defined to include its bottom-right corner, it makes some things easier and others harder. It is important to remember that the definition of the rectangle depends on the specific graphics interpretation.

Rectangles should be normalized, so that *iTl* coordinates are never greater than corresponding *iBr* coordinates. If you perform a calculation on a `TRect` that might violate this condition, call `Normalize()` to clear things up by swapping the *x* and *y* coordinate values as necessary.

Region-related Classes

Several region-related classes are also defined in `e32std.h`. These define a region of arbitrary shape as the union of several disjoint rectangles. The region classes are used extensively by the window server, but only in specialized application programs.

A region can potentially have very many rectangles, so region classes in general can allocate resources on the heap. All the region classes are heavily optimized and are machine-coded on the target. There is also a region class such that, provided relatively few rectangles are needed to define the region, no heap-based allocation is necessary. While points, rectangles and sizes are simple *T* classes that are easy to allocate anywhere and pass around in client-server calls, regions require more careful management and, like *C* classes, need to be deleted or cleaned up when no longer required.

Setting up the Graphics Context

`CGraphicsContext` holds several important items of context for drawing functions.

- The pen defines draw modes (color and style). These are used for drawing lines, the outlines of filled shapes and text. Draw mode options include Boolean operations on the color values of pixels. The most useful Boolean operations are solid (which means ‘use the color specified’), null (which means ‘do not draw’) and XOR with white (which means ‘invert’ and can be useful for cursor selection, rubber-banding and so on). Style options include solid, dotted, dashed and the pen width. The BITGDI that draws screen graphics (which you saw in Chapter 15) does not support combinations of style and pen width – that is, it cannot do thick dotted lines. Use the `SetPenColor()`, `SetPenStyle()` and `SetPenSize()` member functions to control the pen. By default, the pen is black, solid, and one pixel thick.
- The brush defines the fill and background color or pattern. The brush can be null, solid, a hatching pattern or a bitmap. For hatching and bitmaps, you can set an offset so that pattern fills on adjacent drawing primitives join without odd edge effects. Use `SetBrushStyle()`, `SetBrushColor()`, `SetBrushOrigin()`, `SetBrushPattern()` and `DiscardBrushPattern()` to control brush settings; the defaults are null brush, zero origin.
- The font defines how text is drawn. You specify it by passing a `CFont*` object to `CGraphicsContext`. The CONE environment has one font (`iCoeEnv->NormalFont()`), while the UIQ or S60 environments contain several (`iEikonEnv->TitleFont()`, `LegendFont()`, `SymbolFont()`, `AnnotationFont()` and `DenseFont()`). Use `UseFont()` to set a font, `DiscardFont()` to say you no longer wish to use that font and `SetUnderlineStyle()` and `SetStrikethroughStyle()` to set algorithmic enhancements to the font in use. By default, no font and no algorithmic enhancements are in use. Trying to draw text without a font in use results in a panic.
- The current position is set by `MoveTo()` and various `DrawTo()` member functions; it is moved by `MoveBy()` and a range of `DrawBy()` functions. It is also affected by `DrawPolyLine()`. The `MoveBy()` and `DrawBy()` functions support relative moving and drawing. By default, the current position is (0, 0).
- The origin defines the offset, from the device origin, that is used for drawing. You use `SetOrigin()` to control it. By default, the origin is (0, 0).

- The clipping region defines the region to which you want your graphics to be clipped. You can specify a simple rectangle or a region that may be arbitrarily complex. Use `SetClippingRect()` to set a rectangular clipping region and `CancelClippingRect()` to cancel it. By default, no clipping region (other than the device limits) applies.
- Specialized justification settings for a variant of `DrawText()` can be set, although it is best not to call these directly from your own code. Instead, use the FORM component in Symbian OS to create text views for you.

Use `Reset()` to set the context to default values.

Drawing Functions

Once you have set up the graphics context to your liking, there are numerous ways to draw to the screen. All graphics-context functions are virtual, so they can be implemented in derived classes. Furthermore, all graphics-context functions are designed to succeed, and so do not return anything (in C++ declarations, they return `void`). This requirement is so that multiple graphics-context commands can be batched into a single message and sent to a server for execution: this would not be possible if any graphics context command had a return value.

Points and lines

You can plot a single point or draw an arc, a line, or a polyline. These functions all use the current pen. They are declared in `gdi.h`:

```
virtual void MoveTo(const TPoint& aPoint) = 0;
virtual void MoveBy(const TPoint& aVector) = 0;
virtual void Plot(const TPoint& aPoint) = 0;

virtual void DrawArc(const TRect& aRect, const TPoint& aStart,
                    const TPoint& aEnd) = 0;
virtual void DrawLine(const TPoint& aPoint1, const TPoint& aPoint2) = 0;
virtual void DrawLineTo(const TPoint& aPoint) = 0;
virtual void DrawLineBy(const TPoint& aVector) = 0;
virtual void DrawPolyLine(const CArrayFix<TPoint>* aPointList) = 0;
virtual void DrawPolyLine(const TPoint* aPointList, TInt aNumPoints) = 0;
```

Note that line drawing (including arcs and the last line in a polyline) excludes the last point of the line. As with the specification of `TRect`, this is a mixed blessing: sometimes it makes things easier, sometimes harder. If the last pixel were plotted automatically, it would be harder to un-plot it in the cases where this behavior is not desired. It is easy enough to fix tricky cases by using `Plot()`.

Check out the SDK for the interpretation of `DrawArc()` parameters.

`DrawPolyLine()` starts at the current cursor position. Effectively, `DrawPolyLine()` uses `DrawLineTo()` to draw to every point specified.

Filled shapes

You can draw several filled shapes: a pie slice, an ellipse, a rectangle, a rectangle with rounded corners or a polygon. These functions use the pen and the brush. Use only the pen to draw an outline; use only the brush to draw the shape; use both to draw an outlined shape.

Here are the functions:

```
virtual void DrawPie(const TRect& aRect, const TPoint& aStart,
                    const TPoint& aEnd) = 0;
virtual void DrawEllipse(const TRect& aRect) = 0;
virtual void DrawRect(const TRect& aRect) = 0;
virtual void DrawRoundRect(const TRect& aRect,
                           const TSize& aCornerSize) = 0;
virtual TInt DrawPolygon(const CArrayFix<TPoint>* aPointList,
                        TFillRule aFillRule = EAlternate) = 0;
virtual TInt DrawPolygon(const TPoint* aPointList, TInt aNumPoints,
                        TFillRule aFillRule = EAlternate) = 0;
```

The `DrawPie()` parameters are essentially the same as for `DrawArc()`.

`DrawPolygon()` connects all the points specified and fills the resulting enclosed region. Just as for `DrawLine()`, no relative drawing is used or needed. Intersecting polygons may be drawn, in which case the fill-rule parameter specifies the behavior for regions of even enclosure parity. Check the SDK and the `grshell` example for details. Unlike the other drawing functions, the `DrawPolygon()` function can return errors. This is because the parameters can specify an arbitrarily large amount of data and memory may need to be allocated to deal with it.

Bitmaps

You can draw a bitmap at a scale of 1:1 or stretched to fit a rectangle you specify. Here are the functions:

```
virtual void DrawBitmap(const TPoint& aTopLeft,
                       const CFbsBitmap* aSource) = 0;
virtual void DrawBitmap(const TRect& aDestRect,
                       const CFbsBitmap* aSource) = 0;
virtual void DrawBitmap(const TRect& aDestRect, const CFbsBitmap* aSource,
                       const TRect& aSourceRect) = 0;
```

Use the same-size variant for high-performance blitting¹ of GUI icons and the stretch variants for device-independent view code that supports on-screen zooming or printing. If the above functions are called

¹ Blitting is an operation in which several bitmap patterns are combined into one.

with the source and the destination containing the same number of pixels, then they switch to working in a high-performance way. However, there are functions added to the graphics context classes at the `CBitmapContext`-level, called `BitBlt`, that perform one-to-one drawing, which is always fast.

Text

You draw text in the current font using the following functions:

```
virtual void DrawText(const TDesC& aString, const TPoint& aPosition) = 0;
virtual void DrawText(const TDesC& aString, const TRect& aBox,
                      TInt aBaselineOffset, TTextAlign aHoriz = ELeft,
                      TInt aLeftMrg = 0) = 0;
```

The first (and apparently simpler) function uses the graphics context's justification settings, but you shouldn't call it yourself. Instead, use `FORM` if you need to handle properly laid-out text. For general use, use the `TRect` variant that clips the text to the specified rectangle and paints the rectangle background with the current brush.

A graphics context has no default font. If a text-drawing function is called without a previous call to `UseFont()`, a panic occurs. The panic is particularly ugly when you are drawing to a `CWindowGc` because all window-drawing functions are batched together and sent to the window to be executed later. The window server does not detect the absence of `UseFont()` until the buffer is executed – by which time there is no context information about where the panic occurred. Always remember to use `DiscardFont()` to discard the font after use and so avoid a memory leak.

If you do get a panic while drawing and you cannot work out which of your drawing commands is at fault, then you can use the auto-flushing function of the window server. Calling the function `RWSSession::SetAutoFlush(ETrue)` means that every drawing command is sent to the window server as soon as it is generated and so the panic occurs before the next command is executed. This should only be used as a debugging measure as it greatly reduces the performance of drawing and other code and may increase the amount of flickering that the user sees.

17.3 Drawing and Redrawing

In a GUI program, all drawing is done to controls, which form all or part of a screen window, as we saw with `CHelloWorldAppView::Draw()`.

- The derived control class's `Draw()` function is called when drawing is required.

- `Draw()` gets a graphics context using `SystemGc()`.
- It draws into the area defined by its `Rect()` function.

But it is a bit more complicated than that. Your control must not only draw its content, but must also redraw it when the content changes or when the system requires a redraw. System-initiated redraws can occur when the window is first constructed or the window, or part of it, is exposed after having been obscured by some other application or a dialog box.

Application-initiated redraws occur when the application changes the control's content or the drawing parameters (such as color, scrolling or zoom state) and wants these changes to be shown in an updated display.

In addition, there are various other circumstances – partly system-initiated, partly application-initiated – in which redrawing must occur. For example, dismissing a dialog is application-initiated, but the redrawing of the controls underneath is a system-initiated request and the content may have changed due to the user input into the dialog.

To understand redrawing properly, we must first review the model–view–controller (MVC) paradigm. This is a good way to think about GUI systems and using MVC concepts makes the subsequent discussions much easier.

The Model–View–Controller Pattern

The `CHelloWorldAppView::Draw()` function assumes that the data we need is already available. It does not interrogate a database or ask the user for the string to draw; it uses the data that's already there in the `iHelloWorld` member of the control.

This is a standard paradigm in graphics: draw functions simply draw their model data, they do not change anything. If you want to change something, you use another function and then call a draw function to reflect the update. In fact, this pattern is so common that it has a name: model–view–controller, often abbreviated simply to MVC.

- The model is the data that the program manipulates: in the case of 'Hello World', it is the string text.
- The view is the view through which a user sees the model: in the case of 'Hello World', it is the `CHelloWorldAppView` class.
- The controller is the part of the program that updates the model and then requests the view to redraw in order to show the updates. In 'Hello World', there are no updates and therefore there is no controller. When we come to the Noughts and Crosses application, the controller is more sophisticated because updates may be generated from user interaction.

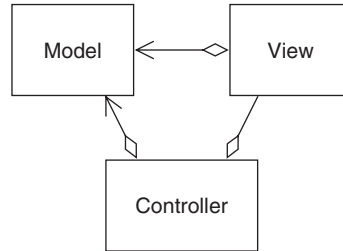


Figure 17.5 Model–view–controller architecture

A strict MVC structure has the architecture shown in Figure 17.5.

- The model is an independent entity; if the program is file-based, the model very often corresponds with the program’s persistent data.
- The view uses the model, since its job is to draw its content.
- The controller coordinates the model and view updates, so it uses both the model and the view.

Depending on the program design, either of the ‘uses-a’ relationships may be upgraded into a stronger ‘has-a’ relationship. The MVC pattern dictates the ‘uses-a’ relationships, but does not force the ‘has-a’ relationships specifically.

Blurring the MVC distinctions

In some programs, the distinctions between model, view, and controller are cleanly reflected by boundaries between classes. But there are many reasons for the boundaries to become blurred in practice.

The ‘Hello World’ program is so simple that there is no point in making such fine distinctions – the `CHelloWorldAppView` contains both the model and the view, and there is no controller at all. The Noughts and Crosses application, on the other hand, is complicated enough to use the MVC structure and greatly benefits from it. However, it turns out that the model (in the MVC sense) isn’t the same as the `COandXEngine` class: it also includes some aspects of the `COandXController`, since knowledge of whose turn it is is contained in this class. Thus the boundary between the MVC model and controller is not quite the same as that between the C++ engine and controller classes.

A sophisticated application uses the MVC pattern again and again – at the large scale for the whole application and at a smaller scale for each interaction within it. You could say that even the button bar has a model (defined by the resource file definitions that construct it) and a controller (somewhere in the application framework).

The MVC paradigm can become particularly blurred when giving feedback to some kinds of user interaction – navigation, cursor selection, animation or drag-and-drop (which admittedly is rare in Symbian OS, though it does exist, for example when re-sizing grid columns in a Symbian OS Sheet). Nonetheless, it remains extremely useful and you can use it to think about the design of many Symbian OS controls and applications.

Terms used in MVC

We should finish this section with a note on nomenclature. A ‘control’ in Symbian OS is not usually a ‘controller’ in the MVC sense. A Symbian OS control does, however, usually contain pure MVC view functionality: its `Draw()` function draws a model without changing it.

In Symbian OS literature, the word ‘view’ is used for a control or some drawing or interaction code, to highlight the fact that the view is entirely separate from the model. A good example is ETEXT and FORM, the Symbian OS rich text components: ETEXT is a model without views and FORM provides views but has no model.

We use ‘application view’ in this sense, while the model is often contained in the document class. In the Noughts and Crosses application, ‘OandX view’ is the name of the control, because the game status model data is kept in a separate class. Often, in Symbian OS literature, the word ‘model’ is used for application data that can be saved to a file.

The `Draw()` Contract

Symbian OS controls use the `Draw()` function to implement MVC view functionality. `CCoeControl::Draw()` is defined in `coecntrl.h` as:

```
IMPORT_C virtual void Draw(const TRect& aRect) const;
```

A derived class overrides this virtual function to draw – or redraw – its model. For the rare cases in which this function is not overridden, there’s a default implementation that leaves the control blank.

Because `CCoeControl::Draw()` is strictly an MVC view function, it should not update the model. It is therefore `const` and non-leaving. Your implementation of `Draw()` must not leave.

This is another reason why most `CGraphicsContext` functions return `void`: if they could fail, `Draw()` could also fail.

Handling redraws

System-initiated redraw handling starts in the window server, which detects when you need to redraw part of a window. In fact, it maintains an invalid region on the window and sends an event to the application that owns the window, asking it to redraw the invalid region. COKE works out which controls intersect the invalid region and converts the event into a call to `Draw()` for all affected controls. A system-initiated redraw must redraw the model exactly as the previous draw.

Application-initiated redraw handling starts (by definition) in the application. If you update a model and need to redraw a control, you can simply call its `DrawNow()` function. `DrawNow()` is a non-virtual function in `CCoeControl` that:

- tells the window server that the control is about to start redrawing
- calls `Draw()`
- tells the window server that the control has finished redrawing.

In theory, then, you do not need to code any new functions in order to do an application-initiated redraw. You can simply call `DrawNow()`, so that your `Draw()` function is called in turn.

Where to draw

It is possible that only part of your control needs to be drawn (or redrawn). To understand this, you need to distinguish between the four regions shown in Figure 17.6.

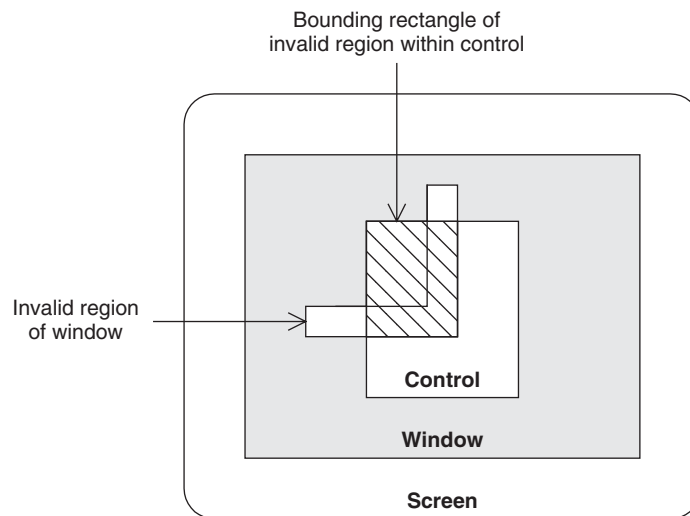


Figure 17.6 Window regions

Your control is part of a window. The window server knows about the window and knows which regions of the window are invalid – that is, the parts that need to be redrawn. Your `Draw()` function must draw the entire invalid region, but it must not draw outside the boundary of the control.

The window server clips drawing to the invalid region – which is clearly bounded, in turn, by the boundary of the window itself.

If your control does not occupy the entire window, you are responsible for ensuring that your redraw does not spill beyond the boundaries of the control.

Often, this turns out not to be too onerous a responsibility: many controls, such as buttons and the Noughts and Crosses application screen, draw rectangles, lines and text that are guaranteed to be inside the control's boundary in any case.

In the few cases where this does not happen, you can issue a `SetClippingRect()` call to the graphics context that ensures that future drawing is clipped to the control's rectangle. Here is an example that is developed later in this chapter:

```
aGc.SetClippingRect(aDeviceRect);  
aGc.SetPenColor(KRgbDarkGray);  
aGc.DrawRect(surround);
```

This is necessary because `surround` could have been bigger than `aDeviceRect`, which is the region of the control that this code is allowed to draw into. You can cancel this later, if you wish, with `CancelClippingRect()`, but since `CGraphicsContext::Reset()` does this anyway and `Reset()` is called prior to each control's `Draw()`, you do not need to do this explicitly from a control.

You can assume that the graphics context is reset before `Draw()` is called. Do not reset it yourself and do not set colors and options that you do not need.

Avoiding wasteful redraws

Drawing outside the invalid region is technically harmless (because such drawing is clipped away by the window server whether it is inside your control's boundaries or not), but it is potentially wasteful. You may be able to save time by confining your drawing activity to the invalid region;

the tradeoff is that you must do some testing to find out what you must draw and what you do not need to draw.

That's the purpose of the `TRect` passed to the `Draw()` function: it is the bounding rectangle of the invalid region. If you wish, you can use this to draw (or redraw) only the part of the control within the passed `TRect`. It is worth doing this if the cost of testing is outweighed by the savings from avoiding irrelevant drawing.

In practice, very few controls gain much by confining their redraw activity entirely to the bounding rectangle: it is simpler and not much slower to redraw the whole control. As a result, the majority of controls are coded to ignore the bounding rectangle. If you are writing a control that does use the `TRect`, remember that you still have to obey the contract to cover the entire invalid region within the boundary of your control and nothing outside your control. (If, however, you are using one of the window server features then you need to draw to the whole of the rectangle, not just the invalid area within it.) You may still have to set a clipping region to ensure this – the system does not set one for you.

Early in its development, Symbian OS passed the invalid region (rather than its bounding rectangle) to `Draw()`. This turned out to be more trouble than it was worth. Regions are data structures of arbitrary size, which are much harder to pass around than `TRect` objects but they were passed whether they were needed or not – and they usually were not. As a compromise, the bounding rectangle of the invalid region was passed.

Breaking the `const` and leave rules

In quite rare circumstances, you may need to do some processing in `Draw()` that is not related to drawing. This could happen, for instance, if your view is very complicated and you are doing lazy initialization of some of the associated data structures in order to minimize memory usage.

In this case, you may need to allocate memory during `Draw()` to hold the results of your intermediate draw-related calculations and this allocation could cause a leave. In addition, you'll want to use a pointer to refer to your newly allocated memory, perhaps in the control. This requires you to change the pointer value, which violates the `const` attribute of `Draw()`.

The solution in this case is to use casting to get rid of the `const` attribute (for more information on the `MUTABLE` macros, see the SDK) and to put your resource-allocating code into a leaving function that is called from a `TRAP()` within `Draw()`. You also have to decide what to draw if your resource allocation fails.

17.4 Drawing Controls

Chapter 15 described the drawing code for the main Noughts and Crosses application. This code draws to the whole of the control each time and attempts to draw each pixel only once so there is no flicker. It is not perfect at this but the approach described there is good enough for most controls. This section illustrates some techniques that allow individual parts of controls to be drawn.

Drawing to Part of a View

Although none of the views in the Noughts and Crosses application require this functionality, it is an important concept for which there are good patterns that can be followed. Let's suppose that, instead of containing one O or X, the Status Window contains two and that they are surrounded with a rectangular box. The code for drawing this might then be:

```
void COandXStatusWin::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc();
    TRect rect = Rect();
    gc.Clear(rect);

    TRect border(iSymbolPos, TSize(2*iSymbolSize, iSymbolSize));
    border = border.Grow(1,1);
    gc.DrawRect(border);

    TRect symRect(iSymbolPos, TSize(iSymbolSize, iSymbolSize));
    DrawSymbol(gc, symRect, iLeftSymbol);
    symRect.Move(iSymbolSize, 0);
    DrawSymbol(gc, symRect, iRightSymbol);
}
```

This function uses four new members, which store the location of the top left of the left symbol and the size at which the symbols are to be drawn; the right symbol is the same size and next to the left one. Now there is much more being drawn in this function and there is a greater chance of flicker. The background is drawn in white and the symbol is drawn on top; however, there is a gap between the drawing of the background and that of the symbol. To avoid this, we can clear the area outside the border to start with:

```
void COandXStatusWin::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc();
    TRect rect = Rect();
    TRect border(iSymbolPos, TSize(2*iSymbolSize, iSymbolSize));
    border = border.Grow(1,1);
```

```
gc.SetPenStyle(CGraphicsContext::ENullPen);
gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
gc.SetBrushColor(KRgbWhite);
DrawUtils::DrawBetweenRects(gc, rect, border);

gc.SetPenStyle(CGraphicsContext::ESolidPen);
gc.SetBrushStyle(CGraphicsContext::ENullBrush);
gc.DrawRect(border);

TRect symRect(iSymbolPos, TSize(iSymbolSize, iSymbolSize));
DrawOneTile(gc, symRect, iLeftSymbol==ETileCross);
symRect.Move(iSymbolSize, 0);
DrawOneTile(gc, symRect, iRightSymbol==ETileCross);
}
```

The `DrawOneTile()` function is defined as:

```
void COandXStatusWin::DrawOneTile(CWindowGc& aGc, const TRect& aRect,
                                   TBool aDrawCross) const
{
    aGc.Clear(aRect);
    DrawSymbol(aGc, aRect, aDrawCross);
}
```

Now the background is cleared immediately before the symbol is drawn on top of it, so there is less chance of flicker.

If only one of the symbols changes, there are several options. First, I could call `DrawNow()` on the control, which may well be good enough for this simple example. It causes the whole of the control to be drawn and the window server processes all the drawing for it. However, I can do much better than this.

One way to stop the window server processing all the drawing is to call `Invalidate()` only on the relevant area (just like the `DrawDeferred()` function does):

```
TRect symRect(iSymbolPos, TSize(iSymbolSize, iSymbolSize));
Win().Invalidate(symRect);
```

This causes the window server to send a redraw event for the rectangle in which this symbol resides. The above function would be called with the rectangle that contains the left symbol; as it does not take any notice of the rectangle, everything is still drawn but the window server clips the drawing to the area that was invalidated, in this case the left symbol. However, if another part of the window also becomes invalid before the redraw event is received then the rectangle passed in the redraw may well be greater. Also, if some window-server features are in operation then the rectangle passed to the redraw function is always the full area of the window.

How can we draw only one of the symbols? The required drawing function is `COandXStatusWin::DrawOneTile()`. The application code does not directly call the `COandXStatusWin::Draw()` function; the `CoeControl` framework does some setup and then calls it. In order to call the `COandXStatusWin::Draw()` function, we need to do part of the setup that is normally done by the framework. The following function draws only one symbol and the window server clips the drawing to the area of that one symbol:

```
void COandXStatusWin::DrawOneTileNow(TBool aLeft) const
{
    TRect symRect(iSymbolPos, TSize(iSymbolSize, iSymbolSize));
    TBool isCross = (iLeftSymbol==ETileCross);
    if (!aLeft)
    {
        symRect.Move(iSymbolSize, 0);
        isCross = (iRightSymbol==ETileCross);
    }
    Window().Invalidate(symRect);
    ActivateGc();
    Window().BeginRedraw(symRect);
    DrawOneTile(SystemGc(), symRect, isCross);
    Window().EndRedraw();
    DeactivateGc();
}
```

This code is written assuming that the window being used is an `RWindow` (a redraw window); if it were a backed-up window (`RBackedUpWindow`), the code would panic as this type of window does not support the functions `Invalidate()`, `BeginRedraw()` and `EndRedraw()`. However, if these lines are omitted then the code works fine for a backed-up window. If a control has to work with both types of window, you could check the window type and, if necessary, skip these lines.

There is one `CCoeControl` function that has to take into account which of the two types of window the control is being displayed in. This is `DrawDeferred()`, whose behavior is shown in the following code:

```
void CCoeControl::DrawDeferred() const
{
    ...
    if (IsBackedUp())
        DrawNow();
    else
        Window().Invalidate(Rect());
    ...
}
```

For a redraw window it just invalidates the window's rectangle. The window is drawn when the control eventually receives a redraw event from the window server. On a control using a backed-up window,

`DrawDeferred()` simply calls `DrawNow()`, causing the window to be drawn immediately.

The `DrawNow()` Pattern

It is useful to pause here to note a few rules about application-initiated redrawing.

- An application-initiated redraw is usually done using a function whose name is of the form `DrawXxxNow()`, where the `Xxx` represents some application-specific graphical element, such as `OneTile`.
- A `DrawXxx()` function (without the `Now`) expects to be called from within an activate-graphics-context and begin-redraw bracket, and to draw to an area that was previously marked as invalid.
- A simple `DrawXxxNow()` invalidates, activates the graphics context, begins the redraw, calls `DrawXxx()`, ends the redraw and deactivates the graphics context.
- A more complex `DrawXxxNow()` function may need to call many `DrawXxx()` functions.
- You should avoid calling multiple consecutive `DrawXxxNow()` functions if you can because it involves (typically) wasteful invalidation, activate-graphics-context and begin-redraw brackets.
- You must in any case avoid calling a `DrawXxxNow()` function from within an activate-graphics-context/begin-redraw bracket, since it causes a panic if you repeat these functions when such a bracket is already active.

You can easily copy the `DrawNow()` pattern for any selective redraws in your own applications.

We see what the activation and begin-redraw functions actually do a little later in this chapter.

17.5 Sharing the Screen

So far, we have covered the basics of drawing and it has been necessary to tell you to do something without explaining why – for instance, the need to call the `ActivateGc()` and `BeginRedraw()` functions in `DrawOneTileNow()`. Now it is time to be precise about how windows and controls work together to enable your application to share the

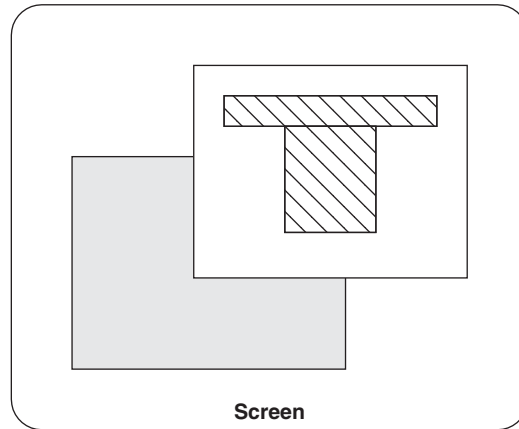


Figure 17.7 Overlapping windows

screen with other applications and to enable the different parts of your application to work together.

Symbian OS is a full multitasking system in which multiple applications may run concurrently. As indicated in Figure 17.7, the screen is a single resource that must be shared among all these applications. Symbian OS implements this sharing using the window server. Each application draws to one or more windows; the window server manages the windows, ensuring that the correct window or windows are displayed, exposing and hiding windows as necessary and managing overlaps.

An application must also share the screen effectively between its own components. These components include the main application view, the button bar and other ornaments: dialogs, menus, and so on. An application uses controls for its components. Some controls – dialogs, for instance – use an entire window, but many others simply reside alongside other controls on an existing window. The buttons on a button bar behave this way, as do the tiles and the status window in the main application view of Noughts and Crosses.

CONE

Every GUI client uses CONE, the control environment, to provide the basic framework for controls and for communication with the window server (see Figure 17.8).

The window server maintains the windows used by all applications. It keeps track of their (x, y) positions and sizes, and also their front-to-back order, which is referred to as a z coordinate. As windows are moved and their z-order changes, parts of them are exposed and need to be redrawn. For each window, the window server maintains an invalid region. When part of a window is invalid, the window server creates a redraw event,

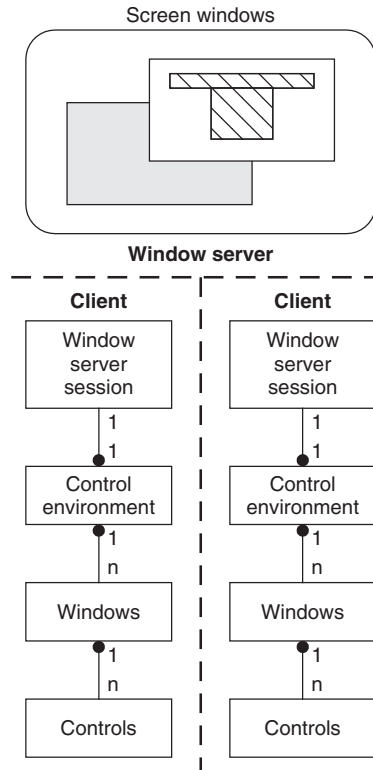


Figure 17.8 Windows and the control environment

which is sent to the window's owning application so that the application can redraw it.

Every application is a client of the window server (see the description of the client–server framework in Chapter 8). It is not necessary to understand the client–server framework in detail for basic GUI programming, because the client interface is encapsulated by CONE.

CONE associates one or more controls with each window and handles window-server events. For instance, it handles a redraw event by calling the `Draw()` function for all controls that use the window indicated and fall within the bounding rectangle of the invalid region.

Window-owning and Lodger Controls

As was pointed out in Chapter 15, there are two types of control.

- A control that requires a whole window is called a 'window-owning' control.
- A control that requires only part of a window, on the other hand, is a 'lodger control' or (more clumsily) a 'non-window-owning' control.

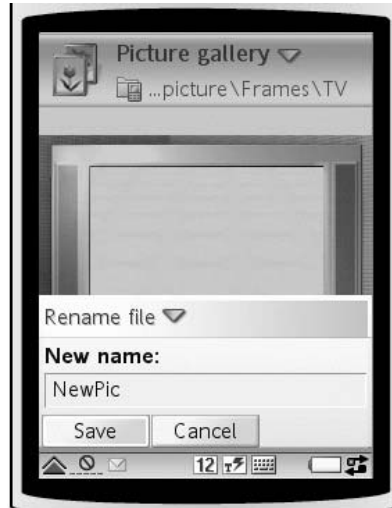


Figure 17.9 A window displaying many controls

The screenshot shown in Figure 17.9 has a single window, but several controls:

- a caption with a pop-out list box
- a captioned text box control
- a button array with two buttons.

Although a window can have many controls, a control has only one window. Every control, whether it is window-owning or a lodger, occupies a rectangle on just one window, and the control draws to that rectangle on that window. A control's window is available via the `Window()` function in `CCoeControl`. There are certain advantages to using lodger controls.

- The client–server traffic between an application and the window server is vastly reduced by lodger controls. Only one client–server message is needed to create an entire dialog, since it includes only one window. Only one event is needed to redraw the whole dialog, no matter how many of its controls are affected. Dialogs are created and destroyed frequently in application use, so these optimizations make a significant difference.
- The overheads associated with complex entities, such as dialogs, are reduced by lodger controls because they are much more compact in memory than windows.

- The processing requirements of lodger controls are less demanding. Windows may move, change z-order and overlap arbitrarily. Lodger controls at peer level on the same window never intersect and occupy only a sub-region of their owning window or control. This makes the logic for detecting intersections much easier than that required for the arbitrarily complex regions managed by the window server.

All these factors improve the system efficiency of Symbian OS, compared to a scenario with no lodger controls. In order to take advantage of these features, most controls should be coded as lodgers, but there are a few circumstances in which you need a window:

- when there is no window to lodge in – this is the case for the application view
- when you need a backed-up window
- when you need to overlap peer controls in an arbitrary way – not according to the stricter nesting rules of lodger controls
- when you need the back-up-behind property, which is used by dialogs and menu panes to hold a bitmap of the window behind them.

Being window-owning is a fairly fundamental property of a control. There isn't much point in coding a control to work as both a lodger and a window-owning control. Decide which it should be and commit to it. Only small parts of a control's code are affected by the decision. So if you find out later that your standalone application view control, for instance, now has a window to lodge in, then you should be able to modify it quite easily.

For instance, in the example in Section 17.9, the `CExampleHelloControl` class adapts the `CHelloWorldAppView` class, turning it into a lodger control. The class declaration changes from:

```
class CHelloWorldAppView : public CCoeControl
{
public:
    static CHelloWorldAppView* NewL(const TRect& aRect);
    ...
    void ConstructL(const TRect& /*aRect*/);
    ...
};
```

to:

```
class CExampleHelloControl : public CCoeControl
{
public:
    static CExampleHelloControl* NewL(const CCoeControl& aContainer,
```

```

...
const TRect& aRect);
private:
void ConstructL(const CCoeControl& aContainer, const TRect& aRect);
...
};

```

The essential change is that we have to pass a `CCoeControl&` parameter to the control's `ConstructL()` function, to tell it which `CCoeControl` to lodge in. The construction changes from something like:

```

void CHelloWorldAppView::ConstructL(const TRect& aRect)
{
    CreateWindowL();
    SetRectL(aRect);
    ActivateL();
    iHelloWorld = iEikonEnv->AllocReadResourceL(R_HELLOWORLD_TEXT_HELLO);
}

```

to:

```

void CExampleHelloControl::ConstructL(const CCoeControl& aContainer,
                                     const TRect& aRect)
{
    SetContainerWindowL(aContainer);
    SetRect(aRect);
    iView = CExampleHelloView::NewL();
    iText = iEikonEnv->AllocReadResourceL(R_EXAMPLE_TEXT_HELLO_WORLD);
    iView->SetTextL(*iText);
    ...
    ActivateL();
}

```

Instead of calling `CreateWindowL()` to create a window of the right size, we call `SetContainerWindowL()` to register the control as a lodger of another window-owning control.

Compound Controls

There needs to be some structure in laying out lodger controls such as those in the Noughts and Crosses application view. As was mentioned in Chapter 15, that discipline is obtained by using a compound control which, by definition, is a control that contains one or more component controls.

- A component control is contained entirely within the area of its owning control.
- All components of a control must have non-overlapping rectangles.

- A component control does not have to be a lodger control: it can also be window-owning. In the majority of cases, however, a component control is a lodger control.

To indicate ownership of component controls to CONE's framework, a compound control must implement two virtual functions from `CCoeControl`:

- `CountComponentControls()` indicates how many components a control has – by default it has zero, but you can override this.
- `ComponentControl()` returns a pointer to the *n*th component, with *n* ranging from zero to the count of components minus one. By default, this function panics (because it should never be called if there are no components). If you override `CountComponentControls()`, you must also override this function to return a component for each possible value of *n*.

If a control contains a fixed number of components, a convenient Symbian OS idiom is to supply an enumeration for the controls, such as:

```
enum
{
    EMyFirstControl,
    EMySecondControl,
    ...
    EMyLastControl,
    ENumberOfControls
}
```

Your `CountComponentControls()` function can then simply return `ENumberOfControls`, which has the advantage that you do not forget to change the return value if, over time, you add or remove controls:

```
TInt anyExampleAppView::CountComponentControls() const
{
    return ENumberOfControls;
}
```

If a control contains a variable number of controls then the return value must be evaluated dynamically. If the control contains a fixed number of components, a simple implementation of `ComponentControl()` might be a `switch` statement, with each case hard coded to the address of the appropriate component control, as in the following example:

```
CCoeControl* anyExampleAppView::ComponentControl(TInt aIndex) const
{
    switch (aIndex)
    {
```

```

case EMyFirstControl: return iMyFirstControl;
case EMySecondControl: return iMySecondControl;
...
case EMyLastControl: return iMyLastControl;
}
return NULL;
}

```

A dialog is also a compound control with, typically, a single window but an unpredictable number of component controls. In this case, instead of hard coding the answers to `CountComponentControls()` and `ComponentControl()` as in the above example, `CEikDialog` uses a variable-sized array to store dialog lines and dynamically evaluates the return values for these functions.

More on Drawing

Drawing to a window is easy for programs but, as indicated by Figure 17.10, it involves complex processing by Symbian OS.

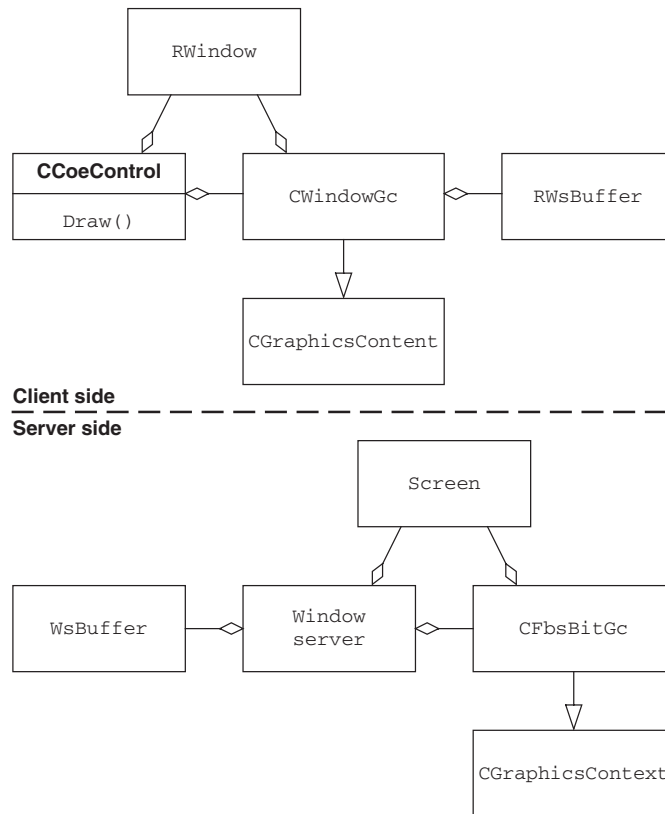


Figure 17.10 Symbian OS drawing architecture

On the client side, an application uses a `CWindowGc` to draw to a window. `CWindowGc`'s functions are implemented by encoding and storing commands in the window server's client-side buffer. When the buffer is full, or when the client requests it, the instructions in the buffer are sent to the window server, which decodes and executes them by drawing directly onto the screen, using a `CFbsBitGc` – a class derived from `CGraphicsContext` for drawing onto bitmapped devices. Prior to drawing, the window server sets up a clipping region to ensure that only the correct region of the correct window can be changed, whatever the current state of overlapping windows on the screen. The window server uses the BITGDI to 'rasterize' the drawing commands. The client-side buffer, which wraps several window-server commands into a single client-server transaction, significantly speeds up system graphics performance.

We can now explain the `DrawOneTileNow()` function that we saw earlier:

```
void COandXStatusWin::DrawOneTileNow(TBool aLeft) const
{
    ...
    Window().Invalidate(symRect);
    ActivateGc();
    Window().BeginRedraw(symRect);
    DrawOneTile(SystemGc(), symRect, isCross);
    Window().EndRedraw();
    DeactivateGc();
}
```

This would be a member function of `COandXStatusWin`, which is derived from `CCoeControl`. The central function is `DrawOneTile()` but this is bracketed by actions necessary to function correctly with the window server.

Invalidating a region

First, we use an `Invalidate()` function to invalidate the region we are about to draw. Remember that the window server keeps track of all invalid regions on each window, and clips drawing to the total invalid region. So before you do an application-initiated redraw, you must invalidate the region you are about to redraw, otherwise nothing will appear (unless the region happened to be invalid for some other reason).

Activating the graphics context

After invalidating the region, the CONE system graphics context must be activated. `CCoeControl::ActivateGc()` is coded as:

```
EXPORT_C void CCoeControl::ActivateGc() const
```

```

{
    CWindowGc& gc = iCoeEnv->SystemGc();
    if(iContext)
        iContext->ActivateContext(gc, *iWin);
    else
        gc.Activate(*iWin);
}

```

The usual case just executes `gc.Activate()` on the control's window, telling the window server's client interface to use the CONE system graphics context to start drawing to it. The function also resets the window graphics context to use default settings. We explain the other case later on, in Section 17.7, when we discuss the control context.

Beginning and ending the redraw

Immediately before drawing, we tell the window server we are about to begin redrawing a particular region. Immediately after redrawing, we tell the window server that we have finished. When the `BeginRedraw()` function is executed by the window server, it has two effects:

- the window server sets a clipping region to the intersection of the invalid region, the rectangle passed into `BeginRedraw()` (the whole window, if no rectangle is passed in), and the region of the window that is visible on the screen
- the window server then marks the region specified by `BeginRedraw()` as valid (or, more accurately, it subtracts the rectangle passed into `BeginRedraw()`, from its current invalid region).

The application's draw code must then cover every pixel of the region specified by `BeginRedraw()`. If the application's draw code includes an explicit call to `SetClippingRegion()`, the region specified is intersected with the clipping region calculated in `BeginRedraw()`.

When the application has finished redrawing, it calls `EndRedraw()`. This enables the window server to delete the region object that it allocated during `BeginRedraw()` processing.

Concurrency

You are probably wondering why the window server marks the region as valid in `BeginRedraw()` rather than in `EndRedraw()`. The reason is that Symbian OS is a multitasking operating system. The following theoretical sequence of events shows why this protocol is needed:

- application A issues `BeginRedraw()` and the affected region is marked valid on application A's window

- application A starts drawing
- application B comes to the foreground and its window overwrites application A's
- application B terminates, so that application A's window is again exposed
- application A's window is now invalid and the window server marks it as such
- application A continues redrawing and issues `EndRedraw()` (and none of this drawing appears on the screen).

At the end of this sequence, the region of the screen covered by the re-exposed region of application A's window is in an arbitrary state. If the window server marked application A's window as valid in `EndRedraw()`, the window server would not know that it needs to be redrawn. Instead, the window server marks A's window as valid in `BeginRedraw()` so that, by the end of a sequence like this, the window is correctly marked invalid and can be redrawn.

You might think that this sequence of events would be rare, but it is possible, so the system has to address it properly.

Redrawing

You should now find it pretty easy to understand how redrawing works. When the window server knows that a region of a window is invalid, it sends a redraw message to the window's owning application, specifying the bounding rectangle of the invalid region. This is picked up by CONE and handled using the following code:

```
EXPORT_C void CCoeControl::HandleRedrawEvent(const TRect& aRect) const
{
    ActivateGc();
    Window().BeginRedraw(aRect);
    Draw(aRect);
    DrawComponents(aRect);
    Window().EndRedraw();
    DeactivateGc();
}
```

This code has exact parallels to the code we saw in `DrawOneTileNow()`: the `ActivateGc()` and `BeginRedraw()` calls are needed to set everything up correctly. However, CONE does not need to call `Invalidate()` here, because the whole point of the redraw is that a region is already known to be invalid. In fact, if CONE did call

`Invalidate()` on the rectangle, it would potentially extend the invalid region, which would waste processing time.

CONE draws the control using `Draw()`, which is passed the bounding rectangle, and then draws every component owned by the control using `DrawComponents()`:

```
void CCoeControl::DrawComponents(const TRect& aRect) const
{
    const TInt count = CountComponentControls();
    for(TInt ii = 0; ii < count; ii++)
    {
        const CCoeControl* ctrl = ComponentControl(ii);
        if(!(ctrl->OwnsWindow()) && ctrl->IsVisible())
        {
            TRect rect;
            const TRect* pRect = (&aRect);
            if(!(ctrl->Flags()) & ECanDrawOutsideRect)
            {
                rect = ctrl->Rect();
                rect.Intersection(aRect);
                if(rect.IsEmpty())
                    continue;
                pRect = (&rect);
            }
            ResetGc();
            ctrl->Draw(*pRect);
            ctrl->DrawComponents(*pRect);
        }
    }
}
```

CONE redraws every visible lodger component whose rectangle intersects the invalid rectangle, and then its components in turn. CONE adjusts the bounding invalid rectangle appropriately for each component control.

CONE also makes an allowance for a rare special case: controls that can potentially draw outside their own rectangle.

Default settings are assured here: the call to `ActivateGc()` sets defaults for the window-owning control that was drawn first; later calls to `ResetGc()` ensure that components are also drawn with default settings.

The loop above does not need to draw window-owning components of the window-owning control that received the original redraw request. This is because the window server sends redraw messages to such controls in any case, in due time. (In fact, it usually sends them first as it tries to send redraw events in a front to back z-order on all the windows of a particular client.)

You can see again how lodger components promote system efficiency. For each component that is a lodger control (rather than a window-owning control), you avoid the client-server message and the calls to activate and begin the redraw. All you need is a single `ResetGc()`, which occupies a single byte in the window server's client-side buffer.

Support for Flicker-free Drawing

As an application programmer, you should be aware of two aspects of the window server that promote flicker-free drawing.

First, the window server clips drawing down to the intersection of the invalid region and the rectangle passed to `BeginRedraw()`, so if your drawing code tends to flicker, the effect is confined to the area being redrawn. You can exploit this in some situations. Imagine that we want to implement a cursor-movement function, but do not want to alter the drawing functions of `COandXAppView` and `COandXTile`. We could write a `DrawTwoTilesNow()` function that accepted the (x, y) coordinates of two tiles to be drawn, enabling us to calculate and invalidate only those two rectangles. We could then activate a graphics context and `BeginRedraw()` on the whole board area, calling `DrawNow()` on the `COandXAppView` class. The window server would clip drawing activity to the two tiles affected, eliminating flicker anywhere else. It is not a very effective flicker-free solution, but in some cases it might just make the difference.

Secondly, the window server's client-side buffer provides useful flicker-free support. For a start, it improves overall system efficiency, so that everything works faster and flickers are therefore shorter. Also, it causes drawing commands to be batched up and executed rapidly by the window server using the BITGDI and a constant clipping region. In practice, this means that some sequences of draw commands are executed so fast that, even if your implementation flickers by nature, no one sees the problem, especially on high-persistence LCD displays. The key here is to confine sequences that cause flicker to only a few consecutive draw commands, so that they are all executed as part of a single window-server buffer.

Finally, and most obviously, the use of lodger controls helps here too, because it means the window-server buffer contains only a single `ResetGc()` command between controls, rather than redrawing and deactivating the graphics context for one control, followed by activating the graphics context and redrawing the next control.

17.6 Support for Drawing in `CCoeControl`

Now is a good time to summarize the drawing-related features of `CCoeControl` that we have seen so far.

First and foremost, a control is a rectangle that covers all or part of a window. All concrete controls are (ultimately) derived from the abstract base class `CCoeControl`. Various relationships exist between controls, other controls, and windows:

- a control can own a window or be a lodger
- a control may have zero or more component controls: a control's components should not overlap and should be contained entirely within the control's rectangle
- a control is associated with precisely one window, whether as the window-owning control or as a lodger
- all lodgers are components of some control (ultimately, the component can be traced to a window-owning control)
- component controls can own a window (say, for a small backed-up region).

Controls contain support for drawing, application-initiated redrawing, and system-initiated redrawing. Applications request controls to draw using the `DrawNow()` function; the window server causes controls to draw when a region of the control's window becomes invalid. In either case, `Draw()` is called to handle the drawing. Functions exist to activate, deactivate and reset a graphics context for use on the control's window.

Control Environment

Each control contains a pointer to the control environment, which any control can reach by specifying `ControlEnv()` (public) or `iCoeEnv` (protected):

```
class CCoeControl : public CBase
{
public:
    ...
    inline CCoeEnv* ControlEnv() const;
    ...
protected:
    CCoeEnv* iCoeEnv;
    ...
};
```

From a derived control or app UI class, including your own application's app UI, you can use `iCoeEnv` to get at the `CCoeEnv`. If you have a pointer to a control or app UI, you can use its public `ControlEnv()` function. If you have access to neither of these things, you can use the static function `CCoeEnv::Static()`, which uses thread-local storage

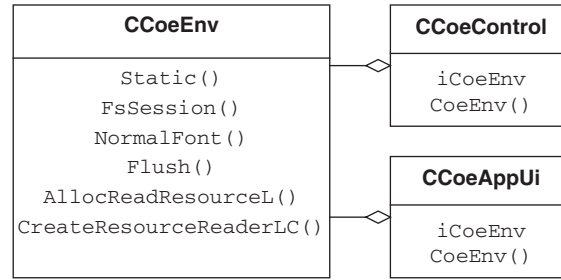


Figure 17.11 The control environment

(TLS) to find the current environment. Since TLS isn't particularly quick, you can store a pointer somewhere in your object for faster access, if you need to do this frequently.

The control environment's facilities (see Figure 17.11) include:

- access to the basic GUI resources: window-server session, window group, screen device and graphics context
- a permanently-available file-server session, available via `FsSession()`
- a font for drawing to the screen (10-point sans serif), available via `NormalFont()`
- a function to `Flush()` the window-server buffer and, optionally, wait for a short period
- convenience functions for creating new graphics contexts and fonts on the screen device
- support for multiple resource files and functions to read resources (see Chapter 13).

See the definition of `CCoeEnv` in `coemain.h` for the full list.

Window-owning and Lodger Controls

A control may either own a window or be a lodger. As illustrated in Figure 17.12, a window-owning control 'has-a' window; a lodger simply 'uses-a' window.

Throughout the lifetime of a control, an `iWin` member points to a drawable window. The drawable window may be standard (`RWindow`) or backed-up (`RBackedUpWindow`) – `RDrawableWindow` is a base class for both of these.

You can call a `CCoeControl` function during the second-phase constructor of a concrete control class, to indicate whether it is

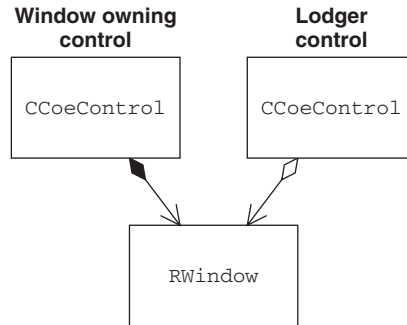


Figure 17.12 Basic control types

window-owning or a lodger. The functions for specifying and testing the window are:

```

class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C virtual void SetContainerWindowL(
        const CCoeControl& aContainer);
    IMPORT_C void SetContainerWindow(RWindow& aWindow);
    IMPORT_C void SetContainerWindow(RBackedUpWindow& aWindow);
    ...
    inline RDrawableWindow* DrawableWindow() const;
    ...
    IMPORT_C TBool OwnsWindow() const;
    IMPORT_C TBool IsBackedUp() const;
    ...
protected:
    ...
    inline RWindow& Window() const;
    inline RBackedUpWindow& BackedUpWindow() const;
    IMPORT_C void CloseWindow();
    IMPORT_C void CreateWindowL();
    IMPORT_C void CreateWindowL(const CCoeControl* aParent);
    IMPORT_C void CreateWindowL(RWindowTreeNode& aParent);
    IMPORT_C void CreateWindowL(RWindowGroup* aParent);
    IMPORT_C void CreateBackedUpWindowL(RWindowTreeNode& aParent);
    IMPORT_C void CreateBackedUpWindowL(RWindowTreeNode& aParent,
        TDisplayMode aDisplayMode);
    ...
protected:
    CCoeEnv* iCoeEnv;
    ...
private:
    RDrawableWindow* iWin;
    ...
};
  
```

The `CreateWindowL()` functions cause a new window – either standard, or backed-up – to be created.

The `SetContainerWindow()` functions tell the control to use an existing standard or backed-up window. This should be used by controls that are themselves components of a control associated with the same window. `SetContainerWindowL()` tells the control to lodge in an existing control and, hence, to use an existing window.

`SetContainerWindowL()` is both virtual and potentially leaving. That's not the best design in Symbian OS; really, it should be neither. You can guarantee that this function won't leave if it is not overridden, so try to think of this function as not having been designed to be overridden. A few classes in Uikon use it for purposes that could be achieved by other means.

Component Controls

A control can have zero, one or more component controls. The three functions that a control uses to manage its component controls are listed in the following extract from the `CCoeControl` class definition:

```
class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C TInt Index(const CCoeControl* aControl) const;
    ...
    IMPORT_C virtual TInt CountComponentControls() const;
    IMPORT_C virtual CCoeControl* ComponentControl(TInt aIndex) const;
    ...
};
```

If you want to implement a container control, you can store controls in any suitable data structure. You override `CountComponentControls()` to indicate how many controls you have and `ComponentControl()` to return the control corresponding to each index value, starting from zero.

By default, `CountComponentControls()` returns zero and `ComponentControl()` panics. These functions work as a pair, so make sure you override them consistently.

`Index()` searches through the component controls one by one, to find one whose address matches the address passed. If none is found, `Index()` returns `KErrNotFound`.

The CCoeControl base class does not dictate how component controls should be stored in a container.

If your container is a fixed-purpose container, such as the Noughts and Crosses application view, which contains only the tiles and one other control, then you can use a pointer to address each component, hardcode `CountComponentControls()` to return `KNumberOfTiles+1` and put code in `ComponentControl()` to return the relevant control.

On the other hand, if your container is a general-purpose container such as a dialog, you may wish to implement a general-purpose array to hold your component controls.

Position and Size of Controls

You can set a control's position and size:

```
class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C void SetExtentL(const TPoint& aPosition, const TSize& aSize);
    IMPORT_C void SetSizeL(const TSize& aSize);
    IMPORT_C void SetPosition(const TPoint& aPosition);
    IMPORT_C void SetRectL(const TRect& aRect);
    IMPORT_C void SetExtentToWholeScreenL();
    ...
    IMPORT_C TSize Size() const;
    IMPORT_C TPoint Position() const;
    IMPORT_C TRect Rect() const;
    IMPORT_C TPoint PositionRelativeToScreen() const;
    ...
    IMPORT_C virtual void SizeChangedL();
    IMPORT_C virtual void PositionChanged();
    ...
    IMPORT_C void SetCornerAndSizeL(TCoeAlignment aCorner,
                                   const TSize& aSize);
    IMPORT_C void SetSizeWithoutNotificationL(const TSize& aSize);
    ...
protected:
    ...
    TPoint iPosition;
    TSize iSize;
    ...
};
```

Position and size are stored in `iPosition` and `iSize`. You can interrogate them with `Position()`, `Size()` or `Rect()` and change them with `SetExtentL()`, `SetPosition()`, `SetSizeL()` and `SetRectL()`.

Changing the size of a control can, in rare cases, cause memory to be allocated, which could fail – so all functions that change size are potentially leaving. `SetPosition()` does not change size, so cannot leave.

- When a control's size is changed, its virtual `SizeChangedL()` function is called.
- A position change is notified by `PositionChanged()`.
- `SetExtentL()` calls `SizeChangedL()` but not `PositionChanged()` – so think of `SizeChangedL()` as always notifying a size change and potentially notifying a position change.
- You can use `SetSizeWithoutNotificationL()` to prevent `SizeChangedL()` being called.
- You can set and interrogate the position relative to the owning window and set the size to the whole screen. `SetCornerAndSizeL()` aligns a control's rectangle to one corner of the screen.

Merely resizing a control should not cause extra resources to be allocated, except in the rare kinds of control which might need to allocate resources in `Draw()` for the extra area. In this case, you should trap any leaves.

Drawing Functions

The following extract from the `CCoeControl` class definition lists the functions related to drawing:

```
class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C virtual void MakeVisible(TBool aVisible);
    ...
    IMPORT_C virtual void ActivateL();
    ...
    IMPORT_C void DrawNow() const;
    IMPORT_C void DrawDeferred() const;
    ...
    IMPORT_C TBool IsVisible() const;
    ...
protected:
    ...
    IMPORT_C void SetBlank();
    ...
    IMPORT_C CWindowGc& SystemGc() const;
    IMPORT_C void ActivateGc() const;
    IMPORT_C void ResetGc() const;
```

```

IMPORT_C void DeactivateGc() const;
IMPORT_C TBool IsReadyToDraw() const;
IMPORT_C TBool IsActivated() const;
IMPORT_C TBool IsBlank() const;
...
private:
    ...
    IMPORT_C virtual void Draw(const TRect& aRect) const;
    ...
};

```

You have to activate a control, using `ActivateL()`, as the final part of its second-phase construction. Assuming that the control's extent is specified and its model is fully initialized by the time `ActivateL()` is called, this makes the control ready for drawing. You can use `IsActivated()` to test whether `ActivateL()` has been called.

`MakeVisible()` sets a control to be visible or not; `Draw()` is not called for invisible controls. `IsReadyToDraw()` returns `ETrue` if the control is both activated and visible. `SetBlank()` is an obscure function that only affects controls which do not override `Draw()`. If you do not `SetBlank()`, then `CCoeControl::Draw()` does nothing. If you do `SetBlank()`, then `CCoeControl::Draw()` blanks the control.

We have already seen that `Draw()` is the fundamental drawing function. `DrawNow()` initiates the correct drawing sequence to draw a control and all its components. `DrawDeferred()` simply invalidates the control's extent, so that the window server sends a redraw message. This guarantees that a redraw is called on the control at the earliest available opportunity, rather than forcing it to happen immediately.

`ActivateL()`, `MakeVisible()` and `DrawNow()` recurse as appropriate through component controls.

`SystemGc()` returns a windowed graphics context for drawing. `ActivateGc()`, `ResetGc()` and `DeactivateGc()` perform the graphics context preparation functions needed for redrawing. Always use these functions, rather than directly calling `SystemGc.Activate(Window())`. They are more convenient and allow control contexts to be supported properly.

17.7 Special Effects

The window server provides many useful special effects to application programs. These include:

- backed-up behind windows
- animation
- debug keys

- a control context
- scrolling
- transparent windows.

Backed-up behind windows were discussed in Chapter 15. This section examines each of the remaining cases in turn.

The details of these special effects depend on the particular user interface that you are using. Transparent windows, for example, needs to be supported by the system; even if two devices are running the same user interface, one may support it and one not.

Animation

Sometimes, you want to do some drawing where timing is an essential feature of the visual effect. This isn't really something that fits well into the MVC paradigm, so we need special support for it.

One kind of animation is used to give reassuring cues in the GUI:

- if you select OK on a dialog – even using the keyboard – the OK button goes down for a short time (0.2 seconds) and then the action takes place
- if you select a menu item using the pointer, the item appears to flash briefly, in a subtle way, before the menu bar disappears and your command executes.

In both cases, this animation reassures you that what you selected actually happened. Or, just as importantly, it alerts you to the possibility that something you didn't intend actually happened. Either way, animation is an extremely important cue: without it, the Symbian OS GUI would feel less easy to use.

Animation isn't the only potential clue that something happened – sound, such as key or digitizer clicks, can be useful too.

Using the flush command

Animation can be achieved very simply. In the case of the dialog button animation, for instance:

- draw commands are issued to draw the button in a 'down' state
- the window server's client-side buffer is flushed, so that the draw commands are executed

- the application waits for 0.2 seconds
- the dialog is dismissed and the relevant action takes place – in all probability, this causes more drawing to occur.

The key thing here is the flush-and-wait sequence. CONE provides `CCoeEnv::Flush()`, which takes a time interval specified in microseconds. The following code therefore implements the flush and wait:

```
iCoeEnv->Flush(200000);
```

The flush is vital. Without it, the window server might not execute your draw commands until the active scheduler is next called to wait for a new event – in other words, until the processing of the current key has finished. By that time, the dialog has been dismissed, so that your draw commands execute ‘flicker-free’ – just at the point when some flicker would have been useful!

Do not use this command to wait for longer than about 0.2 seconds or it will compromise the responsiveness of your application. The entire application thread, with all its active objects, is suspended during this wait. If you need animation to occur on a longer timescale, use an active object to handle the animation task.

Using active objects

Animation using active objects isn’t good enough for some purposes, because active objects are scheduled non-pre-emptively and, particularly in application code, there is no guarantee about how long an active-object event-handler function may take to run. Some animations have to keep running or else they look silly. Examples include the flashing text cursor, or the analog or digital clocks on the bottom of your application button bar. A busy message that appears when a program is running a particularly long event handler would (by definition) prevent any other active object from running.

All such animations run as part of the window server in a window-server animation DLL. They are required to be good citizens of the window server and to have very short-running active objects. See Chapter 6 for a detailed description of active objects.

Uikon Debug Keys

As we saw in Chapter 10, debug builds of the emulator allow you to use various key combinations to control and test your application’s use of memory. These keys are implemented as UIKON components. You can also use the key combinations in Table 17.1 to control your program’s

Table 17.1 Key Combinations for Controlling Drawing

Key	Effect
Ctrl+Alt+Shift+M	Creates a ‘mover’ window that you can move all around the screen, causing redraws underneath it
Ctrl+Alt+Shift+R	Causes the entire application view to redraw
Ctrl+Alt+Shift+F	Causes the window server client API to enable auto-flush, so that draw commands are sent to the window server as soon as they are issued, rather than waiting for the buffer to fill or for a program to issue a <code>Flush()</code> Try this and watch things slow down: it can be useful for naked-eye flicker testing. It is also handy for debugging redraws. You can step through drawing code and see every command produce an instant result on the emulator.
Ctrl+Alt+Shift+G	Disables auto-flush
Ctrl+Alt+Shift+K	Kills the application that currently has keyboard focus (this key is implemented by the window server component)

drawing behavior. Remember that these settings only apply to the current application’s Uikon environment.

Control Context

The `ActivateGc()`, `DeactivateGc()` and `ResetGc()` functions in a control normally pass directly through to window server functions: `gc.Activate(*iWin)`, `gc.Deactivate(*iWin)` and `gc.Reset()`.

These functions, and the way that CONE calls them when recursing through component controls, guarantee that your graphics context is properly reset to default values before each control’s `Draw()` function is called.

In some cases, you do not want your graphics context to reset to system default values: instead, you want to set the default values to something decided by the control. For this purpose, you can use a control context, an interface that overrides the activate and reset behavior of the graphics context.

A control that uses a control context stores a pointer to it in its `iContext` data member. By testing the `iContext` value, a control can determine whether or not to use the system graphics context for all its drawing operations. This explains the second case – first mentioned in Section 17.5, when discussing the activation of the graphics context – in

the action of a control's `ActivateGc()`, which is listed again below, for convenience.

```
EXPORT_C void CCoeControl::ActivateGc() const
{
    CWindowGc& gc = iCoeEnv->SystemGc();
    if (iContext)
        iContext->ActivateContext(gc, *iWin);
    else
        gc.Activate(*iWin);
}
```

See the SDK and examples for further details.

Scrolling

The window server supports scrolling but, once again, this feature may not be available in all user interface implementations. If it is available, you can ask for a region of a window to be scrolled, up, down, left or right. This results in:

- some image data being lost
- the window server moving some image data in the requested direction
- an area of the window becoming invalid.

The (new) invalid area is exactly the same size as the amount of (old) image data that was lost.

Scrolling is clearly always application-initiated. After a scroll, you should call `DrawNow()`: you do not need to invalidate the area that was invalidated by the scroll. You must ensure that the new drawing precisely abuts onto the old data, without any visible joins.

Transparent Windows

Before Symbian OS v8.0, all windows were opaque. In other words, the draw color for each pixel on the screen came from a single window, the top window in the z-order at that point on the screen.

Symbian OS v8.0 added support for transparent windows. An application can say before it activates its window that it wants it to be transparent. It does this by associating a value in the range from 0 to 255 with each pixel of a window. This value is known as the alpha value and windows are said to be 'alpha blended' together. When a window is drawn, this value is used to merge the color from the top window with that of the window underneath it (and the window underneath that if it, too, is transparent). If the value is 255, then you see only the color from the top window; if it is 0, you see only the color from the window below. For

intermediate values, the two colors are merged, with the weighting determined by the particular value. In order to make your window transparent, you must call one of these two functions:

```
TInt RWindow::SetTransparencyBitmap(const CFbsBitmap&
                                   aTransparencyBitmap);
TInt RWindow::SetTransparencyFactor(const TRgb& aTransparencyFactor);
```

If you use `SetTransparencyBitmap()`, then the bitmap should be a gray-scale bitmap and the window server reads values from this bitmap to use as the alpha values. The pixels of the bitmap are matched with the pixels of the window in a one to one fashion. If the bitmap is smaller than the window in one or both dimensions then it is tiled over the window's area. With `SetTransparencyFactor()`, the single color specified (which again has to be a gray color) is used as the alpha value for each pixel of the window (so the level of transparency is constant across the whole window).

In Symbian OS v8.1, a second method of making windows transparent was added. This method can only work if the color mode of the screen is chosen to include alpha values. In this mode, for each pixel, an alpha value is stored, in addition to the usual information about the red, green and blue components. Currently the only display mode that supports this is `EColor16MA`. To make a window transparent by this method, you need to call the following function before the window is activated:

```
TInt RWindow::SetTransparencyAlphaChannel();
```

To draw to a window with this sort of transparency, you need to set the alpha values of the pen and brush colors that are used. These control how much of the color that is already present in the relevant pixels shows through and how much of the color you are drawing that you see. The color that is already there may come from the window (or windows) underneath or from previous draw commands to the same window.

Only certain devices support transparent windows and the `SetTransparency` functions all return an error value of `KErrNotSupported` if this feature is not available. To support transparent windows, the window server has to be configured with the resources that it needs to process them and this is decided when the phone boots up.

17.8 Window Server Features

From Symbian OS v7.0s onward, various features have been added to the window server that complicate exactly when a redraw occurs. They can

also affect what gets drawn on the screen, but only for code that is not well written.

Flicker-Free Redrawing

The way redraws were performed was changed in Symbian OS v7.0s. Before this point all the drawing commands that make up a redraw were applied by the window server directly to the screen. Since then, if the flicker-free redrawing feature is turned on in the system, when the device is booted the window server creates a screen-sized bitmap called the off-screen bitmap. This bitmap is used to construct the result of the redraw and its content is then copied to the screen.

When the client calls `BeginRedraw()`, the window server redirects all the graphics contexts that are active on that window from the screen to the off-screen bitmap. It also draws the background color of the window into the off-screen bitmap for the area being redrawn. Then each draw command is drawn into the off-screen bitmap. When the client calls `EndRedraw()` the area being redrawn is copied from the bitmap to the screen.

Without the off-screen bitmap, flicker can occur in two ways. First, if some of the pixels are drawn twice in the redraw then it is possible (although unlikely for well-written code) that the user of the phone sees those pixels change color twice. Secondly, if there are any sprites visible over the part being drawn, then the sprite flickers with every drawing primitive. With flicker-free redrawing, all drawing operations update the off-screen bitmap and the screen pixels are updated only once. Clearly, this eliminates the first source of flicker entirely and any sprite cannot flicker more than once. In fact, the sprite won't flicker at all, as the off-screen bitmap is copied to the screen in a special way that does not cause the sprite to flicker.

If flicker-free redrawing is in operation, it is even more important that an application's redraw draws to every pixel. Without flicker-free redrawing, an undrawn pixel often continues to have the color that the application had previously drawn on the screen at that point. Although this is never guaranteed in a multitasking system, it does often happen in practice, especially when using `DrawNow()` to update part of your application (that is, to invalidate the area and then redraw it straight away). When using flicker-free redrawing, the off-screen bitmap starts off with the background color of the window, not with its previously drawn content, so you can never get away with this kind of mistake in your drawing code.

Flicker-free redrawing is only in operation if it is configured in the window server at boot time. It can be turned off for individual windows by using the following function, passing the value `EFALSE`:

```
TInt RWindow::EnableOSB(TBool aStatus);
```


Redraw Storing

When only a single window can contribute to the color of pixels at any point on the screen, then it is reasonably efficient to use a single redraw to draw that part of the screen. With the advent of transparent windows, several windows can contribute to the color of a particular group of pixels, so redraws to all of these windows would be needed. It was decided that this was not practical and redraw storing was invented. Although this was designed with transparent windows in mind, if turned on it applies to all windows owned by all applications. This was first added in Symbian OS v8.0, with transparent windows, but there have been improvements to it in later releases.

The first time an application draws its window, it is likely that it is a full redraw to the whole of the area of the window and the window server stores all the drawing commands that the application performs. If, later, the window needs to be redrawn, the window server uses these commands rather than sending a redraw to the client. This is particularly useful for transparent windows, as all of the windows that contribute to the color of pixels in the relevant region can be drawn by the window sever, without needing to involve the client. It can also be useful for a non-transparent window in the case where another window is drawn over the first window and then disappears. The stored commands are used to redraw the window immediately, rather than waiting for the client to perform a redraw.

With redraw storing in operation, the window server can draw the window in many more situations without requiring a redraw from the client. Remember, though, that when the client actually wants the content of its window to change, it has to tell the window server, by a call to `Invalidate()`, to request a redraw. Fortunately, this fits in with the way a client would normally update part of its window, as `DrawNow()` and `DrawDeferred()` both call `Invalidate()`.

However, clients that have more than one view need to take additional actions when redraw storing is active. Consider the case where, while one view is visible, the user performs some action that changes the model of the application and then switches to another view. Without redraw storing, the second view would have received a redraw event and so would have drawn itself to reflect the model changes. With redraw storing in operation, the second view would not receive a redraw event in this situation and thus it would continue to show the unchanged model. To avoid this problem, the client would need to call `Invalidate()` on all non-visible views, either when the model changes or when such a view comes to the foreground.

When redraw storing is in operation for a window, the window server always tries to hold a set of commands that would allow it to draw the window. Thus, if anything happens that makes the stored commands out of date, the window server discards them and issues a full redraw to

the window so that it can create an updated list of stored commands. Operations that cause this to happen include:

- calling `Invalidate()` on part or all of the window
- performing a redraw on part of the window
- performing some non-redraw drawing on the window
- scrolling the window
- changing the window's size.

While redraw storing is in operation on a window, it is best that applications avoid using these operations; if they are used and the window behind is a transparent window, they are in danger of being drawn with the wrong content if the transparent window needs drawing.

The window server is configured at boot time to use, or not use, redraw storing. If transparent windows are enabled, then redraw storing should always be used. It can be turned on or off for a particular window (other than a transparent window) by calling:

```
TInt RWindow::EnableRedrawStore(TBool aEnabled);
```

In Symbian OS v9.1, redraw storing was extended to deal more effectively with the situation of a client performing a partial redraw. Before this change, a partial redraw by a client caused the window server to throw away all the stored commands and then to force the client to perform a full redraw. After the change, the commands corresponding to the client's partial redraw are simply added to the redraw store, along with the part of the window that they refer to. The window server can therefore continue to draw the modified window content, using the stored commands from the initial full redraw, together with those from any later partial redraws.

In some circumstances, following a complex drawing sequence, the window server's repeated execution of the stored instructions can end up adversely affecting an application's performance. You can clear the stored sequence by executing an empty – or nearly empty – `BeginRedraw()` - `EndRedraw()` sequence.

17.9 Device- and Size-Independent Graphics

This section discusses how to write graphics code that is independent of the size of the graphics and the device to be drawn to. First, we study the writing of graphical applications – a topic that involves both size- and device-independent drawing code. Then, we cover the subjects of blitting, fonts and color as these subjects are heavily influenced by the need for size- and device-independence.

- Size-independent drawing allows zooming of a graphic and is necessary when drawing to different targets, such as different-sized screens.
- Target-independent drawing allows an application to draw to more than one kind of device, such as a screen and a printer.
- Device-independent graphical user interfaces requires not just on-screen drawing, but interaction as well. CONE's principles of pointer and key distribution, focus and dimming (see Chapter 18) can be used for any GUI. But the way that Uikon (and, particularly, a customized layer such as UIQ or S60) builds on CONE is optimized for a particular device and its targeted end-users. It turns out that the design of the GUI, as a whole, is heavily dependent on device characteristics.

Printing is a capability that featured in earlier Symbian OS devices. Although it is not used in the current range of Symbian OS smartphones, it continues to be supported and is likely to be reintroduced in future products. Symbian OS has supported target-independent drawing from the start, so that all components can be written to print or draw to a screen.

The Developer's Quest for Device-independent Code

It is not always worthwhile making your code target-independent. So, when should you make your drawing code device-independent and when do you not need to? There are two extreme cases in which the answer is quite clear:

- if your code is designed exclusively for a screen-based user interface, then it should not be target-independent (though you may wish to build in size independence)
- If your code is designed primarily for printing, with a screen-based user interface for editing, then it should be device-independent

An application toolbar can be highly device-dependent, as can the menu and the dialogs (though some size independence may be useful for dialogs). On the other hand, a text view intended for a word processor should be device-independent; so too should a mapping program that might well be printer-oriented.

This answer is only a starting point, however, and there are many awkward, intermediate cases. You have to take the realities of the device into account, even when writing the most device-independent code.

The influence of target devices on your code is even greater when the devices are relatively limited in CPU power and display resolution. With high-resolution displays and near-infinite CPU power, you can render everything with no thought for rounding errors, scale, etc. With small

displays, slower CPUs and no floating-point processor, you have to take much more care in both graphical design and programming.

In a map application, zooming introduces considerations not only of scale, but also of visibility. In a high-level view of a map, you want to see the coastline, a few major cities, big rivers and any borders. In a zoomed-in view of a city, you want to see district names, underground train stations, public buildings, etc. In the high-level view, you wouldn't try to draw these details at small scale: you would omit them altogether. And this omission is device-dependent: you can include more minor features in a printed view than in an on-screen view at the same zoom level, because most printers have higher resolution than a screen.

There are plenty of other complications with maps, such as aligning labels with features and transverse scaling of linear elements such as roads – and many other applications share these considerations. Fortunately, you do not usually edit maps on a handheld device, so there isn't the need for the very quick reformatting code that there is with word processors. As a result, there may be better code sharing between printer and screen views.

Size- and Target-independent Drawing for Applications

Graphical applications may need to support zooming and drawing to different targets: different screens, printers and different types of smartphone, with different screen sizes.

The classes that support size-independence are the same for each target: `CGraphicsContext`, `MGraphicsDeviceMap`, `MGraphicsDevice` and `TZoomFactor`. These have to be implemented differently for the different targets. They are part of the Graphics Device Interface (GDI). Drawing code in an application uses the functions and settings of the GDI classes.

The example application

The example application in this chapter demonstrates how to implement zooming on a screen device. For screen devices, controls are used (derived from `CCoeControl`) to allow user interaction with the graphic and a GUI is used to display and provide an interface to the application. After going through the example code, we see how the size-independent code may be re-used when drawing to a printer.

You can download the example code for this application from developer.symbian.com/main/academy/press. Figure 17.13 shows a screenshot for the example application. The top-left and bottom-right rectangles are used to illustrate the need to do a 'full redraw' rather than drawing over the existing graphic, a point that we'll come back to a little later.

The application structure is shown in Figure 17.14. In order to set the size-independent drawing code in context, an overview of how the

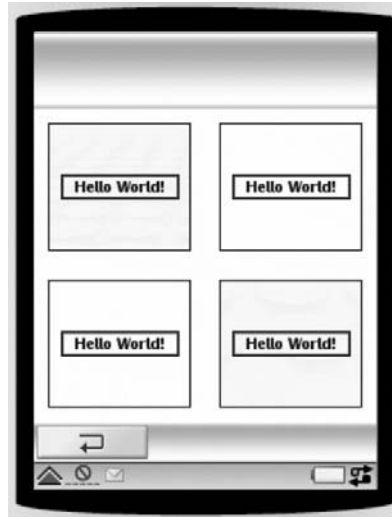


Figure 17.13 The sample application

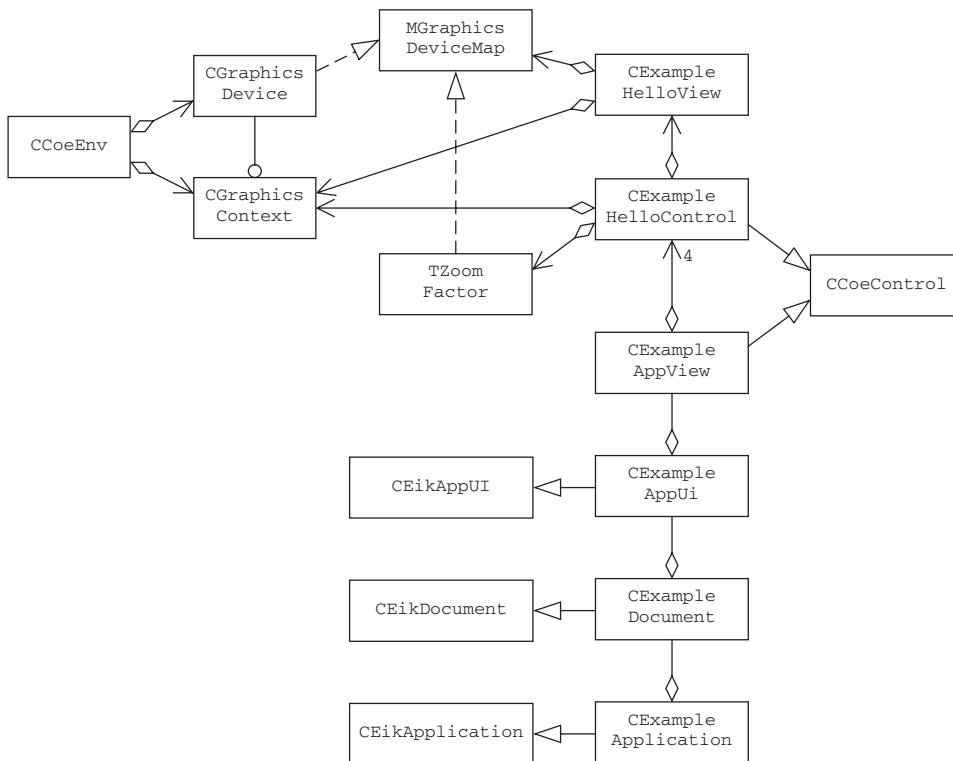


Figure 17.14 The structure of the sample application

example application classes work together is shown in Table 17.2. For greater detail, see the example code.

Table 17.2 Classes in Sample Application

Class	Description
<code>CExampleApplication</code>	Used by the UI framework to create a <code>CExampleDocument</code> object on application start-up
<code>CExampleDocument</code>	Used by the UI framework to create a <code>CExampleAppUi</code>
<code>CExampleAppUi</code>	On construction, creates a <code>CExampleAppView</code> and passes it a rectangle to work with The rectangle is the client area: the area of the screen available to the application for drawing, which does not include the space that is available for the menu and application bands on the screen. The <code>CExampleAppUi</code> class also handles all user interface commands, notably ‘zoom out’ and ‘zoom in’. (<code>Drawing.rss</code> defines the menu options for implementing these commands.)
<code>CExampleAppView</code>	On construction, splits the rectangle into four quarters and creates a control, <code>CExampleHelloControl</code> , for each; in each case, the control is passed a rectangle to work with (and for two of the controls, the ‘full redraw’ parameter is set to false); the user interface commands are passed to these controls
<code>CExampleHelloControl</code>	On construction, creates a view, <code>CExampleHelloView</code> , and sets the text for the view; also used for drawing on application start-up and whenever the view changes: draws a border then calls its view’s draw function
<code>CExampleHelloView</code>	Draws ‘Hello World’ into the centre of the rectangle, then draws a box around the text

`CExampleAppView` and `CExampleHelloControl` are both controls, derived from `CCoeControl`. Such controls are specific to screen devices and so cannot implement target-independent drawing code.

The example separates out the target-independent drawing code into a separate class, which is not a control: `CExampleHelloView`. It has

a drawing function that uses graphics context (CGraphicsContext) drawing functions and a graphics device map (MGraphicsDeviceMap) to access a graphics device's scaling functions. We use the CExampleHelloView class to separate out size- and target-independent code into an independent unit. This allows the size-independent code to be re-used to draw to a printer.

Nevertheless, CExampleHelloControl contains part of the size and target-independent code: it allocates the size and device-dependent font with which to do the drawing using an independent font specification.

The role of each of the GDI classes (see Figure 17.15) is summarized below.

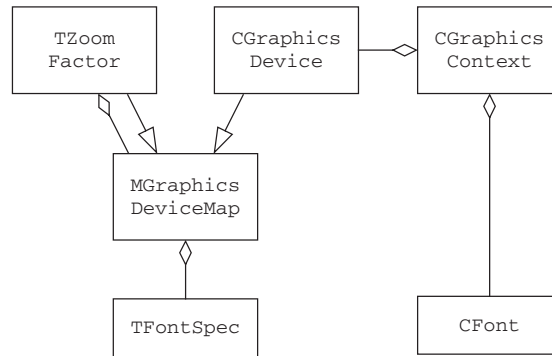


Figure 17.15 The application's GDI classes

- **CGraphicsContext**: the abstract base class created by a graphics device, **CGraphicsDevice**, which contains the main drawing functions. It provides the 'context' in which you are drawing to the associated device in the sense that it holds the pen and brush settings (e.g., color, line styles), font settings (e.g., bold, underline, italic) and the clipping region (the visible drawing area). It deals with pixels of device-dependent size and uses fonts with device-dependent size and representation. The sizes and fonts to be passed to **CGraphicsContext** functions, therefore, need to be converted from size-independent units to size-dependent units. This is done by a class derived from **MGraphicsDeviceMap**: **TZoomFactor** or the **CGraphicsDevice**. This class was described in more detail in Chapter 16 and its handling of colors and bitmaps is described later in this chapter.
- **MGraphicsDeviceMap**: the abstract base class for both graphic devices and zoom factors. It defines the size-dependent functions in a graphics device. These functions convert between pixels and twips and perform font allocation and release. Font allocation involves finding the font supported by the device which is the closest to a device-independent font specification.

- **CGraphicsDevice**: the abstract base class for all graphics devices, which represents the medium being drawn to. It manufactures a graphics context suitable for drawing to itself (using `CreateContext()`), which takes into account the attributes of the device, such as the size and display mode. It allocates (and releases) fonts suitable for drawing to itself and converts between twips and pixels. Important graphic devices are `CScreenDevice`, `CBitmapDevice` and `CPrinterDevice`. (Bitmap applications are discussed later in the chapter.)
- **TZoomFactor**: defines a zoom factor and implements the `MGraphicsDeviceMap` interface. It allocates and releases device-dependent fonts and converts between twips and pixels. It facilitates zooming, because it allows the size of the graphic to become independent of the target size. This class is recursive, because a `TZoomFactor` object can use an `MGraphicsDeviceMap`, which could be a `TZoomFactor` itself. This allows a zoom factor object to contain another zoom factor object, multiplying the effect of the child zoom factor. The top-level zoom factor uses a `CGraphicsDevice` (not another `TZoomFactor`).

In the example application, the zoom factor uses the screen device. This is set up as follows in the control constructor:

```
iZoomFactor.SetGraphicsDeviceMap(iCoeEnv->ScreenDevice());
```

`iZoomFactor` is used to get the appropriate size and device-dependent font and size for drawing.

Device-Independent Drawing

In this section, we study the device-independent drawing code in the `CExampleHelloView` class and the small amount of device-independent code for allocating a font in the `CExampleHelloControl` class. Then, we see how the drawing code is used by the `CExampleHelloControl` class.

```
class CExampleHelloView : public CBase
{
public:
    // Construct/destruct
    static CExampleHelloView* NewL();
    ~CExampleHelloView();

    // Settings
    void SetTextL(const TDesC& aText);
    void SetFullRedraw(TBool aFullRedraw);
```



```

// Draw
void DrawInRect(const MGraphicsDeviceMap& aMap, CGraphicsContext& aGc,
               const TRect& aDeviceRect, CFont* aFont) const;
private:
    void ConstructL();
private:
    HBufC* iText;
    TBool iFullRedraw;
};

```

Firstly, note that `CExampleHelloView` is derived from `CBase`, not from `CCoeControl`. No `CCoeControl`-derived class can be device-independent, because controls are heavily tied to the screen.

The drawing code is located in the `DrawInRect()` function, which takes a device map, a graphics context, and a rectangle within which to draw. `DrawInRect()` can be divided into two sections: it draws the text and a box around the text.

Drawing a box

We start by looking at the part that draws the box. The following code draws a box of a device-independent size. It is two pixels wide, dark gray on the ‘inside’ and black on the ‘outside’.

```

void CExampleHelloView:: DrawInRect(const MGraphicsDeviceMap& aMap,
                                   CGraphicsContext& aGc, const TRect& aDeviceRect,
                                   CFont* aFont) const;
{
    // Draw some text
    ...
    // Draw a surrounding box
    TSize boxInTwips(720,144); // 1/2" x 1/10" surrounding box
    // Converts twips to pixels, using iZoomFactor,
    // to get a box of 1/2" x 1/10"
    TSize boxInPixels;
    boxInPixels.iWidth = aMap.HorizontalTwipsToPixels(boxInTwips.iWidth);
    boxInPixels.iHeight = aMap.VerticalTwipsToPixels(boxInTwips.iHeight);
    TRect box( // Creates a TRect using boxInPixels
              TPoint(aDeviceRect.Center().iX - boxInPixels.iWidth/2,
                    aDeviceRect.Center().iY - boxInPixels.iHeight/2),
              boxInPixels);
    aGc.SetBrushStyle(CGraphicsContext::ENullBrush);
    aGc.SetClippingRect(aDeviceRect);
    aGc.SetPenColor(KRgbDarkGray);
    aGc.DrawRect(box);
    box.Grow(1,1);
    aGc.SetPenColor(KRgbBlack);
    aGc.DrawRect(box); // this makes the box 2 pixels thick
}

```

All graphics-context drawing functions are specified in pixels. So, before we can draw the rectangle, we have to convert the size specified

in twips to a size in pixels. We use the twips-to-pixels functions of the graphics device map. Remember that `aMap` ultimately uses the screen device, which is how it can get the information about the number of twips to a pixel.

Now there is a problem: we cannot be sure that the box is contained entirely within `aDeviceRect` and we should not draw outside `aDeviceRect`. There is no guarantee, either on the screen or on another device, that drawing is clipped to `aDeviceRect`. Because we need that guarantee, we set up a clipping rectangle explicitly.

Even device-independent drawing code must take device realities into account.

We take device realities into account in a different way: although we calculate the size of the surrounding rectangle beginning with twips, we do the expansion explicitly in pixels. Whatever display we draw this rectangle to, we want it to consist of a two-pixel border: two lines spaced apart by a certain number of twips would overlap at small zoom states and be spaced apart at large zoom states.

Getting a font

For this application, we have chosen to draw the message in 12-point SwissA bold text. '12-point SwissA bold' is a device-independent way of specifying a font. We need to get a device-dependent font that meets this specification, taking into account the zoom state.

Symbian OS supports fonts using the following classes (also see Figure 17.15):

- `TFontSpec` is a device-independent font specification, supporting a name, height and style (including italic, bold and superscript/subscript attributes)
- `CFont` is a device-dependent font accessed by `CFont*`; most `CFont` functions, such as `AscentInPixels()`, provide fast access to pixel sizes for any character or for a particular string.

We could allocate (and release) the font along with the rest of the size and target-independent code in `DrawInRect()`. However, `DrawInRect()` is not, and should not be, a leaving function as it is called from the `Draw()` function of the Hello control. However, font allocation could fail and might leave. It is possible to get around this problem using a trap harness, but we do not do this as it is not good practice. It is also bad practice to allocate resources while drawing, so it would make the code unsuitable for large-scale use. Therefore we allocate and release fonts in `CExample-`

`HelloControl::SetZoomAndDeviceDependentFontL()`. This is an appropriate location for the font allocation as it is called both on class construction and as a result of zooming in and out: the size-dependent font is expected to change when the user zooms in or out (and at no other time).

```
void CExampleHelloControl::SetZoomAndDeviceDependentFontL(
                                TInt aZoomFactor)
{
    // Set zoom factor
    ...
    // Allocate a device dependent font for drawing
    iZoomFactor.ReleaseFont(iFont);
    iFont = NULL;
    _LIT(fontName, "SwissA");
    TFontSpec fontSpec(fontName,100);
    fontSpec.iFontStyle = TFontStyle(EPostureUpright, EStrokeWeightBold,
                                    EPrintPosNormal);
    User::LeaveIfError(iZoomFactor.GetNearestFontInTwips(iFont, fontSpec));
}
```

The key function here is `GetNearestFontInTwips()`, a member function of `MGraphicsDeviceMap`. You pass a `TFontSpec` to this function and get back a pointer to a device-dependent font (a `CFont*`). The mapping from `TFontSpec` to `CFont*` is ultimately handled by a graphics device (though the font specification may be zoomed by a zoom factor). Once you have a `CFont*`, you can only use it on the device that allocated it – more precisely, you can only use it for drawing through a graphics context to the device that allocated it. This function usually finds a match but, in the unlikely case that it does not, it returns an error code: we propagate any error by calling `User::LeaveIfError()`.

Notice the need to release the font after use; when you no longer need a `CFont*`, you must ask the device to release it. If you forget to release a font, the effect is the same as a memory leak: your program is panicked on exit from emulator debug builds. There is no error if `ReleaseFont()` is called when the font has not been allocated yet: the function just returns. Therefore it is safe to release the font at the start of the function. The last font allocated is released in the Hello control class destructor.

The code allocates `NULL` to `iFont` because the function may leave before the font is allocated. In this case, the destructor would attempt to release the font again, causing the application to crash.

The font specification uses the `TFontSpec` class, defined by the GDI in `gdi.h`:

```
class TFontSpec
{
public:
    IMPORT_C TFontSpec();
    IMPORT_C TFontSpec(const TDesC& aTypefaceName, TInt aHeight);
    IMPORT_C TBool operator==(const TFontSpec& aFontSpec) const;
    IMPORT_C void InternalizeL(RReadStream& aStream);
```

```
IMPORT_C void ExternalizeL(RWriteStream& aStream) const;
public:
    TTypeface iTypeface;
    TInt iHeight;
    TFontStyle iFontStyle;
};
```

A font specification consists of a typeface, a height and a font style. The `TTypeface` class is also defined by the GDI. It has several attributes, of which the most important is the name.

When you use `GetNearestFontInTwips()`, the height is expected to be in twips, though some devices support a `GetNearestFontInPixels()` function that allows the height to be specified in pixels.

The font style covers posture (upright or italic), stroke weight (bold or normal), and print position (normal, subscript or superscript).

Other font attributes, such as underline and strikethrough, are not font attributes at all – they are drawing effects and you apply them using the `CGraphicsContext` functions.

`TFontSpec` has a constructor that we used for specifying a font in a single statement. It also has a default constructor and assignment operator. Finally, `TFontSpec` has `ExternalizeL()` and `InternalizeL()` functions. These are important: a rich text object, for instance, must be able to externalize a `TFontSpec` object when storing and internalize it when restoring. `TFontSpec` objects can be stored with a rich text object (i.e. a document of some sort). They provide necessary information about how to display the document on a screen or printer. The `TFontSpec` class comes into use whenever a rich text object is stored in or restored from a file. `TFontSpec` is useful to any application that can display text in more than one size or to more than one target.

Zooming and fonts

Before we move on, there is a problem associated with zooming text at very low zoom factors. If this is not dealt with properly then the width of text at low zoom factors does not scale in proportion to the height. This gives rise to the effect illustrated in Figure 17.16, where the text is too wide for the surrounding box.

We may want to display the shape of words, sentences and paragraphs even when the font is too small to read. This requirement applies to a print preview and the Symbian OS rich text view, unlike our application, can properly handle and display fonts even smaller than a pixel. Not all smartphone views have to support this. For many smartphones it is not actually an issue.



Figure 17.16 Text at a very low zoom factor

When you get a `CFont*` from a `TFontSpec`, you convert from a twips size to a pixel size. Pixels are not necessarily square but usually, as you would expect, the height and width of a font scale proportionately. When text is displayed at smaller than a pixel this is not necessarily the case because the smallest unit that a letter can be written in is a pixel, so special scaling support is needed.

The way of getting around this involves calculating the width of the string in twips before converting it to pixels to display. If there is more than one letter per pixel then the string length cannot be properly calculated in pixels.

Drawing the text

We can see how our allocated font is used in the `DrawInRect()` function:

```
void CExampleHelloView::DrawInRect(const MGraphicsDeviceMap& aMap,
    CGraphicsContext& aGc, const TRect& aDeviceRect, CFont* aFont)
{
    // Draw some text
    if (iFullRedraw)
    {
        aGc.SetBrushStyle(CGraphicsContext::ESolidBrush);
        aGc.SetBrushColor(KRgbWhite);
    }
    else aGc.SetBrushStyle(CGraphicsContext::ENullBrush);
    aGc.SetPenStyle(CGraphicsContext::ESolidPen);
    aGc.SetPenColor(KRgbBlack);
    aGc.UseFont(aFont);
    TInt baseline = aDeviceRect.Height()/2 + aFont->AscentInPixels()/2;
    aGc.DrawText(*iText, aDeviceRect, baseline, CGraphicsContext::ECenter);
    aGc.DiscardFont();
    // Draw a surrounding box
    ...
}
```

This uses the version of `DrawText()` that specifies a rectangle and guarantees that drawing will be clipped within that rectangle. However, the `DrawText()` function will only white out the background before drawing if `iFullRedraw` is set. When it is set, the brush setting is solid and white, which causes the background to be drawn in white; when it is not set, there is effectively no brush and the text is drawn straight over whatever was there before, which is the dialog default background image

drawn by the UI framework. In a real program, redrawing the background would not be conditional; depending on the needs of the application, the user interface and the particular phone, an application would always either use an image drawn by the framework or explicitly draw whatever background it requires.

Using the view

CExampleHelloControl has a CExampleHelloView, which it uses for drawing its model:

```
void CExampleHelloControl::Draw(const TRect& /*aRect*/) const
{
    CWindowGc& gc = SystemGc(); //CWindowGc is derived from CGraphicsContext
    TRect rect = Rect();
    rect.Shrink(10,10);
    gc.SetPenStyle(CGraphicsContext::ENullPen);
    gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
    gc.SetBrushColor(KRgbWhite);
    DrawUtils::DrawBetweenRects(gc, Rect(), rect); // whitens the border
    gc.SetPenStyle(CGraphicsContext::ESolidPen);
    gc.SetPenColor(KRgbBlack);
    gc.SetBrushStyle(CGraphicsContext::ENullBrush);
    gc.DrawRect(rect);
    rect.Shrink(1,1);
    iView->DrawInRect(iZoomFactor, gc, rect, iFont);
}
```

First, CExampleHelloControl draws a surrounding 10-pixel border. This code is guaranteed to be on the screen and the 10-pixel border is chosen independently of the zoom state (as it would look silly if the border were to scale to honor the zoom state).

The code is written to ensure that we draw every pixel. The call to `DrawUtils::DrawBetweenRects()` paints white into the region between the rectangle containing the text and the outside of the control. We use the pen to draw a rectangle just inside the rectangle's border. Then the rectangle is shrunk by a pixel and passed to the Hello view's draw function.

The fundamental point that this example shows is that you can implement drawing code that is completely independent of a control and the screen device.

Managing the zoom factor

CExampleHelloControl shows how the zoom factor of a device map relates to the device. Here is the declaration of `TZoomFactor` in `gdi.h`:

```
class TZoomFactor : public MGraphicsDeviceMap
{
    ...
}
```

```

public:
    IMPORT_C TZoomFactor();
    IMPORT_C ~TZoomFactor();
    inline TZoomFactor(const MGraphicsDeviceMap* aDevice);
    IMPORT_C TInt ZoomFactor() const;
    IMPORT_C void SetZoomFactor(TInt aZoomFactor);
    inline void SetGraphicsDeviceMap(const MGraphicsDeviceMap* aDevice);
    inline const MGraphicsDeviceMap* GraphicsDeviceMap() const;
    IMPORT_C void SetTwipToPixelMapping(const TSize& aSizeInPixels,
                                       const TSize& aSizeInTwips);
    IMPORT_C TInt HorizontalTwipsToPixels(TInt aTwipWidth) const;
    IMPORT_C TInt VerticalTwipsToPixels(TInt aTwipHeight) const;
    IMPORT_C TInt HorizontalPixelsToTwips(TInt aPixelWidth) const;
    IMPORT_C TInt VerticalPixelsToTwips(TInt aPixelHeight) const;
    IMPORT_C TInt GetNearestFontInTwips(CFont*& aFont,
                                       const TFontSpec& aFontSpec);
    IMPORT_C void ReleaseFont(CFont* aFont);
public:
    enum {EZoomOneToOne = 1000};
private:
    TInt iZoomFactor;
    const MGraphicsDeviceMap* iDevice;
};

```

TZoomFactor both implements MGraphicsDeviceMap's interface and uses an MGraphicsDeviceMap. TZoomFactor contains an integer, iZoomFactor, which is set to 1000 to indicate a one-to-one zoom, and proportionately for any other zoom factor. In order to implement a function such as VerticalTwipsToPixels(), TZoomFactor uses code such as the following:

```

EXPORT_C TInt TZoomFactor::VerticalTwipsToPixels(TInt aTwipHeight) const
{
    return iDevice->VerticalTwipsToPixels((aTwipHeight*iZoomFactor)/1000);
}

```

TZoomFactor scales the arguments before passing the function call on to its MGraphicsDeviceMap. Other functions combine the zoom and conversion between pixels and twips:

- a pixels-to-twips function scales after calling pixels-to-twips on the device map
- a get-nearest-font function scales the font's point size before calling get-nearest-font on the device map.

The function names in TZoomFactor indicate several ways to set the zoom factor. The method we use in CExampleHelloControl is the most obvious one:

```

void CExampleHelloControl::SetZoomAndDeviceDependentFontL(
    TInt aZoomFactor)

```

```
{
    iZoomFactor.SetZoomFactor(aZoomFactor);
    ...
}
```

GetZoom() works the other way round:

```
TInt CExampleHelloControl::GetZoom() const
{
    return iZoomFactor.ZoomFactor();
}
```

The SetZoomInL() function works in a reasonably standard way, cycling between six hard-coded zoom factors:

```
void CExampleHelloControl::SetZoomInL()
{
    TInt zoom = GetZoom();
    zoom = zoom < 250 ? 250 :
        zoom < 500 ? 500 :
        zoom < 1000 ? 1000 :
        zoom < 1500 ? 1500 :
        zoom < 2000 ? 2000 :
        zoom < 3000 ? 3000 :
        250;
    SetZoomAndDeviceDependentFontL(zoom);
}
```

SetZoomOutL() works the other way round. The rest of CExampleHelloControl is the usual kind of housekeeping that, by now, should hold few surprises. See the source code for the full details.

Printing

CExampleHelloView is a device-independent view: it contains no dependencies on any screen device. That means it can be re-used in some interesting contexts – especially printing. Symbian OS contains other views that are designed in a manner similar to CExampleHelloView. The best example of this is rich text views, delivered by the CTextView class.

CExampleHelloView can be used for printing and print preview, where this is supported by the user interface, without any changes. The GDI specifies a conventional banded printing model that contains, at its heart, an interface class with a single virtual function:

```
class MPageRegionPrinter
{
public:
```



```
virtual void PrintBandL(CGraphicsDevice* aDevice, TInt aPageNo,
                      const TBandAttributes& aBandInPixels) = 0;
};
```

TBandAttributes is defined as:

```
class TBandAttributes
{
public:
    TRect iRect;
    TBool iTextIsIgnored;
    TBool iGraphicsIsIgnored;
    TBool iFirstBandOnPage;
};
```

If you want your application to support printing, use GUI dialogs to set up a print job and start printing. You need to write a class that implements MPageRegionPrinter and then pass a pointer to it (an MPageRegionPrinter*) to the print job. The printer driver then calls PrintBandL() as often as necessary to do the job.

The way in which a driver calls PrintBandL() depends on the characteristics of the printer. Clearly, the driver calls PrintBandL() at least once per page. Drivers may:

- call PrintBandL() exactly once per page, specifying the page number and, in band attributes, a rectangle covering the whole page
- save memory by covering the page in more than one band
- work more efficiently by treating text and graphics separately so that, for instance, all text is covered in one band, while graphics are covered in several small bands.

As the implementer of PrintBandL(), you clearly have to take account of the page number, so that you print the relevant text on every page. Whether you take any notice of the band attributes is rather like whether you take any notice of the bounding rectangle passed to CCoeControl::Draw(): if you ignore these parameters, your code may work fine but you may be able to substantially speed printing up by printing only what's necessary for each band.

You can re-use the view code in CExampleHelloView to implement PrintBandL() for a single-page print:

```
void CMyApplication::PrintBandL(CGraphicsDevice* aDevice,
                              TInt /*aPageNo*/,
                              const TBandAttributes& /*aBandInPixels*/)
{
    // Allocate the font
```

```

TFontSpec fontSpec(_L("SwissA"), 6*10);
fontSpec.iFontStyle = TFontStyle(EPostureUpright, EStrokeWeightBold,
                                EPrintPosNormal);

CFont* font;
User::LeaveIfError(aDevice.GetNearestFontInTwips(font, fontSpec));
// Draw the view
CGraphicsContext* gc;
User::LeaveIfError(aDevice->CreateContext(gc));
TRect rectInTwips(TPoint(2880, 2880), TSize(2880, 1440));
TRect rect(aDevice->HorizontalTwipsToPixels(rectInTwips.iTl.iX),
           aDevice->VerticalTwipsToPixels(rectInTwips.iTl.iY),
           aDevice->HorizontalTwipsToPixels(rectInTwips.iBr.iX),
           aDevice->VerticalTwipsToPixels(rectInTwips.iBr.iY));
iView->DrawInRect(*aDevice, *gc, rect, font);
delete gc;
// Release the font
aDevice.ReleaseFont(font);
}

```

Because `PrintBandL()` is a leaving function, we can allocate the font within it.

This code prints the text in a rectangle 2 inches by 1 inch, whose top left corner is 2 inches down and 2 inches right from the edge of the paper – regardless of the paper size and margins. More realistic print code would take account of the paper size and margins, which are set up by the GUI dialogs.

The `CGraphicsDevice::CreateContext()` function creates a graphics context suitable for drawing to the device; it is **the** way to create a graphics context. Calls such as `CCoeControl::SystemGc()` simply get a graphics context that has been created earlier by the CONE environment, using `iScreenDevice->CreateContext()`.

Rich text and other views

The `CTextView` rich text view class provides a powerful view that hides extremely complex functionality underneath a moderately complex API. This class supports formatting and display of rich text (model objects derived from `CEditableText`), printing, editing, and very fast on-screen update – enough to allow high-speed typing in documents scores of pages long. User interfaces can provide controls derived from `CEikEdwin`, such as UIQ's `CEikRichTextEditor`, that applications can use for rich-text editing.

More on the Graphics Device Interface

Device-independent drawing is supported by the Symbian OS GDI. All graphics components and all components that require a graphics object, such as text content, depend ultimately on the GDI. The GDI defines:

- basic units of measurement – pixels and twips – that are used by all drawing code
- basic definitions for color
- graphics devices and graphics contexts
- fonts
- bitmapped graphics
- device mapping and zooming
- printing.

The GDI is well illustrated by examples in the SDKs. The UIQ SDK's sample program is `GraphicsShell`. As its name suggests, it is a shell with several examples inside it, including the basic drawing functions supported by `CGraphicsContext`, bitmapped graphics, the `CPicture` class, zooming, off-screen bitmap manipulation and the built-in fonts.

We have already seen most of the GDI. In this brief section, we review and develop a few more themes: bitmap handling, font management, printing, color and display modes and web browsing.

Blitting and bitmaps

Displays are fundamentally bitmap-oriented. Graphics primitives such as `DrawLine()` have to rasterize or render the line – that is, they must determine which pixels should be drawn in order to create a bitmap image that looks like the desired line. Rasterizing in the Symbian OS is the responsibility of the BITGDI, which implements all drawing functions specified by `CGraphicsContext` for drawing to on- and off-screen bitmaps.

Another approach to updating a bitmapped display is simply to blit to it: to copy a bitmap whose format is compatible with the format on the display. Blitting is extremely efficient if the source and destination bitmaps are of identical format.

Any GUI worth its salt takes advantage of the efficiency of blitting to optimize certain operations:

- on-screen icons are not rendered using drawing primitives but pre-constructed in a paint program and blitted to screen when required
- flicker-free screen re-draws are performed by rendering to an off-screen bitmap (potentially slow) and then blitting to the screen when needed (usually quick); the off-screen bitmap is maintained by `RBackedUpWindow`; whenever a redraw is required and the window contents haven't changed, the image can simply be blitted from the screen and not redrawn from scratch; redrawing is done entirely from the backup bitmap

- animation is a special case of flicker-free update that can be implemented using a sequence of blits, one for each image frame
- screen fonts are blitted from the font bitmaps onto the screen (or off-screen bitmap).

Blitting is a very useful technique but it isn't always the best thing to use. Bitmaps use a lot of memory; if you can construct a picture from a short sequence of drawing primitives, it is often more compact than storing the picture as a bitmap. Bitmaps cannot be scaled effectively: you lose information if you scale them down and they look chunky if you scale them up. Scaling is generally slow, which eliminates one of the major advantages of using bitmaps. Bitmaps are fixed: you can only use them to store pre-drawn or pre-calculated pictures, or to cache calculated images for reuse over a short period of time. Bitmaps are highly efficient for screens, but highly inefficient for printers, because they involve large data transfers over relatively slow links. They also involve scaling, but usually that's acceptable on printers because the scaling is to a size similar to that which would have been used for the bitmap on screen anyway.

Figure 17.17 shows a UML diagram of the classes that support bitmaps in Symbian OS.

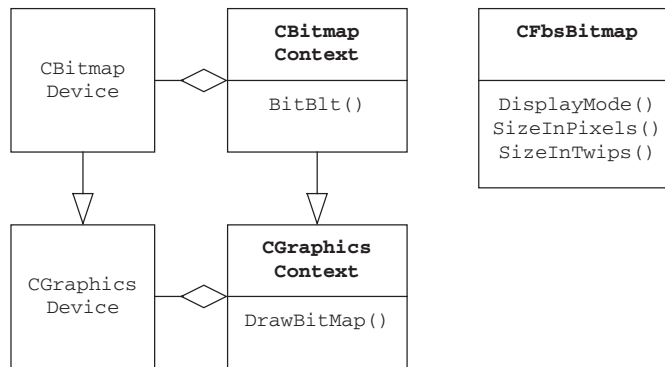


Figure 17.17 Support classes for bitmaps

The class for bitmaps is `CFbsBitmap`, which is defined in `fbs.h`. Key properties of a bitmap include:

- display mode – the number of bits per pixel and the color or gray encoding scheme (see the `TDisplayMode` enumeration in `gdi.h` and the list below)
- size in pixels
- size in twips
- bitmap data, which you can get using `GetScanLine()` and similar functions.

Note that the `CFbsBitmap` constructor sets a pixel size only, but you can also set the size in twips. (The size in twips is optional and defaults to 0.0.) There are two functions for setting the size in twips: you can pass in a size in twips directly or in a graphics device map, which can be used for scaling, to convert the internally stored size in pixels to a size in twips. These functions are defined as follows:

```
void SetSizeInTwips(const MGraphicsDeviceMap* aMap);  
void SetSizeInTwips(const TSize& aSizeInTwips);
```

Functions are provided to set and access all the bitmap properties and also to internalize and externalize bitmaps using streams. `CGraphicsContext` requires that any graphics device (and hence any graphics context) can do four basic operations with any `CFbsBitmap` object. The `DrawBitmap` functions of `CGraphicsContext` draw in one of the following fashions:

- from the source bitmap to a region of the device identified by its top-left corner; the draw size depends on its size in twips, which must be specified; the target device converts from twips to pixels, which means that the bitmap can be drawn according to an actual size, regardless of the twips-to-pixels mappings of the source and target
- from the source bitmap to a region of the device identified by its bounding rectangle; the bitmap is scaled to fit the target rectangle, which is specified in pixels
- from a rectangular region of the source bitmap to a rectangular region of the device; the bitmap region is scaled to fit the target rectangle and both rectangles are specified in pixels.

You can also use a bitmap in `UseBrushPattern()`, for background painting.

The GDI defines a bitmapped graphics device, `CBitmapDevice`, and a bitmapped graphics context, `CBitmapContext`. You can read pixels and scan lines from a `CBitmapDevice` and create a `CBitmapContext` for drawing. You can perform actions such as clear, copy rectangles, blit and 'blit under mask' to a `CBitmapContext`. (The blit-under-mask functions are used for drawing icons with transparent backgrounds.) `CBitmapContext::BitBlt()` always does one-to-one pixel blitting, regardless of pixel size. Compare it with `CGraphicsContext::DrawBitmap()`, which always scales if it needs to, even when copying from a bitmap to a bitmapped device.

Bitmaps are managed by the font and bitmap server. Pre-built bitmaps are built into MBM files, from Windows BMP files, using `bmcconv` – usually one per application or component. MBM files can be built into ROM in a format corresponding to the bitmap layout of the Symbian OS device's

normal screen mode: this makes blitting from them particularly efficient. Bitmaps delivered with non-ROM components can be built into a compressed MBM file from which bitmaps are loaded into the shared heap of the font and bitmap server (FBS) as needed, before being blitted elsewhere. Off-screen bitmaps may be allocated by applications: they reside in the FBS's shared heap.

More on fonts

Symbian OS can use bitmap fonts, for which character bitmaps at various sizes are stored, and scalable fonts, for which algorithms to draw characters are stored. Bitmap fonts are stored in a preset range of sizes: for other sizes they can be algorithmically scaled, but the quality is unlikely to be good. Scalable fonts, however, as the name implies, can produce any size with equal quality.

For applications using Western character sets, scalable fonts are useful; for Far Eastern character sets, they are vital, since the font information for even a single size is enormous. By using scalable font technology, information is only needed for one size. Other sizes, and rasterization for printers, can be handled by the scalable font system.

A number of systems for scalable fonts have been invented, such as Apple's TrueType and the open-source FreeType. Symbian OS has a framework called the Open Font System, that allows rasterizer plug-in DLLs to support particular systems. Such a plug-in reads font files and generates character bitmaps, which are then handled exactly as bitmap fonts.

Scalable fonts are not always required. For example, the P800 uses only a small and fixed number of bitmap fonts because it has no application requirement, such as a word processor, for a large set of fonts. All applications generally know which font and size they require for any specific widget. These fonts could be requested by UID rather than by a device-independent `TFontSpec` specification. This is an optimization issue.

If using `TFontSpec`, clients cannot tell whether a font originated as a bitmap or from a scalable font rasterizer. The method that we have already seen is always used:

- a `TFontSpec` (and its supporting classes) specify a font in a device-independent way
- an `MGraphicsDeviceMap`, which leads to a `CGraphicsDevice`, gets a device-dependent font, using `GetNearestFontInTwips()` and the `TFontSpec`.

You can find out what fonts are available on a device through its typeface store, implemented by `CTypefaceStore`. You can ask how many typefaces there are, iterate through them, and get their properties. The `FontShell` example in the UIQ SDK does exactly this.

Fonts for the screen (and off-screen bitmaps) are managed by the font and bitmap server (FBS). When you allocate a font, using `GetNearestFont()` or similar functions, it creates a small client-side `CFont*` object for the device and ensures that the bitmaps for the font are available for blitting to the screen (or off-screen bitmap). For built-in bitmap fonts, the bitmaps can be in ROM, in which case the `CFont*` acts as a handle to the memory address. Getting a font is a low-cost operation for such fonts. Alternatively, fonts that are installed or generated are loaded into RAM and made accessible so that all programs can blit them efficiently from a shared heap. The `CFont*` acts as a handle to an address in this heap.

Releasing a font releases the client-side `CFont*` and, in the case of an installable font, decrements a usage count that causes the font to be released when the usage count reaches zero. Installable fonts are a main reason why a `GetNearestFont()` call may fail (because of a potential out-of-memory error). It is worth releasing a font as soon as possible to free up the memory.

Sometimes, you want a device-dependent font. For instance, you may want a font of a particular pixel size, without going through the trouble of mapping from pixels to twips and then back to pixels again. For this, you can use `GetNearestFontInPixels()` on most graphics devices: this uses a `TFontSpec` but interprets its `iHeight` in pixels rather than twips. Or, you may want one to use a special character from a particular symbol font. For this, you can use `GetFontById()`, which requires you to specify a UID rather than a `TFontSpec`.

An application such as a word processor may utilize a large number of fonts, because there would generally be a number of fonts to select from, and each would have a large range of sizes.

Sometimes the device-independence implied by a `TFontSpec` isn't device-independent enough. You cannot rely on any particular typeface being present on any device. In response, Symbian applications usually contain font specification information in resource files, so that this aspect of an application can be localized: there are `FONT` and `NAMED_FONT` resource structures for this.

Additionally, `TFontSpec` can specify a font by typeface attributes. For example, most Western-locale devices have a sans serif typeface such as Arial or SwissA. To avoid hard-coding device-dependent names such as these, you can simply set the proportional attribute of the typeface and leave the others, including the name, clear. You can do this by calling `TFontSpec::iTypeface.SetIsProportional(ETTrue)`. Then when you request a font of a given size, the best match for it is found.

Do not over-generalize: text-layout conventions are different for Far Eastern applications, so you may have to change other things if you want to support Far Eastern locales. You do not need to make any special font-related changes to support Unicode-based Western-locale machines.

More on printing

A comprehensive print model is built into the GDI and implemented by higher-level components of Symbian OS. Note however that printing is usually only relevant to larger, communicator-type phones. More compact devices usually find it unnecessary and cut out the support.

To provide system support for particular printers, printer drivers are written as plug-ins that implement the `CPrinterDevice` interface and stored in the `/system/printers` directory. On the client-side, the usual approach is to use the provided GUI dialog (if present, it is usually called `CEikPrinterSetupDialog`) to allow the user to set up and start printing. Print setup options can include:

- page setup: paper size, margins, rich text for header and footer, page numbering, header on first page, footer on first page
- print setup: the number of copies you want to print and the driver you want to use
- print preview: a preview showing the layout.

In your document model, you should externalize the print settings. Of course, such options can also be set up directly: `CPrintSetup` is the key class here. On-screen print preview support is provided through `CPrintPreviewImage`. And, as we have already seen, at the heart of any print-enabled application you have to implement the `PrintBandL()` function of `MPrintProcessObserver`.

Programming printer support to this extent is not very difficult. However, if you do need to print from your application, then the requirements to write code that is independent of a particular user interface increase hugely, to where it becomes a key consideration for many elements of the application.

For example, an application toolbar can be highly device-dependent, as can the menu and the dialogs (though some size independence in the latter case would be useful). On the other hand, a text view intended for a word processor should be highly device-independent.

The requirements for printing text efficiently and for fast interactive editing of potentially enormous documents are, however, substantially different. So the Symbian OS text view component contains much shared code, but also a lot of quite distinct code, for these two purposes. Less-demanding applications have a greater proportion of shared code.

To take another example, without considering printing, a spreadsheet could be considered to be an abstract grid of cells, each containing text, numbers, or a formula. If you take that view, your drawing code can be pixel-oriented and it won't be too difficult if you decide to support zooming. But this changes if people need to print the spreadsheet, with sensible page breaks and embedded charts. If you write a spreadsheet

that supports all this, you need to design for printing from the beginning and optimize your on-screen views as representations of the printed page.

Color management

The basic class for color is the `TRgb`: a red–green–blue color specification. A `TRgb` object is a 32-bit quantity, in which eight bits are each available for red (R), green (G), and blue (B). The other eight bits are often unused; in some situations, they are used for alpha (transparency) information.

Wasted memory in such a fundamental class in Symbian OS? Actually, the waste is small. First, it is hard to process 24-bit quantities efficiently in any processor architecture. More importantly, `TRgb`s do not exist in large numbers – unlike, say, the pixels on a screen or bitmap. Bitmaps are stored using only the minimum necessary number of bits.

Constants are defined for the set of 16 EGA colors.² The following definitions in `gdi.h` show how the R, G and B values are combined in a `TRgb`:

```
#define KRgbBlack      TRgb(0x000000)
#define KRgbDarkGray  TRgb(0x555555)
#define KRgbDarkRed    TRgb(0x000080)
#define KRgbDarkGreen  TRgb(0x008000)
#define KRgbDarkYellow TRgb(0x008080)
#define KRgbDarkBlue   TRgb(0x800000)
#define KRgbDarkMagenta TRgb(0x800080)
#define KRgbDarkCyan   TRgb(0x808000)
#define KRgbRed         TRgb(0x0000ff)
#define KRgbGreen       TRgb(0x00ff00)
#define KRgbYellow      TRgb(0x00ffff)
#define KRgbBlue        TRgb(0xff0000)
#define KRgbMagenta     TRgb(0xff00ff)
#define KRgbCyan        TRgb(0xffff00)
#define KRgbGray        TRgb(0xaaaaaa)
#define KRgbWhite       TRgb(0xffffffff)
```

We use `#define` rather than `const TRgb KRgbWhite = TRgb(0xffffffff)` because GCC 2.7.2 didn't support build-time initialization of class constants. Initialization of a `TRgb` from one of these 'constants' is no more expensive than if we used `const TRgb`. All `CGraphicsContext` color specifications for pens and brushes use `TRgb` values. The graphics device then converts these into device-dependent color values internally.

² These colors are named after the Enhanced Graphics Adapter of the IBM PC, which supported them and introduced them into character set attributes.

With measurements and fonts, you have to convert to device-dependent units (pixels and `CFonts`) before calling `CGraphicsContext` functions. The same approach could have been taken with colors but it wasn't, because the meaning of a color is less device-dependent than the size of a pixel or the bitmap for a font.

Concrete color values such as `KRgbBlack` are useful in many situations. There are also some logical color values, defined in `TLogicalColor`, which refer to colors used in the user interface scheme, such as the colors used in menus, menu highlights, toolbar buttons or window shadows. The choice of which physical colors these logical values correspond to belongs to the device OEM. The mapping is stored in a `CColorList` object, which supports:

- logical-to-RGB color mappings loaded from resource files or specified programmatically
- independent sections for the system and applications: a section is identified by a UID and a logical color by an enumerated constant
- mappings for four-grayscale and 256-color schemes: the 256-color scheme is used and looks good if the screen mode supports 16 or more colors; otherwise, the four-grayscale scheme is used.

An application can access the color list through `CEikonEnv::ColorList()`.

A key thing you have to know about a device is how many colors it supports. Actually, the number of supported colors depends not only on the device, but also on the current display mode of the device. Most devices have a preferred display mode, and some support multiple display modes: you can check the display modes supported by a window-server screen device and set your window to use a required display mode if it is supported. Some display modes consume more power than others, so the window server changes the display mode in use to the one with the minimum power requirement for any visible window.

You can create bitmaps in any display mode. When you blit them onto another bitmap or a display them in a particular mode, the bitmap data is contracted or expanded as necessary to match the display mode of the target bitmap. The display modes supported by Symbian OS are defined in the `TDisplayMode` enumeration in `gdi.h` (see Table 17.3).

The window server sets the screen's display mode to the least capable mode required by any currently visible window and supported by the hardware. So the 'preferred screen display mode' is actually implemented as a 'default window display mode'. There may be a trade-off here between higher display modes and lower power consumption. ROM bitmaps should be generated in the preferred display mode, so that no bitmap transformations are required when blitting to the screen.

Table 17.3 TDisplayMode Enumeration

Mode	Bits	Type	Comment
ENone			A null value that shouldn't be present in any initialized TDisplayMode object
EGray2	1	Grayscale	KRgbBlack and KRgbWhite; mainly used for bitmap masks
EGray4	2	Grayscale	Only KRgbBlack, KRgbDarkGray, KRgbGray and KRgbWhite; the default display mode on the Psion Series 5
EGray16	4	Grayscale	16 shades of gray; the alternate, higher-definition, display mode on the Psion Series 5
EGray256	8	Grayscale	256 shades of gray; mainly used for alpha-channel bitmaps for blending
EColor16	4	Color	Full EGA color set: all standard KRgb values; never used as a display mode on a real device
EColor256	8	Color	The Netscape color cube: represents all 216 combinations of R, G and B in multiples of 0x33, plus all remaining 40 combinations of pure R, pure G, pure B and pure gray in multiples of 0x11; never used as a display mode on a real device
EColor64K	16	Color	High color: represents 5 bits of R, 6 bits of G and 5 bits of B, so that 0xffffffff, 0xgggggggg, 0xbbbbbbbb converts to TRgb(0xffffffff00, 0xgggggg00, 0xbbbbb000), with the least significant bits of each color being dropped; commonly used on older devices
EColor16M	24	Color	Represents 8 bits each for R, G and B; never used in practise because it is very inefficient
ERgb	32	Color	A buffer of TRgb objects; not actually a valid display mode – only used for exporting data

Table 17.3 (continued)

Mode	Bits	Type	Comment
EColor4K	16	Color	Represents 4 bits each for R, G and B; 4 bits are wasted
EColor16MU	32	Color	Represents 8 bits each for R, G and B; 8 bits wasted; the most common display mode in use today
EColor16MA	32	Color	Represents 8 bits each for R, G and B and 8 bits alpha; the semi-transparent display mode

If a color is passed to a `CGraphicsContext` function that is not supported exactly on the device, then the nearest supported color is used instead. We are quite used to having real-world colors mapped down onto black-and-white or low-fidelity color. The best approach for real-world colors is to allow the `TDisplayMode` mappings to do their thing.

Finally, you may want the user to select a color, for example in a drawing program. Some GUIs supply a control for this: in UIQ, it is `CQikColorSelector`.

This nearest-color transformation is done before any other operation uses the color, including logical operations such as XOR. This can produce unexpected effects, but logical operations are in any case of dubious value on windowing systems – with the exception of XORing with `KRgbWhite`, which has its uses and always works as expected.

Some GDI definitions mention ‘palettes’; these were added to the design before support for any form of color display was implemented (in Symbian OS v5). When support for a color display mode was added, palettes were not used to optimize the shades available in a 256-color display mode. The fixed Netscape color cube set was used instead. This reduced the complexity of the API, while losing no worthwhile features for devices in this class and avoiding the oddities that were occasionally seen on Windows PCs when the palette was optimized for a foreground window. This proved to be a sensible decision as the 256-color display mode was never used in a real device.

Web browsing

Web-browsing technology is often confused on the subjects of target and size-independence. It has evolved without clear distinctions between print and on-screen graphics and without a consistent way of re-sizing. Text is

specified in HTML – which is device-independent – while pictures (Gifs, Jpegs or BMPs) are specified in pixels. The physical size of the graphics then vary depending on the resolution of the screen – they are smaller on higher resolution screens. There is no clear relationship between text sizes and graphics, which means they do not scale together on most browsers. However, there are ways around these problems and text and pictures scale together on Symbian OS browsers.

Summary

In this chapter, we have concentrated on graphics for display – how to draw and how to share the screen. More specifically, we have seen:

- the `CGraphicsContext` class and its API for drawing, getting position and bounding rectangles
- the MVC pattern, which is the natural paradigm for most Symbian OS applications; MVC updates and redraws work well with standard `RWindow` objects; for non-MVC programs, or programs whose updates are particularly complex, backed-up windows may be more useful instead
- how the work is shared between controls and windows
- the use of compound controls to simplify layout
- the difference between window-owning and lodger controls
- application- and system-initiated drawing
- techniques for flicker-free drawing and how `CONE` helps you achieve this
- the use of `ActivateGC()` and `BeginRedraw()` in functions such as `DrawNow()`.
- the `CCoeControl` functions for drawing
- how to animate, scroll and use back-up-behind and transparent windows
- use of the window server's drawing features: flicker-free redrawing and redraw storing
- drawing in device-independent ways: fonts, zooming views, printing, bitmaps and color.

18

Graphics for Interaction

In Chapter 17, we saw the way that Symbian OS uses the MVC pattern, and how to draw in the context of controls and windows. In this chapter, we cover how to use controls to enable user interaction with your programs.

In theory, it is easy. Just as controls provide a virtual `Draw()` function for drawing, they also provide two virtual functions for handling interaction: `HandlePointerEventL()` for pointer events and `OfferKeyEventL()` for key events. You simply work out the coordinates of the pointer event, or the key code for the key event, and decide what to do.

Basic interaction handling does indeed involve knowing how to use these two functions and we start the chapter by describing them. We also show that interaction is well suited to MVC: you turn key and pointer events into commands to update the model and then let the model handle redrawing. However, this description of key- and pointer-event handling is really just the tip of the interaction iceberg. There are plenty of other issues to deal with:

- key events are normally associated with a cursor location or the more general concept of focus – but not all key events go to the current focus location and some key events cause focus to be changed
- a ‘pointer down’ event away from the current focus usually causes focus to be changed; sometimes, though, it is not good to change focus – perhaps because the currently focused control is not in a valid state or because the control where the event occurred doesn’t accept focus
- focus should be indicated by some visual effect (usually a cursor or a highlight); likewise, temporary inability to take focus should be

indicated by a visual effect (usually some kind of dimming or making a control invisible)

- the rules governing focus and focus transition should be easy to explain to users – better still, they shouldn't need to be explained at all; they should also be easy to explain to programmers.

The big issues in interaction are intimately related: drawing, pointer handling, key handling, model updates, component controls, focus, validity, temporary unavailability, ease of use and ease of programming. During early development of Symbian OS, the GUI and the control environment were rewritten twice to get these issues right.

Explaining these things is only slightly easier than designing them!

18.1 Key, Pointer and Command Basics

The basic functions that deal with interaction are `HandlePointerEventL()` and `OfferKeyEventL()`. In Chapter 15, we showed how these functions were implemented in `COandXAppView`, the interactive heart of the Noughts and Crosses application.

We can handle pointer and key events in three ways (as can be seen by closer examination of the way that Noughts and Crosses works):

- we can ignore an event if it is not a key that we recognize or want to use at this time or if it is not a pointer-down event; another reason for ignoring a pointer event would be if the part of the control that was clicked on was not one that could respond to pointer events
- we can handle an event internally within the control, by, for example, moving the cursor or (if it was a numeric editor) changing the internal value and visual representation of the number being edited by the control or, for a pointer event, by drawing something at the location of the click
- we can generate some kind of command that is handled outside the control; for example, a choice-list item in a dialog may respond to an up- or down-arrow key event by saying that the value displayed in the choice list has been changed; the Noughts and Crosses application responds to a pointer event by passing a request to play in a tile to the game engine.

In some cases, a key press to a control may be interpreted as a command to the application that has a wide-ranging effect on an application's state or it might affect a control or view in a very different part of the application to itself. A way to handle such commands is to define an `M` class, a mix-in interface, that can be implemented by a central class in the application and used by every control that might want to call such a command.

18.2 User Requirements for Interaction

From the preceding discussion, it should be apparent that it is not handling pointer and key events that makes programming an interactive graphics framework complex. Rather, it is handling the relationships between different parts of the program, and the visual feedback you give in response, that can cause difficulty.

Here are some user requirements, expressed in non-technical terms. First, the user needs to be able to understand what's going on. Consider the S60 screenshots in Figure 18.1. In Figure 18.1a, the action is taking place in the menu, which is in front of everything else; the window that was previously in the foreground is faded. In Figure 18.1b, the field where the action is taking place is highlighted: it looks different to the other fields and there is a cursor (which flashes).



Figure 18.1 An S60 application showing a menu (left) and a text editor (right)

Secondly, the user should not be able to enter invalid data through the dialog. If the user presses Done, when a value in a numeric editor is outside the valid range, the dialog should complain that the value is invalid, and should not carry out the action associated with Done. The dialog should stay active, with the highlight on the offending field. In Figure 18.2, if the alarm time is set after the start time of the meeting, the highlight is returned to the alarm time field and a message is put up to tell the user what is wrong. When an invalid field value is entered, the user should be notified; for example, if 14 is entered for the hour value of the alarm, it is reset to 12. If the user presses Cancel, the dialog should close without carrying out any validation.



Figure 18.2 An S60 application showing a numeric editor

A third requirement is that the user should not be able to enter data that is inconsistent with other settings in the dialog. In Figure 18.3, if the user clicks on the Day: item the click is ignored because it makes no sense to set a day when the alarm is for the next 24 hours; this is indicated by the Day: item being shown dimmed out. If the user changes the When: item to a value where a day is required, the Day: item should automatically become active and be redrawn.

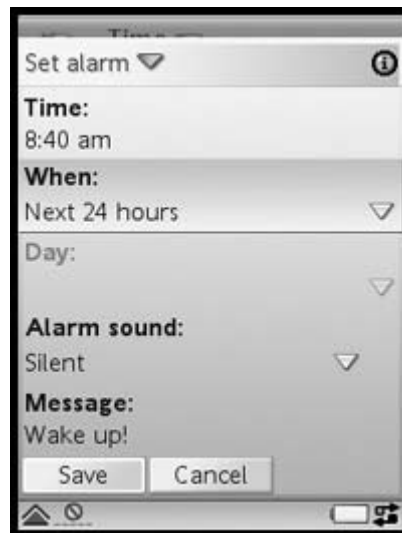


Figure 18.3 A UIQ application showing a dimmed control

18.3 Some Basic Abstractions

Based on the list of user requirements, you can see some abstractions beginning to take shape.

- The highlight (or, in a text editor, the cursor) is a visual indication of focus. The control that currently has focus receives the majority of key events; it ought to know whether it has focus or not and draw any necessary highlight accordingly. (Not all key events go to the control with focus: the hardware Confirm key causes the default button – usually Done – to be pressed, rather than going to the highlighted control.)
- A control needs to be able to refuse interactions such as pointer events. Invisible or dimmed controls should certainly refuse interactions. A control should know that it is dimmed and draw itself in a suitable way.
- A control needs to be able to say whether it is in a valid state and to respond to queries about its state.
- If a control's state changes, it needs to be able to report that to an observer, such as the dialog, so that it can handle any knock-on effects.

These are the user requirements. Throughout the rest of this chapter, we describe how the GUI framework makes it possible for you to meet them.

Programmer Requirements

If ease of use matters to end users, it certainly matters to programmers who have some requirements concerning the way these ideas should hang together.

- It should be possible to invent new dialogs with rich functionality and to implement the validation rules with sufficient ease that programmers want to use these facilities to deliver helpful and usable dialogs.
- It should be possible to use any control in such dialogs – not only the stock controls provided by Uikon, S60 or UIQ, but also any new control that you wish to include in a dialog. (Not all controls are designed to be included in dialogs: there's unlikely to be a need to include `COandXAppView` in a dialog.)
- It should be possible to write code that supports only the things you require for a particular control, without having to worry about implementing things that are unnecessary. Furthermore, you need to be confident that you have included only those things that need including and excluded only those things that need excluding.

Compound Controls

We introduced the idea of compound controls in relation to drawing. Compound controls also make it very much easier to implement general-purpose containers such as dialogs. Figure 18.3 shows a variety of controls within the same dialog:

- a title
- a help icon
- captioned control arrays, including numeric editors, text editors and drop down lists
- a button group, with two buttons (Save and Cancel).

As these controls are all together it makes it much easier for a change in one control to cause another control to be updated in response.

Key Distribution and Focus

Here is a simplified account of how a dialog processes `OfferKeyEventL()` (for more details, search for `OfferKeyEventL()` in the SDK).

At any one time, precisely one of the controls in the captioned control array has focus. That means the line is highlighted or displays a cursor and is the recipient of most key events. When the dialog is offered a key event, it handles some special cases itself (for instance, it offers `Confirm` to the dialog buttons) but otherwise it offers the key to the general-purpose control in the currently focused control.

A general-purpose control is one that can be used both in dialogs and application views. To make a control intended for dialogs usable in an application view is not always difficult and it is a good thing to aim for. To make a control intended for application views usable in a dialog is rarely necessary and you shouldn't try to do so without good reason.

There are plenty of special cases, but this description is enough to illustrate the role of focus and to begin to show why keys are offered but why they are not always consumed.

It is important to give a clear visual indication of focus and all the components work together to achieve this:

- the dialog is the topmost window: it has focus merely by being there
- the buttons and the title bar can never receive focus, so they don't need to change their drawing code in response to whether they're focused or not

- the general-purpose control designed for use in a dialog should show the cursor, if it has one, when focused and not otherwise; many editors include some kind of cursor to indicate that they have focus.

Dimming and Visibility

There are two ways to indicate that a control in a dialog cannot receive focus:

- make it invisible
- dim it.

You can make an entire line – prompt and control – invisible. The control environment has full support for this action: invisible controls are omitted from redraw and pointer handling and dialogs do not allow invisible lines to be given focus.

If you dim a control, it is omitted from pointer handling. Dialogs do not allow a dimmed line to gain focus. However, the control environment does not omit a dimmed control from redrawing; the implementer of the control has to code `Draw()` explicitly to dim the control.

The good thing about dimming is that the user can still see the control and the value it contains even though they can't change it. But dimming is a nuisance: the writer of the control has to add support in `Draw()` (using logical colors can help with this – see Chapter 17 for more information). Unavailable options might be entirely meaningless, in which case you don't want to dim the control, you want to hide it altogether.

Some dialogs in UIQ use a compromise: when you call `MakeLineVisible(EFalse)`, the dialog dims the prompt and makes the general-purpose control invisible. The user sees the right effect, the general-purpose control doesn't have to support dimming explicitly and there's no attempt to display meaningless values.

The functions related to dimming are found in `CCoeControl`:

```
class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C virtual void SetDimmed(TBool aDimmed);
    ...
    IMPORT_C TBool IsDimmed() const;
    ...
};
```

If you need to, take account of `IsDimmed()` in your `Draw()` function.

Validation

While a control has focus and the user is editing it, it may become temporarily invalid. However, when focus is taken from a control – for a

change in dialog line or because Done was pressed – it is important that the control should contain a valid value.

`PrepareForFocusLossL()` is called on any control from which focus is about to be removed. The default `CCoeControl` implementation of this function is empty: there is no need to override it for a control whose internal state can never be invalid (for example, a choice list, a button, a checkbox or a text editor). If your control could be invalid, however, you should implement this function to check the current validity of the control. If it is invalid, you should:

- issue some kind of message to inform the user that the control is invalid
- reset the control to the nearest valid value.

If resetting the value is not possible (for instance, if the user has entered text into a numeric editor), the code should leave, usually with error code `KErrNone`. This informs the dialog that the control is invalid and prevents it from changing the current line, continuing with Done button processing, etc.

You can issue a single call to `IEikonEnv->LeaveWithInfoMsg()` to display an information message and leave without displaying Uikon's standard alert dialog.

Control Observers

Compounding produces a hierarchy of controls that is strictly related to coordinates. Components are contained within their container's extent, and peer controls do not overlap. As we've seen, that greatly simplifies drawing and you can probably guess that it makes pointer-event processing easier too.

Drawing and pointer-event processing operate from top to bottom in the hierarchy, that is from the dialog down to the controls. Controls report events, such as whether their state changed. Event reporting usually goes up the hierarchy – from a control to the dialog – so that you can handle the event with the dialog's `HandleControlStateChangeL()` function.

For this reason, many systems handle a chain of events by passing them up their window-ownership hierarchy. However, making the event-reporting hierarchy the exact opposite of the compounding hierarchy turns out to be very awkward.

A key design decision in Symbian OS was to avoid fixing any association between the observer of an event and the container control. The observer does not have to be the container control or a control in the containment hierarchy; it doesn't even have to be a control at all.

If the container is the observer, why not just say ‘the container’ and have an ownership hierarchy as in some other systems? First, because controls can be used in containers which don’t need to observe them. Secondly, because the dialog isn’t actually the immediate container of, say, a choice-list control – the dialog contains a captioned control that contains the choice list. But the dialog is a direct observer of the choice list.

In Symbian OS, each control contains a member called `iObserver` that it can use to report various general-purpose events required by all controls that live in dialogs:

```
class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C void SetObserver(MCoeControlObserver* aObserver);
    IMPORT_C MCoeControlObserver* Observer() const;
    ...
protected:
    ...
    IMPORT_C void ReportEventL(MCoeControlObserver::TCoeEvent aEvent);
    ...
private:
    ...
    MCoeControlObserver* iObserver;
    ...
};
```

`MCoeControlObserver` is an interface that can be implemented by any class that wishes to observe controls. It is defined in `coecobs.h` and has just one member function:

```
class MCoeControlObserver
{
public:
    ...
    virtual void HandleControlEventL(CCoeControl* aControl,
                                     TCoeEvent aEventType) = 0;
    ...
};
```

If you’re writing a control, you call `ReportEventL()` to report events to the observer, which gives the observer an opportunity to do something about them.

The available event types are defined in `MCoeControlObserver::TCoeEvent`:

```
enum TCoeEvent
{
```

```
EEventRequestExit,  
EEventRequestCancel,  
EEventRequestFocus,  
EEventPrepareFocusTransition,  
EEventStateChanged,  
EEventInteractionRefused  
};
```

More details on these events and an example of a control observer in the Noughts and Crosses application can be seen in Chapter 15.

State changed event

If you are implementing a general-purpose control, the only useful event to report is `EEventStateChanged`. If you report this event from within a dialog, the dialog framework calls `HandleControlStateChangeL()`, which the dialog implementer can use to change the values or visibility of other controls in the dialog.

For some controls, state-change reporting is optional. Edit windows, for example, only report state changes if they're asked to – otherwise, every key press (except for navigation keys) would cause a state change.

You should certainly not report a state change until the control is in a valid state – it would be inappropriate to report a state change while the number in a numeric editor is invalid, for instance.

Other events

The `MCoeControlObserver` interface is designed for the general-purpose requirements of contained controls (reacting to changes in state) and containers (responding to refused interactions, preparing focus transitions and requesting focus events). Special-purpose events should be handled by special-purpose interfaces, such as `MEikMenuObserver` for menus.

Containers

Dialogs are just one example of a container: not every control you write needs to go into a dialog. Many controls are designed for use in your application view instead or perhaps for some other kind of container. Your container may be general purpose (such as a dialog, which can contain an arbitrary number of controls of any type) or special purpose (such as an application's view).

If your container is general-purpose, you should use design patterns similar to those used by dialogs to handle validation, focus, changes of state, dimming, etc. For navigating around the container, views and dialogs, you should use different design patterns.

In both S60 and UIQ the keypad has ways to generate direction keys (up, down, left and right) as well as a Confirm key. In dialogs on devices

that support pens, the pen is the best way to navigate between controls, so pressing the direction keys has no effect; in views, the direction keys can be used for navigation.

Most views use the left and right keys to navigate around the view or between views. For instance, a view with tabs normally uses the horizontal direction keys to move between the tabs. So, if your control is used in a view, it should only consume the direction keys if it really needs to. Then the view can consume the events instead.

When you tap on a choice list, it displays a vertical (potentially scrolling) list of the items. The pop-out is modal: it captures the pointer and consumes all keys offered to it. You have to tap outside the choice list, or select an item in the list, to pop the list back in again – then you can use the rest of the dialog or view.

Similarly, if you tap on a date control, you may get a pop-up calendar (depending on the user interface) for a whole month, which you can navigate without worrying about the dialog or view as a container. Only when you tap to select a date does the calendar pop back in again.

18.4 Processing Key Events

Now we have seen:

- what a simple control does with a key event in its `OfferKeyEventL()` member function
- the idea of focus
- how keys are handled either by the dialog or by the currently focused control
- how a control may or may not consume a key offered to it.

This puts us in a better position to understand the full picture of key distribution, as handled by the window server and the control environment together.

When a key is pressed, it is initially passed from the keyboard driver to the window server. If the key is a window server hotkey (such as `Ctrl+Alt+Shift+K`, which kills the process associated with the foreground window), then it causes the action associated with the hotkey and is not routed to any application. The window server also maintains a list of keys that applications have captured – if the key press is in this list then it is routed to that application.

The window server routes most key events to the application in the foreground, where they are detected by `CCoeEnv::RunL()` and passed on to `CCoeAppUi::HandleWSEventL()`, which identifies the event as a key event and calls `CCoeControlStack::OfferKeyL()`. The control environment's control stack is responsible for offering keys to controls that can then handle them further.

If the key event was generated by the on-screen virtual keyboard FEP, the window server is not involved – the FEP simulates a key event, passing it directly to `CCoeAppUi::HandleWSEventL()`, as though it had come from the window server. The handwriting recognition FEP is slightly different – it generates key events and passes them to the window server, which adds them to its event queue. In both cases though, the result is the same – the key event ends up being offered to controls on the control stack.

The Control Stack

In the UIQ environment, there are typically five types of control on the control stack (other user interfaces work in a similar way):

- the debug keys control: this is an invisible control that consumes all Ctrl+Alt+Shift keys (for instance, Ctrl+Alt+Shift+R causes the current screen to be completely redrawn); if the debug keys control doesn't consume the key, it gets offered further down the stack
- active FEPs that need to intercept key events: most FEPs own a control that is added to the control stack at a higher priority than any visible controls to ensure that the FEP receives first refusal of all key events; in UIQ, the virtual keyboard FEP needs to be on the control stack – when it is active, it consumes all key events, except those it generates itself; the handwriting recognition FEP does not consume any key events, so is not on the control stack
- active dialogs: the key is offered to the topmost active dialog, which consumes the key; if no dialog is active, the key is offered further down the stack
- the menu bar: in some user interfaces (not UIQ), the menu bar processes shortcut keys defined by the application; when the menu bar is invisible, it consumes only the shortcut keys and the key that causes the menu bar to be displayed; when the menu bar is visible, it consumes all keys offered to it; the menu bar is lower down the control stack than any dialog, which prevents the menu bar from being invoked when a dialog is active
- application views: an application view should usually ignore keys (and therefore not consume them) if it is invisible; otherwise it should handle and consume them.

Figure 18.4 illustrates the control stack for an application with three views (one of which is active), an active FEP and two active nested dialogs.

Key events are offered down the control stack in order of priority (highest to lowest) and then in stack order (most recently added to least

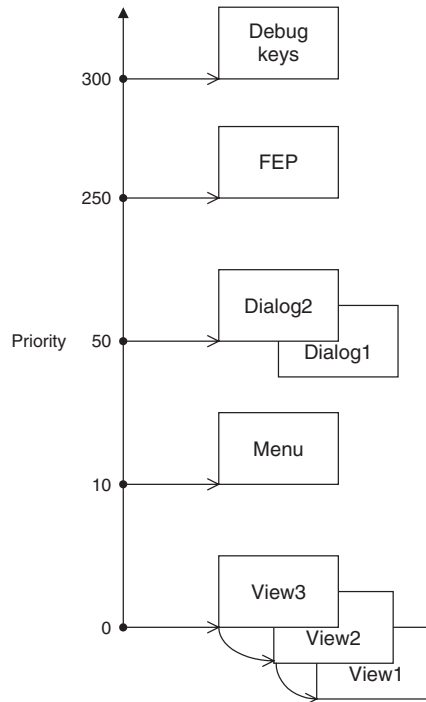


Figure 18.4 The control stack

recently added, within each priority). The priorities shown in Figure 18.4 are defined in an enumeration in `coeui.h`. Using this stack structure, Symbian or an application programmer can insert something new into the GUI environment without having to re-write the control environment or the existing GUI components.

Exactly how a control on the stack handles a key event depends on the control. Dialogs are general containers and we have already seen how they offer keys around their component controls. Application views are often also containers and they use their own logic, including at least some of the patterns used by dialogs, to offer keys to component controls. The debug keys control is a single control without any components: it either consumes a key or not.

The menu bar, when visible, includes at least two controls – the menu bar and a menu pane – and possibly more, since there may be cascaded menus. The menu bar offers and handles keys among these controls to implement conventional menu navigation and item selection.

Different types of FEP handle key events in different ways. UIQ's virtual keyboard FEP consumes most key events but does no processing on them. Other FEPs do sophisticated processing to compose an output string, which is committed to the underlying application. Predictive text input FEPs or FEPs designed to enable Far Eastern text input are examples of this.

Focus

We can see again how focus and key handling are usually – but not always – related:

- the window server sends most keys to the application with focus or to the FEP, if active – but not its own hotkeys or other captured keys
- the control stack mechanism usually results in keys being handled by a control with focus: the topmost dialog, the menu if visible or the application view; keys are always offered to the debug keys control, which is always invisible; if no dialog is showing, keys are always offered to the menu bar, even if it is invisible; in other words, keys can be handled even though they're not associated with focus
- dialogs normally channel keys to the focused line but the direction keys and the Confirm key are handled differently
- the menu bar maintains a focused pane and a focused item and normally offers keys to them; shortcut keys, if supported, are handled independently of focus.

Focus doesn't play a big part in the way that the Noughts and Crosses application handles its key events. As we have already seen, the application view passes key events that it does not process to the tile that currently has focus.

Focus is supported throughout Symbian OS graphics components. First, the window server associates focus with a window group. The window server routes keys to the application whose window group currently has focus. A window group owns all an application's windows, including application views, menus, dialogs and any others. The window server sends focus-gained and focus-lost events to applications as their window groups gain and lose focus.

In response to window server focus-changed events, most applications do nothing. There is no point in redrawing their controls to indicate that they have lost focus, since unfocused applications can't be seen on the screen anyway.

The control environment maintains the top control that can accept focus on the control stack and calls `FocusChanged()` on it to indicate when focus has changed. Similarly, container controls should also call a contained control's `FocusChanged()` function when they change the focus given to it.

A control should change its appearance depending on whether it has focus. If you need to indicate focus visually, use `IsFocused()` in your drawing code, to draw an appropriate highlight or cursor. Handle `FocusChanged()` to change between focus states (for example, to redraw, activate or deactivate the cursor). `FocusChanged()` includes

a parameter `TBool aRedrawNow` to indicate whether an immediate redraw is needed.

You can control and interrogate focus with `SetFocus()`. This does not iterate through component controls: each container handles propagation of `SetFocus()` according to its own requirements.

You can use `SetNonFocusing()` and related functions to set whether a control allows itself to be focused. This can be a permanent state for some controls (they simply don't handle input) and a temporary state for others (this is analogous to dimming).

To summarize, here are `CCoeControl`'s focus-related functions:

```
class CCoeControl : public CBase
{
public:
    ...
    IMPORT_C virtual void PrepareForFocusLossL();
    IMPORT_C virtual void PrepareForFocusGainL();
    ...
    IMPORT_C void SetFocus(TBool aFocus, TDrawNow aDrawNow = ENoDrawNow);
    ...
    IMPORT_C TBool IsFocused() const;
    ...
    IMPORT_C void SetNonFocusing();
    IMPORT_C void SetFocusing(TBool aFocusing);
    IMPORT_C TBool IsNonFocusing() const;
    ...
protected:
    ...
    IMPORT_C virtual void FocusChanged(TDrawNow aDrawNow);
    ...
};
```

The Text Cursor

Focus may be associated with a cursor. The window server provides a text cursor and there can only be one cursor on the screen at any one time. By convention, the text cursor must be in the focused control in the focused window. If your control uses the text cursor, you should be sure to implement `FocusChanged()` so that you can turn the cursor off when you lose focus and on again when you regain it.

A control that uses the text cursor does not have to use it all the time. For instance, the secret editor (for entering passwords) only displays it at the end of the password.

18.5 Processing Pointer Events

The window server ensures that a pointer event gets to the right window and the control environment framework ensures that it gets to the right control. The event can then be handled by `HandlePointerEventL()`.

Interaction Paradigms

A control should interpret the entire pointer sequence.

Press-and-release is appropriate for many types of buttons:

- pointer down inside the button: provide visual feedback that makes it obvious that the button is pressed
- pointer may stay down, and may be dragged, for an arbitrary length of time
- pointer up: if the pointer is outside the button, the button must be redrawn in its neutral (unpressed) state and its associated action is not executed; if the pointer is inside the button, the button must be redrawn and then the action associated with the button must be executed.

Other, related, sequences are possible. For instance, UIQ supports buttons that activate as soon as you press them, whose state toggles, or which expand to show a drop-down list when pressed, so that you can drag and release on one of the items. There are many alternatives.

In UIQ, a single tap to select and open is used where other systems use a double-click or two taps. Pointer down on an item selects and opens it – in `COandXTile`, ‘open’ means ‘play in this tile’; in the case of the Application launcher, ‘open’ means open the application (or, if it is already open, switch to it).

Pick Correlation

Pick correlation means associating the correct object with a pointer event. In the Noughts and Crosses application, it was done by the `CCoeControl::HandlePointerEventL()` function for the `COandXAppView` class. In this application, pick correlation is very easy to do:

- there is only one object type that can be selected: a tile
- the object is rectangular
- the object is part of a simple grid.

This means that selecting the object is a matter of simple bounds checking and division.

In more complicated cases (such as selecting text in a word processor or an object in a vector graphics package), pick correlation can be an awkward business and can involve even more optimization and complexity than is involved in drawing. However, you can use a few handy techniques to make life easier. Object orientation makes designing

for pick correlation easier, just as it does most things. You should construct a pick list optimized for easy checking when a pointer event occurs.

You can often use the same code to handle both incremental-redraw and pick-correlation requirements, since both are a matter of working out which control is at what location. The net effect is that your redraw and pick-correlation code can be optimized together. Often, optimizing feels like inventing two solutions to solve one problem. In this case, one solution solves two problems, which is nice.

Grabbing the Pointer-down Control

When you drag outside a control – for example, in the press-and-release scenario – you usually want all your pointer events, including the release, to go to the control in which the pointer went down. The framework therefore has to remember with which control the pointer-down event was associated, and to channel all subsequent drags and the pointer-up event to that control.

Symbian OS calls this ‘pointer grab’ and there are grab-related APIs in the window server and the control environment to support it: the window server has to remember the right window, while the control environment has to remember the right control. If you write a control that contains multiple press-and-release type objects, but for various reasons you don’t want to implement those objects as component controls (perhaps they are not rectangular), then you must also implement grab logic.

Capturing the Pointer

If an application launches an application-modal dialog (such as most UIQ dialogs) then, although the application view window may still be visible above the dialog window, no pointer events should be allowed into the application view. This is pointer capture: the dialog captures the pointer.

You can use `CCoeControl::SetPointerCapture()` to capture all pointer events to a control. Clearly, the window server has to be involved: displayable windows also support a pointer-capture API.

Grab and capture can seem confusing. Grab means keeping pointer events associated with the control on which the pointer-down event occurred. Capture means preventing the pointer-down being handled outside a particular control.

Getting High-resolution Pointer Events

The window server normally amalgamates drag events so that, after an application has handled the previous pointer event and asks the window

server for another event, the window server only tells the application the coordinates to which the pointer has now been dragged.



Figure 18.5 Handwriting sampled at a low rate

This is the right thing to do when handling non-time-critical MVC-type interactions. But for time-critical applications, such as handwriting recognition, it is rather awkward. The communication between the window server and the application is too slow to handle events at the rate produced by handwriting. Without any special support, the system will effectively sample handwriting at too low a rate, with very poor results (as shown in Figure 18.5).

Some Symbian OS engineers refer to this affectionately as ‘thrupenny-bitting’, after the twelve-sided three-pence coin (‘thrupenny bit’) that went out of circulation in the UK in 1971. It is enough of a challenge to write a screen device driver that doesn’t ‘thrupenny-bit’, let alone convey all this through the window server, client-server interfaces, handwriting recognition software for Western or Far Eastern alphabets, and finally to a mere application.

To avoid this problem, an application can ask the window server to buffer pointer events and deliver the whole buffer in a single large event. The window server sends an event to say that there is a buffer waiting as soon as it enters the first event into the buffer. By the time the event has been passed to the client (which is running at a lower priority than the window server) and the client asks for the buffer it is likely that many events are waiting in the buffer.

Controls handle a full pointer buffer by overriding the `HandlePointerBufferReadyL()` function, which is called when an `EEvent-PointerBufferReady` event is received. The window server has corresponding APIs.

How a Pointer Event Is Processed

We now have all the pieces of the puzzle, so let us run through the processing of a pointer event from start to finish.

Pointer events are initially generated by the digitizer driver and passed to the window server. A pointer event is usually associated with a

window whose on-screen region includes the pointer event. However, pointer grab may be used to associate the event with the window that was associated with the pointer down event or pointer capture may be used to force the event to go to a dialog.

The window is a member of a window group that corresponds to an application. The window server sends the event to the relevant application. The control environment's `CCoeEnv::RunL()` is called, which then passes the event on to `CCoeAppUi::HandleWsEventL()`, using the same process that is used for key events. `HandleWsEventL()` identifies it as a pointer event associated with a particular window.

The control environment finds the window-owning control associated with that window and calls its (non-virtual) `ProcessPointerEventL()` function, which calls `HandlePointerEventL()` to handle the event. It also does some pre-processing, which we explain shortly.

`HandlePointerEventL()` is virtual: when implementing a simple control, you implement it to handle a pointer event however you wish. If you do not implement this function, you get the default implementation (in the `CCoeControl` class), which searches through all the visible, non-window-owning, component controls to find the one which includes the event coordinates and then calls `ProcessPointerEventL()` on that control. This default implementation is good for compound controls and you override it at your peril.

So, the pointer event is ultimately channeled to the correct non-compound control, where you can handle it by overriding `HandlePointerEventL()`. Note, however, that the complications we have already seen in this section affect both `ProcessPointerEventL()` and `HandlePointerEventL()`:

- the control environment's pointer-processing functions support pointer grab, so that once a pointer is grabbed, all subsequent events until the pointer-up event are channeled to the same control
- the control environment doesn't support pointer capture: you have to use windows (and window-owning controls) for that.

`ProcessPointerEventL()` implements the event reporting in the container needed for focus transfer between components: it generates interaction-refused events for dimmed controls, a prepare-focus-transition event for a non-focused control on pointer-down, and a request-focus event, after `HandlePointerEventL()` has been called, for controls that were not refused focus at prepare-focus-transition time.

If you use the pointer buffer to capture high-resolution pointer-event sequences, the control environment handles it with `ProcessPointerBufferReadyL()` and you have to handle it with `HandlePointerBufferReadyL()`.

The window server also provides support for customizing the sounds made when the screen is tapped or a key is pressed. If the phone allows it (the manufacturer can prevent you from doing this), you can change the sounds by creating a DLL that implements the `CClickMaker` interface. You can then load and enable the DLL using the `RSoundPlugIn` class. When your DLL is enabled, its implementations of `KeyEvent()` and `PointerEvent()` are called whenever key or pointer events occur. The function that tells you whether you can unload the phone's current DLL and load your own is:

```
TBool RSoundPlugIn::IsLoaded(TBool& aIsChangeable);
```

The return value tells you if one is loaded and the parameter returns whether you are allowed to change it. For more information on the classes and functions involved, see the SDK.

18.6 Window Server and Control Environment APIs

Now is a good time to review the classes provided by the window server and control environment and how they affect the application framework.

Communication Between an Application and the Window Server

In every Symbian OS GUI application, four main system-provided framework classes are used to ensure that the application can communicate properly with the window server (see Figure 18.6).

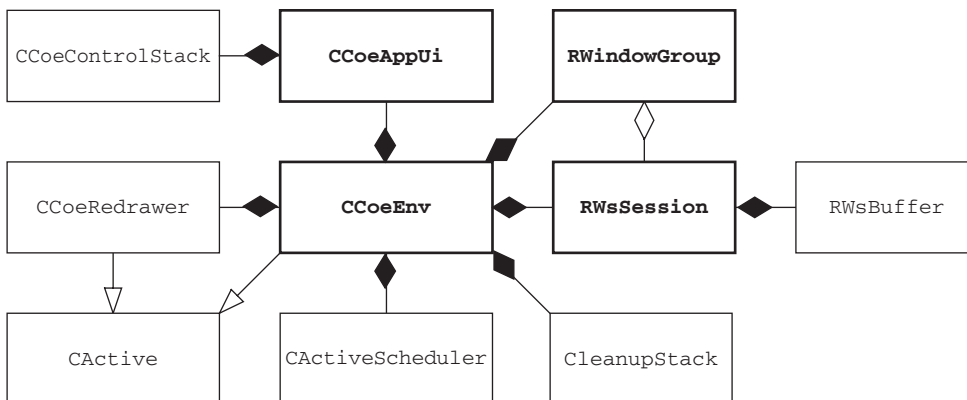


Figure 18.6 Framework classes

- `RWsSession` (defined in `w32std.h`): the window server session provides a client-server session from the application to the window

server. All window server classes (window groups and windows) use this session for communication with the window server. The session also owns the application's client-side buffer, in which drawing and window manipulation commands are batched up before being sent to the server for execution.

- `RWindowGroup` (defined in `w32std.h`): the window group is the client-side version of the window at the top of the application's window hierarchy. A window group is associated with keyboard focus.
- `CCoeEnv` (defined in `coemain.h`): the control environment base class encapsulates the window server session in active objects whose `RunL()` member functions are invoked when events are received from the window server. The `RunL()` functions analyze the events and call framework functions to handle them. The control environment sets up a cleanup stack for graphics programs. It also contains many useful utility functions.
- `CCoeAppUi` (defined in `coeau1.h`): the app UI base class handles the control stack for first-level, key-event distribution. The app UI also performs some other, more incidental functions.

The control environment provides some pre-requisites for cleanup handling (a cleanup stack) and event handling (an active scheduler). It wraps the window server session API in two active objects, to handle events generated by the window server. As we saw in Chapter 6, all event handling in a GUI program – key, pointer, redraw and others – takes place under an active object `RunL()`.

A higher-priority active object deals with user-initiated events and a lower-priority active object deals with redraw events. If both a user-input event and a redraw event occur while the previous user-input event is being handled, the framework handles the user-input event first. Otherwise, the use of two active objects makes no difference at all to the application programmer.

There's some interesting design history here. Earlier in the development of Symbian OS, only a single active object was used. It seemed natural, then, to derive `CCoeEnv` directly from `CActive`. But then the redraw event stream was separated from the user-input event stream, creating a second active object. The result is that `CCoeEnv` is-a `CActive` (for user events) and has-a `CActive` (for redraw events). If it had been designed from scratch, `CCoeEnv` would surely have two active objects. This is a good case study for two of the guidelines that are expressed elsewhere in this book: don't use inheritance where ownership will do, and hide active objects from your interfaces.

Types of Window

In Chapter 17, we concentrated on the drawing-related interactions between CCoeControl and RWindow (and, occasionally, RBackedUpWindow). In this chapter, we’ve introduced key-event processing, which brings in RWindowGroup and CCoeAppUi. The window server classes involved here are part of a small window class hierarchy, defined in w32std.h (illustrated in Figure 18.7 and described in Table 18.1).

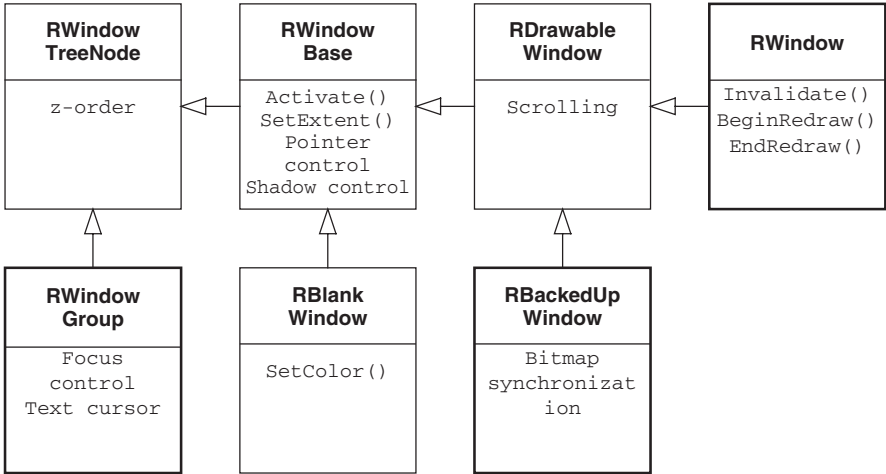


Figure 18.7 Window class hierarchy

For much of the time you can call the functions of these classes through the control environment. Nevertheless, it is useful to understand them because the control environment is not designed to encapsulate the window server. Rather, the control environment provides a convenience layer for lodger controls and compound controls, and for the window

Table 18.1 Window Classes

Class Name	Description
RWindowTreeNode	Base class for all windows: a node in the tree that defines z-order
RWindowGroup	Unit of keyboard focus and top-level owner of displayable windows
RWindowBase	Base class for all displayable windows
RBlankWindow	Entirely blank window
RDrawableWindow	Base class that defines windows which support drawing
RBackedUpWindow	Backed-up window: window server redraws invalid areas
RWindow	Standard window: application redraws invalid areas

server's major functions such as drawing and pointer- and key-event handling.

We do not provide detailed information on these facilities: but we give enough of an overview that you can understand what is available and find the information you need in the SDK.

For most application programming, the most important concrete classes are `RWindow` and `RWindowGroup`. Because all displayable windows are ultimately owned by a window group, a window group is the top-level node in the tree that defines z-order. This means that all windows belonging to an application move back and forth in the z-order as a group. We can therefore use the terms 'foreground application' and 'application whose window group has focus' interchangeably.

The window server allows applications to have more than one window group but the control environment supports only a single window group per application. This assumption is built into other Symbian OS components as well.

The window server provides other features that aren't supported by the control environment, such as blank windows and non-rectangular windows whose shape is defined by a region.

Standard Windows

A standard window has functionality inherited from the chain right up to `RWindowTreeNode`. However, the interesting functionality starts with `RWindowBase`, the base class for all displayable windows, which includes:

- the `Activate()` function
- position- and size-setting functions
- pointer control functions
- shadow control functions
- backed-up behind functions.

You can use the `Activate()` function as the final step in a three-phase construction:

1. Use the constructor to create an `RWindow` with an `RWsSession` object. It is then just an empty client-side handle.
2. Use `Construct()` to connect the `RWindow` to the window server using the `RWsSession` object it already has.

3. After you have set all the window parameters, use `Activate()` to display the window and enable it to receive events. In the case of an `RWindow`, the whole window is normally invalid immediately after you activate it, so (unless you proceed immediately to redraw it) you get a redraw event. If you are using redraw storing, you could draw it before it is activated and the window server will draw it as soon as it becomes visible.

These functions can only be associated with a window that has a visible extent on the screen, so that is why they are introduced with `RWindowBase`. `RWindowBase` serves as a base class for blank windows and also for `RDrawableWindow` objects. If you can draw a window, you can also scroll its contents, so scrolling functions are introduced here.

We have already seen that the most important functions are introduced in `RWindow`: `Invalidate()`, `BeginRedraw()` and `EndRedraw()`. However, there are a few other interesting functions too:

- `Construct()` requires you to pass an `RWindowTreeNode`, to serve as this window's parent, and a 32-bit handle; the parent can be another displayable window or a window group; all events relating to the window include the handle; the control environment passes the address of the control that owns this window; when the control environment fields an event, it simply casts the handle to the address in order to associate it with the correct control
- easier-to-use variants of `SetSize()` and `SetExtent()` than those provided by `RWindowBase`
- two variants of `SetBackgroundColor()`
- `GetInvalidRegion()` allows you to get the exact region that is invalid.

The only type of window for which a client doesn't need to specify a parent is a window group. All window groups from all applications sit at the same level in the window tree. The window server holds them in a linked list which gives them order, their z-order. The order is dynamic – it changes as applications are moved to the foreground or background. For each application, its window group serves as its top-level window.

In the past, it would have been possible to use the invalid region to highly optimize the drawing code of an application, so it would only draw the exact set of pixels that contained the wrong content. If the window is using redraw storing, such optimization works against the benefits that redraw storing provides. Thus from Symbian OS v8 onward it is best not to do this.

Fading is used in Symbian OS to change a window's colors so that other windows stand out. It is implemented by re-mapping color values

to a more limited range and, optionally, making them lighter or darker. For example, in UIQ, when a dialog is displayed, the window that was previously in the foreground is redrawn faded and darkened; in S60 windows that are drawn faded are lighter.

In UIQ, the default fading values are set to zero for black and 190 for white (unfaded windows have values of 0 and 255), although you can override these defaults through the window itself (the functions to do this are defined in `RWindowTreeNode`) or through the graphics context used to draw it.

Window Groups

The primary role of the `RWindowGroup` class is to handle focus and key handling. Window groups handle focus because they are the top-level nodes in the z-order. The only window group that can reasonably grant focus is the foreground group.

Focus-related functions enable you to say that a window group can't take focus or that it should automatically get focus (and foreground) when a pointer event occurs. The window server supports a flashing text cursor, whose window, position, shape, etc. can be controlled through the window group. This is clearly the right place to do it, since the cursor is associated with focus.

One implication of this is that there can only be one text cursor per application. An application such as the UIQ agenda displays a flashing cursor in its detail view when the view has focus. But when a dialog is displayed, the view loses focus and it must stop the cursor flashing. The same is true when adding a meeting in the S60 Calendar and the menu is brought up. Moreover, the view should relinquish control of the cursor, so that it can be used for editing any text fields that might appear in the dialog.

`RWindowGroup` also allows you to configure the way in which key events are generated when a key is held down for a certain length of time. For instance, on some phone keypads, holding down an alphanumeric key rather than releasing it immediately can cause the generated key event to switch between alphabetic and numeric; it may also be sent to a different application from the one currently in the foreground.

The default behavior is for a standard key event to occur on key down; after the key has been held down for a short time interval (usually a fraction of a second), it repeats automatically. You can override this using the `CaptureLongKey()` function of the `RWindowGroup`. You can customize several things. First, when the user presses the key but releases it before the required time interval, the standard key event can occur when the key is pressed or when it is released. Secondly, if the key is held down long enough, a different key event can be generated from the one that was captured, and this may or may not repeat automatically.

The event can also be sent to a different window group from the one with focus. For instance, if the key event was changed from numeric to alphabetic, you might want to send it to the address book, rather than the phone number display.

Summary

In this chapter, we have covered the control and window server framework that supports GUI interaction and shown how you should use this framework. We have seen:

- key and pointer events
- how events are turned into commands
- how to handle focus, using window groups
- how the control stack for key-event handling routes events to various destinations
- dimming controls and validation
- using observers to preserve the MVC pattern
- the difference between pointer-grab and pointer-capture
- the relationship between the control environment and the window server.

19

Plug-ins and Extensibility

This chapter, together with Chapters 20, 21 and 22, takes a look at some of the system services provided by Symbian OS. System services is a collective name for the range of libraries and servers that are included with the Symbian OS in order to make developing applications easier.

We've already seen some libraries and services, such as the File Server (Chapter 7), the Application Architecture (Chapter 11) and BAFL (Chapter 13). For our purposes, extensibility is the ability to extend a program after it has been deployed and it is an important characteristic since it allows third parties to add value to software. After a general overview of extensibility, we take a quick look at the extensibility of some system services and provide an in-depth treatment of ECOM, a system service developed with the express purpose of enabling application extensibility.

19.1 System Services

Communications and Messaging Services

Since Symbian OS targets mobile phones that are heavily communications-enabled, Chapter 20 is devoted to communications and messaging services: it covers serial communications (RS232), Bluetooth, infrared, SMS, MMS and email.

You are introduced to the Serial Comms Server, also known as the Comms Server, or just C32. The main API classes are `RCommServ` and `RComm`; although designed to access RS232 serial ports, C32 also

implements emulation layers that allow you to use infrared¹ and outgoing Bluetooth connections as serial ports though the same APIs.

Something similar happens with the Sockets Client API, also called Sockets Server or ESOCK.² It provides an API that is similar (allowing for the usual Symbian OS idiosyncrasies) to the Berkeley Sockets API³ used on TCP/IP networks. The main API class is `RSocket` with others, such as `RHostResolver`, implementing additional functionality. As with the main C32 classes, `RSocket` is a class that allows us to access infrared and Bluetooth connections through the same interface.

On the messaging front, you have to become familiar with the Messaging Architecture. To quote the Developer Library, 'The full messaging architecture is of some complexity. Developers who need only to add message sending capabilities to their application are recommended first to experiment with Send As, which provides this ability.' Chapter 20 discusses the `SendAs` server and its main API class `RSendAs`.

Multimedia Services

Chapter 21 introduces us to the world of multimedia services:

- the Multimedia Framework, with support for audio (playing, recording and conversion), audio streaming, tone playing, and video playing and recording
- the Image Conversion Library, with support for encoding and decoding images in various formats (Jpeg, Exif, MNG and more)
- the Camera API
- the Tuner API which allows things such as playback and recording of audio from the tuner and station-scanning functionality.

The design of Multimedia Services makes heavy use of ECOM as an extensibility mechanism.

RDBMS

Symbian OS has had a good database implementation for a long time. It is called DBMS. Although very successful (it is used aggressively by Contacts, COMMDB and an unspecified number of third-party applications),

¹ The Symbian documentation on infrared contains frequent references to Infrared Data Association (IrDA, see www.irda.org/), and most of the time actually refers to the Symbian component that implements the IrDA specifications.

² The initial E in ESOCK is a reminder that Symbian OS used to be called EPOC in a previous incarnation (EPOC Socket Server, hence ESOCK).

³ See http://en.wikipedia.org/wiki/Berkeley_sockets.

it does have some limitations. In particular, it implements only a subset of SQL (specifically, it does not support joins between tables) and the performance in certain common scenarios is not ideal. The performance issues are mostly due to historical reasons: DBMS was designed for the hardware of the Psion devices and current phones have very different storage characteristics.

Rather than heavily modify DBMS, Symbian is introducing a new database system called RDBMS in parallel with the old DBMS. Chapter 22 talks about the new RDBMS.

Task Scheduler

The Task Scheduler, also called the Schedule Server or SCHSVR, offers an API (through the `RScheduler` class) that provides similar functionality to the Unix CRON command or the Windows AT command: it schedules programs to be run at specific times. To schedule a program, you need to:

1. Register the EXE that you want to run with the Task Scheduler. Create a schedule that indicates when the program should be run. You can indicate that you want the program to run repeatedly (e.g., every X hours, days, months or years) and you can also specify a certain time interval called the 'validity period' to restrict when the program can run.
2. Add one or more tasks to the schedule (possibly including some task-specific data, such as the contents of a message to send). Tasks can specify a limit to the numbers of times that they run (for example, run according to the schedule but only five times) and the priority of this task compared to other tasks from the same client.
3. Wait until the Task Scheduler calls your program and use `CScheduledTask` to access the schedule and task data so your program knows what to do.

Task Scheduler distinguishes between persistent and transient schedules and provides several handy overloads to make scheduling easier.

EZLIB

EZLIB stands for EPOC zlib. It is a straightforward port to Symbian OS of the free, PKZIP-compatible, zlib compression library.⁴ zlib supports the ZIP file format and the underlying compression and decompression algorithms. EZLIB is a DLL and can be used to read anything that uses the ZIP file format and compression algorithms. Java JAR files and OpenOffice

⁴ See the official zlib website at <http://zlib.net>. zlib was written by the same developers who wrote the Info-ZIP compressor-archiver utilities (www.info-zip.org) and the formats are now an RFC (RFC1950, http://zlib.net/zlib_docs.html).

documents are good examples of file formats that are really ZIP files in disguise. Take a look at the `CZipFile` class to get started.

XML Framework

Symbian OS includes a full-blown XML and WBXML⁵ parser – look for classes `CParser` and `MContentHandler`. The API is loosely based on the SAX API⁶ and the default XML parser that Symbian ships is based on James Clark's free Expat XML Parser Toolkit⁷ (although phone manufacturers might replace it with their own).

You have to register with the parser an object whose class inherits from `MContentHandler`. Once parsing starts through a call to one of the `ParseL()` functions, the parser calls back functions such as `OnStartElementL()`, `OnEndElementL()` and `OnContentL()` at appropriate times.

One thing that helps to make XML-processing programs faster, but adds slightly to the learning curve, is that all the string-related functions make use of the BAFL StringPool API, rather than using 'normal' Symbian descriptors. The StringPool (see class `RString`) allows you to compare strings in a way that is much faster than comparing descriptors. Since a lot of the work in an XML-processing application involves comparing the parsed information with known strings (such as `<p>` or `<blockquote>` in an XHTML document), using the StringPool speeds things up.

Further References on System Services

You can find more information about the system services in an SDK. Open the Symbian Developer Library (devlib) and select the Symbian OS guide, then look under the headings indicated below.

For information about Communications and Messaging services, look at:

- >> Comms infrastructure >> Using Sockets Server (ESOCK) >> Using Sockets Client
- >> Bluetooth >> Using Bluetooth Sockets
- >> Using HTTP Client >> A simple HTTP Client session
- >> Networking >> Using TCP/IP (INSOCK) >> Introduction to TCP/IP >> TCP/IP client programs
- >> Messaging >> Using Messaging Framework >> Message Client Applications >> Send-As Messaging

⁵ See www.w3.org/TR/wbxml

⁶ See www.saxproject.org/apidoc

⁷ See www.jclark.com/xml/expat.html

- » Messaging » Using Messaging Framework » Introduction to the Messaging Architecture.

You can find additional information about Multimedia services in the Symbian Developer Library at » Symbian OS Reference » C++ Component Reference » Multimedia TUNER.

For information about RDBMS: » System libraries » Using DBMS » DBMS rowsets » DBMS SQL.

19.2 What Is a Plug-in?

A plug-in⁸ is a way to extend the functionality of an application, normally through the use of a special type of DLL. Symbian OS uses plug-in DLLs extensively and also provides ECOM, a special library and supporting server that makes it easy to write extensible applications. Plug-ins work particularly well when you have a clear interface that has many different possible implementations; they allow you to ship a range of implementations separately from the main application. They are also invaluable in cases where third parties need to extend your application.

- An imaging application may have plug-ins to handle the different image formats (JPEG, TIFF, PNG, GIF, etc.). The application selects the right plug-in after looking at the file extension or the MIME type.
- A video player may have plug-ins to handle Mpeg, Quicktime, etc.
- A cryptographic library may select the right plug-in depending on the type of operation (symmetric encryption, cryptographic hash, etc.) and the specific algorithm selected by the user (AES, Serpent, SHA, MD5, etc.)
- Drivers are good candidates for plug-ins since they have to conform to a specific interface with strict guarantees with respect to performance, resource usage, etc. and there is an endless variety of hardware devices (printers, pointer devices, keyboards, etc.).

In general, a plug-in is a good option if the performance requirements of the extension mean that it should be written in C++. For operations, such as syntax-highlighting, that are 'interactive' rather than real-time or extensions that are mostly data-driven, an extension language might be a more flexible solution.

Before we go on to describe how plug-ins are used in Symbian OS, we present an example of a design for an Instant Messaging application. We then try to make it possible to extend this application with third-party

⁸ See <http://en.wikipedia.org/wiki/Plugin> for a good, long definition

plug-ins using only the basic Symbian OS APIs. The purpose of this exercise is to give a better understanding of what type of problems you can solve with plug-ins and what goes on behind the scenes when we use a plug-in library, in general, and ECOM, in particular.

If you are already familiar with how plug-ins work and why they are useful, feel free to skip to Section 19.3.

How the Instant Messaging Client Application Works

When the application starts, it asks us to log in to the particular instant messaging (IM) server that we are using. It is a simple user interface that shows the names of our friends and allows a Send Message option when we click on one of the names. Imagine we've implemented a prototype version that uses the Jabber protocol⁹ since a lot of other instant messaging systems use Jabber internally (e.g., Google Chat). But we already know that we want to extend it to include support for AIM, ICQ, IRC, MSN, Yahoo!, Skype, Lotus Sametime and others. The only problem is time!

We could release a Beta version that only supports Jabber and then ask users to upgrade, but we are not sure they will want to do that. We would prefer to release a basic version and then let third-party developers extend our application with support for more protocol types. How can we do that?

Let us look at a simple class design first. We separate the IM-specific classes from the rest of the GUI classes so that we can test them independently before the GUI designers have finished. We are not sure how to implement the IM code yet, but this is how we imagine somebody could use the classes:

```
// IM servers expect you to log in to the service so that other users
// know you are there and the server knows where to send the response
// messages, etc.
User::LeaveIfError(jabber->Login("me@gmail.com", "mememe"));

// Now add our list of friends
jabber->AddFriendL("Joe@gmail.com", "Joe");
jabber->AddFriendL("Mike@aol.com", "Mike");

// Now send messages
jabber->SendMessageL("Joe@gmail.com", "Hi Joe, I'm finally online!");
```

Polymorphism

Because we know that it will be important to extend the application with new protocols, we've encapsulated the protocol functionality behind a `CImProtocol` abstract class that uses pure virtual functions, so that we can extend the application easily without modifying the rest of the code:

⁹ See <http://en.wikipedia.org/wiki/Jabber>

```

class CImProtocol : public CBase
{
public:
    static CImProtocol* NewL(const TDesC& aProtocolName);
    virtual void HBufC* GetNameL() = 0;
    virtual void LoginL(const TDesC& aUserId,
        const TDesC& aNickname, const TDesC& aPassword) = 0;
    virtual void AddFriendL(const TDesC& aUserId,
        const TDesC& aNickname) = 0;
    virtual void SendMessageL(const TDesC& aUserId,
        const TDesC& aMessage) = 0;
    ...
};

```

The rest of the application does not know the details of the different protocols and just uses pointers to `CImProtocol`. For each protocol that we support, we need a concrete class that inherits from `CImProtocol` and implements the needed functionality:

```

class CJabberProtocol : public CImProtocol
{
public:
    void HBufC* GetNameL();
    void LoginL(const TDesC& aUserId, const TDesC& aNickname,
        const TDesC& aPassword);
    void AddFriendL(const TDesC& aUserId, const TDesC& aNickname);
    void SendMessageL(const TDesC& aUserId, const TDesC& aMessage);
};

```

The ability for objects of different types to implement the same interface defined in the base class is called ‘polymorphism’. If we want to implement IRC, we have to implement `CIrcProtocol`.

The `CImProtocol` base class also has a `NewL()` function. The mission of this class is to know what protocol implementations are available and create the right implementation when the client requests it (i.e. `CImProtocol::NewL('Jabber')` should create a new instance of `CJabberProtocol` and return a pointer to it). This works if `CImProtocol::NewL()` knows the names of all the implementation classes at compile time. We would like to allow third parties to add protocol implementations without us having to re-deploy this application. People need to be able to add new protocol implementation classes without us having to recompile the application and make the new version available for download. We need to find a way to split the functionality into separate binaries so that each of them can be delivered separately. After that, we need some functionality in our application to pull all those protocol implementations out of their respective binaries so that our main application can use it.

Packaging Binaries

We are back to the original problem that we wanted to solve in our application: to be able to release a basic version and then let third-party developers extend our application with support for more protocol types without having to modify the main application in any way. It is clear that our initial design (see Figure 19.1), where everything is part of one big EXE file, does not allow us to do this.

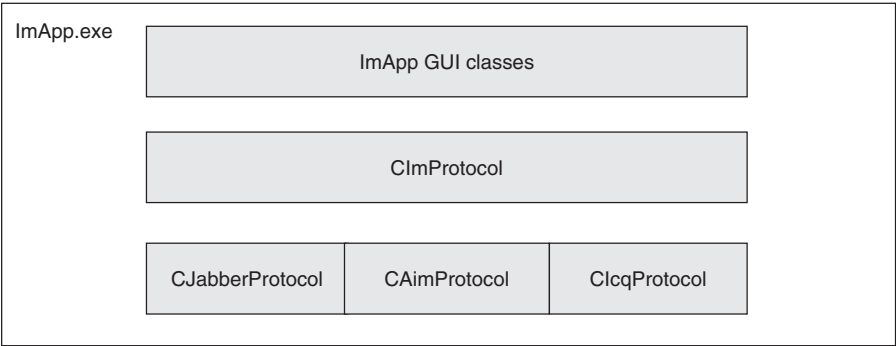


Figure 19.1 Monolithic application

We split the application into several separate binaries to see if this helps. First, we move the IM-related classes into their own DLL (Figure 19.2). This makes sense and makes it somewhat easier since upgrades and extensions would now only need to replace this file rather than the whole EXE.

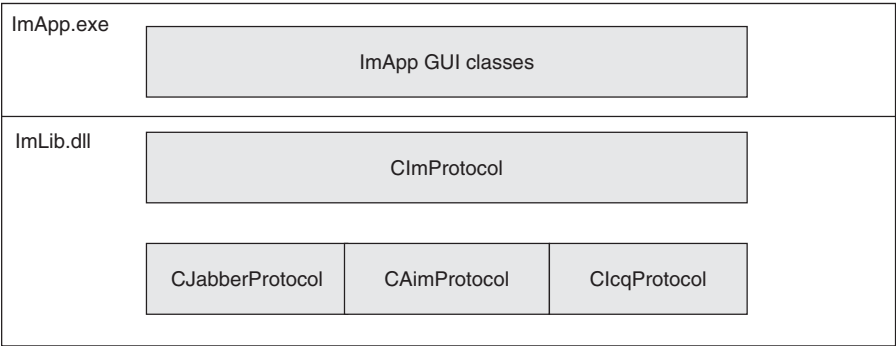


Figure 19.2 Separate IM library DLL

Then we move each protocol class into its own DLL (see Figure 19.3); now all the Jabber protocol functionality is in `jabber.dll`, etc. This is awkward but could work.

The initial version of our library would ship with a protocol DLL for each known protocol. If we have not had time to implement the protocol, we just provide a ‘stub’ DLL that returns `KErrNotSupported` when you

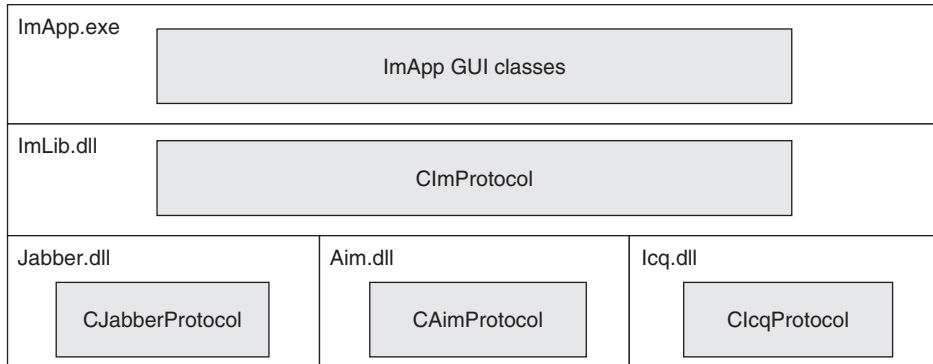


Figure 19.3 Separate IM library and a DLL for each protocol

call any of the provided functions. If a third-party developer implements it properly, they can provide a SIS file that overwrites the original stub DLL with their application.

This approach falls short on several counts:

- we still need to know the names of all protocols in advance: the MMP file of our application (or at least the `IMLib.dll`) links against all the DLLs that handle protocols
- third parties cannot easily provide competing implementations of the same protocol since there is only one DLL per protocol and there is no way to have two installed at the same time or to select one rather than the other
- Platform Security is not particularly fond of random third parties overriding each others' DLLs and the approach might prove complicated if we wanted to get our application signed.

Thinking about it a little more, it is clear that the approach of linking to each DLL by name in the MMP file is not flexible enough for our problem. Luckily, Symbian OS allows you to load a DLL statically or dynamically and dynamic loading might be just what we need.

Loading Polymorphic DLLs

There are two ways to load a DLL: statically and dynamically. The most common scenario is loading a DLL statically (this is what happens when you specify the name of a LIB file in your MMP file).

Dynamically loaded DLLs work in a different way.¹⁰ After loading the DLL using `RLibrary::Load()`, you use another E32 function,

¹⁰ See the Symbian Developer Library's Symbian OS guide, at » Base » Using User Library (E32) » Dynamically Loading Link Libraries

`RLibrary::Lookup()`, to access a function pointer exported by the DLL. You then use this pointer to access the functionality present in the DLL. In practice, the code looks like this:

```
RLibrary library;
User::LeaveIfError(library.Load(aDLLName));
// The provider of the DLL has set things up so that the first DLL
// entry point is the NewL() function of some object implemented
// in the DLL
TLibraryFunction func = library.Lookup(1);

// func() is equivalent to calling CDllStuff::NewL()
CDllStuff* stuff = (CDllStuff*) func();
stuff->DoWhatever();

// Bye
Cleanup(stuff);
library.close();
```

Most Symbian OS literature refers to this type of DLL as a ‘polymorphic interface DLL’ since it is an extension of the normal polymorphism we described before: each DLL provides a different implementation to the same abstract interface. Polymorphic interface DLLs are used extensively in Symbian OS (see Section 19.4).

Managing Plug-in DLLs

However, we have not finished with our IM application yet. We want several of these polymorphic DLLs (most people just call them plug-in DLLs): in particular, we want one for each new protocol implemented by a third party. And then we need a way for the IM library to find out how many of these plug-in DLLs are available at run-time and which protocol each of them implements.

The easiest approach would be to use a special extension for plug-in DLLs or some other file-naming convention and then use File Server APIs to get a list of the DLL names, which we can load using `RLibrary::Load()`. This would even work if we wanted to provide more than one implementation for each protocol (e.g., `jabber-1.dll` and `jabber-2.dll`, see Figure 19.4).

Several plug-in libraries used this approach before Symbian OS v9 but Platform Security necessitates a change: retrieving a list of available binaries from `\sys\bin\` is considered a dangerous operation and is protected by strong capabilities.

We could provide a text file with the name of the plug-in DLL. Then we can ask for a directory listing of the text files (which is not protected) and load each individual DLL by name.

Even with this improvement, our home-made plug-in system still has limitations:

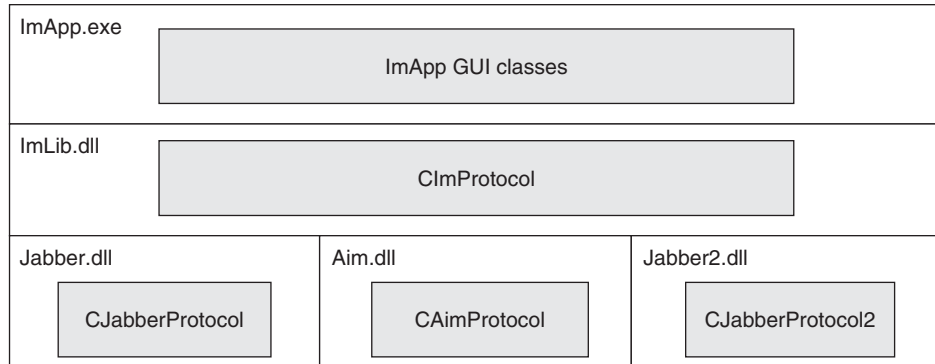


Figure 19.4 Plug-in DLLs with multiple implementations of the same interface

- you cannot have more than one protocol implemented in the same DLL
- you cannot choose between alternative protocol implementations.

We could achieve this by extending the format of the text file in various ways, but luckily we do not have to do any of this as Symbian OS provides ECOM, which allows us to do it quite easily. So let us stop toiling to get the IM application working using the manual approach and instead switch to ECOM, with a good appreciation of what ECOM can do for us.

19.3 The ECOM Library

ECOM is a Symbian OS library, supported by its own server, that makes it easier to write plug-in systems such as the instant messaging library that we have presented as an example. The main hurdle to learning how to use it is just becoming familiar with the terminology and some of the operations that happen behind the scenes but the APIs are actually quite simple.

There are three types of user of ECOM:

- people using an ECOM-based plug-in system (in our example, the authors of the GUI part of the IM application)
- people creating a plug-in system on top of ECOM (in our example, the designers of CImProtocol)
- people creating plug-ins for a custom plug-in system (in our example, third parties extending the IM application, such as the implementers of jabber.dll).

Using ECOM Plug-ins

In theory, the fact that a library uses ECOM could be totally hidden from the library user – this was one of the design goals behind ECOM. In practice, library users frequently need to know about ECOM. This is partly due to some technical intricacies that we discuss later (see the section on `REComSession::FinalClose()`) but also because a lot of library designers have decided that hiding ECOM behind an abstraction layer is not worth the effort, since you risk introducing defects with some code that might not add much value in itself. If you want to see this for yourself, just go into the `\epoc32\include` directory of whatever SDK you are using and search for `DestroyedImplementation` (one of the key ECOM functions). This allows us to take a peek behind the scenes and notice that quite a lot of ECOM users just provide a thin inline wrapper around ECOM.

We saw in the instant messaging example that we could give the IM library the name of the instant messaging protocol that the implementation should support. The library transparently instantiates the correct object (`CJabberProtocol` in this case) from the correct DLL. That is the main service that we can expect from a plug-in-based library, and ECOM makes it easy.

ECOM distinguishes between classes used as *interfaces* (`CImProtocol` in the example) and classes used as *implementations* (`CJabberProtocol`). Since interfaces and implementations are going to be provided by various third parties it makes sense to come up with some kind of naming scheme that allows us to identify them uniquely and also avoids collisions between them. As we know by now, the favorite way to solve naming collisions in Symbian OS is to use UIDs.

Each interface class and each implementation class is identified by a UID. To create an object of a specific implementation class, you just tell ECOM the UID of the interface and some library-specific data (the name of the IM protocol, in our case) that ECOM can use to select the right implementation out of all those that are available. ECOM calls the process of selecting one implementation *resolving*. Most plug-in based libraries hide this step behind the API. Alternatively, you can obtain from the interface UID a list of all the implementations that implement that interface, and from that choose the right one for creating the object.

One important thing to keep in mind is that, with ECOM, the relationship between a specific interface class (such as `CImProtocol`) and its interface UID is mostly implicit, that is, you have to know it. Unless you look at the inline code in some library header file to find out, you just have to trust the documentation for the plug-in library. Some document or header files tell you the name of the class and the corresponding UID. There is no central tool, database or Symbian Developer Library section that lists all available plug-in interfaces with their header file, class name and interface UID.

Designing Plug-in Interface Classes

Our IM example made a big simplification: we skipped over the part of the program that would have to deal with the loading and the unloading of the plug-in DLLs. It also assumed that each DLL would only contain one implementation and that only one instance of that implementation class would be instantiated. ECOM makes no such assumptions. It allows several implementations of the same interface to co-exist in the same DLL with any number of implementations of any other interfaces. To enable this flexibility and to be able to manage DLL loading and unloading, ECOM uses reference counting. This means that implementation objects cannot be created and destroyed directly, but only through a pair of ECOM functions:¹¹

- REComSession::CreateImplementationL()
- REComSession::DestroyedImplementation().

This is normally hidden from library users. ECOM-based objects are generally provided with a small header file that contains mostly inline functions. It defines the interface classes exported from the plug-in DLL and uses the REComSession static functions to keep ECOM informed about what is happening.

This is how it would look in our instant messaging application:

```
class CImProtocol : public CBase
{
public:
    // Given the protocol name, returns a pointer to a new object
    // of the right implementation class
    CImProtocol* NewL(const TDesC8& aMatchString)
    {
        // Use the resolver to do the dirty work
        TEComResolverParams resolverParams;
        resolverParams.SetDataType(aMatchString);
        resolverParams.SetWildcardMatch(EFalse);

        TAny* i = REComSession::CreateImplementationL(KProtocolInterfaceUid,
            _FOFF(CImProtocol, iDtor_ID_Key), resolverParams);
        return REINTERPRET_CAST(CImProtocol*, i);
    }

    // plug-in object needs to be destroyed through ECOM
    virtual ~CImProtocol()
    {

```

¹¹ This is another ECOM design decision: it would have been possible to do this by forcing all ECOM plug-ins to inherit from some common interface (e.g., MPluginInterface), but the designers wanted it to be possible to use any common type of Symbian OS class as an interface to a plug-in DLL. That's why some plug-ins export C classes, while others export M classes and yet others use R classes.

```

    REComSession::DestroyedImplementation(idtor_ID_Key);
}
...
private:
    TUuid idtor_ID_Key; // Helps ECOM identify the object
};

```

The `idtor_ID_Key` member is needed by ECOM to reliably identify each ECOM object for its reference counting, etc. It is initialized in the `CreateImplementationL()` call, (either by using the special macro, `_FOFF`, which calculates the offset of the key in the interface class or by passing it directly to the ECOM framework) and used in the `DestroyedImplementationL()` calls.

Writing an ECOM Plug-in

The resource file

Every interface may have several implementations (implementations can also have versions, but interfaces cannot). ECOM does not restrict how these implementations are packaged in DLLs: you might have a plug-in DLL with just one implementation of one interface or several DLLs each implementing several implementations of several interfaces. An ECOM-library user does not care, but as a plug-in writer your main job is to decide what this mapping of implementations to DLLs is going to look like. Are you providing just one DLL containing all the code, or would you rather have one DLL for each implementation? It is your choice.

Most of the mapping of implementations and interfaces to plug-in DLLs is done in a special resource file that is provided with each plug-in DLL:

```

// <dll uid>.RSS
#include "RegistryInfo.rh"

RESOURCE REGISTRY_INFO theInfo
{
    dll_uid = <dll uid>;
    interfaces =
    {
        INTERFACE_INFO
        {
            interface_uid = <uid for the interface>;
            implementations =
            {
                IMPLEMENTATION_INFO
                {
                    implementation_uid = <uid for the Jabber implementation>;
                    version_no = 1;
                    display_name = "Jabber Protocol plug-in";
                    default_data = "Jabber";
                    opaque_data = "";
                },
                IMPLEMENTATION_INFO
            }
        }
    }
}

```

```

    {
        implementation_uid = <uid for the AIM implementation>;
        version_no = 1;
        display_name = "Aol IM Protocol plug-in";
        default_data = "AIM";
        opaque_data = "";
    }
};
}
};
}

```

The resource file uses the array resource format to allow us to specify any number of interfaces and any number of implementations for each interface.

The interface sections are quite uninteresting as they contain just the UID needed to identify them in calls to `CreateImplementationL()` and similar functions. The implementation sections are more interesting; they are composed of:

- `implementation_uid`, which is used to identify the particular implementation; it can be used in some overloads of `CreateImplementationL()` and is returned by `ListImplementationsL()` but it should mostly be transparent to plug-in users
- `version_no`, which enables ECOM to support several different versions of the same implementation; ECOM transparently loads the most recent version of the implementation
- `display_name`, which is used indirectly in user interface elements; some GUIs allow the user to select a particular implementation
- `default_data`, which contains a text string used to help in deciding which implementation is the right one to use (our example uses 'Jabber' or 'AIM' to identify the IM protocol); its interpretation depends on what the plug-in is designed to do
- `opaque_data` is a binary field that can be used to provide more structured information about the capabilities of a plug-in; plug-in libraries that use `opaque_data` also provide a special type of plug-in, a *custom resolver*, that implements the logic of selecting the right implementation based on some external information (such as the extension of the file to be converted) and the information in the opaque data.

The `default_data` item supports the meta-characters '|', meaning 'OR' (A|B means 'A' OR 'B'), and '*' indicating that the plug-in can handle various data types.¹² Some examples of its various uses are:

¹² This and the `TEComResolverParams` class show that ECOM originated in the Browsing/HTTP group within Symbian. The wildcard handling and the fact that `default_data`

- MIME types that a browser plug-in can handle: `default_data='text/wml'` or `default_data='text/*'`
- file extensions that an image converter can handle: `default_data='PNG||JPEG'`
- hash algorithms implemented by the plug-in to a cryptographic library: `default_data='SHA-1'`.

Now we are in a better position to understand what the `CImProtocol::NewL()` code does: the resource file for the Jabber plug-in has `default_data='Jabber'` and the client code calls something like `CImProtocol::NewL('Jabber')`. This is translated into a call to `CreateImplementationL()` that passes 'Jabber' to ECOM. ECOM searches through the plug-ins to find the one that has that string in the resource file and loads the DLL. It then calls the DLL's entry point to instantiate the object from the right class and passes a pointer to the instance back to the client. The client can then manipulate the instance through the base interface `CImProtocol`.

Low-level plug-in code

The basic MMP file for an ECOM plug-in is actually quite simple:

```
// Jabber.mmp
TARGET jabber.dll
TARGETTYPE PLUGIN

// First UID means it is an ECOM plugin
// Second UID has to match the dll_uid field in the plug-in resource file
UID 0x10009D8D 0x<DLL UID>

SOURCEPATH \MyPlugin
SOURCE Jabber.cpp

USERINCLUDE \Bar
SYSTEMINCLUDE \epoc32\include

START RESOURCE <DLL UID>.rss
    TARGET jabber.rsc
END

LIBRARY euser.lib ECOM.lib
```

Plug-ins are polymorphic DLLs: instead of exporting a whole series of entry points in the same way as statically-linked DLLs, ECOM plug-ins are only expected to have one entry point: a function that returns a

is at times called 'data type' (`TEComResolverParams::DataType`) indicates that its origin was oriented towards MIME-type handling.

pointer to `TImplementationProxy`, which gives a table of pairs of implementation UIDs and `NewL()` factory functions, and the size of the table so that when somebody tries to instantiate a certain implementation UID, ECOM knows which ‘factory function’ to call. It looks like this:

```
// Jabber.cpp
#include <e32std.h>
#include <ImplementationProxy.h>

#include "JabberProtocol.h"

const TImplementationProxy ImplementationTable[] =
{
    IMPLEMENTATION_PROXY_ENTRY(KJabberImplementationUID,
                                CJabberProtocol::NewL),
    IMPLEMENTATION_PROXY_ENTRY(KJabberEnhancedImplementationUID,
                                CJabberProtocolEnhanced::NewL)
};

EXPORT_C const TImplementationProxy* ImplementationGroupProxy(
    TInt& aTableCount)
{
    aTableCount = sizeof(ImplementationTable) /
        sizeof(TImplementationProxy);

    return ImplementationTable;
}
```

Providing a custom resolver

An important bit of ECOM functionality is resolving: taking a list of ECOM implementations for a certain interface and selecting a specific implementation to be loaded. The default behavior is known as the default resolver and you can see how it works by looking at the `ExampleResolver.cpp` code that comes with the SDKs. The default resolver goes through all candidate implementations, looks at their `default_data` and tries to match it against the `TEComResolverParams` provided in the ECOM creation call (`CreateImplementationL()`). The only complication is that it accepts the ‘|’ and ‘*’ meta-characters in the `default_data` field, which might be limiting for some uses.

Going back to our IM example, imagine that two competing Jabber plug-ins are provided. Which one should ECOM load? Perhaps some plug-ins are experimental and it should select only the ones marked as stable. Or perhaps some of them implement encryption of passwords that are saved on disk and we prefer those to the plug-ins that don’t implement it. The plug-in library could mandate that plug-in implementers have to provide some more structured information in `opaque_data` and then write a custom resolver that is able to understand this information.

As designers of the IM library, we could mandate that the binary field must include a flag that indicates if it is a stable or experimental release

of the plug-in and a flag that indicates if passwords are encrypted on disk or not. We can then write a custom resolver that uses some complicated selection logic, such as 'prefer plug-ins that support encryption and prefer stable releases over experimental ones'.

Writing a custom resolver is quite straightforward since it is just about implementing a special ECOM plug-in interface. You provide a plug-in that implements interface UID 0x10009DD0 (you have to use this in the plug-in resource file). You also have to fill out the implementation UID and the version number, but the other fields (`display_name`, `default_data` and `opaque_data`) can be left empty. The C++ class exported from the plug-in has to inherit from `CResolver`, which is declared in `\epoc32\include\ecom\resolver.h`.

The complicated part is coming up with a sensible binary format and writing the custom-resolver logic. There is a code example in the Symbian Developer Library's Symbian OS reference, at » C++ component reference » Multimedia MMF » `CMMFFormatImplementationInformation`.

After you've written the custom resolver, you need to make sure that any calls to `CreateImplementationL()` and `ListImplementationsL()` use the resolver UID argument with the implementation UID of your custom resolver. You can do this by documenting it somewhere or by providing some header files that do it on behalf of the client.

Bringing It All Together

Since this is all quite a bit of information to take in, let us try to go step-by-step through what happens.

1. We call `CImProtocol::NewL()` with a text string that identifies the particular protocol ('Jabber'). `CImProtocol::NewL()` calls an overload of `REComSession::CreateImplementationL()` that uses the default ECOM resolver and passes the interface UID for `CImProtocol` and the protocol string to ECOM.
2. The default resolver gets a list of all the plug-ins that implement the specified interface UID.
3. The default resolver goes through the plug-in list and compares the `default_data` provided in the RSS file of each plug-in with the protocol string that came through `CImProtocol::NewL()`.
4. The default resolver takes the implementation UID of the first implementation that matches (has the same interface UID and matching data in `default_data`) and uses the implementation UID to call another overload of `REComSession::CreateImplementationL()` to load that specific implementation.
5. ECOM loads the DLL if necessary and calls the entry point of the DLL to get the list of implementations.

6. It looks for the matching implementation UID in the implementations table and calls the factory function (the second entry in the exported implementations table).
7. `REComSession::CreateImplementationL()` returns a `TAny*` pointer to the new plug-in object.
8. `CImProtocol::NewL()` casts the pointer to the right base class (`CImProtocol`).
9. The client uses the pointer, mostly ignorant of the fact that it is indeed an ECOM object.

This list is representative of what happens when we use any ECOM-based library. Having a good idea of what's going on behind the scenes is also helpful when debugging ECOM-related problems.

REComSession::FinalClose()

One of the differences between designing normal classes and designing plug-in interface classes is the object lifecycle. For plain old C++ objects, it comprises:

1. Instantiate object.
2. Use member functions.
3. Delete object.

With plug-in objects, the lifecycle is like this (simplified):

1. Load the plug-in DLL.
2. Instantiate the object.
3. Use some member functions.
4. Delete the object.
5. Unload the DLL.

ECOM attempts to make Steps 1 and 5 transparent to the plug-in writer, but they are really only transparent if the designer of the plug-in interface class takes some additional measures to ensure it.

ECOM uses reference counting on the plug-in objects to detect when a plug-in DLL can be unloaded. However, this reference counting is not enough to enable immediate unloading of the plug-in DLL. Since Symbian OS v9.1, immediate unloading of the plug-in DLL would cause a crash, because the code in the plug-in DLL is still executing (in particular the destructor of the plug-in implementation class). Because of this, ECOM

was modified to implement delayed unloading. A member function called `REComSession::FinalClose()` triggers some final clean up. If you don't call `FinalClose()`, your application will have a small memory leak. This is not a huge problem in most cases, but some applications monitor memory leaks and might cause a panic.

Plug-in interface designers have two options for dealing with `FinalClose()`:

- provide a class that is part of the plug-in framework, that has to be instantiated before any plug-in and that is destroyed after we are finished with all plug-ins; the destructor of this class can safely call `FinalClose()` to make sure that the additional internal clean-up happens
- as an alternative, you could provide a pair of static functions (`init()` and `close()` or similar) that the clients of the framework must call
- document as part of your plug-in framework that `FinalClose()` may have to be called to avoid the memory leak in DEBUG mode.

Although the first option is a cleaner solution, the second option also has advantages in certain cases. For existing plug-in libraries, it is less work to tell clients to call `FinalClose()` in certain cases. Interestingly, it might also be less work for the users of the library (since most code is unchanged). On balance, use the first option if you are designing a new library; if you have legacy code, you should probably go for the second option.

Uses of ECOM

- The Multimedia Framework makes extensive use of ECOM (see Chapter 21)
- The Converter Architecture (CONARC) uses ECOM to implement converters (see `CConverterBase2`)
- Front-End Processors (FEPs) are implemented as ECOM plug-ins
- Data and File Recognizers
- Character Converters (`Charconv`)
- HTTP Filters
- The XML Framework implements the parsers (XML and WBXML) as ECOM plug-ins
- Rasterizers in the Open Font System are implemented as ECOM plug-ins.

References

You will find more information about ECOM in the Symbian Developer Library, in both the Symbian OS guide and Symbian OS reference sections.

In the Symbian OS guide, look under the following headings:

- >> System libraries >> Using ECOM
- >> Platform security >> Symbian OS changes for platform security >> System libraries >> ECOM

In the Symbian OS reference section, look under:

- >> C++ component reference >> Syslibs ECOM.

19.4 Plug-ins in Symbian OS

ECOM was introduced in Symbian OS v7.0. At the time, several custom plug-in systems were already in use Symbian OS. Slowly these custom systems have started moving over to using ECOM and now ECOM is used to provide extensibility for all kinds of system services.

Messaging MTMs

We mentioned earlier that Messaging supports both a simple-to-use SendAs server and a more complicated Messaging Architecture. The Messaging Architecture is built around the concept of messaging-type modules (MTM).¹³ MTMs are plug-ins that extend the messaging functionality with new message types.

Comms Extension Modules

The Serial Comms server C32 also uses a plug-in mechanism to allow extensibility. C32 plug-ins are called comms extension modules (CSYs). As we see in Chapter 20, Symbian OS provides CSYs that work with RS232 serial ports, Bluetooth and infrared.

ETel Extension Modules

ETel is the Symbian OS Telephony server. The ETel server supports server-side extension modules that translate client requests (such as making a

¹³ A lot of non-ECOM plug-in frameworks use the file extension of the plug-in DLL to make it easy to distinguish their plug-ins from all the others. Most short names you see (such as, MTM, CSY and TSY) have historically also been the extension for that particular type of plug-in.

call or hanging up) into hardware-specific commands. These modules are called ETeI Extension modules (TSYs).

ESOCK Protocol Modules

The socket server uses plug-ins called ESOCK Protocol modules (PRT). The support for different protocols has been implemented by providing protocol modules, such as `tcpip.prt`.

Summary

If extensibility is important to you, use ECOM. There's no real excuse not to use it. It takes a lot of the pain out of enabling your application to use plug-ins. Since Symbian and phone manufacturers use it intensively, you can be pretty sure that it is stable and fast.

Even if you are not interested in application extensibility, you find that it is not only the APIs from Symbian that are based on ECOM. Whatever SDK you use, phone manufacturers also use it a lot. The ECOM knowledge you have acquired in this chapter is very useful in the following chapters since plug-ins, and ECOM in particular, show up again and again.

20

Communications and Messaging Services

In this chapter, we demonstrate how to exchange data between Symbian OS phones using SMS, MMS, email, Bluetooth, infrared and a serial connection.

Throughout the chapter we use the term ‘transport’ to refer to a means of sending payloads from an application running on one phone to a second instance running on another phone. The term is often used informally to describe the means by which data is transferred to another entity. Note this is not exactly equivalent to the ‘transport layer’ defined by the OSI Protocol Stack.¹

We use the transports to enable a game of noughts and crosses between two phones. The OandX protocol is not significant in itself, but it demonstrates how common game logic can use a wide range of transports.

20.1 Communications in Noughts and Crosses

Applications use transports by talking to a Symbian OS server. The message server supports SMS, MMS and email; the socket server supports Bluetooth, infrared and TCP/IP; and the communications (RS-232) server supports Bluetooth, infrared, and serial cable connections. This one-to-many relationship between servers and transports promotes the reuse of code because the application can support additional transports without rewriting all of the code that communicates with the server. The Noughts

¹ The OSI Protocol Stack is described at [http://standards.iso.org/ittf/PubliclyAvailableStandards/s025022_ISO_IEC_7498-3_1997\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/s025022_ISO_IEC_7498-3_1997(E).zip)

and Crosses application uses almost exactly the same code to support SMS and email because the message server provides them both.

Although there are several classes in this inheritance tree, much of the logic is contained in the intermediate classes such as `CMessageTransport`. The leaf classes such as `CEmailTransport` customize the code for a specific transport.

Although it is straightforward to support different transports on a single server, there are still fundamental differences between some of the transports, which have to present a common interface to the application controller.

Consider the difference between transports that maintain a connection with the remote phone and those that send isolated messages. A Bluetooth connection occurs when a client phone connects to a server and that connection is maintained for the game's lifetime. This contrasts with using SMS, which involves sending a series of isolated messages.

To support the asymmetry of one device starting a game, and a second device joining it, and to provide a simple way of deciding which player is noughts and which player is crosses, the user can either offer a game or join a game that is offered by another player. In connection-based transports, the offering player is the server, and it waits for the joining player to connect. In all cases, the offering player goes first and plays with crosses.

The current player selects a tile and the transport sends its location to the remote phone. The remote player then selects a tile and responds with their move.

In some protocols, the receiving phone sends an acknowledgement to the sender as soon as it has received the payload and determined that it is not corrupt. This informs the sender that the payload was successfully received, so they can get on with another task.

The Noughts and Crosses application does not use acknowledgements because that would double the number of messages that are exchanged between the phones. This is not a problem for local area connections such as Bluetooth, but it is inappropriate for the messaging transports that have higher latency and may incur a charge for every message that is sent, such as SMS or MMS. (Connection-based transports such as Bluetooth may use acknowledgements to manage the data flow between phones, but this is transparent to the application that uses the protocol.)

20.2 Communication Between Controller and Transport

The application's controller uses two classes to communicate with remote phones. It instantiates a subclass of `CTransportInterface` via an ECOM plug-in. The transport is implemented as a concrete subclass,

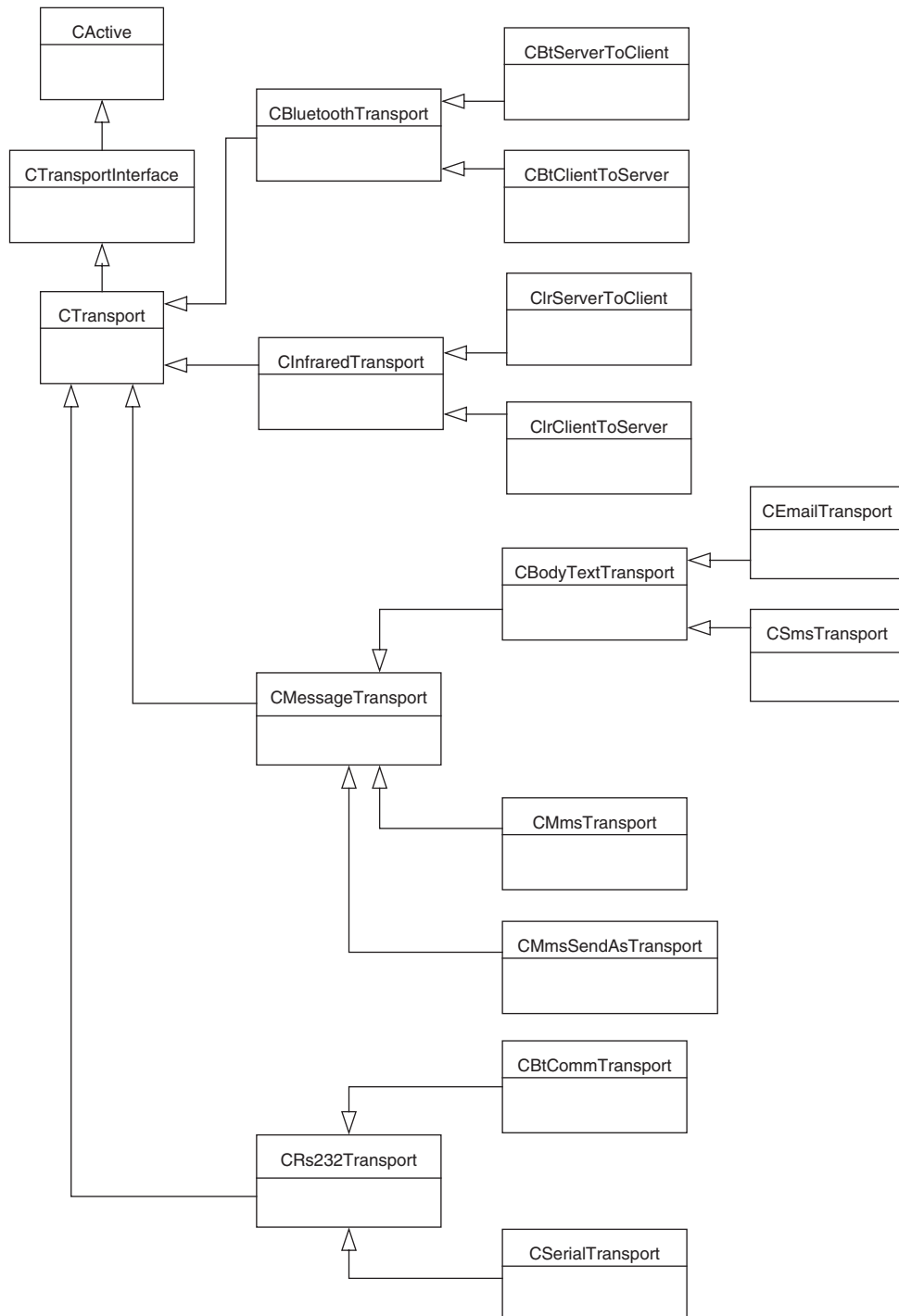


Figure 20.1 Transport class hierarchy for the Noughts and Crosses application

which uses Symbian OS APIs to send data to and receive data from a remote phone. The controller implements `MTransportObserver`, which declares functions that are called back when a transport event, such as receiving data or losing the connection, occurs. The interesting parts of the definitions are reproduced below.

```
class CTransportInterface : public CActive
{
public:
    IMPORT_C static CTransportInterface* NewL(TUId aTransportUid,
        MTransportObserver& aObserver, const TDesC& aAddress,
        TBool aInitListen);

    IMPORT_C virtual ~CTransportInterface();

    virtual void SendPayload(const TDesC& aPayload) = 0;
};

class MTransportObserver
{
public:
    virtual void SentPayload() = 0;
    virtual void ReceivedPayload(const TDesC& aPayload) = 0;
    virtual void LostConnection(TInt aError) = 0;
};
```

Together, these classes satisfy five essential requirements for remote communications on a mobile phone.

- They are asynchronous. It takes time for the local phone to send a message and for the remote phone to receive it. After the remote phone has received the message, it may wait for input from the remote user, in this case selecting a tile, before it responds. During this time, both applications must remain responsive to their users. Symbian OS provides the active object framework to handle asynchronous events efficiently. While this is most likely used by the implementation of `CTransportInterface`, the controller should not have to worry about this level of detail – it just implements `MTransportObserver` to be notified when an event occurs.
- They are transport-independent. `CTransportInterface` provides the API to connect to a remote phone and exchange data with it, and the subclass implements the transport mechanism, such as Bluetooth, SMS or email. This extensibility is useful because smartphones support an expanding range of connectivity options and some older transports, such as serial cables and infrared are no longer supported on all phones. When there is a choice of transports, the user may prefer one to another. For example, the user may choose Bluetooth rather than SMS to avoid network charges. ECOM enables the application to load

a transport plug-in at run time, which in turn enables this decoupling of the controller and the transport.

- They do not contain application logic. The transport classes send and receive unstructured data, which is passed to and from the controller as descriptors. The transports do not understand what the data means. They do not know that the single-character payload of the Noughts and Crosses application represents the single-digit text of an integer that indexes a tile on the board. Decoupling this encoding and decoding logic from the transport makes it easier to reuse the transports in another application, and avoids duplicating application logic in each transport.
- They do not assume the connection is always available. Applications that communicate with remote phones must gracefully handle the loss of those connections. Depending on the transport, the application may be notified of a lost connection. When a Bluetooth connection is broken, a pending read fails with `KErrDisconnected`, so the application notices immediately. If the game is played over SMS, there is no maintained connection and the local player is left waiting for the remote player to respond.
- They are power-conscious. The object derived from `CTransportInterface` can be destroyed when the connection is no longer required. This is useful for transports that maintain a connected session because the connection can be switched off, to conserve power, at the end of the game. A user can play one game over Bluetooth and then another over SMS. The Bluetooth connection is taken down when the first game is over.² As well as conserving power, this also frees system resources such as memory. In the case of connection-based transports, it frees the connection so other applications can use it.

20.3 Serial Communications

As a way of connecting a smartphone or a PDA to a PC, serial cables have long been superseded by infrared, Bluetooth and USB. Nonetheless, serial or, more precisely, RS-232 connections have several advantages for developers.

RS-232 is a simple protocol, which makes it ideal for developing and testing software. A null modem cable is sufficient to run a two-player

² This consideration led to the decision to include a Close menu item in the UIQ version's menu. UIQ applications do not usually have explicit Close or Exit menu items, because it does not typically matter if they are left running in the background.

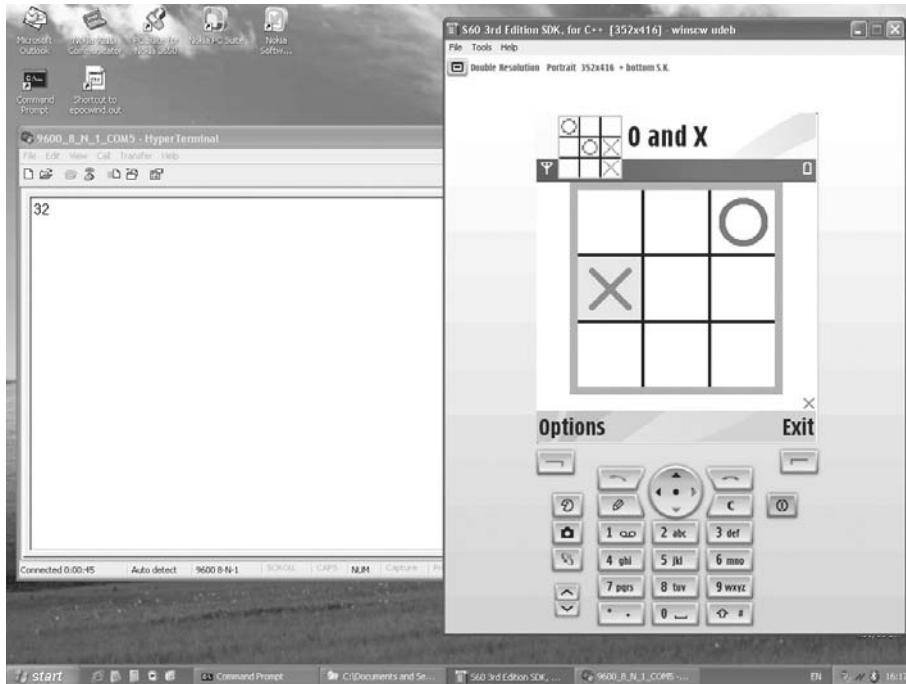


Figure 20.2 Game between S60 emulator and Hyperterminal

game of Noughts and Crosses between two instances of the emulator. If the application only sends human-readable text between phones, then a single instance can be tested with a terminal program such as Hyperterminal, where the developer manually exchanges data with the application.

Because many applications have been written to use RS-232 interfaces, emulation profiles are available for infrared, Bluetooth, and USB. The Noughts and Crosses application uses infrared and Bluetooth emulation, which are discussed later. USB is a host-to-client connection, designed for connecting peripheral devices, such as smartphones, to host devices, such as PCs; it is not generally used for peer-to-peer communications.³

RS-232 nonetheless has its limitations. It has error detection (parity bits) and flow control (Xon/Xoff, CTS/RTS, DSR/DTR) but no addressing, session, acknowledgement or retry features. All of these must be provided by a higher-level protocol. Despite these limitations, it is sufficient to test that the engine, controller, UI and loading of the transport plug-in all work on the emulator.

³ Symbian OS v9.3 supports USB 2.0 On-The-Go (OTG) which provides a way to connect peripherals to a phone.

Opening a Serial Connection

To use a serial cable on the emulator, or an emulated infrared connection on a phone, the application loads the required logical and physical device drivers (LDDs and PDDs). Logical and physical drivers are discussed at length in [Sales 2005, Chapter 12]. To use a serial cable or an emulated serial connection over infrared, the application must load the required drivers.

The LDD is called ECOMM on both the emulator and an actual phone. The PDD is ECDRV on the emulator and EUART on hardware.

```
// Name of serial logical device driver on both hardware
// and emulator.
_LIT(KLddName, "ECOMM");

// Installed name of logical device driver on emulator.
_LIT(KLddInstallName, "Comm");

#if defined(__WINS__)
    // Name of physical device driver on emulator.
    _LIT(KPddName, "ECDRV");
#elif defined(__EPOC32__)
    // Name of physical device driver on hardware.
    _LIT(KPddName, "EUART");
#endif

// ...

EXPORT_C void SerialUtils::LoadDeviceDriversL(TBool& aLoadedLdd,
                                              TBool& aLoadedPdd)
{
    aLoadedLdd = aLoadedPdd = EFalse;

    TInt r;

    r = User::LoadLogicalDevice(KLddName);
    if (r == KErrAlreadyExists)
        r = KErrNone;
    User::LeaveIfError(r);
    aLoadedLdd = ETrue;

    r = User::LoadPhysicalDevice(KPddName);
    if (r == KErrAlreadyExists)
        r = KErrNone;
    User::LeaveIfError(r);
    aLoadedPdd = ETrue;
}
```

The application only cares that the drivers are available, and so it is acceptable to fail with `KErrAlreadyExists`. After the communications port has been closed, the drivers must be freed with `User::FreeLogicalDevice()` and `User::FreePhysicalDevice()`. These are both demonstrated later when closing the connection.

Applications initialize a serial connection via the Comms Server, which is sometimes referred to by its component name, C32.⁴ The application establishes an RCommServ session with the server, and uses this to load the appropriate communications module:

```
TInt r;

r = iCommServ.Connect();
if (r == KErrNone)
    r = iCommServ.LoadCommModule(aCommModuleName);
```

iCommServ is an instance of RCommServ and aCommModuleName is the name of the communications module. A communications module is a plug-in DLL, identified by its CSY extension. Symbian OS provides several CSYs, the most useful of which are:

- ECUART: RS232 serial port
- BTComm: Outgoing Bluetooth serial emulation
- IRComm: Infrared serial emulation.

The next step is to open the real or emulated communications port. RComm is a subsession hosted on the RCommServ session.

```
r = iComm.Open(iCommServ, aPortName, ECommExclusive, ECommRoleDTE);
```

iComm is an instance of RComm and aPortName is the port name, which performs the same role as COM1 on a Windows PC. On Symbian OS it is formed from the required protocol and the unit number. To use the PC's serial ports, specify COMM: : 0 for COM1, COMM: : 1 for COM2, and so on. On a phone, BTComm: : 0 opens an outgoing Bluetooth connection and IRComm: : 0 opens a connection over infrared.

In the Noughts and Crosses application, the user supplies the port name for a cable connection in the address dialog. This makes it possible to run two instances of the emulator on a single PC because they can use two different COM ports, which are connected by a null modem cable.

Once opened, but before any reads or writes have been queued, the port can be configured with RComm::SetConfig(). This API takes a descriptor, which is a TCommConfig or TCommConfig2 instance, packaging an instance of TCommConfigV01 or TCommConfigV02 respectively.

The latter two classes are structures that contain standard serial configuration parameters, such as bits per second and the number of data bits and stop bits, and so they are not discussed further here.

⁴ Some legacy example code shows the Comms Server being started explicitly with StartC32. This is not actually necessary, because RCommServ::Connect calls this function itself if the server is not available.

Exchanging Data over a Serial Connection

Once the communications port has been opened, it can be used to transfer data with `RComm::Read()` and `RComm::Write()`.⁵ Both of these functions have several overloads, all of which are asynchronous. They take a reference to an instance of `TRequestStatus`, which is completed when the operation completes, successfully or otherwise. The application typically uses an active object to be notified when the read or write completes.

`CTransport::LaunchRead()` asks the transport-specific subclass to start an asynchronous read from the remote phone to retrieve the other player's move. `DoLaunchRead()` is declared as a pure virtual function in `CTransport`, which is implemented by the transport-specific subclass.

```
void CTransport::LaunchRead()
{
    DoLaunchRead();
    SetActive();
}
void CRs232Transport::DoLaunchRead()
{
    iComm.Read(iStatus, iPayload8);
    // SetActive is called by CTransport::LaunchRead
}
```

When the application has finished with the serial connection, it closes the `RComm` subsession and the `RCommServ` session. It does not have to explicitly unload the communications modules by calling `RCommServ::UnloadCommModule()` because this occurs automatically when the session ends. Any pending reads or writes must be cancelled by calling `RComm::Cancel()`.

```
void CRs232Transport::DoCancel()
{
    iComm.Cancel();
}
```

Finally, if an LDD or PDD was loaded, then it must be unloaded.

```
EXPORT_C void SerialUtils::FreeDeviceDrivers(TBool aLoadedLdd,
                                              TBool aLoadedPdd)
{
    TInt r;

    if (aLoadedLdd)
    {
        User::FreeLogicalDevice(KLddInstallName);
    }
}
```

⁵ Some versions of the system documentation for `RComm::Write` state that the number of bytes to write is taken from the descriptor's maximum length. It is taken from the descriptor's current length.

```

if (aLoadedPdd)
{
    TFindPhysicalDevice fpd;
    TFullName installedName;
    _LIT(KPddInstallNamePattern, "Comm.*");
    fpd.Find(KPddInstallNamePattern);
    r = fpd.Next(installedName);
    if (r == KErrNone)
        User::FreePhysicalDevice(installedName);
}

// ...
}

```

Serial Communications over Infrared

Applications can exchange data over an emulated RS232 connection running over infrared. The process is almost exactly the same as for communicating over a serial cable – load the device drivers, optionally configure the port, and use the RComm read and write functions. The only differences are the CSY name – IRCOMM instead of ECUART – and the port name – IRCOMM: :0 instead of, say, COMM: :0.

The two techniques are so similar that the Noughts and Crosses application implements them as specializations of a generic serial connection.

```

CSerialTransport* CSerialTransport::NewIrCommTransportL(
    TAny* aTransportCreateInfo)
{
    _LIT(KIrCommCsyName, "IRCOMM");
    _LIT(KIrCommPortName, "IRCOMM: :0");

    const TTransportInterfaceCreateInfo& tci =
        *reinterpret_cast<TTransportInterfaceCreateInfo*>
            (aTransportCreateInfo);

    return New2L(tci.iObserver, KIrCommCsyName, KIrCommPortName,
                tci.iInitListen);
}

CSerialTransport* CSerialTransport::NewSerialCommTransportL(
    TAny* aTransportCreateInfo)
{
    _LIT(KSerialCommCsyName, "ECUART");

    const TTransportInterfaceCreateInfo& tci =
        *reinterpret_cast<TTransportInterfaceCreateInfo*>
            (aTransportCreateInfo);

    return New2L(tci.iObserver, KSerialCommCsyName, *tci.iAddress,
                tci.iInitListen);
}

CSerialTransport* CSerialTransport::New2L(

```

```

MTransportObserver& aObserver, const TDesC& aCsyName,
    const TDesC& aPortName, TBool aInitListen)
{
    CSerialTransport* self = new(ELeave) CSerialTransport(aObserver);
    CleanupStack::PushL(self);
    self->ConstructL(aCsyName, aPortName, aInitListen);
    CleanupStack::Pop(self);
    return self;
}

```

With serial communications over infrared, the developer does not have to decide which phone is the host and which is the client. Applications that want to send data more quickly, or to have finer control over the infrared connection, should use the Sockets API directly.

An application needs the LocalServices capability to open a serial connection over infrared.

Serial Communications over Bluetooth

An application can communicate with other phones over Bluetooth with an emulated RS232 connection or with sockets. The emulated RS232 connection is simpler and very similar to using a serial cable or IRCOMM. However, this can only be used for outgoing connections, e.g. to talk to an application on a PC. It cannot be used to communicate between two phones.

The Noughts and Crosses application can use BTCOMM to play against an emulator or an instance of Hyperterminal on a PC.

Where Symbian OS applications support an emulated serial connection over Bluetooth, the CSY BTCOMM hosts the port, which is typically called BTCOMM: : 0.

To identify the COM port to which a Windows application can connect on XP SP2, select Bluetooth Devices from the control panel or by double-clicking on the Bluetooth icon on the taskbar. Select the COM Ports tab in the dialog (see Figure 20.3) and look for the incoming connection from the phone. This COM port can be opened on the PC, e.g. with Hyperterminal, and used to exchange data with the phone.

An application needs the LocalServices capability to open a serial connection over Bluetooth.

20.4 Socket-based Communications

An application can communicate with remote phones over sockets. The APIs are more complex than serial communications but the application can also exercise finer control over the connection, configuring settings that are specific to the particular transport. For example, it can advertise

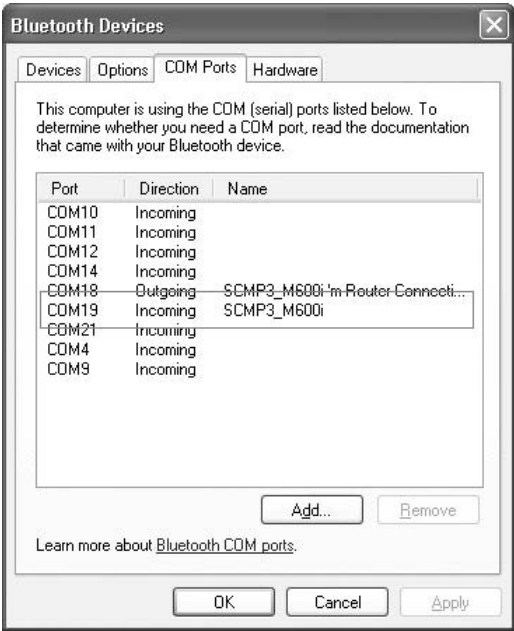


Figure 20.3 BTCOMM connection, as seen by Windows

a service such as printing over Bluetooth. Symbian OS provides a set of APIs that talk to the socket server.

The Socket Server

The socket server supports a range of transports such as Bluetooth, infrared and TCP/IP. The application connects to a remote phone with an instance of RSocket or a class such as CBluetoothSocket that wraps its functionality.

The socket server is a standard Symbian OS server that supports a range of protocols via plug-in protocol modules. These DLLs can be identified by the PRT suffix. Both the host and the client create a session to the socket server with RSocketServ.

The socket-based communications which are described below are connection-based. One phone – the host or server – accepts an incoming connection from the second phone – the client.

The general process of establishing a socket-to-socket connection is similar to the Berkeley sockets mechanism:

- 1. The host opens a listening socket and binds it to a local port.
- 2. The host listens for incoming connections on the listening socket. It calls RSocket::Listen(), passing in a queue size, which specifies the number of connections that it will accept on that port.

3. The host opens a blank socket which is not yet connected to a remote socket and passes it to the listening socket via `RSocket::Accept()`.
4. The host waits for a client to connect.
5. The client opens a connecting socket with the host phone's address, protocol and port number. The format of these values depends on the type of connection. For a Bluetooth phone, the address is the remote phone's 48-bit identifier; the protocol is L2CAP or RFCOMM; and the port is an L2CAP or RFCOMM channel.
6. The client calls `RSocket::Connect()` and waits for the host to accept the connection.
7. When the host detects an incoming connection, its accept operation completes and it establishes a connection between the connecting client and the blank socket. The blank socket, which may now be referred to as the accept socket or the data socket, can now exchange data with the remote phone.
8. On the client side, the connect operation completes. The connect socket can now exchange data with the accept socket on the host.

Socket-based Communications over Infrared

IRCOMM is simple to use, but it does not allow the application to configure IR-specific settings such as whether to use the `Irmux` protocol, which is unreliable, or `TinyTP`, which is reliable.

For the purposes of the Noughts and Crosses application, which does not provide reliable communications anyway, either protocol can be used. The application uses exactly the same code for both protocols, demonstrating how the socket server supports code reuse. When the IR Socket menu option is selected, a dialog box is displayed, into which the user should enter the required protocol – '`Irmux`' or '`IrTinyTP`'.

The process for connecting two phones over infrared follows the generic process described above. The following sections describe the process from the server's and the client's point of view.⁶

Extracting infrared protocol information

The server and client phones must both load the same device drivers as for serial communications over infrared. They then connect to the socket server and get information about the protocol they intend to use. `RSocketServ::FindProtocol()` is given the protocol name and an instance of `TProtocolDesc` which the server populates.

⁶ The terms 'server' and 'client' are used for consistency with the Bluetooth example. In other contexts, the term 'host' may be used instead of 'server'.

```
void CInfraredTransport::ConstructL(const TDesC& aProtocolName)
{
    SerialUtils::LoadDeviceDriversL(iLoadedLdd, iLoadedPdd);

    TInt r = iSocketServ.Connect();
    if (r == KErrNone)
        r = iSocketServ.FindProtocol(aProtocolName, iProtoDesc);
    User::LeaveIfError(r);
}
```

iSocketServ is an instance of RSocketServ, which is a session to the socket server.

iProtoDesc is an instance of TProtocolDesc, which contains information about the protocol, which is used later to open a socket.

Offering a connection over infrared

The server opens the listening socket, passing in the information that it retrieved from FindProtocol().

```
TInt r = iListenSocket.Open(iSocketServ, iProtoDesc.iAddrFamily,
                           iProtoDesc.iSockType, iProtoDesc.iProtocol);
User::LeaveIfError(r);
```

iListenSocket is an instance of RSocket.

The server binds the socket to an infrared address, which uses a designated port number. This port number must be agreed in advance by the client and the server.

```
TIrdaSockAddr sockAddr;
sockAddr.SetPort(KIrSocketPortNum);
iListenSocket.Bind(sockAddr);
r = iListenSocket.Listen(*qSize* 1);
User::LeaveIfError(r);
```

Notice that TIrdaSockAddr is a subclass of TSocketAddr that adds infrared-specific information, such as the 'sniff status'. An instance of this object is passed to RSocket::Bind() which takes a reference to a TSocketAddr object. Transport-specific subclasses of TSocketAddr are also used for Bluetooth (TBTSockAddr, which is itself subclassed by TL2CAPSockAddr and TRfcommSockAddr) and TCP/IP (TInetAddr).

The application opens the blank socket and calls Accept() on the listen socket.

```
r = iSocket.Open(iSocketServ);
User::LeaveIfError(r);

// accept an incoming connection
iListenSocket.Accept(iSocket, iStatus);
SetActive();
```

iSocket is an instance of RSocket.


```
iSocket.Connect(irdaSockAddr, iStatus);  
SetActive();
```

Exchanging data between infrared sockets

Once the client and server have connected to each other, they can exchange data with the normal RSocket functions such as Read() and Write(). The sockets, including the server's listen socket, should be closed when they are no longer needed.

Further infrared functionality

This section has covered the basic steps required to exchange data between two phones over infrared. Symbian OS provides further infrared functionality including IrTranP, a protocol that is designed to exchange digital images. The socket APIs also allow the application to configure the connection in greater detail. For example, it can configure the baud rate with RSocket::SetOpt().

```
TPckgBuf<TBps> setbaudbuf(EBps9600);  
baudsock.SetOpt(KUserBaudOpt, KLevelIrlap, setbaudbuf);
```

The system documentation describes how to use the advanced features of the infrared sockets API.⁷

Socket-based Communications over Bluetooth

It is more complex to communicate with a Bluetooth socket than over an emulated serial connection as it requires Bluetooth-specific code. The benefits are that the application can specify the link in greater detail – it can advertise or search for a service, such as printing, instead of making a blind connection – and it can establish device-to-device communications. Symbian OS provides a Bluetooth PRT module that supports L2CAP and RFCOMM.

Logical link layer and adaptation protocol (L2CAP)

The wireless connection between two phones is organized as a set of logical channels. L2CAP provides these logical channels so that applications can exchange data without knowing what is being transmitted over other, unrelated channels.

⁷ See details in the Developer Library at » Symbian OS guide » Infrared » Using IrDA Sockets » Advanced IrDA Sockets API.

Each channel is assigned a Channel Identifier (CID, see [Bluetooth SIG 2006, 3:A, Section 2.1]) so the channels can be multiplexed over a single phone-to-phone connection. The channel itself supports a set of multiplexed protocols such as SDP or RFCOMM, both of which are used by the Noughts and Crosses application. Packets sent over an L2CAP logical channel contain a Protocol/Service Multiplexer (PSM⁸) to identify the service.

L2CAP can exchange data packets up to 64 KB, although the packets may be transparently sent in smaller chunks, depending on what is supported by the lower levels of the Bluetooth stack implementation. It also supports retransmission and flow control, which can be configured with `TL2CapConfig`.

Serial emulation over Bluetooth (RFCOMM)

RFCOMM supports legacy applications by emulating a serial port. It is a protocol that runs over an L2CAP channel and has its own reserved PSM. A phone can use up to 60 RFCOMM sessions (30 incoming and 30 outgoing) running over a single L2CAP channel. Each RFCOMM channel is assigned a channel number, much like L2CAP channels.

From the application's point of view, RFCOMM is managed with sockets in a very similar way to L2CAP. A truly serial-like interface, one that uses `RComm`, is provided by `BTComm`, as described in Section 20.3.

Connecting Two Phones over Bluetooth

The process for connecting two phones over RFCOMM or L2CAP is an extension of the generic process. Because many services can be offered over Bluetooth, such as Serial Port Profile and Dial-up Networking, the host must advertise which services it provides, and the client must select the service to which they want to connect.

Listening for incoming connections

A phone that wants to provide services over Bluetooth begins by connecting to the socket server. After that, it creates a listening socket, which is an instance of `CBluetoothSocket`.⁹

The socket object is created with the protocol's name ('RFCOMM' or 'L2CAP'.) The following code from the Noughts and Crosses application creates an instance of `CBluetoothSocket` and binds it to an available L2CAP or RFCOMM channel:

⁸ Defined PSMs are listed in [Bluetooth SIG 2006, 3:A, Section 4.2].

⁹ `CBluetoothSocket` was introduced in Symbian OS 8.0. It is supported from S60 Second Edition, FP 2 and UIQ 3. It should be used instead of `RSocket` for Bluetooth connections.

```

iListenSocket = CBluetoothSocket::NewL(*this, iSocketServ, aProtocolName);

// channel security settings. No authentication, authorisation,
// or encryption.
TBTSserviceSecurity oandxSecurity;
oandxSecurity.SetUid(KUidServiceSDP);
oandxSecurity.SetAuthentication(EFalse); // do not require key
// (PIN) exchange
oandxSecurity.SetAuthorisation(ETrue); // require local user
// to confirm accept

oandxSecurity.SetEncryption(EFalse);
oandxSecurity.SetDenied(EFalse);

TInt r;
if (aProtocolName == KL2CAPDesC)
{
    TL2CAPSockAddr l2SockAddr; // find an available L2CAP channel
    l2SockAddr.SetPort(KL2CAPPassiveAutoBind);
    l2SockAddr.SetSecurity(oandxSecurity);
    r = iListenSocket->Bind(l2SockAddr);
}
else // if (aProtocolName == KRfcommDesC)
{
    TRfcommSockAddr rfSockAddr; // find an available RFCOMM
    // channel
    rfSockAddr.SetPort(KRfcommPassiveAutoBind);
    rfSockAddr.SetSecurity(oandxSecurity);
    r = iListenSocket->Bind(rfSockAddr);
}

```

- aProtocolName is a descriptor which is 'RFCOMM' (KRfcommDesC) or 'L2CAP' (KL2CAPDesC).
- *this refers to the instance of CBtServerToClient. This class is defined by the Noughts and Crosses application and implements the MBluetoothSocketNotifier interface, which declares functions that the Bluetooth stack calls to notify the application about events such as receiving an incoming connection.
- iListenSocket points to an instance of CBluetoothSocket.

Because Bluetooth allows phones to communicate without line of sight, it is useful where phones cannot be physically connected or where it would be awkward to line up two phones' infrared transceivers, for example, to exchange phonebook contacts at a train station. However, this usability brings with it security considerations – a user does not want an arbitrary device to be able to push unwanted data or applications onto their phone.

On a Symbian OS smartphone, general Bluetooth settings such as whether it is enabled or whether the phone is discoverable can be set via the built-in settings application. Here we look at how to configure the settings for a specific connection.

The application sets the port's security settings by passing a reference to an instance of TBTSserviceSecurity to TBTSockAddr::

`SetSecurity()`. This class defines whether the incoming connection requires authentication (both users must enter the same PIN), authorization (the user of the receiving phone must confirm the connection) or encryption.

The listening socket does not exchange the application's data with remote phones. It listens for incoming connections and, when it detects one, it marries a blank instance of `CBluetoothSocket` to the incoming connection, creating an accept socket. From then, the accept socket can exchange data with the client.

In coding terms, the application calls `CBluetoothSocket::Listen()` on the listening socket, passing the queue size as an argument. The queue size is the number of incoming connections that it matches up with local blank sockets.

```
// accept one incoming connection on this socket
if (r == KErrNone)
    r = iListenSocket->Listen(*qSize* 1);
User::LeaveIfError(r);
```

The listen socket's port was set to `KL2CAPPassiveAutoBind` or `KRfcommPassiveAutoBind` instead of a specific L2CAP PSM or RFCOMM channel. This instructs the Bluetooth protocol to find an available channel number, which the application can then retrieve with `CBluetoothSocket::LocalPort()`.

```
TInt channel = iListenSocket->LocalPort();
```

The application then calls `CBluetoothSocket::Accept()` on the listening socket, passing in the (currently blank) accept socket.

```
iBtSocket = CBluetoothSocket::NewL(*this, iSocketServ);
iListenSocket->Accept(*iBtSocket);
```

When the client attempts to connect, the Bluetooth framework calls the application's implementation of `MBluetoothSocketNotifier::HandleAcceptCompleteL()`.

So far, this is similar to using `RSocket` objects directly. The main difference is that the framework notifies the application when events occur via the `MBluetoothSocketNotifier` interface instead of completing an instance of `TRequestStatus`.

Before a client can connect to the server, it must first determine that the server supports the service for which it is looking – in this case the Noughts and Crosses application – and the client must also find out the protocol and channel on which the service is available. Bluetooth

servers advertise their services with Service Discovery Protocol (SDP, see [Bluetooth SIG 2006, 3:B]).

The server should call `CBluetoothSocket::Accept()` before it advertises the service; if it does not, a client could attempt to connect before the server is ready to accept it.

Publishing a service

Bluetooth devices advertise their services, such as printing, via SDP. Remote devices can read the available services and then decide to connect to the server device.

SDP runs over L2CAP and has a reserved PSM. Any Bluetooth device that offers services to other devices must implement an SDP server. The Bluetooth standards use the term ‘server’ to refer to the component that provides access to a set of service records. This happens to be implemented as a Symbian OS server but it is generally referred to as the ‘SDP database’ on Symbian OS. As illustrated in Figure 20.4, the database contains a number of service records. Each service record is identified by a service handle, and contains a number of attributes. Each attribute contains an ID and a value. The attribute’s type and size are encoded in its value.

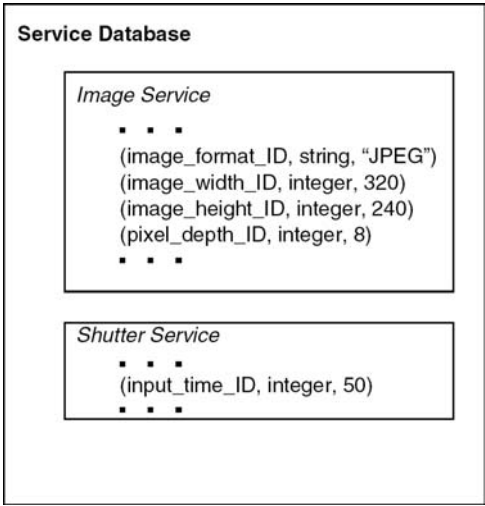


Figure 20.4 Example service database contents

The server application adds a service record to the SDP database. A client reads this database to find a service in which it is interested and then connects to the host. The database is there to publish the information about available services. It is not involved in the subsequent connection.

In Symbian OS terms, the application on the server phone connects to the SDP database server using the familiar session and subsession

mechanism. `RSdp` creates a session to the server and `RSdpDatabase`, which is used to add and later delete the record, runs a subsession over it.

```
TInt r = iSdp.Connect();
if (r == KErrNone)
    r = iSdpDb.Open(iSdp);
User::LeaveIfError(r);
```

`iSdp` is an instance of `RSdp`, which derives from `RSessionBase`, and `iSdpDb` is an instance of `RSdpDatabase`, which indirectly derives from `RSubSessionBase`.

Each Bluetooth service has a `ServiceClassIDList` attribute (see [Bluetooth SIG 2006, 3:B, Section 5.1.2]). This is a sequence of 128-bit identifiers, known as Universally Unique Identifiers (UUIDs, see [Bluetooth SIG 2006, 3:B, Section 2.7.1]) that describe the service. A sequence is used instead of a single number because a service may match several increasingly specific functions. The specification provides an example of a printer that is described by four classes – printer, postscript printer, color postscript printer and duplex color postscript printer.

Organizations such as the Bluetooth SIG and Symbian reserve ranges of UUIDs to identify their service classes and specific services. Reserved Bluetooth UUIDs are commonly referred to in a 16-bit or 32-bit short form. Symbian OS defines the `TUUID` class to hold UUIDs, and provides constructors and the `SetL()` function to expand short-form values from UUIDs from the Symbian or Bluetooth SIG ranges.

An application specifies the class ID when it creates the service record. For demonstration purposes, a value from the reserved Symbian range is used, although applications should use their own values. For the developer's convenience, `RSdpDatabase::CreateServiceRecordL()` provides an overload which takes a single UUID and internally converts it to a service class list.

```
iSdpDb.CreateServiceRecordL(iServiceUuid, iServRecHandle);
```

`iServRecHandle` is an instance of `TSdpServRecordHandle` which is set to a value which identifies the record within the database.

The application then builds the service record by adding attributes. These can be standard (although not compulsory) attributes.¹⁰ They can also be attributes which are defined by the service provider, and which are only meaningful within the context of that service. Attribute IDs up to `0x1fff` are reserved by the Bluetooth specification (see [Bluetooth SIG 2006, 3:B, Section 5.1.17]).

¹⁰ [Bluetooth SIG 2006, 3:B, Section 5.1] lists the Universal Attribute Definitions. Only `ServiceRecordHandle` and `ServiceClassIDList` are required and these are both set when the record is created with `CreateServiceRecordL()`.

Several types of attributes are defined, including integers, UUIDs, lists, Booleans and strings (see [Bluetooth SIG 2006, 3:B, Section 3.2]). `RSdpDatabase` defines several overloads of `UpdateAttributeL()` for adding these. During development, it is useful to browse the SDP database from a remote phone to ensure the service is being published. An easy way to identify the record is to set the service name.

```
iSdpDb.UpdateAttributeL(iServRecHandle,
    KSdpAttrIdBasePrimaryLanguage + KSdpAttrIdOffsetServiceName,
    KOandXServiceName);
```

Notice the service name attribute value `KSdpAttrIdOffsetServiceName` is added to the primary address offset. A further attribute `LanguageBaseAttributeIDList` (see [Bluetooth SIG 2006, 3:B, Section 5.1.7]) can be added to support multiple natural languages.

Once it has found the service record that it requires, the client connects to the service. The service record includes a `ProtocolDescriptorList` attribute that tells the remote application which protocols and ports it must use (see [Bluetooth SIG 2006, 3:B, Section 5.1.5]). If the host used an RFCOMM channel running over an L2CAP PSM, it could produce a list like the following.

```
((L2CAP, RFCOMM-PSM) (RFCOMM-PSM, 2))
```

Symbian OS defines a class `CSdpAttrValueDES` to construct Bluetooth list attributes. The code to allocate a list with this class looks strange when written down, but it is actually a convenient way to allocate such a structure. The indentation reflects the list's structure and there is no need to remember which object is being modified – it is a single C++ statement.

```
const TInt KRFCOMMPSM = 0x03; // The PSM on which RFCOMM
                               // resides
CSdpAttrValueDES* protoDesc = CSdpAttrValueDES::NewDESL(NULL);
TSdpIntBuf<TUint16>
    l2capChannelBuf(static_cast<TUint16>(KRFCOMMPSM));
TSdpIntBuf<TUint8> channelBuf(static_cast<TUint16>(aChannel));
CleanupStack::PushL(protoDesc);
protoDesc
->StartListL()
->BuildDESL()
    ->StartListL()
    ->BuildUUIDL(KL2CAP) // 0x0100
    ->BuildUIntL(l2capChannelBuf) // 0x0003
    ->EndListL()
->BuildDESL()
    ->StartListL()
    ->BuildUUIDL(KRFCOMM) // 0x0003
    ->BuildUIntL(channelBuf)
    ->EndListL()
->EndListL();
```

The protocol descriptor list is a standard, but not compulsory, attribute. Its structure and interpretation is up to the applications. In practice, an application is likely to know which protocol is being used to offer the service – it is probably talking to another instance of itself – and only needs to extract some run-time information such as the port number.

The accept operation completes when the client socket attempts to connect to the service. The Bluetooth stack notifies the application by calling its implementation of `MBluetoothSocketNotifier::HandleAcceptCompleteL()`. Once the server accepts the connection, the accept socket can exchange data with the client's connect socket.

Exchanging data with a remote phone

CBLuetoothSocket defines the `Read()` member function, which starts an asynchronous read operation. This can be cancelled with `CancelRead()` and the stack calls `HandleReceiveCompleteL()`, when the read completes. `Write()`, `CancelWrite()` and `HandleSendCompleteL()` are complementary functions for sending data.

When the application has finished with the connection, it should remove the SDP record and destroy the socket objects. The SDP record can actually be removed at any time. It is only there for informational purposes so the client knows which channel and protocol to connect to.

```
if (iSdpDb.SubSessionHandle() != KNullHandle && iServRecHandle != 0)
{
    TRAP_IGNORE(iSdpDb.DeleteRecordL(iServRecHandle));
}

iSdpDb.Close();
iSdp.Close();
```

The client phone, which connects to the Bluetooth service, asks the user to select a remote phone, and then searches that phone's SDP database to find out how to connect to the service, before making the actual connection. How the application asks the user to select a remote device differs between S60 and UIQ.

Asking the user to select a remote device on S60

S60 uses the notifier framework to provide a dialog from which the user can select a nearby Bluetooth device. The application connects to the extended notifier server with `RNotifier`. It then calls `RNotifier::StartNotifierAndGetResponse()`, which takes four arguments:

[illegible]

- `aRs` is completed when the notifier is dismissed or an error occurs.
- `aNotifierUid` identifies the type of notifier which the application wants to display. In this case, `KDeviceSelectionNotifierUid` raises a dialog that asks the user to select a Bluetooth device.
- `aBuffer` contains data which the application uses to initialize the notifier. An instance of `TBTDeviceSelectionParamsPckg` is used to filter the devices according to whether they support a given service, although this may not be supported by the UI implementation.
- `aResponse` is populated by the notifier to return information to the application. The Bluetooth device selector expects an instance of `TBTDeviceResponseParamsPckg`, in which it stores the selected device's address.

The client application uses the server's device address to search its SDP database for the service's protocol descriptor list, which tells the client how to connect to it. (This stage is unnecessary if the client and server agree in advance to use a hard-coded L2CAP or RFCOMM channel, but this can clash with other applications.)

```
void CBTClientToServer::AskUserToSelectHostL(TBTDevAddr& aDevAddr)
{
    TInt r;

#ifdef __SERIES60_3X__
    // ask user to select a device via the extended notifier server
    RNotifier ntf;
    r = ntf.Connect();
    User::LeaveIfError(r);

    TRequestStatus rs;

    // filter the displayed devices by those which support the
    // OandX service. (This may not be supported by the UI.)
    TBTDeviceSelectionParamsPckg devFilter;
    devFilter().SetUUID(iServiceUuid);

    TBTDeviceResponseParamsPckg response;

    ntf.StartNotifierAndGetResponse(rs, KDeviceSelectionNotifierUid,
                                     devFilter, response);

    User::WaitForRequest(rs);
    ntf.Close();

    // ensure a valid device was selected
    r = rs.Int();
    if (r == KErrNone && ! response().IsValidDeviceName())
        r = KErrNotFound;
    User::LeaveIfError(r);

    aDevAddr = response().BDAddr();
#endif
}
```

```
#else
// UIQ dialog code
```

Asking the user to select a remote device on UIQ

UIQ defines a class, `CQBTUISelectDialog`, which displays a dialog from which the user can select a device. The dialog populates an instance of `CBTDeviceArray`, which is an array of `CBTDevice` objects, each of which describes a selected device.

```
typedef CArrayPtrFlat<CBTDevice> CBTDeviceArray;
```

The application allocates an instance of `CQBTUISelectDialog` and calls its `RunDlgLD()` member function.

```
CBTDeviceArray* btDevArray = new (ELeave)CBTDeviceArray(1);
BTDeviceArrayCleanupStack::PushL(btDevArray);

CQBTUISelectDialog* btUiSelDlg = CQBTUISelectDialog::NewL(btDevArray);
TInt dlgRet = btUiSelDlg->RunDlgLD(KQBTUISelectDlgFlagNone);
```

`PushL()` is the only public function in `BTDeviceArrayCleanupStack`. It ensures the instances of `CBTDevice` are deleted along with the array.

The value passed to `RunDlgLD()` is a bitmask composed from values in `TQBTUISelectDialogFlags`, where `KQBTUISelectDlgFlagNone` means ‘select a single device’. The return value is from the `TBTSelectControlsAndButtons` enumeration set.

If the user selects a device, the application can extract its address by using `CBTDevice` in a similar way to `TBTDeviceResponseParams`, which is packaged by `TBTDeviceResponseParamsPckg` in the S60 code.

```
if (dlgRet != EBTDeviceSelected)
    r = KErrNotFound;
else
{
    const CBTDevice* dev = (*btDevArray)[0];
    if (! dev->IsValidBDAddr())
        r = KErrNotFound;
    else
    {
        aDevAddr = dev->BDAddr();
        r = KErrNone;
    }
}

User::LeaveIfError(r);
CleanupStack::PopAndDestroy(btDevArray);
```

Searching for services on a remote device

To search the remote device's SDP database, the client allocates an instance of `CSdpAgent`. It supplies an implementation of `MSdpAgentNotifier` and the server device's address. The former is an observer interface that is called back for each record in the SDP database.

To limit the search to records that match a given service ID, the application can additionally allocate an instance of `CSdpSearchPattern`, which it supplies to the agent object.

The application starts the search by calling `CSdpAgent::NextRecordRequestL()`.

```
iSdpAgent = CSdpAgent::NewL(/* MSdpAgentNotifier& */ *this, aDevAddr);

// only process SDP entries which match the OandX service class
CSdpSearchPattern* searchPattern = CSdpSearchPattern::NewL();
CleanupStack::PushL(searchPattern);
searchPattern->AddL(iServiceUuid);
iSdpAgent->SetRecordFilterL(*searchPattern);
CleanupStack::PopAndDestroy(searchPattern);

iSdpAgent->NextRecordRequestL();
```

The implementation of `MSdpAgentNotifier::NextRecordRequestComplete()` should parse the record's attributes by calling `CSdpAgent::AttributeRequestL()`. It calls this function with the record's identifier, and the ID of the attribute in which it is interested, which is the protocol descriptor list in this case.

```
void CBtClientToServer::NextRecordRequestComplete(TInt aError,
    TSdpServRecordHandle aHandle, TInt aTotalRecordsCount)
{
    if (aError == KErrNone)
    {
        TRAP(aError, iSdpAgent->AttributeRequestL(aHandle,
            KSdpAttrIdProtocolDescriptorList));
    }

    if (aError != KErrNone)
        FinishedSearching(aError);
}
```

The framework calls the implementation of `MSdpAgentNotifier::AttributeRequestResult()`, to which it passes a pointer to an instance of `CSdpAttrValue`, which represents the attribute. The implementation of `AttributeRequestResult()` parses the attribute to extract the protocol information, i.e. how to connect to the service. It should delete the attribute object once it has finished with it. `CSdpAttrValue::AcceptVisitorL()` takes a reference to an implementation

of `MSdpAttributeValueVisitor` which is called back with the attribute's values.

```
void CBTClientToServer::AttributeRequestResult(
    TSdpServRecordHandle aHandle,
    TSdpAttributeID aAttrID,
    CSdpAttrValue* aAttrValue)
{
    TRAPD(r, aAttrValue->AcceptVisitorL(
        /* MSdpAttributeValueVisitor& */ *this));
    if (r != KErrNone)
        FinishedSearching(r);

    delete aAttrValue;
}
```

The implementation of `MSdpAttributeValueVisitor::VisitAttributeValueL()` reads the protocol type (RFCOMM or L2CAP) as a UUID and the channel number as an integer.

```
void CBTClientToServer::VisitAttributeValueL(CSdpAttrValue& aValue,
    TSdpElementType aType)
{
    switch (aType)
    {
        case ETypeUUID:
        {
            const TUUID protocolUuid = aValue.UUID();
            if (protocolUuid == TUUID(KL2CAP))
                iProtocolName = &KL2CAPDesC;
            else if (protocolUuid == TUUID(KRFCOMM))
                iProtocolName = &KRFCOMMDesC;
            else
                User::Leave(KErrNotSupported);
        }
        break;
        case ETypeUint:
            iProtocolChannel = aValue.Uint();
            break;
        default:
            // ignore other attribute types.
            break;
    }
}
```

To summarize, the client allocates an instance of `CSdpAgent` to parse the server's SDP database. It filters the records by service ID with an instance of `CSdpSearchPattern`. It implements `MSdpAgentNotifier` to be called back when the agent finds a record.

The implementation of `MSdpAgentNotifier` extracts the required attribute within the record by calling `CSdpAgent::AttributeRequestL()` with the attribute's ID. It then parses the contents of that attribute (which may be a list or some other Bluetooth attribute type) by calling `CSdpAttrValue::AcceptVisitorValueL()` with an instance of `MSdpAttributeValueVisitor`.

The implementation of `MSdpAttributeValue::VisitAttributeValueL()` extracts the required values, such as UUIDs and integers from within the attribute.

Connecting to a remote service

Once the client has discovered how to connect to the remote service, typically by parsing the `ProtocolDescriptorList` attribute, it then has to make the actual connection.

It constructs the service's Bluetooth address from the server's device address and the protocol channel. The client can also set a security policy for this connection, and so can the server for its listen socket. Once the Bluetooth address has been set up, the application then passes it to `CBluetoothSocket::Connect()`:

```
// open the socket and connect it to the remote device
iBtSocket = CBluetoothSocket::NewL(*this, iSocketServ, *iProtocolName);

TBTSockAddr btSockAddr;
btSockAddr.SetBTAddr(devAddr);
btSockAddr.SetPort(iProtocolChannel);

// channel security settings. No authentication, authorisation
// or encryption.
TBTSecurity oandxSecurity;
oandxSecurity.SetUid(KUidServiceSDP);
oandxSecurity.SetAuthentication(EFalse); // do not require key
                                         // (PIN) exchange
oandxSecurity.SetAuthorisation(ETTrue);  // require local user
                                         // to confirm accept

oandxSecurity.SetEncryption(EFalse);
oandxSecurity.SetDenied(EFalse);
btSockAddr.SetSecurity(oandxSecurity);

r = iBtSocket->Connect(btSockAddr);
User::LeaveIfError(r);
```

The Bluetooth stack notifies the application when the connection has completed, successfully or otherwise, by calling its implementation of `MBluetoothSocketNotifier::HandleConnectComplete()`. From then on the client and service are connected and they can exchange data.

20.5 Messaging

The Message Server

The messaging application on a Symbian OS phone gives the user access to messages that are owned by the message server. SMS, MMS and email

are all examples of messages which are managed by the server. The message server is a Symbian OS server, so applications communicate with it via a session.

The messaging framework supports a diverse range of message types with a set of plug-in DLLs called Message Type Modules (MTMs.) Each message type has its own UID. For example, `KUidMsgTypeSMS`, which is defined in `smut.h`, identifies short messages, and `KUidMsgTypeMultimedia`, which is defined in `mmsconst.h`, identifies multimedia messages.

Each MTM provides a set of classes for applications to construct and send a message, and to bring up an editor with which the user can modify the message. Because each message type is handled by an MTM, the messaging system is both extensible, because new message types can be added as MTM plug-ins, and promotes code reuse, because the same or similar code can construct and parse different types of message. From an application's point-of-view, it may be simpler to use the Send As server,¹¹ rather than use the MTMs directly. If the application wants the user to select a transport or to edit the message, it can use `SendUi` on `S60` or `Send As` classes, such as `CQikSendAsDialog`, on `UIQ`.

The Message Tree

Symbian OS organizes messages into a tree structure. The higher-level entries broadly correspond to folders that are visible from the messaging application, such as `Inbox`, `Sent` and `Drafts`. Incoming SMS or MMS messages appear in the inbox and an incoming email is stored in that account's entry.

Applications that use the Send As server to send messages do not generally need to know about the message tree, beyond the fact that saving a message stores it in the `Drafts` folder. Applications that handle incoming messages may need to identify the parent entry.

Sending a Message with the Send As Server

To send a message, an application connects to the Send As server with an instance of `RSendAs`. It then uses `RSendAsMessage` to construct an individual message. Once constructed, the message can be sent to a remote phone or saved to the `Drafts` folder for later processing.

The following example is taken from the Noughts and Crosses application. First, create a session with the Send As server (`iSendAs` is an instance of `RSendAs`):

¹¹ `RSendAs` and `RSendAsMessage` were introduced in Symbian OS v9.1 and replace `CSendAs`.

```
User::LeaveIfError(iSendAs.Connect());
```

Secondly, open a subsession and create a message, supplying the message type UID:

```
iSendAsMessage.CreateL(iSendAs, iSendMtm);
iSendAsMessage.AddRecipientL(*iRemoteAddress,
    RSendAsMessage::ESendAsRecipientTo);
```

- `iSendAsMessage` is an instance of `RSendAsMessage`.
- `iSendMtm` is a message type UID.
- `*iRemoteAddress` is a descriptor whose contents are interpreted according to the message type. For example '07712345678' would be appropriate for an SMS, created with `KUIdMsgTypeSMS`, and 'abc@def.com' would be appropriate for an email, created with `KUIdMsgTypeSMTP`.

Thirdly, populate the message as required. This could involve adding recipients, subject text, body text and attachments. The following examples are used to build an email message and assume the recipient has already been set as above.

```
iSendAsMessage.SetBodyTextL(aPayloadText);
```

`aPayloadText` is a descriptor which contains the email body. In the Noughts and Crosses application, it is the location of the selected tile.

```
_LIT(KEmailSubjectLine, "OandX move via email");
iSendAsMessage.SetSubjectL(KEmailSubjectLine);
```

Not all message types support all operations. For example, calling `SetSubjectL` for an SMS leaves with `KErrNotSupported`.

Finally, send the message or save it to the Drafts folder. The following code sends the message asynchronously:

```
iSendAsMessage.SendMessage(iStatus);
```

Overloads are available to send the message synchronously, to ask the user to confirm the sending and to close the `RSendAsMessage` subsession when the message has been sent. If the application does not have the required capabilities to send the message, then the user is prompted to confirm the sending, irrespective of whether the application

asks them to confirm the operation, e.g. by calling `RSendAsMessage::SendMessageConfirmed()`. If the user declines to send the message, the send operation is still completed with `KErrNone`.

An application needs the `NetworkServices` capability to send data via the telephone network; if it does not have it, the user is asked to confirm every message that is sent.

Receiving a Message

An application can register to be notified when entries are added, changed or deleted. The application implements `MMsvSessionObserver` and opens a session with the message server. If the session is only used to get event notifications, as opposed to modifying the message store, for example by creating a message, then it can be created with `CMsvSession::OpenAsObserverL()` instead of `OpenSyncL()` or `OpenAsyncL()`.

```
if (aObserverSession)
    iMsvSession = CMsvSession::OpenAsObserverL(*this);
else
    iMsvSession = CMsvSession::OpenSyncL(*this);
```

- `aObserverSession` is an instance of `TBool` which describes whether this session can be opened as an observer only.
- `iMsvSession` points to an instance of `CMsvSession`.
- `*this` implements `MMsvSessionObserver`. This interface declares a single function `HandleSessionEventL()` that takes an operation ID followed by three `TAny*` arguments.

```
virtual void HandleSessionEventL(TMsvSessionEvent aEvent,
                                TAny* aArg1, TAny* aArg2, TAny* aArg3) = 0;
```

When an application implements this function, it should silently ignore event codes that it does not recognize, and otherwise cast the `TAny*` arguments to the appropriate type. The documentation for `MMsvSessionObserver`¹² lists the supported message codes along with the required casts.

Receiving a message is not an atomic operation, so `EMsvEntriesCreated` may be followed by a number of `EMsvEntriesChanged` events as the full message is downloaded and constructed. In the

¹² See details in the Developer Library at » Symbian OS reference » C++ component reference » Messaging Framework » `MMsvSessionObserver`.

Noughts and Crosses application, `CMessagingTransport::HandleSessionEventL()` implements the function.

A message can appear when it is saved to the Drafts folder, or when it is sent, which creates a message in the Outbox folder that is later moved to the Sent folder. The Noughts and Crosses application does not process messages which appear in these folders.

```
// ignore messages which are being sent or created
TMsvId parentEntry = *reinterpret_cast<TMsvId*>(aArg2);
if (parentEntry == KMsvGlobalOutBoxIndexEntryId ||
    parentEntry == KMsvSentEntryId ||
    parentEntry == KMsvDraftEntryId)
    return;
```

From the arguments, it gets an instance of `CMsvEntrySelection`, which contains an identifier for each added or changed entry. The code iterates through the entries, searching for payload text, which was sent from the remote phone.

To extract the payload text, the application gets the message's entry data. An instance of `CMsvEntry` provides access to further information about the message, such as its type and content.

```
CMsvEntry* msve = iMsvSession->GetEntryL(aEntryId);
```

The Noughts and Crosses application tests if the incoming message's type matches the type of transport that is being used to conduct the game. (It does not check if the source matches the phone or account to which the original message was sent.) It also ensures that the message has a store. A message store, which the client can access via an instance of `CMsvStore`, provides access to the message's content, such as body text or attachments. As described in the system documentation, client applications should generally create and retrieve message content with `CMsvEntry` and the relevant MTMs. However, `CMsvStore` is used in a few cases, such as adding attachments to an MMS.

```
if (!(ShouldUseReceivedMtmUid(msve->Entry().iMtm) && msve->HasStoreL()))
    User::Leave(KErrOandXMessageNotFound);
```

The Noughts and Crosses application declares `ShouldUseReceivedMtmUid()` as a pure virtual function in `CMessageTransport`. It takes a message type UID and decides whether it matches the type of message that was sent. This cannot be implemented as a simple comparison with `iSendMtm`, which was used with `RSendAsMessage::CreateL()` above, because a transport may use different MTMs

to send and receive messages. For example, email is sent with `KUIdMsgTypeSMTP`, but received with `KUIdMsgTypePOP3` or `KUIdMsgTypeIMAP4`.

Exactly how the payload text is stored varies between message types. SMS and email both have body-text sections, but MMS stores the text as an attachment. Like `ShouldUseReceivedMtmUid()`, this is delegated to a virtual function which is implemented in a subclass.¹³

```
CMsvStore* msvs = msve->ReadStoreL();
CleanupStack::PushL(msvs);

// extract the payload from body text or an attachment
HBufC* plainText = ExtractPlainTextLC(*msvs);
```

Finally, the Noughts and Crosses application searches for the payload text within the extracted buffer. It searches within the buffer instead of comparing it directly against the expected payload format because the text may have been embellished before the phone receives it. For example, the remote player's email server may have appended a disclaimer or an advertisement.

```
TInt prefixPos = plainText->Find(KMtPylPrefix);
if ((prefixPos == KErrNotFound) ||
    (prefixPos + KMtPylPrefixLen + KPayloadLen > plainText->Length()))
    User::Leave(KErrOandXMessageNotFound);
```

The above code demonstrates how different message types – email, SMS, and MMS – can be handled with common code much of the time. Nonetheless, the underlying differences, such as support for body text, mean they necessarily differ in some cases.

An application requires the `ReadUserData` capability to read messages.

When an SMS or an MMS is sent on the S60 v3 emulator, it stores a representation of the message in a directory on the emulated C drive. By default, the emulated directories are `c:\smsout` and `c:\mmsout`, although these can be changed with the emulator's Utilities settings. The emulator also defines two directories – `c:\smsin` and `c:\mmsin` – into which files can be dropped to simulate receiving a message.

¹³ The Noughts and Crosses application uses the term 'plaintext' to mean the text in a message body or attachment after formatting has been removed. It is not used in the cryptographic sense of 'unencrypted'.

Email

Three MTMs support email: SMTP constructs and sends outgoing emails; POP3 and IMAP4 read incoming emails.

Before the application can use email as a transport, the user must have configured at least one email account in the messaging application. When `RSendAsMessage::CreateL()` is called with `KUIdMsgType-SMTP`, it uses the default email account, as configured in the messaging application.

When an application constructs an email with `RSendAsMessage`, it can add recipients, a subject, body text and attachments. `RSendAsMessage::AddRecipientL()` takes a recipient's address and type (To, CC or BCC):

```
IMPORT_C void AddRecipientL(const TDesC &aAddress,
                           TSendAsRecipientType aRecipientType);
```

Although email supports all three types of recipients, other transports, such as SMS, do not. `RSendAsMessage::ESendAsRecipientTo` can be used for both types of message.

`RSendAsMessage` can be used to set rich body text:

```
IMPORT_C void SetBodyTextL(const CRichText &aBody);
```

SMS does not support rich text, so the Noughts and Crosses application uses unformatted text as a common denominator:

```
IMPORT_C void SetBodyTextL(const TDesC &aBody);
```

These two examples demonstrate that although email supports richer functionality than SMS, it is sometimes better to use a common denominator where the additional features are not required, because it enables greater code reuse.

The message's body text is extracted from its store. The following code retrieves the body text from an SMS or from an email. The text is retrieved from the store as rich text, even for an SMS.

```
HBufC* CBodyTextTransport::ExtractPlainTextLC(CMsvStore& aStore) const
{
    if (! aStore.HasBodyTextL())
        User::Leave(KErrOandXMessageNotFound);

    // the body text is extracted as rich text and then
    // converted to plaintext.
    CParaFormatLayer* pfl = CParaFormatLayer::NewL();
```

```

CleanupStack::PushL(pfl);
CCharFormatLayer* cfl = CCharFormatLayer::NewL();
CleanupStack::PushL(cfl);
CRichText* rt = CRichText::NewL(pfl, cfl);
CleanupStack::PushL(rt);
aStore.RestoreBodyTextL(*rt);

// the document length includes non-printing characters, and
// so establishes an upper limit on the number of plaintext
// characters.
TInt docLen = rt->DocumentLength();
HBufC* plainText = HBufC::NewL(docLen);
TPtr ptDes = plainText->Des();
rt->Extract(ptDes);

CleanupStack::PopAndDestroy(3, pfl);
CleanupStack::PushL(plainText);

return plainText;
}

```

Once the application has the body text, it can process it as required. The Noughts and Crosses application searches for the expected payload text that describes the remote player's move.

Short Messaging Service (SMS)

From the perspective of a developer working with the messaging framework, short messages can be considered as a simplified version of email. They only support 'To' recipients, with no subject field. For this reason, the SMS and email transports in the Noughts and Crosses application share most of their logic, except that short messages do not support a subject field. Notice that the body text is extracted as rich text, just as it is for email.

Multimedia Message Service (MMS)

Multimedia messages include a subject line and recipients but, unlike email and SMS, they do not support body text. Their contents, such as text, music and graphics, are added as attachments.

Multimedia messages can be sent in two ways on Symbian OS. An application can use the Send As server, which provides the same API that was used to construct SMS and email messages. The Send As server can construct messages with attachments, but it cannot organize how those attachments are presented to the user. For example, it can add an image and a text file to the message, but it cannot tell the remote phone to display the image for three seconds followed by the text for three seconds.

An application can control how a multimedia message is displayed if it constructs the message with the MMS client MTM. This, however, can be more complex than using the Send As server.

Sending multimedia messages with the Send As server

A multimedia message has subject and recipient fields, just like an email, so `RSendAsMessage::CreateL()` and `RSendAsMessage::AddRecipientL()` can be used here.

The application cannot use `RSendAsMessage::SetBodyTextL()`. Instead it must use `AddAttachmentL()` or `CreateAttachmentL()` to add the text and any other content such as an image as an attachment. `AddAttachmentL()` adds a file that already exists, which makes it useful for static content such as an image. All of its overloads in `RSendAsMessage` are asynchronous, so they must be handled with active objects. `CreateAttachmentL()` creates a new file into which the application can write data. This makes it useful for dynamically generated content. In the Noughts and Crosses application, it is used to write the payload data, i.e. which tile the local player has selected.

```
RFile textFile;
iSendAsMessage.CreateAttachmentL(KPayloadAttachmentName, textFile,
                                  /*aMimeType*/ KMmsTextPlain);
CleanupClosePushL(textFile);

HBufC8* framedPayload =
    MmsUtils::BuildFramedPayloadLC(aPayloadText);

TInt r = textFile.Write(*framedPayload);
if (r == KErrNone)
    r = textFile.Flush();
User::LeaveIfError(r);

CleanupStack::PopAndDestroy(2, &textFile);    // framedPayload
```

`MmsUtils` is a class defined by the Noughts and Crosses application which converts the native-width payload text into an 8-bit descriptor.

Sending multimedia messages with the MMS Client MTM

A Synchronized Multimedia Integration Language (SMIL, pronounced 'smile')¹⁴ document describes how media objects¹⁵ in a multimedia message are presented to the user. The following is used to lay out the MMS which contains a player's move:

¹⁴ www.w3.org/TR/SMIL

¹⁵ A 'media object' is something which can be rendered in a presentation, such as an image, text or an audio clip.

```

<smil>
  <head>
    <layout type="text/smil-basic-layout">
      <region id="r1" fit="fill" title="main region" />
    </layout>
  </head>
  <body> <!-- this forms an implicit sequence -->
    
    <text src="payload.txt" region="r1" type="text/plain" dur="3s" />
  </body>
</smil>

```

This document displays a graphic of a noughts-and-crosses board for three seconds and then displays the move text for three seconds.

This document must be the message's root attachment. (In this context, 'root' identifies the layout document. The application does not have to organize the attachments into a tree structure and it is unrelated to the Symbian OS message tree.) This cannot be set with `RSendAsMessage`, so the application must use the client MTM, `CMmsClientMtm` directly.

Symbian OS does not define an MMS MTM. At the time of writing, both S60 and UIQ support an MMS MTM. The Noughts and Crosses application uses the implementation provided by the S60 3rd Edition MR SDK. Similar APIs are available in the M600i and P990 extension packages for UIQ. With the release of the P910 on UIQ 2.1 and Symbian OS 7.0, Sony Ericsson provided an MMS MTM that was largely source-compatible with Nokia's S60 implementation. Both interfaces have been revised for Symbian OS v9.1 (which forms the basis of S60 v3 and UIQ 3) and they remain broadly compatible.

Before it can allocate an instance of the client MTM, the application must first allocate an instance of `CClientMtmRegistry`. This is a factory object that allocates the required MTM.

```

iClientMtmRegistry = CClientMtmRegistry::NewL(*iMsvSession);
iMmsClientMtm = static_cast<CMmsClientMtm*>(
    iClientMtmRegistry->NewMtmL(KUidMsgTypeMultimedia));

```

- `iClientMtmRegistry` points to an instance of `CClientMtmRegistry`.
- `iMsvSession` points to a message server session, an instance of `CMsvSession`.
- `iMmsClientMtm` points to an instance of `CMmsClientMtm`. This is a subclass of `CBaseMtm`, a pointer to which is returned by `NewMtmL()`.

The MTM identifier passed to `NewMtmL()` is the same UID which is passed to `RSendAsMessage::CreateL()`.

An MTM supports a ‘context’ which identifies the current entry. The following creates a new, draft MMS:

```
iMmsClientMtm->SwitchCurrentEntryL(KMsvDraftEntryId);

// create a new message using the MMS service
TMsvId serviceId = iMmsClientMtm->DefaultServiceL();
iMmsClientMtm->CreateMessageL(serviceId);
```

From then on, the MTM provides functions such as `AddAddresseeL()` and `SetSubjectL()`, which operate on the current entry in a similar way to their equivalents in `RSendAsMessage`. After it has set the subject and address field, the application should call `SaveMessageL()`, which is declared in the `CBaseMtm` superclass:

```
iMmsClientMtm->AddAddresseeL(*iRemoteAddress);
iMmsClientMtm->SetSubjectL(KMmsSubjectLine);
iMmsClientMtm->SaveMessageL();
```

Having saved the context, the application adds attachments to the message store, before calling `CMsvStore::CommitL()`.

```
// add the graphic, payload text and SMIL document as attachments
CMsvEntry& mtmEntry = iMmsClientMtm->Entry();
CMsvStore* s = mtmEntry.EditStoreL();
CleanupStack::PushL(s);

AddFileAttachmentL(*s, KImageAttachmentName, KImageAttachmentMimeType8);
AddTextAttachmentL(*s, aPayloadText, KPayloadAttachmentName);
TMsvAttachmentId smilId = AddFileAttachmentL(*s, KSmilFileName,
                                             KMmsApplicationSmil);

// SMIL document must be root
iMmsClientMtm->SetMessageRootL(smilId);

s->CommitL();
CleanupStack::PopAndDestroy(s);
```

`AddFileAttachmentL()` and `AddTextAttachmentL()` are helper functions defined by the Noughts and Crosses application in `CMmsTransport`. `CMmsClientMtm` implements `AddAttachmentL()` and `CreateTextAttachmentL()`, which both provide a straightforward way to create an attachment. However, they do not allow the caller to specify the attachment’s MIME headers. Specifically, the content-location header (see [IETF 1999, Section 4]) is not set to the attachment’s

filename. This matters because in the SMIL file, the media object's `src` attribute¹⁶ refers to the attachment's content-location or content-ID value, both of which are MIME headers.

Referring back to the example SMIL document,

```
<text src="payload.txt" region="r1" type="text/plain" dur="3s" />
```

Here `payload.txt` identifies which media object should be displayed. The section of the message that contains the text file should have 'payload.txt' as its content-location value. The application sets this value by calling `CMmsClientMtm::CreateMessage2L()`.

```
virtual void CreateAttachment2L(CMsvStore& aStore,
                                RFile& aFile, TDesC8& aMimeType,
                                CMsvMimeHeaders& aMimeHeaders,
                                CMsvAttachment* aAttachmentInfo,
                                TMsvAttachmentId& aAttaId);
```

- `aStore` is the message entry's store, which the application uses to manage attachments.
- `aFile` is an open handle to the attachment file. The application should not expect the MTM to seek to the start of this file, so it must ensure the file is rewound before calling `CreateAttachment2L()`.
- `aMimeType` is the value of the content-type MIME header, such as 'text/plain' or 'image/gif'.¹⁷
- `aMimeHeaders` is used by the application to set the required MIME headers. The Noughts and Crosses application sets the content-location value to the identifier that is used in the SMIL document.
- `aAttachmentInfo` describes the attachment in some detail. Although it is a required argument, it is only used in this case to specify that the attachment is a file. The MTM takes ownership of this object, so the application must not delete or otherwise use it after calling `CreateAttachment2L()`.
- `aAttaId` is set to the newly-created attachment's identifier. After the application has added the SMIL document, it uses this value to specify the new attachment as the 'root' attachment.

The following example from the Noughts and Crosses application demonstrates this API in use:

```
TMsvAttachmentId CMmsTransport::AddFileAttachmentL(CMsvStore& aStore,
```

¹⁶ Synchronized Multimedia Integration Language (SMIL 2.1), §7.4.1.

¹⁷ Defined MIME types are listed at www.iana.org/assignments/media-types.

```

                                RFile& aFile, const TDesC&
                                aContentLocation, const TDesC8& aMimeType)
{
    CMsvMimeHeaders* headers = CMsvMimeHeaders::NewLC();
    headers->SetContentLocationL(aContentLocation);

    CMsvAttachment* attach =
        CMsvAttachment::NewL(CMsvAttachment::EMsvFile);
    CleanupStack::PushL(attach);

    // Construct a non-const TDesC8 object for the MIME type.
    // This is necessary because CreateAttachment2L takes a
    // TDesC8& instead of a const TDesC8&.
    TPtrC8 mt8(aMimeType);

    TMsAttachmentId id; // CreateAttachment2L sets this to the
                        // new attachment's ID
    iMmsClientMtm->CreateAttachment2L(aStore, aFile, mt8,
                                    *headers, attach, id);
    CleanupStack::Pop(attach); // ownership transfers to MTM
    CleanupStack::PopAndDestroy(headers);

    return id;
}

```

Once the attachments have been added, the application sets the message's root attachment with `CMmsClientMtm::SetMessageRootL()`, passing in the attachment identifier that was set by `CreateAttachment2L()`.

```

TMsAttachmentId smilId = AddFileAttachmentL(*s, KSmilFileName,
                                           KMmsApplicationSmil);
iMmsClientMtm->SetMessageRootL(smilId); // SMIL document must be root

```

When a message is created with an MTM, it is marked as invisible and in preparation. These must be reversed before the message is sent.

```

TMsEntry e = mtmEntry.Entry();
e.SetVisible(ETTrue);
e.SetInPreparation(EFalse);
mtmEntry.ChangeL(e);

```

Sending a message is an asynchronous operation. `CMmsClientMtm::SendL()` returns an instance of `CMsvOperation`, which an application can use to track the operation's progress. This object can also be used to cancel the sending operation while it is in progress. The application must delete this object when it is no longer required.

An application requires the `WriteUserData` capability to modify the message store directly and `NetworkServices` capability to send the message via an MTM.

Receiving a multimedia message

When the phone receives a multimedia message, the Noughts and Crosses application extracts the payload from the text-file attachment. The process is the same whether the message was sent with the Send As server or with the client MTM.

The application gets the message's attachment manager. This is an implementation of `MMsvAttachmentManager` that allows it to retrieve each attachment's MIME headers and content.

Each individual attachment is accessed with an instance of `CMsvAttachment`, which the Noughts and Crosses application uses to determine if the attachment is the payload text file.

```

HBufC* MmsUtils::ExtractPlainTextLC(CMsvStore& aStore)
{
    MMsvAttachmentManager& attachMan =
        aStore.AttachmentManagerL();
    TInt attachCount = attachMan.AttachmentCount();

    for (TInt i = 0; i < attachCount; ++i)
    {
        // is this a text file attachment with the expected name?
        CMsvAttachment* msva = attachMan.GetAttachmentInfoL(i);
        TBool payloadFile =
            msva->Type() == CMsvAttachment::EMsvFile
            && msva->MimeType() == KMmsTextPlain
            && msva->AttachmentName() == KPayloadAttachmentName;
        delete msva;

        if (payloadFile)
            return ExtractPlainTextFromFileLC(attachMan.GetAttachmentFileL(i));
    }

    // payload not found in any attachment
    User::Leave(KErrOandXMessageNotFound);
    return NULL;    // avoid "return value expected" warning
}

```

Note that the attachment name in this case is not the same as the content location header value that was set explicitly with the MTM. In practice, they may have the same value, e.g. "payload.txt", but they are two distinct entities. `RSendAsMessage` can be used to set the attachment name, but not the content-location MIME header that is used by the SMIL document.

Once the application has found the attachment, it calls `MMsvAttachmentManager::GetAttachmentFileL()` which returns an open file handle. This is an effective but slightly unusual way to open a resource on Symbian OS and the application must remember to close the file handle when it is no longer required. Given the file handle, the application can read its contents, as it would for any file.

```

HBufC* MmsUtils::ExtractPlainTextFromFileLC(RFile aFile)
{
    CleanupClosePushL(aFile);

    TInt fileSize;
    User::LeaveIfError(aFile.Size(fileSize));

    HBufC8* contents8 = HBufC8::NewLC(fileSize);
    TPtr8 ptr8 = contents8->Des();
    User::LeaveIfError(aFile.Read(ptr8));

    HBufC16* contents =
        CnvUtfConverter::ConvertToUnicodeFromUtf8L(*contents8);

    CleanupStack::PopAndDestroy(2, &aFile); // contents8, aFile
    CleanupStack::PushL(contents);

    return contents;
}

```

20.6 Security

Although the wide range of connectivity options that are available in a modern smartphone enable rich, valuable applications, they also create security risks. An application receives and acts on information from remote and possibly untrusted devices. It should validate information that is supplied from an external source.

Data Validation

Smartphone security is discussed at length in [Heath 2006]. Here we look at a specific attack vector and how it is handled in the Noughts and Crosses application.

The Noughts and Crosses application receives a single wide character from the remote phone.¹⁸ This character should be between ‘0’ and ‘8’ inclusive and identify the tile that the remote user has selected. When it receives that character, the application converts it to an integral value between 0 and 8 and marks the corresponding tile with the remote player’s symbol.

The tile values are stored in an array. The attacker therefore supplies an array offset. Without validation, this would allow the attacker to write a known (but not selected) 4-byte value into a memory location of their choice within 256 KB¹⁹ around the start of the tile array.

¹⁸ The character is an unsigned 16-bit value, although it probably enters the device as an 8-bit value.

¹⁹ The array, `COandXEngine::iTileStates`, is a sequence of `TInt` values, so the range is `0x10000 * sizeof(TInt) = 256 KB`.

Symbian OS protects applications' memory space in several ways. It imposes restrictions on executables which run in user-mode such as applications.

- An executable cannot see heaps and stacks for other processes. In the moving-memory model, it can see all loaded code. In the multiple-memory model, which requires an ARMv6 processor, it can see execute-in-place (XIP) code²⁰ and other DLLs which it has loaded, but not DLLs which have been loaded by other processes.
- An executable cannot access memory-mapped IO.
- An executable cannot modify code that has been loaded.

Where MMU support is available, stack and heap memory are marked as non-executable. This also requires ARMv6. These and other security measures are discussed further in [Sales 2005, Chapter 7].

The application can also use Symbian OS constructs, such as descriptors, and containers, such as `RArray<T>`, all of which provide bounds checking. Note that `TFixedArray<T, S>` provides debug-mode checking for operator `[]` and always checks `At()`.

Although these safeguards are valuable, they are the last line of defense. While it is better to kill a compromised or defective application than to let it continue, it does not provide a good user experience.

The Noughts and Crosses application detects the invalid index in `COandXController::ReceivedPayload()` with a simple range comparison. If the index is invalid, then it alerts the user and terminates the current game.

On its own, bounds checking is not sufficient – the tile may already contain a nought or a cross. `ReceivedPayload()` also tests for this case. Use the Send Index menu item to send an arbitrary value to the remote phone. The value can be negative, greater than the size of the array, or describe a tile that has already been selected. This menu option can only be used when it is the local phone's turn to send an index:

```
void COandXController::ReceivedPayload(const TDesC& aPayload)
{
    __ASSERT_DEBUG(iState == EWaitRemoteMove, Panic(ERpBadState));

    // extract the tile number from the payload
    TInt tileIndex = aPayload[0] - '0';

    TBool validTile = (tileIndex >= 0) && (tileIndex < KNumberOfTiles);
    COandXEngine& eng = Engine();
    validTile = validTile && eng.SquareStatus(tileIndex) == ETileBlank;
```

²⁰ Execute-in-place code runs from a fixed memory location in the same way as a traditional ROM image. This contrasts with non-execute-in-place code which is loaded from executable files at run time, and whose code and data addresses are 'fixed up' by the operating system.


```
if (! validTile)
{
    _LIT(KInvalidMsg, "Received invalid tile index.");
    TerminateGame(KInvalidMsg);
}
else
{
    // handle valid move
}
}
```

By validating the local user's input, the sending application should make it difficult – ideally impossible – to send invalid data. For example, it should not allow the user to select a tile which already contains a symbol. Independent of security considerations, an application provides a better user experience if the user cannot enter invalid data.

Nonetheless, the receiving application should still validate the input that it receives. [Heath 2006, Chapter 4] discusses checking input from local users and remote sources.

Platform Security

An application requires different capabilities to use the various Symbian OS communications and connectivity APIs. This chapter has provided examples of where LocalServices, NetworkServices, WriteUserData, and ReadUserData are required.

As described in Chapter 9, an executable or DLL (A) cannot link to a DLL (B) that does not have all the capabilities which A has. (B can have additional capabilities that A does not have, but if A has a capability then B must also have that capability.)

The application executable loads the transport plug-ins into its own process with the ECOM framework. Not only does the application need of all the capabilities that any of its transport plug-ins might require, but each plug-in needs to at least match that set of capabilities, else it could not be loaded into the process. Therefore the application and all of its plug-ins need all the capabilities mentioned above.

This approach was used for the Noughts and Crosses application, to keep the emphasis on the communications APIs without adding a layer of indirection. However, it is undesirable from a security perspective, because it means that binaries are given capabilities for functionality they do not use. For example, an application does not need NetworkServices to communicate over Bluetooth. It also means that if a new transport plug-in is written which needs yet more capabilities, then the application executable and all of the existing plug-ins need to have the new capability.

This can be addressed with the Symbian OS client–server architecture. Instead of loading plug-ins that call the Symbian OS APIs directly, an application loads a plug-in that talks to a server running in a separate process. The plug-in only needs enough capabilities to be loaded into the application’s process. The server is given the capabilities that are required to use the Symbian OS APIs. Secure plug-ins are discussed at length in [Heath 2006, Chapter 6].

Summary

This chapter demonstrated how to exchange data between phones over serial cables, infrared, Bluetooth, SMS, MMS and email. It explained how applications use the communications, socket and messaging servers, which each support a range of transports. It emphasized the need to be vigilant when accepting data from untrusted sources.

Symbian OS provides the functionality to connect to services and to other users. This can be used to create richer, more valuable applications for end users.

21

Multimedia

This chapter describes the Multimedia Framework and the associated Multimedia APIs that are present in Symbian OS v9.1. These APIs can be used to play and record audio and video data, perform image processing and manipulation, and access any camera and radio tuner hardware present on a phone.

21.1 The Multimedia Framework

The Multimedia Framework (MMF) is a framework, based on ECOM plug-ins, that is used for playing back and recording audio and video clips. It allows phone manufacturers and third parties to add plug-ins to provide support for more audio and video formats. For the application programmer, it provides APIs that abstract away from the underlying hardware, thereby simplifying the code needed to record and play the supported formats. The streaming APIs, which bypass large parts of the MMF, provide a lower-level interface to allow streaming of audio data to and from the audio hardware.

The MMF is a lightweight framework that makes use of a worker thread to process the audio or video data. It makes extensive use of ECOM to allow for extensibility via plug-ins. Figure 21.1 shows its basic structure during audio- or video-clip playback or recording.

Applications use the client APIs to control the playback or recording of media. The underlying controller framework creates a new thread to process the media data and uses the Symbian client–server framework to communicate with this thread. The audio streaming APIs and the audio tone API interface directly to the MDA layer; they do not use a separate thread.

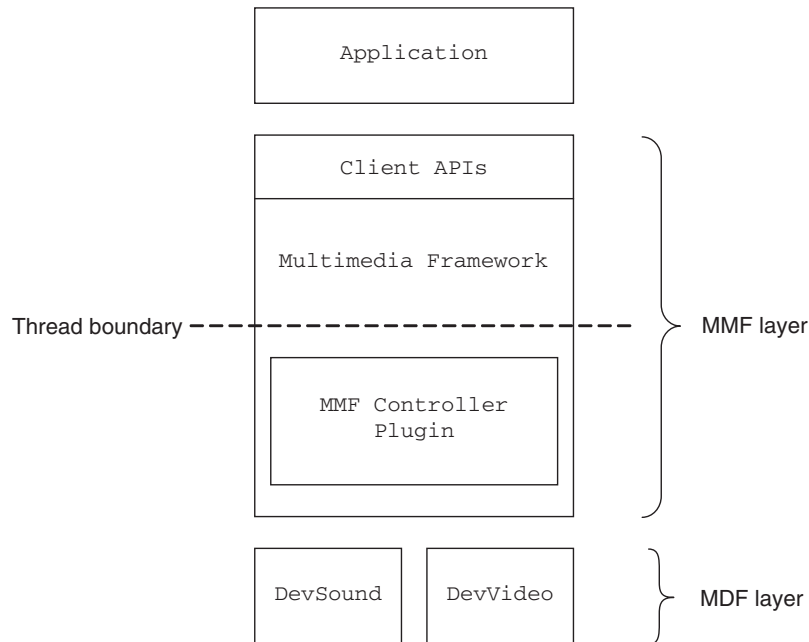


Figure 21.1 MMF configuration when using audio- or video-clip APIs

The main processing in the worker thread is done by a controller plug-in. In the case of playback, a controller plug-in is responsible for

- reading data from the source, typically either a file or a URL
- performing any processing required on the data
- writing the data to a sink, either the audio device or the screen.

In the case of recording, the data flow would be in the opposite direction.

The layer below the controller plug-in is the Media Device Framework, or MDF layer. This provides a hardware-abstraction layer that is tailored to each hardware platform on which the MMF runs. The MDF consists of DevSound, for audio input and output, and DevVideo for video.

The MDF also makes use of ECOM plug-ins that allow compressed audio or video formats to be written to, or read from, the MDF layer and allow the MMF to make use of any hardware acceleration present on a phone. These are known as hardware device plug-ins and are intended to be provided by device manufacturers to make use of specific hardware present on a phone.

The presence of these plug-ins means that it is often possible to read and write encoded audio formats directly to and from the MDF layer; most

Symbian OS v9.1 phones support adaptive multi-rate (AMR) encoding and decoding in the MDF layer and decoding of other popular formats such as MP3.

The audio policy forms a part of the MDF. It is responsible for allocating the available audio resources to applications that request them. Many phones are not capable of playing back more than one audio stream at a time and the audio policy must enforce this. The policy can pre-empt an audio playback or recording activity. This is essential if a phone call is received, for example, as any audio playback must usually be stopped before the call can begin.

The audio policy does not have a directly accessible API. You cannot directly request permission to play or record audio; the MDF layer does this for you when you attempt to play or record any such data.

Since the audio policy is part of the MDF layer, manufacturers customize it for each phone. It may therefore behave differently on different phones.

Support for Media Formats

Symbian OS v9.1 provides the ‘Symbian Audio Controller’ to support playback and recording of uncompressed Pulse Code Modulation (PCM) data. It provides playback and recording support for PCM data contained in Wav, Au and Raw files. It supports 8- and 16-bit PCM formats and the following PCM variants: IMA-ADPCM (Adaptive Differential PCM); Mu-law; A-law; and GSM 610. Conversion of these PCM variants to normal 16-bit PCM is performed in the MDF layer by a hardware device plug-in.

Phone manufacturers and third parties can supply support for other audio formats via MMF controller plug-ins. Most Symbian OS v9.1 phones provide support for playback of many popular formats such as MP3, advanced audio coding (AAC) and AMR. These compressed formats are generally much more useful due to decreased storage requirements. Since phones use the MMF for ring-signal playback and other system sounds, it is possible to use these controller plug-ins to play these sounds.

Symbian OS v9.1 does not provide any video plug-ins for playback or recording, hence these must be provided by phone manufacturers or third parties. Generally, video controller plug-ins are usually tightly integrated with the hardware on a particular phone in order to provide optimum performance and so are provided by the phone manufacturer.

Overview of APIs

The MMF client APIs can be broadly split into two types. APIs that deal with ‘clips’ are distinguished by the fact that no media data is passed

directly into the API: a filename or URL is passed in instead. The MMF controller is responsible for reading data from, or writing data to, the clip.

Streaming APIs do not deal with clips, but allow a client to pass audio data into the API, or read data directly from it. The word 'streaming' when applied to the MMF APIs is intended to mean that audio data is streamed into or out of the API. This should not be confused with streaming data over an Internet connection – something that can be done with the clip APIs, by passing in a URL.

The Observer pattern

For each API, there is a corresponding mixin class that must be implemented by a client of that API. When instantiating any of the MMF APIs, it is necessary to provide a reference to a class that implements the appropriate mixin class. This allows the MMF to notify the client of events that occur during the activity and of errors that occur.

You must have an active scheduler in your thread before using any of the MMF APIs, but this is not normally a problem since the application framework supplies every GUI application with its own active scheduler.

Many of the MMF APIs feature the letters MDA in their name, for historic reasons. Prior to Symbian OS v7.0s, all multimedia services were provided by the Media Server, abbreviated to MDA. The names remain to provide backward source compatibility with applications written for these early versions of Symbian OS.

Audio clip APIs

The two audio clip APIs are `CMdaAudioPlayerUtility` and `CMdaAudioRecorderUtility`, which provide playback and recording, respectively, of audio clips. You can use any clip format for which a controller plug-in is present on the device.

`CMdaAudioPlayerUtility` provides functionality to play back clips, either in their entirety or from a defined playback window. Clips can be repositioned and paused during play back and the volume can be changed. You can also access any metadata present in a clip; for example, you may be able to access the artist name, album name and song title when playing back an MP3 file.

`CMdaAudioRecorderUtility` provides recording, editing and playback functionality of audio clips. It also allows you to access and edit metadata in an audio clip.

Note that `CMdaAudioPlayerUtility` provides a subset of the functionality of `CMdaAudioRecorderUtility`. However, it is slightly

easier to use and therefore recommended if you do not need to use the additional functionality of the latter.

Video clip APIs

The video clip APIs, `CVideoPlayerUtility` and `CVideoRecorderUtility`, provide playback and recording, respectively, of video clips. As with the audio clip utilities, they support any format for which a suitable controller plug-in is present in the device.

Audio tone API

The audio tone API, `CMdaAudioToneUtility`, provides playback of DTMF tone sequences, fixed tone sequences and custom tone sequences. The fixed tone sequences may be customized by a phone manufacturer, as may the format of the custom tone sequences.

Note that much of the MMF framework is not used when you use the audio tone API (see Figure 21.2). Specifically, a new thread is not created to process the audio data, so you must ensure that every active object `RunL()` in your thread finishes quickly to allow the tone utility to run in a timely fashion. Otherwise you may find that the tone playback breaks up or the hardware underflows, causing playback to terminate prematurely.

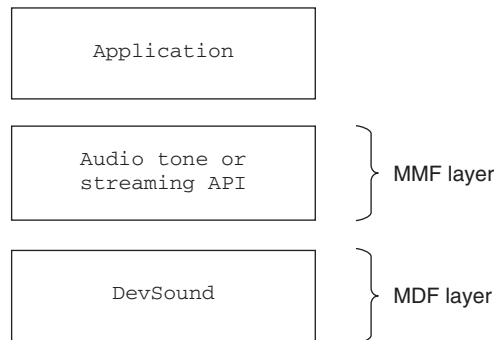


Figure 21.2 Structure of the audio tone and streaming APIs

Audio streaming APIs

The audio streaming APIs, `CMdaAudioInputStream` and `CMdaAudioOutputStream`, allow you to stream raw audio directly to and from the audio hardware.

`CMdaAudioOutputStream` is used where audio data is being received or generated as it is played, making the clip APIs unsuitable. Similarly, `CMdaAudioInputStream` is used in cases where you want to access the recorded data immediately.

There are no explicit video streaming APIs. However, the Camera API (see Section 21.3) can be used to achieve input streaming of video and direct screen access (see [Sales 2005, Chapter 11]) can be used to achieve output streaming.

DRM support

Symbian OS v9.1 has support for playback of DRM protected audio and video content by making use of the Content Access Framework (CAF). The abstract `TMMSource` class provides the basis of this support; the audio- and video-clip playback APIs contain overloads of their open methods that take a `TMMSource` argument. The following concrete subclasses of `TMMSource` are defined:

- `TMMFileHandleSource`, allowing a file handle to be provided
- `TMMFileSource`, allowing a file to be provided by name.

The `TMMSource` class and its subclasses also allow you to provide a DRM access intent, indicating what you intend to do with the DRM protected data, and a unique identifier to specify a content object within a file, if appropriate. The DRM intent that you specify may affect whether or not you are granted access to the content in question. These API methods work similarly for any non-protected DRM content.

In order to access any DRM protected content using the Symbian multimedia APIs, your process must have the DRM capability. For more information on the Content Access Framework, see the Symbian Developer Library.

Audio Clip Playback

Playback of audio files is achieved using the `CMdaAudioPlayerUtility` class, defined in `MdaAudioInputStream.h`. The class supports the playback of audio clips from files, descriptors or URLs. Please note that in order for playback from a URL to be supported, the controller plug-in in use must support this; many do not.

You do not need any capabilities to play audio using these APIs. However, if you want to play audio with a high priority, you need the `MultimediaDD` capability. This policy is intended to stop untrusted applications holding the audio resource, which could interfere with system functions such as ring signal playback or setting up a call. If you do not have `MultimediaDD` and you request a high audio priority, it may be silently downgraded. The exact behavior depends on the audio policy, which differs between phone manufacturers and models.

In order to use the class, you must implement the observer mixin class `MMdaAudioPlayerCallback`. When instantiating `CMdaAudioClipUtility`, you have to provide a reference to a class implementing this mixin; typically, the client itself does so.

```
class CAudioPlayer : public CBase, public MMdaAudioPlayerCallback
{
    ...
    // from MMdaAudioPlayerCallback:
    virtual void MapcInitComplete(TInt aError,
        const TTimeIntervalMicroSeconds& aDuration);
    virtual void MapcPlayComplete(TInt aError);
};

void CAudioPlayer::ConstructL()
{
    iPlayer = CMdaAudioPlayerUtility::NewL(*this);
}
```

To play back an audio clip from a file, you must first open the file using the `OpenFileL()` method. This locates a suitable controller plug-in to play the file you have given and calls back asynchronously to the `MapcInitComplete()` method. An error code is given; if this is `KErrNone`, you are free to start calling other methods in the utility or to initiate playback. If the error you receive is `KErrNotSupported`, it means that no suitable MMF controller plug-in could be found to open the file you provided. Other errors are possible, such as `KErrNotFound` if the specified file does not exist.

```
void CAudioPlayer::PlayFileL(const TDesC& aFileName)
{
    // close the player in case it has previously been
    // opened or if we are already playing a clip.
    iPlayer->Close();
    iPlayer->OpenFileL(aFileName);
}
```

To start playback, you call `Play()`. If you simply want to play the file, this can be done from the callback method.

```
void CAudioPlayer::MapcInitComplete(TInt aError,
    const TTimeIntervalMicroSeconds& /*aDuration*/)
{
    if (aError == KErrNone)
    {
        iPlayer->Play();
    }
    else
    {
        // handle the error
    }
}
```

In a slightly more complex application, you may not want to start playback immediately but simply to update your state, to indicate that further operations are now possible.

```
void CAudioPlayEngine::MapcInitComplete(TInt aError,
    const TTimeIntervalMicroSeconds& /*aDuration*/)
{
    if (aError == KErrNone)
    {
        iState = EOpen;
        // notify the UI of our new state
        iObserver.MapeoStateChanged();
    }
    else
    {
        // notify the UI of the error that has occurred
        iObserver.MapeoError(aError);
    }
}
```

Playback can then be initiated at a later time. Now this clip is open, we can access the metadata entries and other information associated with it.

The audio player utility can pause a clip during playback, by calling the `Pause()` method. When a clip is paused, the current playback position is preserved; we can also reposition the clip while it is paused, something that we cannot do while it is stopped. Use the `SetPosition()` method to do this.

Once playback of the clip has completed, we receive another callback:

```
void CAudioPlayEngine::MapcPlayComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        iState = EOpen;
        // notify the UI
        iObserver.MapeoStateChanged();
    }
    else
    {
        // handle the error
    }
}
```

An error of `KErrNone` indicates that playback ended because we reached the end of the clip – this is the usual case. Other error codes are possible, for example if your playback is pre-empted by another audio playback.

To stop playback before the end of the clip is reached, simply call `Stop()`. This resets the clip but does not close it, so playback can be restarted simply by calling `Play()` again. Note that if you call `Stop()`, you do not receive a `MapcPlayComplete()` callback, so you must update any internal state.

```
void CAudioPlayEngine::Stop()
{
    // it only makes sense to stop the clip if it is currently
    // playing or paused.
    __ASSERT_ALWAYS((iState == EPlaying) || (iState == EPaused),
                    Panic(EAudioPlayerEngineWrongState));

    iPlayer->Stop();
    iState = EOpen;
    // notify the UI
    iObserver.MapeoStateChanged();
}
```

If you have previously set a play window by calling the `SetPlayWindow()` method, a call to `Stop()` resets the clip position to the start of the play window, not the start of the clip.

Notifications of Audio Resource Availability

When you are playing audio, you may be pre-empted by the system or by another audio playback. When this happens, it is possible to receive a notification from the system when the audio resource has become available again so that playback can be resumed.

The behavior of the resource notifications depends on the internal audio policy implementation and therefore it varies from phone to phone. For example, on the Sony Ericsson P990 family of phones, notifications are only provided when the application is pre-empted by certain system events, such as a phone call, not if it is interrupted by another application such as the music player.

To do this, your class must derive from the `MMFAudioResourceNotificationCallback` mixin, which contains the single method `MarncResourceAvailable()`. Then, register for the notification when you start playback.

```
void CAudioPlayEngine::Play()
{
    ...
    iPlayer->Play();

    TInt err = iPlayer->RegisterAudioResourceNotification(*this,
                                                         KMMFEventCategoryAudioResourceAvailable);
    if ((err != KErrAlreadyExists) && (err != KErrNone))
    {
        // handle the error
        ...
    }
    ...
}
```

The `UID_KMMFEventCategoryAudioResourceAvailable` is used to indicate that it is audio-resource events that you are interested in. When your audio playback is pre-empted, you receive an error via the usual callback:

```
void CAudioPlayEngine::MapcPlayComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        ...
    }
    else if ((aError == KErrInUse) || (aError == KErrAccessDenied))
    {
        // Playback has been pre-empted by another audio activity,
        // or playback failed to start due to an ongoing audio
        // activity. Pause the clip so that we can resume when the
        // audio resource is available again.
        iState = EPaused;
        iObserver.MapeoStateChanged();
    }
    else
    {
        ...
    }
}
```

In this example, the engine enters the paused state. Then, when the resource becomes available again, we receive another callback:

```
void CAudioPlayEngine::MarncResourceAvailable(TUId aNotificationEventId,
                                              const TDesC8& aNotificationData)
{
    if ((aNotificationEventId == KMMFEventCategoryAudioResourceAvailable)
        && (iState == EPaused)
        && (aNotificationData.Size() >= sizeof(TInt64)))
    // aNotificationData may have 0 size if some error occurred.
    {
        // retrieve the resume position from the notification data:
        TPckgBuf<TInt64> resumePosition;
        resumePosition.Copy(aNotificationData);
        // seek to the appropriate position
        iPlayer->SetPosition(resumePosition());
        // and resume playing
        iPlayer->Play();
        iState = EPlaying;
        iObserver.MapeoStateChanged();
    }
}
```

The notification data provided with the callback tells us what position in the clip we were at when the interruption occurred. We retrieve this

information and perform a seek before restarting playback to ensure that we resume from the correct position.

Note that although the example code here is given for `CMdaAudioPlayerUtility`, identical functionality is provided in all the audio APIs.

Audio Clip Recording

Recording of audio clips is achieved using class `CMdaAudioRecorderUtility` which is defined in `MdaAudioSampleEditor.h`. You can use this class to record audio in any format for which a suitable controller plug-in exists. The class contains a superset of the functionality of `CMdaAudioPlayerUtility` and so is often used for playback of clips. Note however that the class does not contain a `Pause()` method, unlike the player utility.

To perform audio recording, you need the `UserEnvironment` capability. High audio priorities are policed in the same way as with the player utility, using the `MultimediaDD` capability.

To use the audio recorder utility, you must derive from mixin class `MMdaObjectStateChangeObserver` which defines a single method:

```
class MMdaObjectStateChangeObserver
{
public:
    virtual void MoscoStateChangeEvent(CBase* aObject,
                                      TInt aPreviousState, TInt aCurrentState, TInt aErrorCode) = 0;
};
```

`MoscoStateChangeEvent` is called every time the utility changes state; the `aPreviousState` and `aCurrentState` parameters have values corresponding to `CMdaAudioClipUtility::TState`:

```
enum TState
{
    ENotReady = 0,
    EOpen,
    EPlaying,
    ERecording
};
```

When using the utility for audio playback, the usage is very similar to the player utility so it is not discussed again here. One point worth mentioning however is that you receive a callback to `MoscoStateChangeEvent` following a call to `PlayL()` indicating that playback has started; no equivalent callback exists when using the player utility.

To perform audio recording, a controller is opened by the recorder utility. You can record to a descriptor or to a file; it is usual to record to a file due to the memory requirements of using a descriptor.

When you open the recorder utility, you must indicate which controller to use for the recording. You can do this either by specifying explicitly the UID of the controller to use or by letting the controller framework select the controller automatically based on the extension of the filename given. For example, if you open a file with the extension `.wav`, the Symbian audio controller is selected automatically.

Selecting a controller by UID

As stated earlier, all controller plug-ins are ECOM plug-ins. Hence we can use ECOM to query the controllers present on a phone and the formats that they support. Symbian provides some helper classes to make this job easier:

- `CMMFControllerImplementationInformation`
- `CMMFFormatImplementationInformation`
- `CMMFControllerPluginSelectionParameters`
- `CMMFFormatSelectionParameters`.

You can use these classes to query the controllers present on a phone and find out what media formats they support. They are all defined in the header file `mmf\common\mmfcontrollerpluginresolver.h`.

The `mmfcontrollerframework.lib` library that you need to link against when using these classes is not present in the S60 3rd Edition SDK from Nokia. You must download an SDK extension plug-in separately before you can use the classes discussed here.

First, create an instance of `CMMFControllerPluginSelectionParameters` and indicate what sort of controller you are interested in. Here, we specify that we are only interested in controllers that support audio:

```
CMMFControllerPluginSelectionParameters* controllerSelection =
    CMMFControllerPluginSelectionParameters::NewLC();

RArray<TUid> mediaIds;
CleanupClosePushL(mediaIds);

mediaIds.AppendL(KUidMediaTypeAudio);
controllerSelection->SetMediaIdsL(mediaIds,
    CMMFPluginSelectionParameters::EAllowOnlySuppliedMediaIds);
```

If we wanted to find out about video controllers, we could have specified `KUIdMediaTypeVideo` instead of `KUIdMediaTypeAudio`. In this example, we have specified that we are interested in controllers that support only audio, by giving the value `EAllowOnlySuppliedMediaIds`. Other options here are:

- `ENoMediaIdMatch` – do not perform matching on media ID
- `EAllowOtherMediaIds` – allow controllers that support the specified media ID in addition to others. This would show controllers supporting audio and video.

The next step is to specify the media formats we are interested in:

```
CMMFFormatSelectionParameters* formatSelect =
    CMMFFormatSelectionParameters::NewLC();

controllerSelection-> SetRequiredRecordFormatSupportL(*formatSelect);
```

We could have chosen a specific format at this point by calling `SetMatchToMimeTypeL()` and giving the MIME type of the audio format we are interested in. In this example, we list all controllers that support audio recording.

The final step is to retrieve the list of controllers that meet the criteria we have specified.

```
RMMFControllerImplInfoArray iControllers;
...
controllerSelection->ListImplementationsL(iControllers);
```

The array `iControllers` now contains a list of controllers that support audio recording, each one represented by an instance of `CMMFControllerImplementationInformation`.

Each controller can support any number of audio playback or recording formats. The list of formats supported can be accessed using the `RecordFormats()` and `PlayFormats()` methods. Each format is represented by an instance of `CMMFFormatImplementationInformation`.

Recording the sound

Once we have chosen the format we wish to record in, the controller can be opened by the utility. If we want to let the MMF choose a controller and format based on the extension of the filename we give, simply pass in this filename.

```
iRecorder->OpenFileL(fileName);
```


If we are specifying a format to record in by controller and format UID, we pass these in to the `OpenFileL()` method.

```
void CAudioRecordEngine::OpenL(const TDesC& aRecordFileName,
                               TRecordFormatParameters aParameters)
{
    // in case it has previously been used:
    iRecorder->Close();
    iState = ENotReady;

    // Append a suitable extension to the filename we are given.
    const CDesC8Array& extensions =
        aParameters.iFormat->SupportedFileExtensions();
    if (extensions.Count() > 0)
    {
        // just use the first one
        InitializeFileNameL(aRecordFileName, extensions[0]);
    }

    iRecorder->OpenFileL(iFileName,
                        aParameters.iController->Uid(), // controller UID
                        KNullUid, // play controller UID (unused here)
                        aParameters.iFormat->Uid() // record format UID
                        );
}
```

In this example, we are appending a suitable file extension to the filename given. This step is not necessary but adds completeness.

As with the player utility, we expect a callback after we have called `OpenFileL()`.

```
void CAudioRecordEngine::MoscoStateChangeEvent(CBase* /*aObject*/,
                                               TInt /*aPreviousState*/, TInt aCurrentState, TInt aErrorCode)
{
    switch (iState)
    {
        {
            case ENotReady:
                // The OpenFileL call is complete
                if (aErrorCode == KErrNone)
                {
                    iState = ERecordReady;
                    // notify the UI
                    iObserver.MareoStateChanged();
                }
            else
            {
                // notify the UI
                iObserver.MareoError(aErrorCode);
            }
        }
        break;
        ...
    }
}
```

The `aObject` parameter to this method indicates which object the state change relates to; in this example, since we only use a single recorder utility, we know that `aObject == iRecorder`.

If we want to set the parameters for the recording, now is the time to do it. The recorder utilities provides a number of methods allowing you to change the number of channels, bitrate and other settings. You should query which values are supported before setting them, for example by using the `GetSupportedBitRatesL()` method before calling `SetDestinationBitRateL()`.

Now the recorder is ready to begin recording.

```
void CAudioRecordEngine::RecordL()
{
    __ASSERT_ALWAYS(iState == ERecordReady,
        Panic(EAudioRecorderEngineWrongState));
    iRecorder->RecordL();
}
```

We receive another callback to `MoscoStateChangeEvent()` when the recording starts successfully; at this point, we move the engine into state `ERecording`. The recording continues until we stop it or an error occurs, such as if we run out of disk space or the descriptor we are recording into becomes full. If we call `Stop()`, we do not receive a callback notifying us of the state change.

Playing back the recording

The recorder utility is capable of playing audio clips as well as recording them. Once the clip is recorded, we can play it back. This can be done simply by making a call to `PlayL()` after the call to `Stop()`; however this requires that the same controller supports recording and playback of the format in use, since only one controller is used at a time. While this is true for the Symbian audio controller, it is often not the case for controllers that record in compressed formats.

So, to play the clip reliably, we need to close and reopen the controller. This gives the MMF the chance to load a new controller if required. In this example, we do it immediately after the recording finishes.

```
void CAudioRecordEngine::Stop()
{
    __ASSERT_ALWAYS(iState != ENotReady,
        Panic(EAudioRecorderEngineWrongState));
    iRecorder->Stop();
    switch (iState)
    {
        ...
        case ERecording:
            OpenControllerForPlay();
    }
```

```

        break;
    }
    iObserver.MareoStateChanged();
}
...
void CAudioRecordEngine::OpenControllerForPlay()
{
    iRecorder->Close();
    TRAPD(err, iRecorder->OpenFileL(iFileName));
    if (err == KErrNone)
    {
        iState = EPlayReady;
        // don't notify our observer yet - wait until we get another
        // state change event notifying us that the player is now
        // ready to go
    }
    else
    {
        iState = ENotReady;
        iObserver.MareoError(err);
    }
}

```

We then receive a callback to `MoscoStateChangeEvent()`, indicating that the utility is ready to begin playback.

Audio Streaming

The audio streaming APIs allow you to provide or receive audio data while it is being played or recorded respectively. This allows you to generate audio data for playback while it is being played and process audio data while it is being recorded.

When streaming, you are using a client utility API that interfaces directly to the MDF layer. Hence, no MMF controller plug-in is used and the subthread associated with controllers is not used either. The streaming takes place directly in the application's thread. This makes it important to ensure that the `RunL()` implementations for all the active objects in your application are kept short to avoid interrupting the streaming – see Chapter 6. If your application needs to perform processor-intensive tasks, it may be advisable to delegate usage of the streaming API to a separate thread.

Since the streaming APIs interface directly to the MDF layer, the audio formats that are supported by the streaming APIs are determined by the audio formats supported in each phone. All Symbian OS v9.1 phones support PCM formats. Many also support compressed formats such as AMR and MP3. To determine which formats a particular phone supports, you must use the `CMMFDevSound` interface.

Output streaming

Audio output streaming is performed using class `CMdaAudioOutputStream`. You must implement the corresponding mixin class `MMdaAudioOutputStreamCallback` in order to use it. The capability restrictions associated with this class are identical to the audio clip playback API.

As with the audio clip utilities, you must initialize the streaming utility after it has been instantiated.

```
iStream = CMdaAudioOutputStream::NewL(*this);
...
// open the audio stream. We get a callback to
// MaoscOpenComplete indicating success or failure.
iStream->Open(NULL);
```

The `TMdaPackage*` parameter is present only for historical reasons and is not used in Symbian OS v9.1. You should therefore just specify `NULL`.

Once the stream has been opened, you can configure it for the audio type that you intend to stream to it. In this example, we stream PCM data but most phones also support some compressed formats. In this example, the stream is configured from inside the callback method; this could be done any time between receiving the callback and starting streaming.

```
void CAudioStreamOutEngine::MaoscOpenComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        // set the data type to 16-bit PCM
        TRAP(aError, iStream->SetDataTypeL(KMMFFourCCCodePCM16));
    }
    if (aError == KErrNone)
    {
        // setup the sample rate and number of channels
        TRAP(aError, iStream->SetAudioPropertiesL(iSampleRateCaps,
                                                iChannelCaps));
    }
    if (aError == KErrNone)
    {
        // setup complete
    }
    else
    {
        // setup failed - handle the error
    }
}
```

The `SetDataTypeL()` method is given a four-character code specifying the format of the audio data to be streaming. The four-character codes are specified in `mmffourcc.h`.

The parameters to the `SetAudioPropertiesL()` method are values from enumeration `TMdaAudioDataSettings::TAudioCaps`. Valid values for the sample rate setting are of the form `ESampleRateXxxxHz`. The number-of-channels setting should be either `EChannelsMono` or `EChannelsStereo`.

Once the stream has been set up successfully, streaming is started simply by writing some data to the stream:

```
iStream->WriteL(buffer);
```

Buffer is a descriptor filled with audio data of the appropriate type. Once you have written a buffer of audio data to the stream, you should not modify the data in the buffer, nor the buffer itself, until it is returned to you via the callback `MaoscBufferCopied()`.

You must supply a sufficiently frequent stream of data to the output stream to prevent it running out of data, known as an underflow. This can be done simply by writing a large number of buffers to the stream all in one go. They are queued and played in turn. However, it is usual to use two buffers – this allows for one buffer to be filled up while the other is being played by the audio stream. If you use a single buffer, the stream may run out of data while you are refilling it, leading to an underflow and a break in the audio. Using more than two buffers may reduce the risk of underflow when the system is heavily loaded, but clearly has a larger memory overhead.

When a buffer is returned to you, you should refill it and write it back to the utility if you wish to continue streaming.

```
void CAudioStreamOutEngine::MaoscBufferCopied(TInt aError,
                                              const TDesC8& aBuffer)
{
    // the audio device has finished with the data
    // in a buffer. Refill it, and send it back to
    // the device.
    TInt bufferNum = KErrNotFound;
    if (aError == KErrNone)
    {
        // find the buffer that has just been emptied so that we
        // can refill it.
        for (TInt i=0; (i<KBufferCount) && (bufferNum == KErrNotFound); ++i)
        {
            if (&aBuffer == &iBuffer[i])
            {
                bufferNum = i;
            }
        }
        // we can be sure here that bufferNum != KErrNotFound
        // because the stream only passes back buffers
        // that we have given it.
        aError = FillThisBuffer(iBuffer[bufferNum]);
    }
}
```

```

if (aError == KErrNone)
{
    // write the new data to the output stream.
    iStream->WriteL(iBuffer[bufferNum]);
}
// we get a KErrAbort for each queued buffer
// after a call to Stop. Ignore these errors.
else if (aError != KErrAbort)
{
    iStream->Stop();
    // handle the error
}
}

```

In this example, we have a small array of buffers. We find the buffer that has just been emptied in this array, refill it and then write it back to the stream.

To stop streaming, you have two options. Either you can just stop writing buffers back to the stream when they are returned to you – in this case, you get a callback to `MaoscPlayComplete()` with an error of `KErrUnderflow` when all the data you have provided has been played completely. This method ensures that all the data is played fully.

If you want to stop the streaming immediately, you should call `Stop()` on the audio stream.

```
iStream->Stop();
```

In this case, you should expect the following callbacks:

- `MaoscBufferCopied()` with an error of `KErrAbort` for each buffer that you have written to the stream but which has not yet been returned
- `MaoscPlayComplete()` with an error of `KErrCancel`.

These callbacks happen synchronously from within the call to `Stop()`. By the time this call returns, you can be sure that streaming has terminated.

Choosing buffer sizes

When using the audio streaming APIs, you should take a little time to decide what size buffers you should use and how many of them. Some compressed formats dictate a fixed buffer size; for PCM, any size buffer can be used.

- If you want low-latency sound, such that the sound heard can react quickly (for example, to events on screen), you should use a smaller

buffer size. This carries the risk of underflow when the system is heavily loaded.

- If the reliability of the audio is important, larger buffers should be used. This reduces the risk of underflow occurring during the stream.

If you need reliable and low latency audio, you should consider having a dedicated high-priority thread to process the audio.

Some extra overheads are introduced when using the audio streaming utilities. The data is copied from the buffers in which you provide it into another buffer that is owned by the audio device. This introduces extra processing overheads and increases the latency slightly. For performance-critical applications, you should consider interfacing directly to the DevSound interface, which avoids these overheads.

Input streaming

The audio input-streaming interface, `CMdaAudioInputStream`, is similar in structure and usage patterns to the output streaming interface. To use it, you must provide a class that derives from `MMdaAudioInputStreamCallback`. As with audio clip recording, you need the `UserEnvironment` capability to perform audio input streaming.

As with the output-streaming API, you must perform the following sequence of events to start input streaming:

1. Instantiate the class `CMdaAudioInputStream`.
2. Initialize it by calling `iStream->Open(NULL)`.
3. Wait for a callback to `MaiscOpenComplete()`.
4. Configure the stream using the `SetDataTypeL()` and `SetAudioPropertiesL()` methods.
5. Start streaming by making a call to `iStream->ReadL()`.

You should provide an empty descriptor to the `ReadL()` method. As with the output stream, to increase the reliability of the stream and reduce the risk of an overflow, you may want to issue more than one call to `ReadL()` when the streaming starts.

When a buffer full of audio data has been recorded, it is provided to you via the `MaiscBufferCopied()` callback. You should then process the new data in the buffer and issue another read request if you wish to continue streaming.

To stop the input streaming, simply call `iStream->Stop()`. After this call, you receive a callback to `MaiscBufferCopied()` for each buffer that you had previously given to the utility via a call to `ReadL()`. Note that these buffers may not all be empty: any data that was recorded

between the last callback to `MaiscBufferCopied()` and the call you make to `Stop()` is provided to you at this time, to ensure that you do not miss any audio that was recorded immediately prior to when you stopped the stream.

The same tradeoffs that apply to output streaming in terms of buffer sizes also apply to input streaming. If you want access to the data very soon after it was recorded, you should use smaller buffers. For maximum reliability, use larger buffers.

Audio Tone Playback

Symbian OS v9.1 supports playback of basic, predefined tone sequences and DTMF strings. The class `CMdaAudioToneUtility` provides this support; the tones are generated as they are being played back and consist of one or more sine waves mixed together.

The predefined tone sequences that can be played back using the class are known as fixed sequences. They are defined by the phone manufacturer and cannot be changed. Each fixed sequence has a human-readable name which can be retrieved using the `FixedSequenceCount()` and `FixedSequenceName()` methods.

To use this class, you should implement the `MMdaAudioToneObserver` mixin. The usage is similar to the audio clip utility.

Once the clip has been instantiated, it must be initialized. Here, we initialize it to play a DTMF string contained in a descriptor:

```
iUtility->PrepareToPlayDTMFString(aDtmfString);
```

This results in a callback to `MatoPrepareComplete()`; playback can then be started.

```
void CAudioToneEngine::MatoPrepareComplete(TInt aError)
{
    if (aError != KErrNone)
    {
        // handle the error
    }
    else
    {
        iUtility->Play();
    }
}
```

The `CancelPrepare()` and `CancelPlay()` methods can be used to stop the utility during the initialization phase and during playback respectively. `CancelPlay()` is analogous to the `Stop()` methods seen in the other utility classes. When playback of the tone completes, you get a callback to method `MatoPlayComplete()`.

As with the audio streaming APIs, the audio tone utility interfaces directly to the MDF layer and does not use a subthread to process audio data. The utility must be able to receive timely `RunL()` calls during playback of the tone if it is to play successfully. In order to avoid an audio underflow during tone playback, you should ensure that every active object `RunL()` in your application completes promptly.

Video Playback

Video playback in Symbian OS is similar in principle to audio playback. An MMF controller plug-in is used, along with its associated subthread. Symbian OS does not provide any concrete video playback support, but video controller plug-ins are provided by phone manufacturers to support video playback.

No capabilities are required to play video. However, if you are playing a video with an audio track, the restrictions regarding the audio priority are the same as those that apply to the audio clip playback API.

Video playback is performed using the `CVideoPlayerUtility` class. You need to provide a class that derives from `MVideoPlayerUtilityObserver` in order to use it.

When you instantiate this class, you must provide a number of parameters to allow the video playback to be configured:

```
iPlayer = CVideoPlayerUtility::NewL(*this,
                                     EMdaPriorityNormal,
                                     EMdaPriorityPreferenceNone,
                                     iCoeEnv->WsSession(), // window-server session
                                     *iCoeEnv->ScreenDevice(), // software screen device
                                     *window, // display window
                                     drawingRect, // display-window dimensions
                                     drawingRect ); // clipping area
```

The first parameter is the observer class deriving from `MVideoPlayerUtilityObserver`; the next two are the priority and priority preference that are used when accessing the audio device, if the video in question has an audio track.

The window server session, software screen device and display window are the parameters required to set up direct screen access, which is used to render the video. See [Sales 2005, Chapter 11] for more information on direct screen access. The display window dimensions specify the rectangle on the screen in which the video is to be rendered. Note that the rectangle is relative to the origin of the entire screen – not the origin of the control or window. The clipping area – also relative to the entire screen – specifies the region of the video that should be rendered. It is possible to display just a portion of a video.

Once the utility is instantiated, we can then open a video file and await the callback:

```
iPlayer->OpenFileL(aFileName);
...

void CVideoPlayEngine::MvpuoOpenComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        iPlayer->Prepare();
    }
    else
    {
        // handle the error
    }
}
```

The next phase that must be carried out before the video can be played is to call `Prepare()` on the utility. This allows the video controller plug-in to finalize its configuration for video playback. In this example, it is done immediately; it could be done any time after the open complete callback is received. Once the `Prepare()` call has been completed, we receive another callback.

```
void CVideoPlayEngine::MvpuoPrepareComplete(TInt aError)
{
    TSize videoSize;
    if (aError == KErrNone)
    {
        TRAP(aError, iPlayer->VideoFrameSizeL(videoSize));
    }
    TSize controlSize = iCoeControl->Size();
    if (aError == KErrNone)
    {
        if ((controlSize.iHeight < videoSize.iHeight)
            || (controlSize.iWidth < videoSize.iWidth))
        {
            // video is bigger than the control so we need to scale it
            TReal32 scalePercentX = controlSize.iHeight * 100.0
                                   / videoSize.iHeight;
            TReal32 scalePercentY = controlSize.iWidth * 100.0
                                   / videoSize.iWidth;
            // scale both directions by the same amount so that
            // the aspect ratio is preserved:
            TReal32 scalePercent = Min(scalePercentX, scalePercentY);
            TRAP(aError, iPlayer->SetScaleFactorL(
                scalePercent,
                scalePercent,
                // use anti-alias filtering if available
                ETrue));
        }
    }
    if (aError == KErrNone)
    {

```

```

        // Success! play the video
    }
    else
    {
        // handle the error
    }
}

```

Once the callback indicating the completion of the `Prepare()` call is received, we can query the dimensions of the video clip that we have just opened. We then work out by what factor it should be scaled so that it fits in the area into which we are rendering it.

Now, to start the playback of the video, we simply call `Play()`:

```

iPlayer->Play();

```

We then receive another callback when the clip ends naturally, or when it is interrupted by the system. This could happen, for example, if the audio resource is pre-empted by another audio client.

Any metadata present in the video clip can be accessed in exactly the same way as with the audio-clip player utility.

Video Recording

Video recording can be achieved using the `CVideoRecorderUtility` class which is defined in `VideoRecorder.h`. It is similar in principle to audio recording, but slightly more complex. The callback class that you must implement to use `CVideoRecorder` is `MVideoRecorderUtilityObserver`. Video is recorded from the built-in camera on the phone.

Since the `UserEnvironment` capability is required to record audio and to use the camera, this capability is required in order to use the video recording API.

When recording video, you must select a format for the recorded data. Unlike the audio-recorder utility, you cannot let the MMF select the controller plug-in automatically based on the extension of a filename that you provide; instead, you must supply explicit code to find the UID of the controller to use.

```

CMMFControllerPluginSelectionParameters* controllerSelection =
    CMMFControllerPluginSelectionParameters::NewLC();

RArray<TUid> mediaIds;
CleanupClosePushL(mediaIds);

mediaIds.AppendL(KUidMediaTypeVideo);
controllerSelection->SetMediaIdsL(mediaIds,
    CMMFPluginSelectionParameters::EAllowOtherMediaIds);

```

Here, we indicate that we are interested in video controllers by specifying `KUidMediaTypeVideo`. When calling `SetMediaIdsL()`, we must specify `EAllowOtherMediaIds`, as most video controllers also support audio. Querying of the associated formats can now proceed in the same way as when querying audio controllers.

Once you have selected the UID of the controller and format that you wish to use for the recording, you can open the recorder utility using `OpenFileL()`, `OpenDesL()` or `OpenUrlL()`. Each of these methods requires the UID of a controller and format. You are also required to supply the handle to a camera from which to record. The handle can be obtained using the `CCamera::Handle()` method; see Section 21.3 for more information on `CCamera`.

After the call to an `OpenXxx()` method, you receive a callback to `MvruoOpenComplete()` indicating success or otherwise. Once the utility has been successfully opened, you can perform any configuration necessary. For this purpose, `CVideoRecorderUtility` provides a number of `SetXxx()` methods that can be called at this time:

- call `SetAudioEnabledL(ETrue)` if you wish the recorded video clip to contain an audio track
- use `SetAudioTypeL()` and `SetVideoTypeL()` to configure the specific audio and video encoding to use. You should check the supported types beforehand, using `GetSupportedAudioTypesL()` and `GetSupportedVideoTypesL()`.

Due to the complexity of video controllers, plug-in suppliers may also provide a number of custom commands to configure the controller.

Once the video recorder is configured, you must prepare it by calling `Prepare()`. This is another asynchronous method and result in a callback to `MvruoPrepareComplete()`. Once this callback has been received with an error of `KErrNone`, the recorder utility is ready to start the recording itself.

To begin the video recording, simply call `Record()`, which starts the capturing and encoding of video frames from the camera. Recording continues until you stop it, the recording destination becomes full or some other error occurs.

To stop the recording yourself, call `Stop()`. In this case, you do not receive any callback. If recording stops for some other reason, you receive a callback to the `MvruoRecordComplete()` method.

Other asynchronous events from the controller plug-in may be delivered to you via the `MvruoEvent()` callback method. These events are defined by the video controller plug-in; to interpret them you need to check the details of the controller plug-in in use.

Using DevSound

The DevSound API, `CMMFDevSound`, forms the interface audio component to the Media Device Framework (MDF) layer. It is the lowest-level audio API available on Symbian OS phones.

The DevSound implementation is replaced by phone manufacturers to interface to the audio hardware present on each phone; the interface remains unchanged between all Symbian OS v9.1 phones and should behave in the same manner.

The DevSound API can be used to query the capabilities of the audio hardware on any given phone, to determine which audio formats are supported. This determines which formats you can use with the audio input and output streams described earlier. You can also use it to perform high-performance streaming to and from the audio hardware.

Querying MDF capabilities

The `GetSupportedOutputDataTypesL()` and `GetSupportedInputDataTypesL()` methods indicate which audio formats are supported for audio output and audio input, respectively. Both return an array of four-character codes that identify an audio encoding format. All Symbian OS phones can be expected to support playback and recording of various PCM formats. Many also support encoded formats such as AMR. The header file `mmffourcc.h` defines the four-character codes for these audio formats.

The `Capabilities()` method indicates what audio parameters are supported by the sound device. It returns a `TMMFCapabilities` object; the members of this class are interpreted as follows:

- `iRate` is a bit field of values taken from enumeration `TMMFSampleRate`, indicating which audio sample rates are supported for audio input and output
- `iEncoding` represents which PCM encoding formats are supported, with values taken from enumeration `TMMFSoundEncoding`; this functionality has been superseded by the methods described above and should not be relied on
- `iChannels` indicates how many channels are supported by the audio device; it is a bit field of values from enumeration `TMMFMonoStereo`
- `iBufferSize` is the maximum size of an audio buffer that can be handled by DevSound.

The sample rates supported for audio input and output may differ; by default DevSound reports the capabilities supported for playback. To get the recording capabilities, you should initialize DevSound for recording

before calling this method. Note also that the number of channels and the sample rate may be constrained by the audio format in use: for example, when using AMR-encoded audio, only 8 kHz mono may be supported.

Audio playback and recording using DevSound

The DevSound API can be used for high-performance audio input and output streaming, avoiding the overheads associated with the streaming utilities. It also allows more control over the audio device. While a full description of how to do this is outside the scope of this book, a brief discussion is worthwhile.

Before streaming can begin, DevSound must be initialized using one of the `InitializeL()` methods. An observer class implementing mixin `MDevSoundObserver` must be provided.

```
IMPORT_C void InitializeL(MDevSoundObserver& aDevSoundObserver,
                        TFourCC aDesiredFourCC, TMMFState aMode);
```

The parameter `aMode` should be set to `EMMFStatePlaying` to perform playback or `EMMFStateRecording` for recording. Note that it is possible to play fixed tone sequences and DTMF strings directly using DevSound, but this is not recommended since it provides no advantage over the easier-to-use class `CMdaAudioToneUtility`.

Following an initialization call, you receive a callback to the `InitializeComplete()` method in your observer indicating the success or otherwise of the initialize operation.

You can now configure the parameters of your playback or recording using the appropriate methods. To begin the playback or recording activity, you should now call either `PlayInitL()` or `RecordInitL()`, which request permission from the audio policy and then begin.

During a playback operation, the following sequence of events occurs:

1. DevSound calls its observer's `BufferToBeFilled()` method, requesting some audio data.
2. The observer fills the buffer with audio of the appropriate type.
3. The observer passes the buffer back to DevSound via the `PlayData()` method. This can be done synchronously or asynchronously.

When DevSound requires more data, `BufferToBeFilled()` is called again. To end playback, the final buffer is marked as the last buffer by calling `CMMFBuffer::SetLastBuffer(ETTrue)` on it.

During a recording operation, the sequence of events is similar. Full audio buffers are passed to the observer via the `BufferToBeEmptied()` method; the observer returns the buffers to `DevSound` via the `RecordData()` method.

`DevSound` may report errors to the observer at any time during playback or recording via the asynchronous callback methods `RecordError()` or `PlayError()`. Other events may be passed to the observer via the `SendEventToClient()` method. Event types that are passed into this method include the following:

- `KMMFEventCategoryPlaybackComplete` when a playback or recording activity is complete
- `KMMFEventCategoryAudioResourceAvailable` if you have requested a notification when the audio resource is available.

Writing Controller Plug-ins

The MMF is designed to allow extensibility via ECOM plug-ins. The controller plug-in is the mechanism by which support for new audio formats are added to the system. The following discussion should provide you with the information you need to start writing controller plug-ins; it is not intended to be a comprehensive description.

All MMF controller plug-ins derive from class `CMMFController`. They are ECOM plug-ins, which are described in detail in Chapter 19. The details of creating the ECOM plug-in itself are therefore omitted here.

The behavior of an MMF controller largely dictates the behavior of the client-side clip APIs. A badly behaving controller plug-in can cause a well-written client application to stop responding or to behave incorrectly. Some care must therefore be taken when writing controller plug-ins to preserve the expected behavior of the client-side APIs.

Defining the plug-in

The MMF requires that each MMF controller plug-in has at least one associated format, a type of file that the plug-in can read or write. A basic controller plug-in has a single format; a more complex plug-in can have any number. A format is presented as another ECOM plug-in to the system; however, the format is not instantiated by ECOM unless the associated controller does this explicitly; hence it is not necessary to actually write a format plug-in, but merely to describe it in the controller plug-in resource file.

The `opaque_data` section in a controller's resource file describes what media types and formats are supported. The section is split with tags of the following forms:

- `<s>` specifies the supplier of the controller plug-in
- `<i>` specifies the media type UUIDs supported by the controller
- `<p>` specifies the ECOM interface UUID of play formats supported by the controller
- `<r>` specifies the ECOM interface UUID of record formats supported by the controller.

For example:

```
opaque_data = "<s>Symbian<i>0x101F5D07<p>0x101F0001";
```

The UUIDs following an `<i>` tag are defined in `MmfDataSourcesink.hrh`. Valid UUIDs are:

- `KUuidMediaTypeAudio: 0x101F5D07`
- `KUuidMediaTypeVideo: 0x101F5D08`.

An audio controller specifies only audio as a supported media type. A video controller would typically specify audio and video. In this case, there would be two separate `<i>` tags in the `opaque_data` section.

The format plug-ins must have interface UUIDs matching those given in the `<p>` or `<r>` tags in the controller. The `opaque_data` section for a format plug-in is similar to that for a controller; the following tags are recognized:

- `<s>` specifies the supplier of the format;
- `<i>` specifies the media IDs supported by the format;
- `<m>` specifies the MIME type of the format;
- `<e>` specifies the file extension given to files of this format;
- `<h>` specifies header data to match to recognize a file of this format.

For example:

```
opaque_data =
    "<s>Symbian<i>0x101f5d07<e>.ogg<h>OggS*vorbis<m>application/ogg";
```

The `<i>` tags in the format would usually be the same as those in the controller. The `<e>` and `<m>` tags can be specified multiple times, if more than one file extension or MIME type is commonly associated with the format.

The format of the string following an `<h>` tag is the same format as that accepted by the `TDesC8::MatchF()` method. If more flexibility is required, you can write a separate recognizer plug-in that matches the files to a MIME type supported by the format plug-in. See Chapter 19 for more information on writing recognizer plug-ins.

Implementing the plug-in

As mentioned earlier, a controller plug-in derives from class `CMMFController`. You need therefore to derive from this class. To properly support the MMF client-side APIs, you also need to derive from a number of other classes, to support custom commands. Custom commands are those commands that need to be supported by some controllers, but not all.

An audio playback controller should implement the following custom command mixin classes:

- `MMMFAudioPlayDeviceCustomCommandImplementor`, allowing the volume and balance to be set and queried
- `MMMFAudioPlayControllerCustomCommandImplementor`, to support playback windows.

`MMMFAudioControllerCustomCommandImplementor` can also be implemented by an audio playback controller if it wishes to allow its client to query the properties of the audio clip, such as the bitrate and number of channels.

An audio recording controller should implement the following:

- `MMMFAudioRecordControllerCustomCommandImplementor`, for recording information to be provided
- `MMMFAudioControllerCustomCommandImplementor`, to allow the parameters of the recording to be queried and set.

The following should be implemented by video controller plug-ins:

- `MMMFVideoControllerCustomCommandImplementor` for querying and setting basic video information
- `MMMFVideoPlayControllerCustomCommandImplementor` in a video-playback controller
- `MMMFVideoRecordControllerCustomCommandImplementor` in a video-recording controller.

Additionally, `MMMFResourceNotificationCustomCommandImplementor` should be implemented by any controller that wishes to support audio resource notifications.

For each custom command mixin class implemented by a controller plug-in, a custom command parser should be added when the plug-in is constructed. Each custom command implementer interface has a corresponding custom command parser. For example:

```
void COggVorbisController::ConstructL()
{
    ...
    CMMFAudioPlayDeviceCustomCommandParser* audPlayDevParser =
        CMMFAudioPlayDeviceCustomCommandParser::NewL(*this);
    CleanupStack::PushL(audPlayDevParser);
    AddCustomCommandParserL(*audPlayDevParser);
    CleanupStack::Pop( audPlayDevParser ); //audPlayDevParser
    ...
}
```

This allows the MMF to find the custom command implementer and send the appropriate commands to it.

Any custom commands that are not implemented by a controller result in a `KErrNotSupported` error when a client calls the corresponding client-side API. In many cases, it may cause the client-side APIs to fail entirely if vital custom commands are not implemented.

Operation of the plug-in

Your controller plug-in is instantiated by the MMF when it is opened by a client. The subthread is created for you. After your controller has been opened, you can add one or more data sources and data sinks, via the `AddDataSourceL()` and `AddDataSinkL()` methods. Which specific sources and sinks are added depends on the media IDs that you support (as specified in the plug-in definition), and whether you are performing playback or recording. For example, an audio playback controller has a file, descriptor or URL data source and an audio output data sink. The principal job of the controller is to read data from the sources and write data to the sinks.

When a data source or sink is added, you can verify its type by checking its UID. These UIDs are defined in header file `MmfData-sourcesink.hrh`. Once all the appropriate data sources and sinks have been added, the controller is primed via the `PrimeL()` method. Here, you should prime each of your sources and sinks and perform any other initialization necessary to begin the playback or recording activity.

The `CMMFController::PlayL()` method is used to begin both a playback and a recording activity. The two are similar to the extent that your controller is reading from its sources and writing to its sinks. The fact that `PlayL()` is called to begin recording should not therefore be a source of confusion.

Platform security limitations

Prior to Symbian OS v9.1, third-party controller plug-ins could be added to phones to allow new formats to be supported in the built-in media player applications, and to allow those new formats to be used as ringtones and other system sounds. However, due to the capability-based DLL loading restrictions introduced in Symbian OS v9.1, many of these use cases are not available to the majority of third-party developers. A built-in media player application typically has a DRM capability and the process responsible for playing ringtones usually has MultimediaDD capability to allow it to play system sounds. These are both protected capabilities not easily available to third-party developers.

You should therefore note that when writing a new controller plug-in, it may not be usable in any of the built-in applications. However, it does not prevent a third-party media application to be written that supports all the formats available on the phone, including those provided in the phone's firmware and those added via third-party controller plug-ins.

21.2 The Image Conversion Library

The Image Conversion Library, or ICL, is a library based on ECOM plug-ins that supports encoding and decoding of a wide variety of image formats. ICL image encoder and decoder plug-ins for many formats are provided by Symbian. Other plug-ins may be added by phone manufacturers, to support more formats or to take advantage of hardware-accelerated encoding or decoding on a phone. Plug-ins can also be provided by third parties.

The ICL also supports scaling and rotating of bitmaps once they have been decoded, as well as displaying encoded images on the screen easily.

Overview of the API

The core ICL API consists of basic image encoding and decoding classes. These are used to convert between external, usually compressed, image formats and the native `CFbsBitmap` type. See Chapter 17 for more information about `CFbsBitmap`. The classes used to perform decoding and encoding are `CImageDecoder` and `CImageEncoder` respectively.

The encoding or decoding operation takes place asynchronously, allowing your application to continue responding to other events. This is achieved in one of two ways:

- By using an active object running in the same thread, the image conversion operation is split into numerous smaller tasks, each running within a separate call to the object's `RunL()`. Your application remains responsive during the decode operation, but calls to the

`RunL()` functions of any of the application's other active objects could be slightly delayed during the conversion operation.

- By using a dedicated subthread that the ICL can automatically create for the conversion process; no intensive processing takes place inside your application's thread, ensuring that it always remains responsive.

Note that image conversion plug-ins can indicate that they should always be executed in a separate thread. If this is the case, you cannot override this behavior.

If you simply want to display an image on screen, you can make use of the `CImageDisplay` class. The ICL also provides functionality to scale and rotate native `CFbsBitmaps`, using the `CBitmapRotator` and `CBitmapScaler` classes. These classes perform their functions using active objects in your application's thread. Unlike the image-decoding and -encoding classes, they do not have the facility to perform in a subthread.

Image Decoding

Decoding of images is achieved using the `CImageDecoder` class, defined in `ImageConversion.h`. It supports decoding of an entire image in one go, or 'progressive' or 'streamed' decoding, where the image data arrives in parts. This allows a partial image to be rendered before the entire image has been received.

If you need only display an image on the screen, you should consider using `CImageDisplay`. This class has numerous advantages, such as automatic animation images with multiple frames, which make it better suited to displaying images on screen. However Nokia have excluded it from the S60 SDK.

When we instantiate an image decoder, we must provide all or some of the image data immediately. For example, if we are decoding an image from a file:

```
iDecoder = CImageDecoder::FileNewL(iFs, aFileName);
```

Here, `iFs` is an `RFs` instance. It should already be connected to the file server. The method leaves if the file cannot be recognized or if it is not a supported image file. The file can contain either a partial or a complete image.

There is also a `DataNewL()` method that allows you to instantiate a decoder for processing image data stored inside a descriptor. The initial

descriptor you provide can contain an entire encoded image or only as much as is available at the time. Construction of the decoder fails if there is not enough data to determine its format, unless you specify the format explicitly.

Decoding options

There are several overloads of the `FileNewL()` and `DataNewL()` methods that allow us to specify the image decoder options, the content access intent, the MIME type of the image, the UID of the image type and subtype, and the UID of the specific image decoding plug-in to use.

The image decoder options are defined by enumeration `CImageDecoder::TOptions`:

- `EOptionNone` – no options set; this is the default
- `EOptionNoDither` – do not perform any error-diffusion dithering when loading an image; when decoding a bitmap with a different display mode to the original image, dithering is used by default; you should use this option if you intend to further resize the image after it has been decoded
- `EOptionAlwaysThread` – always perform the image decoding in a separate subthread; consider using this option if you are using the decoder within a GUI, or other, application that requires the `RunL()` functions of all active objects to be serviced promptly; it has extra overhead associated with creating a new thread, so should only be used when it is required
- `EOptionAllowZeroFrameOpen` – allows the creation of the image decoder even when not enough data is present to decode an entire image frame; this should be set when you intend to perform streamed decoding
- `EAllowGeneratedMask` – an image decoder normally only reports that transparency is possible if mask data is encoded with the image; however, some decoders have the ability to automatically generate a mask; setting this flag instructs the decoder to enable this operation, if it is available.

Several options can be combined using bitwise or.

If we specify the MIME type of an image, the decoder plug-in is selected on the basis of this rather than by recognizing the image type based on the image data itself. Similarly, we can indicate the format of the image by specifying the image type and subtype UIDs and the UID of the specific decoder plug-in to use. This means that the decoder plug-in can always be instantiated, even if no data is present at the time of creating the plug-in. The image subtype is usually set to `KNullUid`.

Image type and subtype UUIDs for image formats supported by Symbian OS are defined in `imagecodecdta.h`. Note that other image formats can be added to the ICL by phone manufacturers or third parties, so the list of image UUIDs here may not be complete.

Some image formats do not contain enough data in the header to be recognized on the basis of the encoded data itself. In these cases, it is necessary to specify either the MIME type or the image type when creating a decoder.

Decoder subclasses

When an image decoder is created, the plug-in can optionally provide a subclass of `CImageDecoder` that is returned to the client instead of `CImageDecoder` itself. This allows functionality specific to a particular image format to be provided to the client. For example, when decoding a Jpeg image, we get an instance of `CJPEGExifDecoder` allowing access to EXIF metadata stored within the image.

Basic decoding

If all the image data is available when the decoder is constructed, decoding is quite straightforward. First, we create a new instance of `CImageDecoder` providing the source data. In this example, we let the ICL automatically select an appropriate plug-in for us:

```
iDecoder = CImageDecoder::FileNewL(iFs, aFileName);  
iFrameInfo = iDecoder->FrameInfo();
```

Once the decoder is created, we can be sure that there is enough data for at least one frame since we did not specify the option `EOptionalLowZeroFrameOpen`.

Before we can decode the image, we need to create a destination `CFbsBitmap` object to store the decoded image. Depending upon what we intend to do with it next, we have two options:

- create a destination bitmap with the same size as the frame we are decoding
- scale the image down during the decoding operation to save memory.

If the image is to be encoded using a different format for a conversion operation, we want to decode the entire image. If it is to be displayed on screen, we may want to scale it down. Note that very large images may have to be scaled down, if there is insufficient memory to store the entire decoded image in memory at once.

To scale the image down, we create a `CFbsBitmap` that is smaller than the encoded image. Some decoders support arbitrary scaling of an image during decoding; in this case, we can simply create the `CFbsBitmap` at the exact size that we wish the decoded image to be. In general though, decoders only support scaling down of images by a factor of 2, 4 or 8. `CImageDecoder` provides some methods to help work out what scaling factor should be used. When arbitrary scaling is not supported, we have to perform further scaling once the image has been decoded if we need to fit it into a predefined area. Many user interface components can do this for us, however.

```
if (iFrameInfo.iFlags & TFrameInfo::EFullyScaleable)
{
    TRect fitRect(desiredSize);
    ShrinkToAspectRatio(iFrameInfo.iOverallSizeInPixels, fitRect);
    decodeImageSize = fitRect.Size();
}
else
{
    TInt reductionFactor = iDecoder->ReductionFactor(
        iFrameInfo.iOverallSizeInPixels, desiredSize);
    if (reductionFactor>3) reductionFactor=3;
    User::LeaveIfError(iDecoder->ReducedSize(
        iFrameInfo.iOverallSizeInPixels,
        reductionFactor, decodeImageSize));
}
```

The flag `TFrameInfo::EFullyScaleable` indicates if the decoder supports arbitrary scaling of the image. If it does, we simply match the aspect ratio of the desired size to that of the image. If the flag `TFrameInfo::EConstantAspectRatio` is not set, this is not necessary as the image can be stretched during decoding.

If the decoder does not support arbitrary scaling, we use the `ReductionFactor()` and `ReducedSize()` methods to work out what size the decoder can scale to. The `ReductionFactor()` method gives us the factor by which we need to scale the image, as a logarithm to the base 2. Since the image decoder only supports scaling by factors of 2, 4 and 8, we must make sure that the reduction factor we use is not greater than 3 (since $2^3 = 8$). The `ReducedSize()` method is then used to calculate the final size for the bitmap. It is important that you use this method rather than calculating the final size yourself as different decoder plug-ins may use different rounding algorithms.

Some third-party ICL decoder plugins supplied on Symbian OS phones support reduction factors greater than 3. If your desired reduction factor is greater than 3, you can attempt to call `ReducedSize()` anyway and only reduce it to 3 if this returns an error.

Once the destination bitmap has been created, we can call `Convert()` to start the conversion process. This takes a zero-based frame number as a parameter, for images that consist of more than one frame; it defaults to zero.

```
iBitmap = new(ELeave)CFbsBitmap();
User::LeaveIfError(iBitmap->Create(decodeImageSize,
                                iFrameInfo.iFrameDisplayMode));

iDecoder->Convert(&iStatus, *iBitmap);
SetActive();
```

Here, the destination bitmap has the same display mode as the source image. If we intend to display the image on the screen, it is better to use the same display mode as the screen. This allows the decoder to perform error diffusion if necessary.

When the decode operation has completed, the request is completed and our active object `RunL()` is executed, indicating success or an error. If the decoding operation is successful, the error code is `KErrNone` and we can display the image on the screen using any method that accepts a `CFbsBitmap`, or encode it as another format for a conversion operation. If we receive an error of `KErrUnderflow`, this means that there was not enough data to fully decode the image (see the section on progressive decoding).

Image masks

The presence of a mask in a frame is indicated by the setting of the `ETransparencyPossible` flag in the frame information. The mask can be decoded along with the image by using the overload of `CImageDecoder::Convert()` that takes a second `CFbsBitmap`. The mask bitmap must have the same size as the main bitmap; the display mode is usually different.

The display mode of the mask can be determined by the flag `EAlphaChannel` in the frame information. If this flag is set, then the mask is an 8-bit alpha blend; hence we should use display mode `EGray256` for the mask bitmap. Otherwise, the mask is a simple 1-bit mask and the mask bitmap should have display mode `EGray2`.

Frame information

An image can contain a number of frames, each of which is, essentially, a separate image. Multiple frames are only supported by some image formats. They are often used to achieve animation.

After the image decoder has been instantiated and header processing has completed, we can query the number of frames present using the

`FrameCount()` method. Information about each frame is returned by the `FrameInfo()` method that we saw earlier. In a Jpeg image that contains a thumbnail in the EXIF metadata, two frames are reported: the main image and the thumbnail.

Images that contain frames that are intended to be animated contain timing information in the `iDelay` member of `TFrameInfo`. This determines how long the frame should remain on screen, and the flags `ELeaveInPlace`, `ERestoreToBackground` and `ERestoreToPrevious` determine how to transition from one frame to the next. However, it should not be necessary to implement this functionality yourself since it is provided by the `CImageDisplay` class.

Further information about the frames in an image is represented by the `CFrameImageData` class and is retrieved using the `FrameData()` method. This provides access to data specific to various image formats, such as palettes, compression methods or quality settings. Each decoder can provide its own classes deriving from `TFrameDataBlock` or `TimageDataBlock`, which can be encapsulated within `CFrameImageData`; many are defined in `imagecodecd.h`.

Determining the type of an encoded image

The `CImageDecoder` class provides two static methods for determining the format of an encoded image:

```
static void GetMimeTypeFileL(RFs& aFs, const TDesC& aFileName,
                             TDes8& aMimeType);
static void GetMimeTypeDataL(const TDesC8& aImageData, TDes8& aMimeType);
```

These methods can be used to determine if an encoded image is recognized by the ICL with the available decoder plug-ins. Note that successful recognition of some image data by these methods usually implies that the image can be decoded. It is possible, though unlikely, to get false positives using these methods and the image data could be corrupted, leading to an error when you attempt to decode it.

Progressive decoding

Progressive decoding, also known as streamed decoding, allows you to start decoding an image before all of the data is available. This is useful when the image data is being downloaded over a network, for example. It enables you to render a partial image during downloading or display some frames of a multi-frame image before all frames have been received. Progressive decoding is more complex than basic decoding. The process is shown in Figure 21.3.

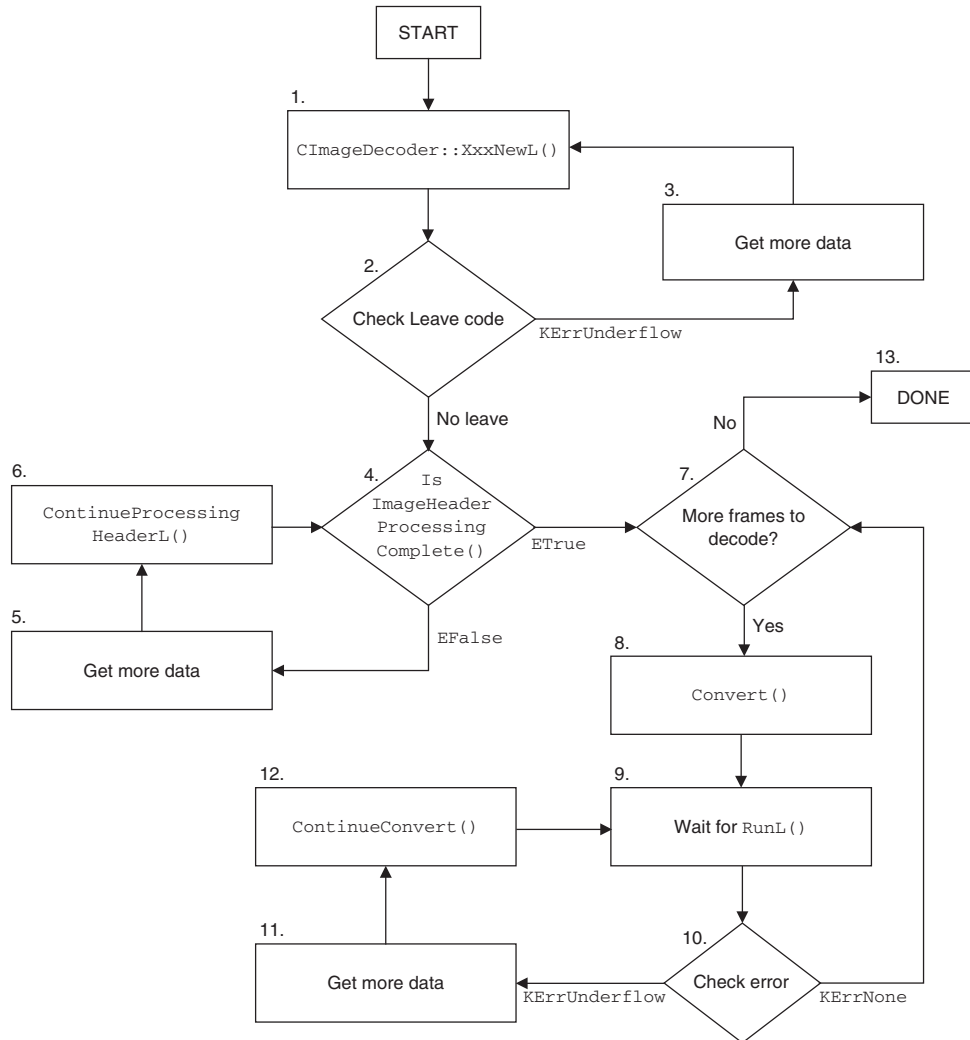


Figure 21.3 Progressive decoding of an image

The first step is to create the image decoder using either `CImageDecoder::FileNewL()` or `CImageDecoder::DataNewL()`. Which-ever method you are using, it is assumed here that the data source does not contain the entire image, merely the first part of it, however small.

If you do not explicitly specify either the MIME type of the data or the type of the image, the construction of `CImageDecoder` fails if there is not enough data present to recognize the image format from the data provided. In this case, you should wait for more data to become available and then try again (Steps 2, 3 and 1).

Once the image data has been successfully recognized and a `CImageDecoder` has been instantiated, we must then continue to read data until header processing has been completed (Steps 4, 5 and 6). We continue appending data to the file or descriptor until `IsHeaderProcessingComplete()` returns `ETrue`; at this point, we can start attempting to decode the first frame.

The first time we reach Step 7, we know that the condition is true as we have not yet decoded any frames and an image is assumed to contain at least one frame.

When we call `Convert()`, we must then wait asynchronously for our `RunL()` (Steps 8, 9 and 10). In the normal case (i.e. when no errors are detected in the image data), we receive an error of either `KErrNone`, indicating that the frame was successfully decoded, or `KErrUnderflow` indicating that more data is required to decode the frame. If we receive an error of `KErrUnderflow`, we must wait for more data to become available; this data should be appended to the existing data source and then `ContinueConvert()` called to continue the decoding process (Steps 11, 12 and back to 9).

Once an entire frame has been decoded, we can either move onto the next frame if it exists (Steps 10, followed by 7, 8, 9 and 10 again) or conclude that the image processing is now complete (step 13).

In Steps 8 and 12, you must provide the same `CFbsBitmap` instance while decoding a single frame. You can use a new `CFbsBitmap` when moving onto the next frame, that is, when moving between Steps 10, 7 and 8.

The details of Steps 3, 5 and 11 depend upon your data source. It is likely that, since you are using progressive decoding, the data is arriving asynchronously and so these stages may involve an asynchronous wait for more data. In this case, the data is receiving concurrently with the decoding; you should be careful to coordinate the two activities properly. As new data arrives, it should be appended to the original file or descriptor.

There are two possible places in Figure 21.3 when the image could be rendered. If the frame being decoded does not have the flag `EPartialDecodeInvalidset`, then we can render a partially decoded frame between Steps 10 and 11. This would allow the user to see the portion of the image that has been received already, which is the main advantage of progressive loading of images.

In all cases, the frame can be rendered between Steps 10 and 7 when an error of `KErrNone` is received, as in this case the frame has been fully decoded.

Figure 21.3 does not show any error conditions, for the sake of clarity. However, there are a number of steps where your code need to consider error cases.

In any of Steps 3, 5 and 11, if no more data is available this means that the source data has become truncated. You should fail with `KErrUnderflow`; you can still render a partial frame if one exists.

If an error other than `KErrNone` or `KErrUnderflow` is received in Steps 2 or 10, again decoding has failed and you should not continue.

Buffered decoding

The class `CBufferedImageDecoder` can be used to simplify the process of progressive image decoding slightly. The class encapsulates the `CImageDecoder` and the descriptor in which the encoded image is stored. It can be created before any image data is present and it can be reused.

Its use is very similar to that of `CImageDecoder` when we perform progressive decoding. However, once we have passed some data into the buffered decoder, we no longer need to keep that data ourselves, as it has been appended to an internal buffer. The upshot of this is that the encoded image data is always copied, meaning slightly greater overheads.

The usage of the buffered decoder is as follows:

1. Call `OpenL()`, optionally specifying a MIME type or image type UIDs, as with `CImageDecoder`.
2. While `ValidDecoder()` returns `EFalse`, add more data to the decoder using the `ContinueOpenL()` method.
3. When `ValidDecoder()` returns `ETrue`, write data via the `AppendDataL()` method and then call `ContinueProcessingHeaderL()`. Continue making this sequence of calls until `IsImageHeaderProcessingComplete()` returns `ETrue`.
4. The frame information can now be queried using the `FrameCount()` and `FrameInfo()` methods. To begin decoding of a frame, create a `CFbsBitmap` of the appropriate size and then call `Convert()`.
5. The decoding process now takes place in the same way as with `CImageDecoder`, except that when more data becomes available we write it to the decoder using the `AppendDataL()` method.

To reuse a `CBufferedImageDecoder`, simply call `Reset()` and then start again.

Image Encoding

Encoding of images is done using class `CImageEncoder`, defined in header `ImageConversion.h`. The class is similar to `CImageDecoder` in the way that it is used, but is somewhat simpler. It encodes a

`CFbsBitmap` to a number of external formats, where the destination for encoded data can be either a memory buffer or a file; it does not support streamed, progressive or buffered encoding.

Clearly, when we are encoding an image, we must specify a destination image format to use. There are two ways of doing this:

- provide a MIME type for the encoded image; an image encoder plug-in is selected based on this MIME type
- specify the image type, subtype and encoder plug-in to use by their UUIDs.

The set of formats that can be encoded into by `CImageEncoder` depends entirely on what image encoder plug-ins are present on the phone. As with decoder plug-ins, further encoding plug-ins may be added by phone manufacturers or third parties.

If you know in advance what image format you wish to encode, and it is supported by the release of Symbian OS that you are targeting, it is easiest to hard code the MIME type of the format. This allows the ICL to automatically select a suitable encoder plug-in for you.

Querying supported encoding formats

The ICL provides the ability to query what image formats can be encoded. Since all of the encoders are ECOM plug-ins, it would be possible to do this by interfacing directly to ECOM; using the ICL to query the available plug-in is far easier, however. Class `CImageEncoder` contains the following methods to achieve this:

- `GetImageTypesL()` returns all image types that can be encoded, together with short descriptions
- `GetImageSubTypesL()` returns the list of image subtypes associated with a given image type
- `GetFileTypesL()` returns the list of MIME types, that can be encoded, together with their associated filename extensions and image type UUIDs.

Encoding an image

As with the decoder, an image encoder is instantiated using the `FileNewL()` or `DataNewL()` methods. The methods are similar to the equivalent ones in `CImageDecoder`; the main difference being that it is mandatory to specify the format.

When writing the image to a file, it is desirable to ensure that the file has the expected extension for the image type in use, as reported by `GetFileTypesL()`. Any existing file with the same name is overwritten.

If you are using the `DataNewL()` method, the destination for the encoded image is given as type `HBuFC*&`. You should pass in a reference to a `NULL` pointer; a descriptor of a suitable size is created and the pointer is updated. Ownership of the descriptor is transferred to the caller.

The image encoder allows you to specify some options when it is created. The options are defined by enumeration `CImageDecoder::TOptions`. The only option available is `EOptionAlwaysThread`, which has the same meaning as the identically named option for the decoder.

```
iEncoder = CImageEncoder::FileNewL(iFs, fileName, mimeType);
```

Once the encoder has been instantiated, we can begin the encoding process. This is done using the `Convert()` method; we provide a `TRequestStatus` to signal when the encoding is complete, the source bitmap and, optionally, some frame image data represented by `CFrameImageData`. This can be used to configure the encoder.

```
iEncoder->Convert(&iStatus, *iBitmap);
SetActive();
```

We then wait for a `RunL()` indicating success or otherwise. You must ensure that the `CFbsBitmap` that you pass into the `Convert()` method is not deleted or altered until the conversion has completed.

The error code we receive in the `iStatus` is `KErrNone` if the encoding was a success; in this case, the data destination we provided when we instantiated the encoder contains an encoded image. Any other value indicates that there was some problem with the encoding process. Likely errors are `KErrNotFound`, if no suitable encoder plug-in could be found, or `KErrNoMemory` if the system runs out of memory.

Configuring the encoder

When we call `convert` on the encoder, we have the option of providing some frame image data. Encoders that support any configuration provide subclasses of `TImageDataBlock` or `TFrameDataBlock` that can be added to the `CFrameImageData` passed into the `Convert()` method. These classes are defined in `imagecodecddata.h`. For example, you can use class `TBmpImageData` to configure the bits per pixel of the Bmp encoder, or class `TJpegImageData` to configure the quality of the Jpeg encoder.

```
TJpegImageData data;
data.iSampleScheme = TJpegImageData::EColor444;
data.iQualityFactor = 75;
```



```

void CImageDisplayer::SetImageSourceL(const TDesC& aFileName)
{
    ...
    User::LeaveIfError(
        iImageDisplay->SetImageSource(TMMFileSource(aFileName));

    iImageDisplay->SetOptions(CImageDisplay::EOptionMainImage |
                             CImageDisplay::EOptionAutoRotate);
    iImageDisplay->SetupL();
    ...
}

```

Note that the filename or descriptor used when instantiating a class derived from `TMMSource` must persist for the lifetime of the `CImageDisplay` object. The `TMMSource` class itself does not need to.

After `SetImageSource()` has been called, we can set the options to use for displaying the image. In this example, we specify to use the main image rather than a thumbnail and to automatically rotate the image so that it has the correct orientation. The available options are as follows:

- `EOptionThumbnail` – use a thumbnail rather than the main image, if available; this saves memory and speeds up the decoding time
- `EOptionMainImage` – use the main image, even if a thumbnail is embedded in the image
- `EOptionRotateCw90` – rotate the image by 90° clockwise
- `EOptionRotateCw180` – rotate the image by 180° clockwise
- `EOptionRotateCw270` – rotate the image by 270° clockwise
- `EOptionMirrorHorizontal` – flip the image about the horizontal axis
- `EOptionMirrorVertical` – flip the image about the vertical axis
- `EOptionAutoRotate` – automatically rotate the image as dictated by the header data.

It is mandatory to specify either `EOptionThumbnail` or `EOptionMainImage`. Failure to do so results in the `SetupL()` call leaving with `KErrArgument`. Once we have set the options, calling `SetupL()` attempts to recognize the image data and find and load a plug-in to decode it.

Before we can display the image, we must set a size for the destination; this would usually be the size at which we intend to draw the image.

```

TSize sizeToUse;
const CImageDisplay::RImageSizeArray& recommendedSizes =
    iImageDisplay->RecommendedImageSizes();

```



```

if (recommendedSizes.Count() == 0)
{
    // no recommended sizes - use our controls size
    sizeToUse = Rect().Size();
}
else
{
    TRect myRect = Rect();
    TSize imageSize = recommendedSizes[0];
    ShrinkToAspectRatio(imageSize, myRect);
    sizeToUse = myRect.Size();
}
iImageDisplay->SetSizeInPixels(sizeToUse);
iImageDisplay->Play();

```

Here, we scale down the recommended size – the size of the decoded image – to fit within the control, while maintaining the aspect ratio. Once the size has been set, we call `Play()` and then wait for a callback.

```

void CImageDisplayer::MiidoImageReady(const CFbsBitmap* aBitmap,
    TUint aStatus, const TRect& /*aUpdatedArea*/, TInt aError)
{
    if (aError == KErrNone)
    {
        if (aStatus & CImageDisplayPlugin::EStatusFrameReady)
        {
            iLastBitmap = aBitmap;
            DrawNow();
        }
        ...
    }
    else
    {
        // handle the error
    }
}

```

If we don't want to animate the image, this is all the we need to do. To achieve the animation, however, a little more work is required.

```

void CImageDisplayer::MiidoImageReady(const CFbsBitmap* aBitmap,
    TUint aStatus, const TRect& /*aUpdatedArea*/, TInt aError)
{
    if (aError == KErrNone)
    {
        ...
        if (!(aStatus & CImageDisplayPlugin::EStatusNoMoreToDecode))
        {
            // there's more to decode
            iImageDisplay->Play();
        }
        else if (iImageDisplay->ImageStatus() & CImageDisplay::EImageAnimated)
        {
            // we have completed a playthrough of the animation
            // play it again

```

```

        iImageDisplay->StopPlay();
        iImageDisplay->Play();
    }
    ...
}

```

The absence of flag `EStatusNoMoreToDecode` indicates that there remains more data to be decoded. This usually means further frames in an animation, but it could also be that we have not fully decoded the first frame. Calling `Play()` again prompts `CImageDisplay` to continue the animation (with the appropriate delay) or finish decoding the image.

When the animation has played through once, the flag `EStatusNoMoreToDecode` is set. If we wish to repeat the animation, we call `StopPlay()` to reset the animation to the beginning followed by `Play()` to play it again. We should only do this if the image is animated; if we were to call `Play()` again with a single frame image, it would cause it to be decoded again resulting in an infinite loop.

Please note that you cannot call `Reset()` or delete the `CImageDecoder` instance from within the `MiidoImageReady()` callback. Attempting to do so results in a panic.

As with many other parts of the multimedia subsystem, `CImageDisplay` is based on ECOM plug-ins. The image formats that are supported by the class are directly influenced by the available plug-ins. There is a generic ICL wrapper plug-in that ensures that all image formats supported by `CImageDecoder` are also supported by `CImageDisplay`. A specific plug-in for Jpeg images ensures that any rotation information contained within a Jpeg EXIF header is honored. Other specific plug-ins exist for other formats, such as Multiple-image Network Graphics (MNG) images.

An image display plug-in can optionally provide an extension interface allowing access to plug-in-specific functionality. The `CImageDisplay::ExtensionInterface()` method provides access to these interfaces, which are identified by UID. Extension interfaces provided in Symbian OS v9.1 include:

- `MExifImageDisplayExtension`, which gives access to Jpeg EXIF metadata and to control the quality of the Jpeg-scaling algorithm. This interface is identified by UID `KExifImageDisplayExtensionUid`
- `MGenIclImageDisplayExtension`, which gives access to some of the functionality of the underlying image decoder plug-in, if one exists. The UID `KUidGenIclImageDisplayPluginExtUid` identifies this interface.

The `CImageDisplay::ExtensionInterface()` method returns `KErrNotSupported` if you request an interface that is not supported by the current image display plug-in. You should be sure to check for this error before attempting to use the interface pointer provided.

Image Transformations

The classes `CBitmapScaler` and `CBitmapRotator`, both defined in `BitmapTransforms.h`, can be used to scale and rotate bitmaps represented by `CFbsBitmap`. They are similar to `CImageEncoder` and `CImageDecoder` in the way that they are used, but do not provide the option to perform the operation in a separate thread.

Both classes have two modes of operation: transforming a bitmap in-place or transforming a source bitmap and writing the result into a separate bitmap, leaving the source unchanged.

Rotating bitmaps

The class `CBitmapRotator` is used to rotate bitmaps. Once the call is instantiated, simply call the `Rotate()` method.

```
iRotator->Rotate(iStatus, *iSourceBitmap, *iDestBitmap,  
                CBitmapRotator::ERotation90DegreesClockwise);  
SetActive();
```

The destination bitmap should be instantiated, but `Create()` does not need to have been called. `CBitmapRotator` atomically creates the destination bitmap of the appropriate size and shape, depending on what rotation angle is applied. The rotation angle must be one of the values from enumeration `CBitmapRotator::TRotationAngle`.

You must ensure that both the source and the destination bitmaps remain in scope for the duration of the rotation operation.

When the rotation is complete, the `TRequestStatus` you provided is signaled. Note that the rotation operation itself requires many separate occurrences of the active object's `RunL()` internally; hence you must yield to the active scheduler after calling `Rotate()`.

Scaling bitmaps

Use of `CBitmapScaler` is very similar to that of `CBitmapRotator`. The `Scale()` method is called to initiate the asynchronous scaling operation.

```
iScaler->Scale(iStatus, *iBitmap, iNewSize, ETrue);  
SetActive();
```

If, as here, you are using the in-place variant of the `Scale()` operation, the size of the new bitmap is specified using `TSize`. If you provide a destination bitmap object to write the scaled bitmap into, the size of that destination determines the scaling factor. You must have already called `Create()` on the destination bitmap, if you provide one.

You can change the quality of the scaled image by calling the `SetQualityAlgorithm()` method before scaling the image. You can choose between minimum, medium and maximum quality. The high-quality algorithm takes longer to process; minimum quality is faster.

The final parameter indicates whether or not to maintain the aspect ratio of the scaled bitmap. If it is set to `EFalse`, the bitmap is stretched to fit the destination size exactly. If it is set to `ETrue`, the bitmap is scaled down so that it fits inside the rectangle defined by the destination size – that is, such that the width or height of the final bitmap match the width or height of the requested destination size, but the other dimension is smaller if the aspect ratio differs.

Again, you must ensure that the source and destination bitmaps remain in scope for the duration of the scaling operation and you must yield to the active scheduler to allow the scale operation to take place.

21.3 Camera API

Symbian OS v9.1 provides the ECAM API to allow access to any onboard camera on the phone. It provides functionality to display a viewfinder and capture still images from the camera. It is also possible to capture video using the ECAM API; however we recommend that you use the video recording API outlined in Section 21.1 instead.

The class `CCamera` provides this functionality; it resides in `ECam.h`. To use this class, you must provide a class that derives from `MCameraObserver2` or `MCameraObserver`. It is recommended that you use `MCameraObserver2` where it is supported, as this class supersedes the older `MCameraObserver`. You must have capability `UserEnvironment` to use `CCamera`.

The implementation of the camera API is not supplied by Symbian since it is highly dependent on the hardware architecture of each phone. Each phone manufacturer provides their own implementation, using the same `CCamera` API. You may therefore find that the behavior differs slightly between phones and manufacturers.

Some Symbian OS v9.1 phones from Nokia do not support `MCameraObserver2`; to work with these phones, you have to use the old `MCameraObserver` class.

Selecting a Camera

Before you attempt to use the camera on a Symbian OS phone, you should first check that one exists. The static `CCamera::CamerasAvailable()` method returns the number of cameras present. Many phones capable of video telephony have more than one; often one facing towards the user and one facing away.

To instantiate `CCamera`, simply call `CCamera::NewL()`, specifying the index of the camera you want to use as the `aCameraIndex` parameter. If you want to select a camera to use on the basis of its orientation or capabilities, you should instantiate a `CCamera` instance for each one and check the capabilities using the `CameraInfo()` method. This returns a `TCameraInfo` class, containing a variety of information about the camera in question.

The camera's orientation, `iOrientation`, can take one of the following values:

- `EOrientationOutwards` – the camera faces away from the user and so would usually be used to take pictures
- `EOrientationInwards` – the camera faces towards the user and is probably used for video telephony
- `EOrientationMobile` – the orientation of the camera can be changed by the user; in this case, you can't rely on knowing the present orientation of the camera
- `EOrientationUnknown` – the orientation of the camera is not known.

The member `iOptionsSupported` of `TCameraInfo` is a bit field indicating a number of options that a camera supports. The values for the bits are defined by enumeration `TCameraInfo::TOptions`. The following values are useful in selecting a camera:

- `EImageCaptureSupported` – indicating if still-image capture is supported by the camera
- `EVideoCaptureSupported` – indicating if video capture is supported.

Other information contained in this field is discussed later.

Setting up the Camera

Once we have selected a camera to use and instantiated the `CCamera` instance, the next step is to reserve it. Reserving the camera, if successful, grants our application exclusive access to the camera until our application

releases it or is pre-empted by another application. Camera reservation is based on priority; higher-priority camera clients can pre-empt lower-priority ones. The priority is set when we instantiate CCamera object, using the priority parameter of the NewL() method.

To reserve the camera, simply call Reserve(). This is an asynchronous call and we expect a callback indicating success or failure.

```
iCamera->Reserve();
...

void CCameraEngine::HandleEvent(const TECAMEvent& aEvent)
{
    if (aEvent.iEventType == KUidECamEventReserveComplete)
    {
        if (aEvent.iErrorCode == KErrNone)
        {
            iCamera->PowerOn();
        }
        else
        {
            // handle the error
        }
    }
    ...
}
```

If you are using MCameraObserver rather than MCameraObserver2, the corresponding callback is ReserveComplete(). In this example, we power up the camera immediately. If we were just reserving the camera with the intention of using it later, we would not do this.

A call to PowerOn() results in the camera itself being switched on. This results in greater drain on the battery, so it should not be done until necessary. After this call, we receive another callback.

```
void CCameraEngine::HandleEvent(const TECAMEvent& aEvent)
{
    ...
    if (aEvent.iEventType == KUidECamEventPowerOnComplete)
    {
        if (aEvent.iErrorCode == KErrNone)
        {
            // success! Camera is now active.
            iState = ECameraReady;
            ...
        }
        else
        {
            iCamera->Release();
            // handle the error
        }
    }
    ...
}
```

The corresponding callback for `MCameraObserver` is `PowerOnComplete()`.

You'll have noticed that the `MCameraObserver2::HandleEvent()` callback method is used for multiple purposes. Each event type is given a UID; some event UIDs to expect are as follows:

- `KUideCamEventReserveComplete` – this is a response to a call to `CCamera::Reserve()`, indicating the success or otherwise of the camera reservation
- `KUideCamEventPowerOnComplete` – indicates the success or otherwise of powering on the camera hardware
- `KUideCamEventCameraNoLongerReserved` – this can occur at any time while we have reserved the camera and indicates that the reservation of the camera has been revoked, usually due to another application reserving it.

Displaying the Viewfinder

After we have successfully powered on the camera hardware, we can display a viewfinder. The camera subsystem can render the viewfinder for us, using direct screen access, or we can render the viewfinder ourselves, using bitmaps passed to us at regular intervals. See [Sales 2005, Chapter 11] for more information on direct screen access.

The option to use is determined by the capabilities of the camera. The `iOptions` member of `TCameraInfo` may have one of the following two bits set:

- `EViewFinderBitmapsSupported` – we have to render the viewfinder ourselves using bitmaps supplied by the camera
- `EViewFinderDirectSupported` – rendering of the viewfinder is performed for us using direct screen access.

You should always check which method is supported before attempting to render a viewfinder. It is possible that neither method is supported, in which case rendering a viewfinder is not possible. You may still be able to take still images or videos, however.

A viewfinder of either type can be stopped at any time by calling the `StopViewFinder()` method.

Direct screen access viewfinder

Rendering of a direct screen access viewfinder is the most efficient method. The camera subsystem takes care of the rendering and is usually

optimized to make use of any hardware acceleration present. To draw a direct screen access viewfinder, we use the `StartViewFinderDirectL()` method.

```
void StartViewFinderDirectL(RWsSession& aWs,  
                           CWScreenDevice& aScreenDevice,  
                           RWindowBase& aWindow,  
                           TRect& aScreenRect)
```

The parameters are similar to those used to initialize video playback, which also uses direct screen access. The first three can be retrieved from an application GUI environment; `aScreenRect` is the rectangle, relative to the physical screen, in which the viewfinder is to be rendered. Once a successful call has been made to `StartViewFinderDirectL()`, the viewfinder is visible to the user. No further action is required by the application.

Bitmap-based viewfinder

When a bitmap-based viewfinder is active, the camera passes bitmaps to the application at regular intervals, and the application draws them on the screen as appropriate. The bitmaps are passed at a rate fast enough to ensure a smooth viewfinder display.

To start the viewfinder, make a call to `StartViewFinderBitmapsL()`. This method requires a single `TSize` parameter that determines the size of the bitmaps that you receive. It should be set to the size of the screen area into which you intend to draw the viewfinder.

Following this, bitmaps are passed to you via the `ViewFinderReady()` callback method (or `ViewFinderFrameReady()` if you are using `MCameraObserver`).

With `MCameraObserver::ViewFinderFrameReady`, the bitmap passed to you is simply a `CFbsBitmap` which can be drawn to the screen in the usual way.

If you are using `MCameraObserver2`, the bitmap is represented by class `MCameraBuffer`. The `CFbsBitmap` can be retrieved from this class using the `BitmapL()` method. An `MCameraBuffer` is capable of encapsulating several frames which may be encoded; for the sake of a viewfinder, you should expect a single, unencoded frame in the form of a `CFbsBitmap` instance.

Capturing Still Images

Before still images can be captured, you should query the capabilities of the camera in use. There are two parameters that need to be set based on the capabilities: the format of the captured image and its size.

Selecting the image format

The formats that are supported are determined by the member `iImageFormatsSupported` of class `TCameraInfo`. This is a bit field of values from enumeration `CCamera::TFormat`. Some cameras support a single format. Others may support a number of formats, in which case you choose the one most suitable for your application. Formats that may be supported include the following:

- `EFormatJpeg` and `EFormatExif`; these are encoded formats that need decoding before they can be displayed on the screen; the data arrives encoded in a descriptor
- `EFormatFbsBitmapColorXxx`; these are uncompressed formats in which the captured image is represented by a `CFbsBitmap` object
- `EFormatRgbXxBitRGBXxx` and `EFormatYUVXxx`; these are raw data formats, the characteristics of which are determined by the exact format; the data arrives in a descriptor.

The formats supported by a camera may be influenced by the resolution of that camera. High-resolution cameras are likely to present the image data in a compressed format, as the memory required to store the image in its uncompressed form would be too great.

Once you have selected a format to use, you can enumerate the capture sizes that are supported for that format. The number of image sizes supported is stored in the `iNumImageSizesSupported` member of class `TCameraInfo`. The sizes themselves are returned by the `CCamera::EnumerateCaptureSizes()` method.

```
RArray<TSize> iSizeArray;
...
for (TInt i=0; i<iCameraInfo.iNumImageSizesSupported; ++i)
{
    TSize size;
    iCamera->EnumerateCaptureSizes(size, i, iFormat);
    iSizeArray.AppendL(size);
}
```

The `iFormat` member contains a value from `CCamera::TFormat`, selected from `iImageFormatsSupported`.

Preparing for the capture

Before you can capture an image, you need to prepare the `CCamera` object using the `PrepareImageCaptureL()` method. This allows the camera subsystem to allocate any memory necessary and perform any other setup required to capture an image; it only needs to be called once

for many image captures. It can be done at any time before the image capture takes place; if images are being captured in response to user input, you should prepare the camera in advance to minimize the delay when the capture is performed.

```
iCamera->PrepareImageCaptureL(iFormat, aSizeIndex);
```

The parameter `aSizeIndex` is the index of capture size that you wish to use; it corresponds to the `aSizeIndex` parameter of the `EnumerateCaptureSizes()` method.

After a successful call to `PrepareImageCaptureL()`, the camera is ready to start capturing images. No asynchronous callback is received.

Capturing the image

Image capture is an asynchronous operation. It is initiated simply by making a call to `CaptureImage()`.

```
iCamera->CaptureImage();
```

You then receive a callback to the `MCameraObserver2::ImageBufferReady()` method.

```
void CCameraEngine::ImageBufferReady(MCameraBuffer& aCameraBuffer,
                                     TInt aError)
{
    if (aError == KErrNone)
    {
        // process the image
    }
    else
    {
        // Handle the error
    }
}
```

To cancel an image capture, you can call `CancelImageCapture()`. You must not call `CaptureImage()` again before you receive the `ImageBufferReady()` callback without canceling the ongoing image capture first.

The image is encapsulated by class `MCameraBuffer`. The class can store image data in a number of ways and can store multiple frames. In the case of still-image capture, it contains a single frame of the requested format.

If you requested an `EFormatFbsBitmapColorXxx` format when you configured the image capture, the `MCameraBuffer::BitmapL()` method returns a handle to the `CFbsBitmap` containing the image data.

For other formats where the data is presented in a descriptor, the data can be accessed using the `MCameraBuffer::DataL()` method. The interpretation of this data depends on the format that you requested.

If the format is `EFormatJpeg` or `EFormatExif`, the descriptor can be decoded to a `CFbsBitmap` using `CImageDecoder` as described in Section 21.2; alternatively, the data can be written directly to a Jpeg file.

Some camera implementations are able to store the data in a shared chunk. In this case, the `ChunkL()` and `ChunkOffsetL()` methods return details of it. See [Sales 2005, Chapter 13] for more information on shared chunks.

Once you have finished processing an `MCameraBuffer` object, you must call its `Release()` method. This allows the memory to be freed or reused by the camera subsystem. Failure to do so results in memory leaks.

If you are working with a phone that only supports `MCameraObserver` instead of `MCameraObserver2`, the equivalent callback is `MCameraObserver::ImageReady()`. In this case, you are given a `CFbsBitmap*` and an `HBufC8*`; these can be considered equivalent to the objects returned by `MCameraBuffer::BitmapL()` and `MCameraBuffer::DataL()`, respectively. Only one of them is valid, as determined by the image format in use.

21.4 Tuner API

The tuner API allows an application to control a radio tuner on a phone, and play and record audio from it. Symbian provides the Tuner API, as defined in `tuner.h`, which phone manufacturers can implement on phones that have a radio tuner.

At time of writing, the only phones that support the Tuner API are the Sony Ericsson P990 and W950, which allow control of the FM radio. Symbian OS phones made by other manufacturers that have an FM radio do not yet support the Tuner API.

Tuning

The `CMMTunerUtility` provides the basis for the Tuner API. This class allows you to perform tuning and acts as a factory for the other classes that form the Tuner API. To use it, you must derive from `MMMTunerObserver`.

Before instantiating a tuner utility, you should check that a tuner is present. The static `CMMTunerUtility::TunersAvailable()` method indicates how many tuners are present on the phone. If this method

returns a value greater than 0, you can call `CMMTunerUtility::NewL()`.

The static `CMMTunerUtility::GetCapabilities()` method can be used to retrieve the capabilities of a given tuner; if more than one tuner is present, you could use the capabilities to select a suitable tuner for your needs.

Before attempting to use a tuner, you should check that it is possible to do so currently. A tuner may, for example, require an external antenna to be attached. In addition, you should never attempt to use a tuner if the phone is in ‘flight mode’.

You can determine the requirements of the tuner using the `iAdditionalFunctions` member of class `TTunerCapabilities`; this is a bit field of values from `TTunerCapabilities::TTunerFunctions`. You should check the bits `ETunerFunctionRequiresAntenna` and `ETunerFunctionAvailableInFlightMode`. For example:

```
TTunerCapabilities caps;
User::LeaveIfError(CMMTunerUtility::GetCapabilities(0, caps));
if (caps.iAdditionalFunctions &
    TTunerCapabilities::ETunerFunctionAvailableInFlightMode)
{
    // we can use the tuner regardless of the flight mode setting
    ...
}
else
{
    TBool flightMode;
    User::LeaveIfError(iTuner->GetFlightMode(flightMode));
    if (flightMode)
    {
        // we cannot use the tuner until flight mode is enabled
        ...
    }
    else
    {
        // flight mode is disabled; we can use the tuner
        ...
    }
}
```

The state of the external antenna can be checked using the `IsAntennaAttached()` method. An attempt to use a tuner when either the flight mode or the antenna state does not allow it results in an error of `KErrNotReady`.

Unlike the ECAM API discussed in Section 21.3, the tuner does not have explicit control over the power state of the tuner hardware; there is no `PowerOn()` method. This is because the Tuner API allows multiple clients to use the same tuner hardware concurrently. The hardware remains powered on while at least one client requires it.

If the flight mode and antenna states allow the tuner to be used, you can start using it by issuing a `Tune()` request:

```
iTuner->Tune(TFrequency(98500000), CMMTunerUtility::ETunerBandFm);
```

The class `TFrequency` is used to represent a radio frequency that can be tuned to. The unit of the frequency is Hertz; hence the value given above corresponds to 98.5 MHz in the FM band.

You should check the tuner capabilities to determine which bands are supported, which also determines whether you should use frequencies or channels; you can check the valid range of frequencies or channels for a given band using the `GetFrequencyBandRange()` and `GetChannelRange()` methods.

After making any tune or search call, you receive a callback indicating success or otherwise to the `MToTuneComplete()` method. You must not make any further tune calls until you have received this callback.

You can search for frequencies or channels that carry stations using an appropriate overload of the `StationSeek()` method.

Tuner notifications

There are a number of events for which you can request notification when using a tuner. Each notification category has an associated callback class and request method in `CMMTunerUtility`.

Tuner state change notifications can be requested using the `NotifyChange()` method; you must derive from class `MMMTunerChangeObserver` to receive these notifications. This provides the following notifications:

- changes of tuner state – that is, if the tuner hardware is powered up, if it is playing and if someone is recording from it
- changes to the currently tuned frequency or channel
- antenna attachment and detachment notifications
- flight-mode change notifications
- changes to the squelch setting.

If you derive from `MMMSignalStrengthObserver`, you can call its `NotifySignalStrength()` method to request notifications when the strength of the signal being received by the tuner changes. Similarly, you can request notifications when the audio reception changes between stereo and mono by deriving from `MMMTunerStereoObserver` and calling its `NotifyStereoChange()` method.

Control of the tuner

When you instantiate the tuner utility you have the option to specify a tuner access priority. This is used to arbitrate between multiple applications attempting to use the tuner at the same time. Only one application is granted control of the tuner at any given time; only the application that has control of the tuner can retune it and otherwise change its state.

When you call any method in the tuner utility that requires control of the tuner, a request for control is made for you. When control of the tuner is granted, you receive a callback to `MToTunerEvent()`:

```
void CRadioTuneEngine::MToTunerEvent(MMMTunerObserver::TEventType aType,
                                     TInt aError, TAny* /*aAdditionalInfo*/)
{
    if (aType == MMMTunerObserver::EControlEvent)
    {
        if (aError == KErrNone)
        {
            // we have been granted control of the tuner
            iHaveControl = ETrue;
        }
        else
        {
            // some other tuner client has pre-empted us.
            iHaveControl = EFalse;
        }
    }
    else if (aError != KErrNone)
    {
        // handle the error
    }
}
```

An event type of `MMMTunerObserver::EControlEvent` indicates that we have been granted or denied control of the tuner.

Once granted control of the tuner, we keep it until either we release it explicitly, by calling `ReleaseTunerControl()`, or we are pre-empted by another application with a higher priority for access to the tuner. If we do not wish to prevent other applications from tuning the tuner, we should release it whenever it is granted. As with audio priorities, high tuner priorities are protected by capability `MultimediaDD`.

Note that having control of the tuner does not prevent other applications playing or recording the tuner audio. Permission to do this is handled by the system-wide audio policy, in exactly the same way as for any other audio playback or recording. Other applications can also observe Radio Data System (RDS) data without requiring control of the tuner.

Closing the tuner

When you have finished using the tuner utility, you can call the `Close()` method to release any tuner resources that have been allocated to you. Doing so also affects any of the utilities that have been instantiated using the tuner utility described in the following sections: any ongoing tuner audio playback or recording is terminated and you receive no more RDS data notifications.

Before deleting your `CMMTunerUtility` instance you must ensure that any tuner utilities that you have created using the `GetXxxUtility()` methods have been deleted first. Failure to do so leads to undefined behavior – most likely a crash.

Tuner Audio Playback

Playback of audio from a tuner is performed using class `CMMTunerAudioPlayerUtility`. This class is instantiated by `CMMTunerUtility`.

```
iPlayer = iTuner->GetTunerAudioPlayerUtilityL(*this);
```

You must derive from the `MMMTunerAudioPlayerObserver` class in order to use it. The `CMMTunerUtility` object used to instantiate the tuner player utility (`iTuner`) must persist until after the player utility instance is deleted.

Use of the player utility is quite straightforward. First, you must initialize the utility using the `InitializeL()` method; you can specify an audio priority at this time. This audio priority has exactly the same meaning as in the standard audio-clip player described in Section 21.1. After initializing the utility, you receive a callback to `MTapoInitializeComplete()`.

```
iPlayer->InitializeL();
...
void CRadioTuneEngine::MTapoInitializeComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        iPlayer->Play();
    }
    else
    {
        // Handle the error
    }
}
```

Playback continues indefinitely until it is stopped or some error occurs. If an error occurs during the playback, we receive a callback to the `MTapoPlayEvent()` method. The type of event is determined by the

aEvent parameter, which takes values from enumeration `MMMTunerAudioPlayerObserver::TEventType`. This is either `ETunerEvent`, indicating some problem with the tuner such as the antenna being removed, or `EAudioEvent`, which may mean that the audio policy has revoked our permission to play the audio.

The class `CMMTunerAudioPlayerUtility` also provides a number of methods to change the volume and balance of the tuner audio and some other methods common with the standard audio-clip-player utility.

Tuner Audio Recording

The `CMMTunerAudioRecorderUtility` class can be used to record audio from a tuner. It is very similar to the `CMdaAudioRecorderUtility` class described in Section 21.1, so it is not discussed in detail here. The main difference from the standard audio recorder is that you must derive from a different class to use it, `MMMTunerAudioRecorderObserver`. It is otherwise the same in all important respects; it does not contain any playback-related functionality.

The tuner audio recorder utility is instantiated using the `CMMTunerUtility::GetTunerAudioRecorderUtilityL()` method.

The tuner player and recorder utilities can be used simultaneously. However, before you attempt to play and record audio from the radio at the same time, you should use the `CMMTunerUtility::GetCapabilities()` method to check that the `ETunerFunctionSimultaneousPlayAndRecord` capability is supported.

As with the tuner audio player utility, the `CMMTunerUtility` instance used to instantiate a `CMMTunerAudioRecorderUtility` must not be deleted before the recorder utility itself is deleted.

Using RDS data

Some phones that contain an FM radio also support Radio Data System (RDS) data access. This is used to broadcast small amounts of data along with the audio, containing details such as the name of the station. It can also be used to broadcast application-specific data in custom formats. The class `CMMRdsTunerUtility` allows applications to access the data being broadcast with a radio station. It can be used concurrently with the player and recorder utilities. The tuner utility used to instantiate `CMMRdsTunerUtility` must not be deleted before the `CMMRdsTunerUtility` itself.

The class contains a number of `GetXxx()` methods to access the RDS data. The most recently received RDS data is cached by the tuner system. However, since the data is received asynchronously over the air and may change at any time, it is more useful to receive notifications when it is received or when it changes.

To receive notifications for basic RDS data, you should derive from class `MMMRdsDataObserver`. Notifications can then be requested by calling its `NotifyRdsDataChange()` method. When you request these notifications, you can request a subset of RDS data for which you wish to receive notifications; for example:

```
iRdsTuner->NotifyRdsDataChange(*this, TRdsData::EStationName |
                                TRdsData::EProgrammeType);
```

You receive a callback whenever the data in question changes, or becomes invalid because the tuner has been retuned.

```
void CRadioTuneEngine::MrdoDataReceived(const TRdsData &aData,
                                         TUint32 aValid, TUint32 aChanged)
{
    if (aChanged & TRdsData::EStationName)
    {
        if (aValid & TRdsData::EStationName)
        {
            // PS has changed and is valid
            ...
        }
        else
        {
            // PS is no longer valid
            ...
        }
    }
    ...
}
```

The RDS utility can provide notifications for all of the commonly used RDS data fields. You can also request access to raw RDS frames as they are received over the air if you wish to decode them yourself to access application-specific data. To do this, you should derive from class `MMMRdsFrameReceiver` and request the frames that you wish to receive by calling its `ReceiveRdsFrames()` method.

Summary

This chapter has provided an introduction to the range of Multimedia APIs that are available in Symbian OS, and how to use them.

- The Multimedia Framework supplies the ability to stream, or to record and play, audio and video data.
- The Image Conversion Library supports encoding and decoding of a wide variety of image formats.

- The ECAM camera API provides access to any onboard camera on the phone. It provides functionality to display a viewfinder and capture still images from the camera.
- The radio tuner API allows an application to control a radio tuner on a phone, and play and record audio from it.

22

Introduction to SQL RDBMS

Persistent data storage is a vital component in any general-purpose computer system, including Symbian OS. In this context, 'persistent' means that if the power is removed from the device, the data is still available and usable when power is restored. RAM only retains its data while supplied with electricity.

Persistent data storage is mainly used for preserving large volumes of data that do not fit in the limited reserve of volatile memory, that need to be preserved during power-down and that do not need to be accessible as rapidly as data stored in RAM.

To satisfy this need for accessible, robust and secure data storage, Symbian OS offers a number of alternatives. These include flat files stored in the file system, stream stores, the central repository and the DBMS. Each of these can help satisfy a particular design requirement. For example, attachments to emails, which can be quite large, can reasonably be stored as flat files, as can music or pictures. The messaging folder hierarchy is persisted in a stream store.

The Database Management System (DBMS) is a cut-down version of a more general database management system called a relational database management system (RDBMS). SQL, the successor to DBMS, is an example of an RDBMS, and is the subject of this chapter.

To begin with, we look at what a relational database management system offers that the other storage alternatives do not. Then we move on to look at Symbian SQL in more specific detail.

22.1 Overview of RDBMS

An RDBMS allows the designer to easily abstract the problem domain and to establish semantic relationships between data elements. An example of

such a relationship might be 'has-a.' A car 'has-a' steering wheel. In fact, a car not only 'has-a' steering wheel, but indeed it must have a steering wheel (normally). That relationship, including the compulsory nature of the steering wheel, can be captured in an RDBMS 'schema'.¹ We can see, even with this simple example, the potential power of a relational database over using flat files.

Of course, such relationships and data rules can be established without a relational database. However, the links between data elements and the associated consistency checking then become the responsibility of the application. An RDBMS offers the advantage of providing these useful features 'for free'. That is to say, the code for establishing and enforcing the data relationships and allowing easy abstraction of the problem domain is in the RDBMS itself and does not have to be replicated in each client application.

Symbian DBMS has some limitations and therefore Symbian is replacing it. Originally, full RDBMS functionality was not required by the users of the database; more recently, there are use cases where a full RDBMS provides a more efficient and elegant solution. Exploiting native RDBMS functionality directly, rather than coding it into the client applications, improves performance and robustness.

Symbian therefore has taken the initiative to provide a component with the existing elements of DBMS, together with some additional functionality that would increase its usefulness and bring the Symbian database solution closer to the industry standard. That standard is Structured Query Language (SQL, pronounced either as 'S-Q-L' or as 'sequel'), and it is a syntax for creating, modifying and deleting databases and their data, as well as a formalized way to structure queries.

22.2 SQL Basics

SQL has been around for a very long time, but it wasn't until the early 1980s that the necessary standardization processes started converging the plethora of SQL-like implementations into something approaching a common syntax. The first standard version of SQL was completed by ANSI in 1986 and since then it has gone through a great many revisions. Even today, work continues in this area to keep SQL up to date with newer technologies such as Java and XML.

Two popular open source implementations of SQL are MySQL (available free of charge at www.mysql.com) and SQLite (again free, and found at www.sqlite.org). SQLite is particularly significant in this discussion because it is the underlying SQL database engine that Symbian selected for its relational database.

¹ A schema is a description of the organization of the data within a database.

Although SQL is described as a language, it is much more than that. In addition to supporting the standard syntax, MySQL and SQLite both provide a command-line interface allowing the entire lifecycle of a database to be exercised.² Some of these involve use of the standard SQL syntax, while other operations are proprietary.

Basic SQL Terminology

The *database engine* is the module that allows one or more databases to be manipulated. It may take the form of a command-line interface, or it might be a GUI, or even a set of APIs in other programming languages that allow developers to write code against their databases. The main functions of the engine are to parse the SQL syntax and to enforce standard behavior, although it normally also includes some non-standard services that are specific to the implementation of the RDBMS.

A *database* is made up of one or more files containing data that is encoded in a particular way. The contents of these files are not intended to be human-readable (in most cases, they are in a proprietary binary format and so can only be manipulated using the tools provided by the database vendor). This is not to say that the proprietary format compromises the SQL standard; it simply means that the vendors are free to choose how to implement and encode the data, as long as the syntax and semantics of the SQL standard are honored. The consequence of this is that it is unlikely that a database created by one database engine will be directly usable by a database engine provided by another vendor.

File-storage strategies also vary between implementations. For some SQL implementations there is a one-to-one relationship between the database and the file in which it resides. SQLite strictly obeys such a one-to-one relationship, so it is therefore not possible to have more than one database in a file, nor is it possible to spread a database over several files.

Inside a database are a number of objects that are meaningful to the database engine. Some of these are used by the engine exclusively for its own management activities; others are user data objects.

All databases need to maintain certain system data (often called *meta-data* because it is data that describes data). This metadata contains information about the state of the database and detailed structural information about it. Very often it is stored in a form that makes the metadata look similar to user data (and so can be manipulated via the SQL syntax). Vendors normally choose to keep this metadata in the database file itself, although other vendors might store the data elsewhere. In SQLite, the metadata is stored in the database.

The user objects include tables, which can be thought of in much the same way as flat files in the file system. A *table* has a name and the data

² The lifecycle includes creation and configuration of a database; management of user data, reading, writing and deleting data, copying and deleting a database.

it contains are related, so should be kept together. An example might be a PERSONNEL table containing information about employees.

Tables are arranged into rows and columns which are directly equivalent to records and fields respectively. A *row* is one entry in a table and a *column* is a particular value in that entry.

Rowid	Name	Address	Phone	Fax
1	Rick	123 High Street	1234567	7654321
2	Sally	234 Low Street	23456543	9876868

Figure 22.1 The PERSONNEL table

Figure 22.1 shows a table with two rows and four user-defined columns with the following names: NAME, ADDRESS, PHONE, FAX. The ROWID column is a ‘phantom’ column, which means that it is used by the SQL engine, but is not a specifically user-created column. The word column can have two meanings:

- it can refer to a particular value in a particular row (for example, the value of the PHONE column for Rick is ‘1234567’)
- it can refer to the collection of all column entries (for example, the values of the NAME column are ‘Rick’ and ‘Sally’).

We say that rows are *inserted* into a table. They can also be *updated* or *deleted*, and they can be read by *selecting* them.

One of the useful things about SQL databases is that tables can be *indexed*. This means that when we are looking for a particular piece of data in the table, we can go straight to it rather than having to scan the whole table.³ We can even set up multiple indices on a table so that we can search on different *keys*.⁴ Furthermore, we can set up indices that take parts of different columns as the key. This sounds like a hugely complex exercise, but in fact most of the work is done by the SQL engine. SQLite does not implement all of the myriad indexing possibilities, but most of those that matter are available.

There are some other entities that can be present in a database: stored procedures, events and triggers. However, the basic building blocks that matter most to developers of database solutions are tables and indices. A database with no user tables can only do trivial operations, such as returning the system date or advising the fact that there are no user tables!

³ This is not exactly true. In practice, an index file is walked and then the location of the actual data is retrieved from the index, which is then used to fetch the data. The index is normally in some highly performant format, such as a Btree.

⁴ ‘Keys’ and ‘indices’ are often used interchangeably.

More SQL Features

You are probably thinking that all the services provided by SQL described in the previous section would be reason enough for using it. However, there is even more on offer than that.

SQL has enjoyed a long history and, over the years, database developers have discovered that certain design problems and challenges have come up again and again. SQL has grown and expanded to address many of these issues automatically, rather than having them implemented time and again in user application code. This section looks at some of these innovations.

One of the key requirements in any data storage system is that the data that is stored should be good data. Not in the sense of the data being accurate – that is up to the client application – but in the sense that the data needs to be internally consistent, that is, it needs to make sense in the problem domain.

Going back to the car ‘has-a’ steering wheel scenario and the fact that it must have one, any car that did not have a steering wheel should never find its way into the database. That would be an error. In SQL terms this might be expressed in pseudo-code something like: ‘table CAR has a column called STEERINGWHEEL and this column must not be empty’. So, if that column were to be empty, then our database would not be complying with the domain rules for cars (that they all have steering wheels). This might be down to a programming error somewhere or to someone manually entering a row without a steering wheel.

To avoid situations like this, SQL has a concept of *constraints*. This means that the designer of the database can enforce the steering-wheel requirement, as opposed to it being the responsibility of the designer of the application that uses the database. With this constraint in place, any row that is inserted without a value being present in the steering-wheel column is rejected. The row simply cannot be created without meeting this constraint.

But what if someone inserts a valid row that contains a steering wheel, then goes in afterwards and makes that column empty? Again, SQL catches it and does not allow the row to be changed.

Let us suppose further that I have a table called STEERINGWHEEL where all my known steering wheels are kept. Then I try to create a row in the CAR table with a value specified in the steering-wheel column. Wouldn’t it be ideal if SQL would automatically check in the STEERINGWHEEL table to make sure that I am inserting a valid steering wheel? The good news is that this is exactly what SQL does. This is a concept called *referential integrity*.

Let me try to cheat SQL by creating an entry in CAR that satisfies the referential constraint (i.e., that the steering wheel is in the STEERINGWHEEL table), but then delete that entry from the STEERINGWHEEL table. SQL also detects and prohibits this operation.

Now let's apply another constraint – we insist that every steering wheel must be part of a car, that there cannot be a steering wheel instance existing in isolation. What would happen if we delete the only car that refers to a given steering wheel? The steering wheel would then be *orphaned*.

We can prevent this as well by constraining SQL to only allow the deletion of a CAR entry if all its dependents have been deleted. Or tell SQL to delete automatically all its dependents when the CAR entry is deleted. This latter operation is called *cascading delete*.

This raises another issue. What happens if the car entry is deleted and the system crashes before the steering wheel is deleted? We would again end up with an orphaned steering wheel and therefore an inconsistent database.

This is where an important database concept comes in: an *ACID*-compliant RDBMS ensures that the database operations are *atomic*, *consistent*, *isolated* and *durable*.

- Atomic means that, when a set of operations have to be successfully completed together, either none of the operations are applied or they are all applied. In the example above of the database crash, when the database resumes, it detects the inconsistency and automatically undoes the first part of the operation (the deletion of the car) to make the database consistent again. Atomicity is guaranteed by transactions. A *transaction* is a set of operations that can be undone in totality if any one of the operations comprising that transaction fails.
- Consistent means that the database's constraints have not been violated.
- Isolated means that, in a multi-user situation, the actions of one user cannot result in an inconsistent view of the database from the point of view of another user. It guarantees that every concurrent user sees the database at any given moment as though they were the only one using the database, even if there are other users operating on the same data. For example, user A is updating data in two tables in a transaction. Because it is a transaction, these operations must happen in totality or not at all. Now suppose that user B is reading the data affected. If user B reads the data after part of user A's transaction is complete, but before all of it is complete, then there is the risk that user B reads inconsistent data. Worse than that, if there is a transaction failure that results in user A's change being undone, then the data read by user B would no longer be valid and user B would have no way of knowing. Isolation guarantees that these circumstances cannot arise.
- Durability means that once a transaction fully completes all its operations, the data is *committed* and cannot subsequently revert (without an explicit operation causing it to change). It also means that data not

yet committed is guaranteed not to be durable (until it is committed or reverted).

So we can see that ACID provides us with a good level of confidence that databases are consistent and that the data in them makes sense in the problem domain.

Using Multiple Databases

In SQLite, it is only possible to have one database⁵ per file and one database cannot straddle multiple files. It is, however, possible to create a number of distinct databases and bring them together so that they appear to be one large database. This process is called *attaching* databases and it works by opening a primary database, to which one or more secondary databases can then be attached. Once attached, operations that can normally be done against a standalone primary database can also be done across a set of attached databases.

Databases can be attached and detached at runtime. This gives a powerful and flexible way to create complex relationships between data elements that might have a temporary relationship but do not warrant cohabiting in a single database.

Another important database concept, and one that makes exploiting the ‘relationships’ between data easy, is that of *joining*⁶ data. Joining data can be simple or extremely complex depending on the desired effect. We will describe only a simple joining situation here in order to illustrate the concept.

Suppose we have a PERSONNEL table containing data about people. People normally have contact information associated with them and it is possible to put the contact details for any given person in the PERSONNEL table together with their other data. However, people tend to have a number of contact possibilities: their home phone, mobile phone, fax, email address, Skype address, MSN messenger account, MySpace page, etc. The way to approach this might be to have a separate CONTACT table that holds zero, one or more entries for each person. It is a valid approach, but how do we associate those entries with the person to whom they relate? The answer is to join the data.

The PERSONNEL table would probably include a unique identifier for each person. A personnel number or identifier is the usual approach. If the CONTACT table also has a column carrying that same personnel number, then the obvious way to get at all the contact details for a given person is to look for all the rows that contain the personnel number.

⁵ A ‘database’ in this context is one collection of user and system data that is kept logically together.

⁶ ‘Joins’ are also known as ‘translations’ although ‘joins’ is probably the term of choice these days.

If we wanted the name of a person and all their contact data, the traditional way would be to look up the surname of the person in the PERSONNEL table, pick the correct person with that surname, extract the personnel number for that person and then use that number to look for all the contact entries in the CONTACT table. This all seems to be a bit long-winded and is how it might have to be done in DBMS.

Happily, SQL provides us with some clever syntax that allows joins across two tables to be done without writing complex, and possibly defective, code. In fact, SQL can join across more than two tables and even join a table to itself. This means that even highly complex data relationships can be easily and robustly modeled.

Stored Procedures, Events and Triggers

A *stored procedure* is a means by which a series of complex operations can be written, stored and carried out inside the database. As the name suggests, this is a procedural language, like C or BASIC, etc., but SQL syntax will probably form a substantial part of the procedure.

Stored procedures are stored in the database itself. If there are common complex operations that are done against data, then a stored procedure can be written, stored in the database and invoked by a client. This simplifies the client code and reduces the possibility of clients introducing defects or making errors in their manipulation of the database.

Stored procedures do not form part of the first release of Symbian SQL.

An *event* is used when something outside the database needs to detect that some data has changed in the database. This ‘something’ is normally an application that has an interest in some of the data in the database.

Suppose that there is an application whose job it is to notify the user when a new email comes in. The email reception server (IMAP, for example) receives the email and inserts it into the database. At this point the other application needs to know about this event and notify the user. But how will it know? Well, the application could poll the database periodically looking for new messages, but polling is an expensive and wasteful solution. Another alternative is that the email application could notify the interested application. However, this means that the email program needs to know about all interested applications and how to contact them.

Using the event model, an interested application can simply ask to be contacted by the database when something interesting happens. A *listener* to an event registers an interest in some part of the database. The database doesn’t need to know in advance who wants to be notified, it just accepts listen requests from applications as and when they arrive. When something occurs that matches the interests registered by listeners,

then all listeners to that particular event are informed. They can then take whatever action is required.

The event model is not provided in the first release of Symbian SQL.

Triggers are similar to events, but they occur within the database, so that, if a particular operation occurs inside the database, another database operation can be initiated automatically. An example here might be writing to a log. If a particularly sensitive table is updated, you might want to set up a trigger that records the fact that the table has been altered, and when.

SQL Syntax

You need at least a basic understanding of SQL syntax to follow the discussions about Symbian OS in Section 12.3. If you are already familiar with SQL syntax, then you can skip over this section.

The best way to gain experience with SQL syntax is to experiment with database operations on the command line. In this way, feedback is instantaneous and lessons can be learned rapidly about what works and what does not. In this section, we talk only about how SQLite behaves, although the SQL syntax is more or less the same in MySQL, etc.⁷

The first thing to do is to acquire an SQL implementation. For SQLite, you need to download a distribution from www.sqlite.org. It is free and comes with the complete source, so it is possible to find out how it does things internally (although this level of expert understanding is by no means required in order to be able to work with SQL).

This section covers basic SQL operations, but is not intended to replace other dedicated tutorials nor is it intended to be a reference document on SQLite. For more detailed explanations, visit the documentation page of the SQLite site.

The first operation is to create a database. A database is created or opened automatically when you specify its name when invoking the SQLite shell program (in this case, `sqlite3.exe`):

```
C:>sqlite3 myDatabase
SQLite version 3.2.7
Enter ".help" for instructions
sqlite>
```

⁷ The parts that differ are not included in the SQL standard (for example, schema commands).

This opens the `myDatabase` file if it exists and creates it if it does not exist.⁸

On SQLite start-up, the version number is displayed. Some (limited) help is also available – here is what you might see, although this may change with future releases of SQLite:

```
sqlite> .help

.databases          List names and files of attached databases
.dump ?TABLE? ...   Dump the database in an SQL text format
.echo ON|OFF        Turn command echo on or off
.exit              Exit this program
.explain ON|OFF      Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF   Turn display of headers on or off
.help              Show this message
.import FILE TABLE Import data from FILE into TABLE
.indices TABLE      Show names of all indices on TABLE
.mode MODE ?TABLE?  Set output mode where MODE is one of:
                    csv      Comma-separated values
                    column    Left-aligned columns.  (See .width)
                    html      HTML <table> code
                    insert     SQL insert statements for TABLE
                    line       One value per line
                    list        Values delimited by .separator string
                    tabs       Tab-separated values
                    tcl        TCL list elements

.nullvalue STRING    Print STRING in place of NULL values
.output FILENAME      Send output to FILENAME
.output stdout        Send output to the screen
.prompt MAIN CONTINUE Replace the standard prompts
.quit                Exit this program
.read FILENAME        Execute SQL in FILENAME
.schema ?TABLE?       Show the CREATE statements
.separator STRING     Change separator used by output mode and .import
.show                Show the current values for various settings
.tables ?PATTERN?     List names of tables matching a LIKE pattern
.timeout MS           Try opening locked tables for MS milliseconds
.width NUM NUM ...    Set column widths for "column" mode
```

Note that all these commands start with a dot. If you forget the dot, you see this:

```
sqlite> help
...>
```

The second prompt tells you that SQLite is waiting for more input. This is because, without the leading dot, SQLite thinks that you are entering an SQL statement. SQL statements must terminate with a semi-colon. Since one is not provided in the example above, SQLite is expecting more input

⁸ Note that the database is not created if you do nothing with it before you quit the shell. SQLite does not create empty databases.

followed by the terminating semi-colon. As you will not be able to carry out any other operations until this prompt has been dealt with, get rid of the secondary prompt by simply typing in the expected semi-colon.

```
sqlite> help
...> ;

SQL error: near "help": syntax error

sqlite>
```

The error is produced because 'help' is not a valid SQL statement.

The first thing we do is to create a table, using the following command. Remember to include the terminating semi-colon or nothing will happen.

```
sqlite> create table MyTable (myint INT);
```

This creates an empty table with a single column called `myint`.

The parentheses encapsulate the column list. The column list is the description of the kind of data that will reside in the table. Multiple columns can be specified separated by commas.

We do this again with the entire command in upper case.

```
sqlite> CREATE TABLE MYTABLE (MYINT INT);

SQL error: table MYTABLE already exists
```

Notice that SQLite is not paying any attention to case here.⁹

To see the tables you have in your database, use the following command:

```
sqlite> .tables

MyTable
```

Just to be a bit more general, we create a table with three columns this time.

```
sqlite> create table mytab2 (mytext TEXT, myreal REAL, myint INT);
```

It can be useful, if you are working with tables that have been created by someone else, to be able to find out what kind of data is stored in a table. Schema is the SQL word for the organizational description of the database: tables, columns, indices, etc. In this case we want the table schema for `mytab2`.

⁹ This series of commands was run on a Windows platform

```
sqlite> .schema mytab2  
  
CREATE TABLE mytab2 (mytext TEXT, myreal REAL, myint INT);
```

The `schema` command, above, returns the SQL statement that was originally used to create the table.

The following data types are recognized by SQLite:

- NULL
- INTEGER – the value is a signed integer, stored in 1, 2, 3, 4, 6 or 8 bytes depending on the magnitude of the value
- REAL – the value is a floating-point value, stored as an 8-byte IEEE floating-point number
- TEXT – the value is a text string, stored using the database encoding (UTF-8, UTF-16)
- BLOB – the value is a Binary Large Object, stored exactly as it was input. The data is raw; perhaps a picture or a block of encrypted data.

An important point to make here relates to data-type affinity. This means that SQLite makes some clever decisions about unknown data types. The type affinity of a column is determined by the declared type of the column, according to the following rules:

1. If the data type contains the string 'INT' then it is assigned INTEGER affinity.
2. If the data type of the column contains any of the strings 'CHAR', 'CLOB'¹⁰ or 'TEXT' then that column has TEXT affinity. Notice that the type VARCHAR contains the string 'CHAR' and is thus assigned TEXT affinity.
3. If the data type for a column contains the string 'BLOB', or if no data type is specified, then the column has affinity NONE. This means that SQLite makes no attempt to coerce data before it is inserted.
4. If the data type for a column contains any of the strings 'REAL', 'FLOA', or 'DOUB' then the column has REAL affinity.
5. Otherwise, the affinity is NUMERIC.¹¹

Having created a database, we can add some data to it:

```
sqlite> INSERT INTO myTab2 VALUES("this is some text", 1.23, 123);
```

¹⁰ CLOB is Character Large Object.

¹¹ Warning! a typo in specifying the data type is NOT caught by SQLite. The author made an error and put 'INT myint' instead of 'myint INT'. There was no error reported, but the table ended up with a column called 'INT' with a data type of 'myint' and an affinity of 'INT'!

This simple data insertion is quite self-explanatory. We are telling SQLite to insert values into the table `myTab2` and then providing the values to be inserted. Note that the values are provided in the same order as the definition in the table creation statement.

Now we see what happens if we make a mistake.

```
sqlite> INSERT INTO myTab2 VALUES(123, "this is some text", 1.23, 123);  
SQL error: table myTab2 has 3 columns but 4 values were supplied
```

SQLite reports errors as it finds them. Here it has detected the mismatch in the number of values provided.

The following statement does not produce an error, even though the data types don't match the original table definition.

```
sqlite> INSERT INTO myTab2 VALUES(123, "this is some text", 1.23);
```

This is because SQLite is loosely typed. It does not perform type checking. Obviously this has the potential to allow defects to find their way into your SQL code. We see later that the Symbian OS implementation of SQL manages data typing in a much more robust manner.

To see what is in a table, we use the `select` statement:

```
sqlite> select * from mytab2  
...> ;  
  
this is some text|1.23|123  
123|this is some text|1.23
```

Firstly, notice the effect of putting the semi-colon on a separate prompt line. SQLite processes it exactly as though it had been on the first line. In fact, SQLite does not regard new lines as special; its delimiter between statements is the semi-colon.¹² The only rule about multi-line commands is that the break cannot occur mid-word.

The asterisk in the `select` statement above means 'every column.' So in this case, all the data in the table is returned; all the rows and all the columns.

By default, the columns are delimited by the pipe symbol ('|'); this is configurable (see the appropriate SQLite documentation). The important thing to notice here is that the second insertion, with the mismatched data types, has succeeded! This is proof that SQLite is data-type agnostic.

¹² There are exceptions to this – see the SQLite documentation for more information. Also it is quite possible to include multiple statements on a single line separated by semi-colons, or to put half a statement on one line and the rest on subsequent lines.

Sometimes it is useful see the results in a different format. Output formats are called modes in SQLite. The line mode provides each column returned on a line by line basis, with rows separated from each other by a blank line:

```
sqlite> .mode line

sqlite> select * from mytab2
...> ;

mytext = this is some text
myreal = 1.23
myint = 123

mytext = 123
myreal = this is some text
myint = 1.23
```

To be more specific in interrogating the database, you can specify particular columns that you want retrieved. The following statement returns only the column `myint`:

```
sqlite> select myint from mytab2;

myint = 123

myint = 1.23
```

To look for rows where a particular column matches a particular value, we use the `where` clause.

```
sqlite> select * from mytab2 where myint=123;

mytext = this is some text
myreal = 1.23
myint = 123
```

The `where` clause can be used when selecting specific columns. Note that the column retrieved does not have to be the same as the column used to select it!

```
sqlite> select mytext from mytab2 where myint=123;

mytext = this is some text
```

The important thing to remember with `where` clauses is that they return all rows that satisfy the condition. All of the usual Boolean operators can be used.¹³

¹³ C++ programmers should be careful: the operators used in comparisons are not as you would expect! Check the SQL documentation for a complete list of operators.

```
sqlite> select mytext from mytab2 where myint<124;
mytext = this is some text
```

Clearly, this has not been an exhaustive tutorial in SQL. It was intended just to be enough to allow you to be able to follow the discussion of the Symbian SQL Server component. You need a much better understanding of SQL syntax to be able to do any useful work with it. Since SQL is a standard syntax, there is a wealth of dedicated SQL learning material out there. Be careful about vendor-specific variations though!

22.3 Symbian SQL Server Component Architecture

For brevity, we refer to the Symbian SQL Server Component as Symbian SQL. Where it is important to be clear about a subcomponent, we refer to it more explicitly. This section is concerned with the high-level architecture of Symbian SQL (see Figure 22.2). It describes the sub-components and the high-level interactions that take place.

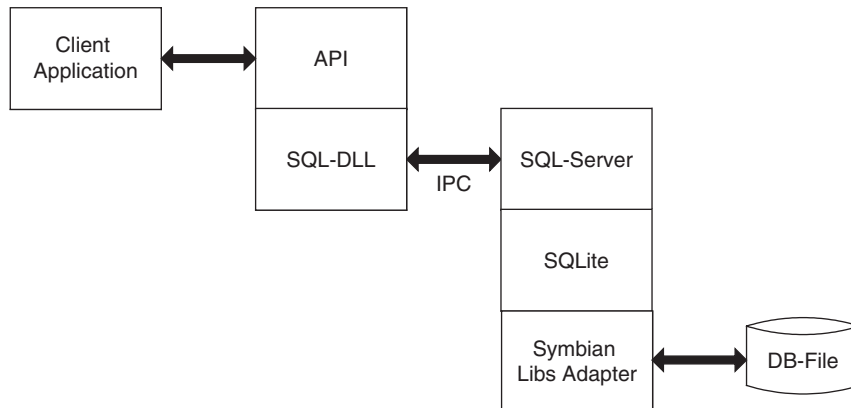


Figure 22.2 The layout of the Symbian SQL component

The component uses the standard Symbian client–server architecture. The two Symbian-delivered parts are the client-side API DLL which we refer to as the API in this chapter, and the server-side EXE, which we refer to as the server.

To make use of Symbian SQL, the client application must link to the API DLL. It is not possible to access the SQL server directly; this is to enforce security and concurrency constraints.

Note that the application linked to the SQL API has the secure UID of the client application not of the SQL server. So, whatever rights and privileges the client application has in accordance with the Platform Security Model are also true of the API DLL. No additional rights are conferred to the client by linking to the DLL.

Once linked, the client can then create a database resource that can be used to talk to a particular database via the server. This communication is via a published C++ API that is described later in the chapter.

All requests for operations against the database are subject to preliminary validation by the API code and, if acceptable, the request is marshaled and sent to the server via IPC. The server then validates the request, asserts any required security policy and packages the request for submission to the underlying SQLite C API.¹⁴

The Symbian Libs Adapter in Figure 22.2 allows the SQLite library to access the Symbian operating system services. As the SQLite engine has been designed to work on multiple platforms, the operating system interface layer has to be re-engineered for each new port. Symbian has developed the appropriate interface layer to allow the SQLite engine to interact with Symbian OS. Using this layer, the SQLite database engine then attempts to execute the client operation on the appropriate database and reports results and return codes back to the server. The server packages up the results and passes them back to the client via IPC and the client-side DLL.

A question that arises at this point is: where do the database files live in the file system? This is an important question because the location in the file system has an impact on behavior and the range of operations that are possible. The short answer is that the database files can exist anywhere in the file system, but are usually in the SQL server's private data cage¹⁵ on the device. The data in this data cage cannot be accessed by any process apart from the one that owns the data cage, in this case the SQL server. However, it is possible that the database may reside in ROM, on removable media, in the client's private data cage, or in any of the public folders. Access rights to the database vary substantially amongst these options.

Client-Side API

In this section, we look at how a client would create and work with a database. We start with simple scenarios and make them more complex as we gain confidence in our understanding of how things work.

The first thing is to ensure that your application links to the `sqldb.lib` library, which is specified in the MMP file. You also need to include the `sqldb.h` header file, which resides in the standard include directory.

In `sqldb.h`, you'll see a class called `RSqlDatabase`. This is the most important class from the point of view of a client-side developer. Lots of things can be done using just this class alone – databases can be created, deleted, copied and configured and data can be inserted.

¹⁴ Although Symbian SQL is implemented in C++, SQLite is delivered as a C library.

¹⁵ See [Heath 2006, Chapter 2] for more information on data cages, etc.

You'll probably have noticed that one important operation is not in the list above: the ability to read data from a database. This is not an oversight!

The reason is that all of the operations described above return only a success or fail indication. If we are returning data then things get more interesting, because we don't know how much data, if any, we might get back. Also, the data is two-dimensional: a certain number of rows by a certain number of columns. We need to be able to manage that complexity. So we have helper classes that simplify what would otherwise be a horrible job on the client side.

RSqlDatabase is the resource class that allows client-side programmers to interact with the SQL Server. This section looks at this class in some detail.

Creating a database

In the simplest case, the only piece of information we need to create a database is its name. In selecting a name for a database, there are a few rules on database-naming conventions that tie in with things such as platform security and the backup and restore engine. For now, we assume we are creating the simplest sort of database: one that is unprotected by platform security and does not need to be automatically backed up. We call it `mydatabase.db`. We also need to decide where to put it. In this simplest case, we can choose a drive and a path. So our full database name is `C:\test\mydatabase.db`. You'll notice that database names conventionally end in `.db`. This is not enforced, but is recommended.

At the appropriate point in our code, we need to create our database resource object: `RSqlDatabase`. This is trivial:

```
RSqlDatabase myDatabase;
```

We can now do simple operations against databases – such as creating one:

```
TInt Create(const TDesC& aDbFileName);  
TInt returnCode = myDataBase.Create(_L("C:\\test\\mydatabase.db"));
```

This call should return `KErrNone`. Other return codes are possible, including Symbian OS system-wide error codes as well as SQL-specific error codes. The `Create()` call could fail because of lack of permission to create the database, a bad database name, the disk being full, trying to create a database in a read-only location, being unable to open the database, etc. See the SQL-error code list in Section 22.4 for a complete list of the SQL-specific error codes.

You may wonder why a 'can't open' error might come back from the create call. This is because, as a convenience, SQL server assumes that

when you create a database, you also have the intention of using it. For that it needs to be open, so it attempts to open the database after it is created.

If the `Create()` call returns `KErrNone`, you have a database created, opened and ready for use. There are a few questions that flow from the above code: Where do we specify a security policy for the database? What about configuring the database? Is there a default configuration and, if so, what is it?

To the first question, we decided that this was going to be a simple database creation operation without applying any protection to the database, so no security policy was needed. The second question is similar. This simple database does not have a special configuration, but instead inherits a default configuration. The default configuration is that our new database is encoded in UTF-16, the page size is 1024 bytes, and the cache size is 64 pages.

Creating a table

The data repositories in an SQL database are tables. Before we can store any data, we must create one or more tables. This follows exactly the same pattern as we saw earlier when creating tables from the command line:

```
TInt rc = myDatabase.Exec(_L("CREATE TABLE mytab2 (mytext TEXT)");
```

The return code we want, of course, is `KErrNone`. Other codes might be returned if, for example, the table already exists, you have no permission to create tables, or there is a syntax error, etc. Note that the descriptor passed into `Exec()` can be either UTF-8 or UTF-16. It is entirely up to you which you use, but be aware that 16-bit descriptors are less performant than 8-bit descriptors.

You may be tempted at this point to check that the table is there, using something like `Exec(_L(".tables"))`. This does not work. The dot commands are not SQL statements¹⁶ and you generate an SQL syntax error if you attempt to pass such commands into the SQL engine.

If the creation operation returns `KErrNone` then the table has been created successfully.

Inserting data into a table

Let's insert something into the table now. We follow the same pattern but apply the alternative programming practice of using `_LIT`:

```
_LIT(KMyStatement, "INSERT INTO mytab2(mytext)
VALUES('This is a text value')");
TInt rc = myDatabase.Exec(KMyStatement);
```

¹⁶ They are commands issued to the SQLite shell interface.

Note the slight syntactic change in the INSERT statement. Here we identify explicitly the name of the field to insert the data into (`mytext`), allowing us to insert data selectively. In the absence of this qualifier, the columns would be filled left to right until the `VALUES` were exhausted. Of course, in this example, because there is only one column in the `mytab2` table and only one value provided, the result is identical whichever way we do it.

An obvious point: the text value to be inserted needs to be quoted. In this case we are using the single quote to avoid having to escape a double quote (which would be needed to disambiguate the end of the `_LIT` string).

Now that we have the basics down, let's make things more interesting. Here's what we're going to do:

- delete the `mytab2` table
- recreate it with more than one column
- create an index on a column in the table
- insert some data
- delete some data
- change some data.

This demonstrates most of the simple operations that can be done on a table.

We include all the code as a complete block rather than piecemeal as before to give a better feeling for how the steps relate to each other.

```
// convenience macros to avoid repetition
#define EXEC0(statement) if((KErrNone!=(rc=db.Exec(statement))))\
{\
    return rc;\
}

#define EXEC(statement) if((rc=db.Exec(statement))<1)\
{\
    return rc;\
}

_LIT(KMyDatabase, "C:myDatabase.db"); // database name

// now for the statements to execute
_LIT(KDeleteTable, "DROP TABLE IF EXISTS mytab2");
_LIT(KCreateTable, "CREATE TABLE mytab2 (id INT, txt TEXT, num INT,\
                                         dbl REAL)");
_LIT(KCreateIndex, "CREATE UNIQUE INDEX myidx ON mytab2 (id)");
_LIT(KInsert0, "INSERT INTO mytab2 VALUES(0,'txt0',0,0.0)");
_LIT(KInsert1, "INSERT INTO mytab2 VALUES(1,'txt1',1,1.0)");
_LIT(KInsert2, "INSERT INTO mytab2 VALUES(2,'txt2',2,2.0)");
_LIT(KInsert3, "INSERT INTO mytab2 VALUES(3,'txt3',3,3.0)");
_LIT(KInsert4, "INSERT INTO mytab2 VALUES(4,'txt4',4,4.0)");
_LIT(KDeleteRow, "DELETE FROM mytab2 WHERE id=3");
```

```

_LIT(KUpdateRow, "UPDATE mytab2 SET dbl=123.456 WHERE id=0");

TInt rc = KErrNone; // our error return code
RSqlDatabase db; // the database resource handle

// the database already exists so we only open it, not create it
if (KErrNone != (rc = db.Open(KMyDatabase)))
{
    return rc;
}

EXEC0(KDeleteTable); // delete the table
EXEC0(KCreateTable); // create the table again
EXEC0(KCreateIndex); // create and index on the 'id' column
EXEC(KInsert0); // insert data

// EXEC(KInsert0); // this would fail if we uncommented it

EXEC(KInsert1); // insert data
EXEC(KInsert2); // insert data
EXEC(KInsert3); // insert data
EXEC(KInsert4); // insert data
EXEC(KDeleteRow); // delete a row
EXEC(KUpdateRow); // change a row

db.Close();

return rc;

```

Initially, we created two convenience wrappers for the `db.Exec()` operations. In `EXEC0()`, we are checking for a return value of `KErrNone`. In `EXEC()`, we are checking for zero or a negative value. The reason for this will become clear later.

We then set up all the descriptors for the SQL statements that we will be using. In real code, you would probably create ones with substitutable parameters to avoid duplication but, for the sake of simplicity, we are just literal.

The first SQL statement is the `DROP` statement, which deletes the table from the database. The `'IF EXISTS'` part is optional. This clause simply avoids returning an error in the case where the table is not present.

Then we create the table. We've seen this before. We have four columns in this new table: `id`, `txt`, `num` and `dbl`. The next step is to create an index, on the column called `id`. We could have created the index during the `CREATE` phase but we chose not to in order to separate the discussion of `CREATE` from the explanation of how `INDEX` works.

Creating an index

Before explaining the syntax, let's describe what this statement achieves. What we want is an efficient way of being able to find rows in the table. The way we are going to search for rows is by means of its `id` column. We

also want to ensure that we don't end up with multiple rows that share the same `id` value. This is why the second insert of `id=0` is commented out in the code!

Once an index is established against a column in a table, select statements such as the following are likely to be a lot faster:

```
SELECT * from mytab2 WHERE id=3;
```

The index is accessed first to find the exact location in the data of the row we want. This avoids traversing the table `mytab2` from the beginning until the desired value is found. The index is in a form that allows a particular value to be found quickly. In the case of SQLite, it is a B-Tree. Here is a non-technical description of how a B-Tree works.

Suppose I want to store the numbers 5, 11, 3, 25, 4 and 12, such that they can be found quickly. One approach would be to reorder the list every time I add a new number. On disk, this might be expensive in time. Another, better, approach is to create an index. Suppose that an index entry looks like this:

```
<POINTER A> <VALUE FOR THIS ENTRY> <POINTER B>
```

POINTER A points to a value that is less than the value for this entry and POINTER B points to an entry that has a bigger value. I start by adding the 5 to an empty index. The index now only contains entry 5:

```
<> <5> <>
```

Here we see an entry with a value, but no pointers to any other entries. I now add 11, which creates a pointer to another entry and put that pointer in the 'bigger' side of the current entry:

```
<> <5> ← <> <11> <>
```

We see now that there is still no lesser pointer, but the bigger pointer now points to an entry containing 11 (an index entry that itself contains no pointers). I now add the 3:

```
<> <3> <> ← <5> → <> <11> <>
```

We now want to add 25:

```
<> <3> <> ← <5> → <> <11> → <> <25> <>
```

The pattern is clearly seen so far. But now what happens when we want to add 4? The pointer to 'less than five' is already taken. It points to the 3 entry.


```
<> <3> <> ← <5>
```

4 is bigger than 3. And we have a spare pointer for 'bigger than three.' So we end up with:

```

              <5> → <> <11> → <> <25> <>
              /
<> <3> → <> <4> <>

```

And it is the same when we add the 12. The 'bigger than 11' pointer is already taken. So we do the same trick:

```

              <5> → <> <11> → <25> <>
              /           \
<> <3> → <> <4> <>         <> <12> <>

```

This is how the index is built up.

Now, to find a value, instead of walking the whole index, all we need to do is start at the root of the index file and ask three questions:

- is this the value? if so, we're done
- is the value we want smaller? if so, follow the left pointer
- is the value we want bigger? if so, follow the right pointer

Of course, associated with each value would be the offset of the data that the index refers to:

```
<left pointer> <value + data offset> <right pointer>
```

So, it can be seen that for columns that might be searched upon often, an index can speed things up quite a lot.

The syntax for creating an index is as follows:

```
CREATE UNIQUE INDEX myidx ON mytab2 (id)
```

The word **UNIQUE** enforces the constraint that there can only ever be one row in this table with a given `id` value. It is possible to leave this keyword out, which would allow multiple entries with the same key. In such a case, searching on the column would return all rows with that key.

Secondly, the index is given a name. There may be multiple indices on the same table and in order to manipulate them it is necessary to have a means of referring to them. The index name serves this purpose.

An index is created on a particular table and a particular column.

It is possible to remove an index from a table:

```
DROP INDEX myidx;
```

Inserting data

We then insert some data, as previously discussed. The reason for using `EXEC ()` and not `EXEC0 ()` here is because `INSERT` operations can return a negative value, zero or a positive value.

A return value of less than zero indicates that an error has occurred and so the return code is the error code; 0 means that no rows were affected by the operation (in the case of `INSERT` it means that no rows were inserted); and greater than zero indicates the number of rows affected. Each of the `INSERT` statements in the example should return 1 (since one row was inserted in each case).

Deleting a row

The following statement deletes only the row that contains 4 in the `id` column.

```
DELETE FROM mytab2 WHERE id=4;
```

The `WHERE` clause is optional, but be careful! If you leave it off, all rows in the table are deleted! This is because you are not limiting the selection: all rows satisfy the implied condition (i.e., no condition).

Updating data

The final statement that is executed is to alter the contents of a row.

```
UPDATE mytab2 SET dbl=123.456 WHERE id=0
```

Here we are saying to update the specified table (`mytab2`) setting the column that is called `dbl` to the value 123.456 but only if the `id` column of the row is 0. Again, if the `WHERE` clause is omitted, the change is applied to the whole table.

Further, note that the `EXEC ()` macros are checking for return codes with values that are less than one (either no rows affected or an error code). Be careful as this is a potential source of defects. If you test only for `KErrNone`, then successes are reported as failures.

Closing the database resource

The following statement closes the database resource:

```
db.Close();
```

This cannot return an error, even if the database is already closed.

Deleting and copying a database

You cannot copy or delete a database using a handle object for that database because the calls to perform these operations are static methods of `RSqlDatabase`.

```
static TInt Copy(const TDesC& aSrcDb, const TDesC& aDestDb);
static TInt Delete(const TDesC& aDbFileName);
```

These are self explanatory. The expected return code on success for both of these is `KErrNone`. For example:

```
TInt rc = RSqlDatabase::Copy(_L("SrcDb"), _L("DstDb"));
TInt rc = RSqlDatabase::Delete(_L("Database"));
```

Obviously there is an assumption with the copy operation that the SQL server has the right to create a file in the destination directory.

Attaching a database

Suppose that an application wants to be able to present SMS messages to the user. In addition to the message itself, this application wants to be able to display information about the sender of the message (assuming this information is available). This might include other contact information, a picture, a virtual business card, etc.

The SMS storage database is unlikely to contain all the information that this application might need. Some information is in the contacts database, for example. Although the application might want to delete or move the message after reading it, it would probably want to prohibit the user from changing the details of the contact from inside that application; perhaps expecting them to use a dedicated contacts management interface (which may or may not be automatically invoked – the details of the user interface are not relevant to this discussion). It would be interesting to have simultaneous access to a read–write message database and a read-only contacts database.

The partitioning of the two databases is useful for two reasons:

- the read–write permissions on each database can be different; such permissions are database wide, so read–write access cannot change across a single database; a database that is read–write for a given user is entirely read–write
- other applications may want access to the contacts information but have no need for the messaging data, so it makes sense to keep them separate.

Database attachment therefore affords us the opportunity to temporarily view these databases as if they were one database with some read-only elements and some read–write elements.

First, let's examine the API that allows the attachment of databases:

```
TInt Attach(const TDesC& aDbFileName, const TDesC& aDbName);
```

Notice that it is an instance method, not a static method. This means that this operation is applied to a database – the ‘primary’ database. This is the database to which all other databases can be attached.

That said, it is now obvious that the first required operation is to open (after possibly creating) the primary database. This is exactly as we described earlier. Any database can be a primary database and it can be primary in one situation, while on another occasion it may be attached to a different primary database. The key point here is that the primary database is the one that features in the `Open()` command. Once opened, we can attach any other existing database to the primary database. It is not possible to attach a database and then subsequently create it.

Symbian SQL supports multiple simultaneous clients. So it is perfectly possible, and reasonable, for client A to `Open()` database X (i.e. consider it to be primary) while another client, B, has attached that same database X to its primary database, Y.

There is an SQL statement that appears to attach a database:

```
ATTACH DATABASE C:\aDatabase.db AS mydb;
```

So why not simply insert that syntax into an `EXEC()` call? Why use a dedicated API call? The reason is platform security. Even though we are working with non-secure databases at the moment, the requirements of platform security still have an impact. Symbian has had to disable the direct SQL syntax approach to attaching databases to ensure that the attachment of databases can be policed by the server when in secure mode. With the direct SQL syntax disabled, another means had to be provided to allow legitimate attachments. This is why the `Attach()` API was provided.

There are two parameters to the `Attach()` method. The first is the name of the database to be attached. This is the file system name as was the case when opening the primary database. The second is a logical name to be assigned to the database so that it can be referenced unambiguously. In effect, the logical database name allows for explicit scoping of the namespace.

Using attached databases

What does this mean in practical terms? Imagine that a primary database has a table called `Table1`. Suppose that an attached database also has a table of the same name. If we perform an operation against `Table1`, which table is affected?

To resolve this ambiguity, the table is not referred to as `Table1`, but rather as `aDatabase.Table1`, where `aDatabase` is the name supplied in the `Attach()` call.

In this example, we perform a join using a single database, then attach a second database and perform another joining statement. We then want to find an integer value in `Table2` and copy it into `Table1`.

Assuming that the rule for finding the integer in Table2 is that it is found in the num column in the row of Table2 that has id equal to 123, the SELECT for this would be:

```
SELECT num FROM Table2 WHERE id=123;
```

This would retrieve the value, but this statement wouldn't insert the value retrieved into Table1. Let's assume the value returned by the SELECT statement was 456 and INSERT this value into the num column of Table1.

```
INSERT INTO Table1(num) VALUES(456);
```

This works nicely, but we had to do it manually in two steps. We also had to remember the number from the first step and plug it into the second step. We could combine these steps as follows:

```
INSERT INTO Table1(num) SELECT num FROM Table2 WHERE id=123;
```

This achieves both operations in one SELECT statement and is good for our one-database example above, but what happens if we have a number of attached databases?

Let us assume that Database1 and Database2 are the logical names of two databases attached to some primary database, and that both databases have a Table1 and a Table2. Now let's look back at the INSERT statement above. We have a problem.

Which Table1 is referred to in the statement? Which Table2 ? The short answer is that we don't know and neither does Sybman SQL. Here is the solution (assuming that the Table1 we want is in Database1 and the Table2 we want is in Database2):

```
INSERT INTO Database1.Table1(num) SELECT num FROM Database2.Table2  
WHERE id=123;
```

This demonstrates why a unique name has to be provided for all attached databases: so that it is possible to be unambiguous about which database is being referred to in SQL statements.

Detaching a database

Databases can be attached and detached as often as required during the run of an application. There is a performance cost in doing this, however, and so decisions should be made about whether lazy attachment or pre-emptive attachment is the best approach.

```
TInt Detach(const TDesC& aDbName);
```

Note that the logical name of the database (not the name of the database file!) is included in the `Detach()` call. The reason why a name is expected in the call is because multiple databases can be attached¹⁷, and the detach API needs to know which one is to be detached.

On the subject of the ordering of attachment and detachment of databases, there are very few rules. Generally, they can be attached and detached in any order. The only restrictions are that:

- the first database to be attached cannot be attached until a primary database exists
- the primary database cannot be detached (it must be closed)
- a database cannot be attached to an attached database.¹⁸

Returning Data from Statements

We have not yet seen how to return data using SQL statements. As mentioned, the complications are that we don't know whether any data will be returned, how many rows, how big the columns are, the column types, etc.

In order to make things simple for developers using the Symbian SQL API, there is a helper class called `RSqlStatement`. This encapsulates an SQL statement, and the data that it might return, if any. Before discussing how this is effected in code, we'll look at some of the concepts behind a 'statement' in the sense of an `RSqlStatement`.

We've seen examples of SQL statements:

```
INSERT INTO tab VALUES(123);
DELETE FROM tab;
UPDATE tab SET col=123 WHERE id=345;
```

An `RSqlStatement` is a representation of an SQL statement but is not itself an SQL statement.

An `RSqlStatement` is a container for an SQL statement. Once inside this container, the SQL statement can be manipulated in certain ways. It can be prepared for execution, executed and, possibly, reused. We work through a simple example to illustrate this and then advance to more complicated ideas.

Inserting dynamic values

Let's suppose that we want to insert some values into a table. So far, we have plugged the values directly into the statement descriptor:

```
_LIT(KStatement, "INSERT INTO tab VALUES(123)");
```

¹⁷ The underlying database engine imposes a limit of ten attached databases.

¹⁸ This only applies to a given client.

This has the value 123 hard coded into the statement, which is not very useful if we don't know the values in advance. We could build the statement programmatically by building up strings as we figure out what the statement is supposed to be, but then we'd have to redo that each time we want to execute a statement like this – obviously not a very efficient strategy!

Instead, we create an `RSqlStatement` object and let it do the work for us:

```
RSqlStatement myStatement;
```

To inform this statement of its duties, we have to prepare the statement. For this we need a database object. Statements have to be associated with a database to be able to do anything. We assume that a `_LIT()` has been used to make the `KStatement` in the following snippet.

```
RSqlDatabase myDatabase;  
myDatabase.Open(_L("aDatabase")); // ignoring errors for clarity  
TInt rc = myStatement.Prepare(myDatabase, KStatement);
```

This is ready to execute:

```
rc = myStatement.Exec();
```

We seem to be doing even more work and using more objects to achieve the same result as when we used:

```
myDatabase.Exec(KStatement);
```

This is because we are not exploiting the advantage of the `RSqlStatement` object. Let's go further and change the `KStatement` descriptor:

```
_LIT(KStatement, "INSERT INTO tab VALUES(:val)");
```

The new part of this statement is the data passed to the `VALUES` statement. This is called a *placeholder* and serves to mark the position where some data is inserted at run time. We've given this placeholder the name `val` so that we can refer to it later. The colon is a syntactic device to identify it as a placeholder.

We can prepare the statement as before:

```
TInt rc = myStatement.Prepare(myDatabase, KStatement);
```

We now need to 'plug' the value that we want into the `:val` placeholder position (say, 123). This plugging-in is called *binding*.

So what we do next is to bind the actual value into the placeholder's position. Before doing that, be aware that the SQL engine doesn't understand the placeholder name we assigned to the placeholder. The text is just a convenience for humans; SQL expects a number. That is to say, that we have to ask SQL to bind the placeholder to a value. This means we have to turn our convenient name `:val` into an index¹⁹ that SQL can understand.

In this case, we only have one placeholder. However, in a real situation, we might have several in the same prepared statement. Hard coding the index would be a nuisance. It is also a possible source of defects (for example, in the future we might add another placeholder earlier in the descriptor – this would have the effect of shunting all the existing placeholders along by one, rendering all our hard coded indices invalid).

So, in order to be robust, we resolve the placeholder.

```
TInt placeHolderIndex = myStatement.ParameterIndex(":val");
```

Now we have the placeholder index and even if we change the number of placeholders in the statement, we always have the correct one (assuming the name assigned to the placeholder is the same of course!²⁰).

So let's bind the value 123 into the statement at the index we have just retrieved:

```
TInt rc = myStatement.BindInt(placeHolderIndex, 123);
```

We can now execute the statement as before and 123 is inserted into the table as expected.²¹

```
rc = myStatement.Exec();
```

Note that in this `Exec()` call, there are no arguments. This is different to the syntax for calling the `Exec()` on the `RSqlDatabase` object. This is because `RSqlDatabase` needs to be told what to execute. But the `RSqlStatement` object has all the information it needs and so just needs to be told to go ahead and do the operation.

Usefully, we can even recycle the statement and bind it with another value (or values, if more than one placeholder is in the statement).

```
rc = myStatement.Reset();  
rc = myStatement.BindInt(placeHolderIndex, 456);
```

¹⁹ Not to be confused with a table index. The index in a binding context is simply an integer which refers to a specific placeholder in a prepared query.

²⁰ If not, SQL tells us with an error code when we try to resolve the unknown name using `ParameterIndex()`.

²¹ Of course '123' in the `BindInt()` call could be replaced by a variable of an appropriate type at left.

Note that the `Prepare()` call was not made between the `Reset()` and the `BindInt()`. This is because the statement has already been prepared, that is, it has previously been parsed and the placeholder positions have been identified. So all that needs to be done is to tell it what the new parameter values should be. All we need to do is to `Reset()` it ready for reuse.

You may have observed that the bind operation was a `BindInt()` call. There are analogous calls for binding other data types:

```
TInt BindNull(TInt aParameterIndex);
TInt BindInt(TInt aParameterIndex, TInt aParameterValue);
TInt BindInt64(TInt aParameterIndex, TInt64 aParameterValue);
TInt BindReal(TInt aParameterIndex, TReal aParameterValue);
Int BindText(TInt aParameterIndex, const TDesC& aParameterText);
TInt BindBinary(TInt aParameterIndex, const TDesC8& aParameterData);
```

The first of these is interesting. You might think that binding something to be empty is a pointless exercise. But in fact, a NULL value in SQL is a valid and meaningful possibility.

A column can be NULL for two reasons:

- it might never have been set; NULL is the default value for a non-initialized column (assuming that the database constraints allow that column to be NULL)
- it can be set to NULL by the `BindNull()` method; again, this only true if the NULL value is not forbidden by the schema.

The `Reset()` method is used to return the statement to a point where it can be reused. Of course, resetting a statement invalidates any data it may contain. The important thing is that the SQL statement itself remains intact (i.e., it does not need to be prepared again).

Retrieving data

Let's prepare a statement that selects some data:

```
LIT(KStatement, "SELECT num FROM tab");
```

This reads all the data in the `num` column in a table called `tab`, as we discussed near the beginning of this chapter.

We assume that there is a lot of data in this table that is returned by this query. How do we get it back? First, we have to ask for it by submitting the above SQL to the SQL engine. So, as before, we prepare a statement (we're ignoring errors on the previously presented methods for clarity):

```
RSqlStatement myStatement;
myStatement.Prepare(myDb, KStatement);
TInt rc = myStatement.Exec();
```

There are no placeholders to bind so we can execute it directly. Having executed it, where is the data and how can we access it?

Before answering this, we need to note the semantics of the return code from the `Exec()` call. In the case of an error, it is the Symbian OS or SQL-specific error code, which is less than zero. Zero means that no rows were affected by the statement. A number greater than zero indicates how many rows are available.

To get the data, we have to ask the statement object for it, row by row. This means we need to be in a loop:²²

```
while ((rc = myStatement.Next()) == KSqlAtRow)
{
    // we do something here which we'll explain later
}
```

As can be seen, we are looping until we get a return code which is not `KSqlAtRow`. This return value tells us that there is valid data at our current iteration position. If we are at a row, we can ask for the data from it. If not, we have run out of data (or an error has occurred) and we exit the loop.

To make life easier on ourselves, we've asked for integers. Remember that we wanted all the values for the `num` column from the table? So inside the loop we tell the statement object to give us an integer at the current iteration position. In this example, each iteration returns a single integer:

```
123
456
789
...
```

However, in real situations, there could be multiple columns returned:

```
123  sometext  1.234
456  moretext  5.678
...
```

We therefore need to tell SQL which column of the current row we want. We do not have to ask for all of them or, indeed, any of them if we do not want to. In the case, where only one column is returned, we could simply say:

```
TInt myInt = myStatement.ColumnInt(0);
```

²² You might wonder why we don't just use a simple `for` loop instead of the `while` loop? After all, we know how many rows are there. It is better to test for an expected result rather than just hoping that the iterations work without problem. Just imagine what would happen if some code inside the loop affected the statement in a way that changed the number of iterations...! Of course, if you check for errors inside the loop then there is no reason not to use a `for`-loop.

In a multi-column result set, this is not robust. Besides, we may know the name of the column but not its index. So the preferred approach is to ask SQL for the index rather than using a hard-coded number in the code.

```
TInt columnIndex = myStatement.ColumnIndex("num");
```

We asking for the index of the column called num. Notice that, although this is similar to the `ParameterIndex` that we saw earlier, the column name does not have a leading colon. This can be the source of hard-to-find bugs, so be careful!

With the index found, the code changes to:

```
TInt myInt = myStatement.ColumnInt(columnIndex);
```

Clearly this is much more robust.

As was the case with `BindInt()` we have here `ColumnInt()`. There are analogous methods for the other data types. But what if we don't know the data type for a given column (unlikely, but possible). How can we know what type of data to ask for?

As before, we simply ask SQL to tell us. This is the method:

```
TSqlColumnType ColumnType(TInt aColumnIndex) const;
```

The possible data types returned are:

- `ESqlNull`
- `ESqlInt`
- `ESqlInt64`
- `ESqlReal`
- `ESqlText`
- `ESqlBinary`

Here are the methods that allow retrieval of the various data types. Some of these need further explanation as they behave very differently from the simple integer example.

```
TBool IsNull(TInt aColumnIndex) const;
TInt ColumnInt(TInt aColumnIndex) const;
TInt64 ColumnInt64(TInt aColumnIndex) const;
TReal ColumnReal(TInt aColumnIndex) const;
TPtrC ColumnTextL(TInt aColumnIndex) const;
TInt ColumnText(TInt aColumnIndex, TPtrC& aPtr) const;
TInt ColumnText(TInt aColumnIndex, TDes& aDest) const;
TPtrC8 ColumnBinaryL(TInt aColumnIndex) const;
TInt ColumnBinary(TInt aColumnIndex, TPtrC8& aPtr) const;
TInt ColumnBinary(TInt aColumnIndex, TDes8& aDest) const;
```

The first one might seem a little strange, since it returns no data. It simply tells us whether the particular column is empty or not. Obviously, you cannot retrieve data from an empty column, so there is no `ColumnNull()` method. The scalar methods `ColumnInt()`, `ColumnInt64()` and `ColumnReal()` all behave similarly. The value is inserted into the return value.

The three text methods allow different approaches to retrieving the data, and it is up to the designer of the application to decide which best suits their needs in terms of ownership of the data buffer and whether resultant data is copied or just referred to *in situ*. The same approach is taken with the `ColumnBinary()` methods which also can return large data buffers.

Where methods simply return a pointer to data, that data becomes invalid once the buffer the pointer points to becomes invalid. This happens when calling `Next()` or `Reset()` or after closing the statement.

When dealing with variable-size data, and possibly having to allocate a buffer for it, it is important to be able to find out in advance of reading it how big it is. The method for this is `ColumnSize()` which returns an integer.

```
RSqlStatement myStatement;
TInt err = myStatement.Prepare(database, "SELECT BinaryField FROM Tbl1");
TInt columnIndex = myStatement.ColumnIndex("BinaryField");
while((err = myStatement.Next()) == KSqlAtRow)
{
    TInt size = myStatement.ColumnSize(columnIndex);
    HBufC8* buf = HBufC8::NewL(size);
    err = myStatement.ColumnBinary(columnIndex, buf->Ptr());
    //<do something with the data>
    delete buf;
}
if(err == KSqlAtEnd)
{
    // OK - no more records
}
else
{
    // process the error
}

myStatement.Close();
```

Here we can see that we are using `ColumnSize()` to decide how big a buffer we need to create for retrieving some binary data. We then create the buffer and pass a pointer to that buffer to the `ColumnBinary()` method. This of course involves a copy operation from the statement buffer into the user-allocated buffer.

Notice the two new features we've introduced here – the `KSqlAtEnd` return code and the `Close()` method on the statement object. `KSqlAtEnd` is returned when all the data that was returned from a query has

been exhausted. We do test for `KSqlAtRow` as before. But it is possible that some other code might be returned by `Next()`. This is the way to determine whether an error occurred or we simply ran out of data.

Closing the statement releases the statement's resources. After closing a statement, it can be reused but you need to `Prepare()` it again, since `Close()` removes the SQL query from the statement. Any returned data is lost when you call `Close()`.

Binary objects and text may be of any size. Using the above methods results in the data being copied, which means that there is a buffer on the server side, on the client side and in kernel space. This could work out to be too expensive in terms of resources. So to address this problem, it is possible to stream the data across the IPC divide:

```

RSqlStatement myStatement;
TInt err = myStatement.Prepare(database, "SELECT BinaryField FROM Tbl1");
TInt columnIndex = myStatement.ColumnIndex("BinaryField");
while((err = myStatement.Next()) == KSqlAtRow)
{
    RSqlColumnReadStream stream;
    err = stream.ColumnBinary(myStatement, columnIndex);
    // do something with the data in the stream
    stream.Close();
}
if(err == KSqlAtEnd)
{
    // OK - no more records
}
else
{
    // process the error
}

myStatement.Close();

```

This is the same pattern of code as we've seen previously. The interesting parts of this latest snippet are the lines:

```

RSqlColumnReadStream stream;
err = stream.ColumnBinary(myStatement, columnIndex);

```

The first statement creates a read stream and the second statement attaches the data to the stream where it can be manipulated by the standard stream operations. The two classes that support the streaming functionality are:

- `RSqlColumnReadStream`, derived from `RReadStream`
- `RSqlParamWriteStream`, derived from `RWriteStream`.

For more information on the methods inherited from the parent stream classes, see the Symbian OS C++ API documentation.

In each of the examples so far, we have used the `Next()` method from the `RSqlStatement` class to cause the statement to be executed and a row returned. You are probably wondering why we did not use `Exec()`. `Exec()` is used to perform a single operation. The `Next()` method has an implied `Exec()`.

There is another way to cause an SQL statement to be executed, and that is by using the `Exec(TRequestStatus &aStatus)` method.²³ The presence of the `TRequestStatus` reference indicates that this is an asynchronous way to execute an SQL statement. This allows us to write code that does not block during a long database operation.²⁴

The non-blocking `Exec()` is not intended to be used for row retrieval. Indeed, row retrieval uses the `Next()` interface, which does not offer a non-blocking version.

This is more or less the entire interface that you would use on a regular basis to manipulate an SQL database.

Platform Security

Since Symbian OS v9, platform security has played an important role (see Chapter 9). This section looks at how platform security is supported in the context of Symbian SQL. The key class to consider is the `RSqlSecurityPolicy` which encapsulates the policy to be applied to a database when it is created.

Database creation is the only opportunity for a security policy to be specified. Once the database is created, then the policy is a permanent part of it. The only way to change a policy is to delete the database and recreate it with the new desired policy.

Here is what the public interface to the `RSqlSecurityPolicy` class looks like:

```
class RSqlSecurityPolicy
{
public:
    enum TPolicyType
    {
        ESchemaPolicy,
        EReadPolicy,
        EWritePolicy
    };
    enum TObjectType
    {
        ETable
    };

    IMPORT_C RSqlSecurityPolicy();
    IMPORT_C TInt Create(const TSecurityPolicy& aDefaultPolicy);
    IMPORT_C void CreateL(const TSecurityPolicy& aDefaultPolicy);
```

²³ The `Exec()` in the `RSqlDatabase` interface also has an overloaded asynchronous version.

²⁴ See Chapter 6 for more information on active objects.

```

IMPORT_C void Close();
IMPORT_C TInt SetDbPolicy(TPolicyType aPolicyType,
                        const TSecurityPolicy& aPolicy);
IMPORT_C TInt SetPolicy(TObjectType aObjectType,
                        const TDesC& aObjectName,
                        TPolicyType aPolicyType,
                        const TSecurityPolicy& aPolicy);
IMPORT_C TSecurityPolicy DefaultPolicy() const;
IMPORT_C TSecurityPolicy DbPolicy(TPolicyType aPolicyType) const;
IMPORT_C TSecurityPolicy Policy(TObjectType aObjectType,
                        const TDesC& aObjectName,
                        TPolicyType aPolicyType) const;
IMPORT_C void ExternalizeL(RWriteStream& aStream) const;
IMPORT_C void InternalizeL(RReadStream& aStream);
};

```

This class is in fact a container for policies that can apply to databases (and, in the future, will apply to database tables). The easiest way to understand how security policies are applied to a database is to work through an example.

```

TSecurityPolicy defaultPolicy;
RSqlSecurityPolicy securityPolicy;
RSqlDatabase database;
TInt err;

securityPolicy.Create(defaultPolicy);
TSecurityPolicy schemaPolicy;
...
err = securityPolicy.SetDbPolicy(RSqlSecurityPolicy::ESchemaPolicy,
                                schemaPolicy);

TSecurityPolicy writePolicy;
...
err = securityPolicy.SetDbPolicy(RSqlSecurityPolicy::EWritePolicy,
                                writePolicy);

TSecurityPolicy tablePolicy1;
...
err = securityPolicy.SetPolicy(RSqlSecurityPolicy::ETable,
    "Table1", RSqlSecurityPolicy::EWritePolicy, tablePolicy1);
TSecurityPolicy tablePolicy2;

err = securityPolicy.SetPolicy(RSqlSecurityPolicy::ETable,
    "Table2", RSqlSecurityPolicy::EReadPolicy, tablePolicy2);

err = database.Create(KDatabaseName, securityPolicy);

securityPolicy.Close();

```

In this code, the first line creates a default security policy. This policy can then be refined to create a specialized policy for the database to which it is applied. We then create a container for the security policy. The next two lines declare the resource handle for the database and a place to put error codes as we progress.

Now we load the container with the default policy we created earlier. However, we override this policy to create a policy with the characteristics

we desire for our database. For this example, we need a holder for the schema policy. This is the same class as is found in the Base E32 component, further details of which are available in the Symbian API documentation.

Once the schema policy is set up to your satisfaction, it can then be added to the SQL security policy container:

```
err = securityPolicy.SetDbPolicy(RSqlSecurityPolicy::ESchemaPolicy,
                                schemaPolicy);
```

This process is repeated for the write policy and the table policy.²⁵ Note that the database (not the table) has had no read policy defined in the above example.

The two database tables Table1 and Table2 have had a write policy and a read policy applied, respectively. This demonstrates how the policies can be reused.

Once each of these steps has been completed, it is possible to create the database, passing in the newly created policy object.

```
err = database.Create(KDatabaseName, securityPolicy);
```

Finally we close the security policy, which then becomes available for reuse.

```
securityPolicy.Close();
```

When creating the database, the information carried in the security policy is persisted inside the database, where it remains, unchanged, until the database is deleted. The original security policy object we used to create the database is no longer needed and can be destroyed without any impact on the created database.

22.4 Symbian SQL Error Codes

Identifier	Value	Description
KSqlErrGeneral	−311	SQL error or missing database
KSqlErrInternal	−312	Internal logic error

²⁵ The first release of Symbian SQL does not include an implementation of the table policy. The syntax is valid but the policy is not enforced.

Identifier	Value	Description
KSqlErrPermission	−313	Access permission denied
KSqlErrAbort	−314	Call back routine requested an abort
KSqlErrBusy	−315	Database file is locked
KSqlErrLocked	−316	Table in a database is locked
KSqlErrNoMem	−317	Out of memory
KSqlErrReadOnly	−318	Attempt to write a read-only database
KSqlErrInterrupt	−319	Operation terminated by <code>sqlite3_interrupt</code>
KSqlErrIO	−320	I/O error
KSqlErrCorrupt	−321	Database file image invalid
KSqlErrNotFound	−322	Table or record not found
KSqlErrFull	−323	Database is full
KSqlErrCantOpen	−324	Unable to open database file
KSqlErrProtocol	−325	Database lock protocol error
KSqlErrEmpty	−326	Database is empty
KSqlErrSchema	−327	Bad schema
KsqlErrTooBig	−328	Too much data for one row
KSqlErrConstraint	−329	Abort to constraint violation
KSqlErrMismatch	−330	Data type mismatch
KSqlErrMisuse	−331	Library used incorrectly
KSqlErrNoLFS	−332	Uses OS features not supported on host

Identifier	Value	Description
KSqlErrAuthorization	−333	Authorization denied
KSqlErrFormat	−334	Auxiliary database format error
KSqlErrRange	−335	Bind parameter is out of range
KSqlErrNotDb	−336	Not a database file
KSqlErrStmtExpired	−360	SQL statement expired – needs to be prepared again

Summary

In this chapter we first introduced the fundamental terminology and concepts of relational database management systems, and discussed some more advanced features such as using multiple databases and stored procedures.

This was followed by an overview of SQL syntax, as employed both from SQLite and via the Symbian OS API. After a look at the Symbian OS SQL server architecture, we used the Symbian OS API to show how to manipulate data, including considering support for platform security.

Finally, the chapter presented a list of the Symbian OS SQL error codes that you may encounter when using the API.

Appendix

Developer Resources

Download the source code for this book at
developer.symbian.com/main/academy/press/books/scmp_v3.

Symbian OS Software Development Kits

Software development kits (SDKs) are based on a particular user interface (UI) for Symbian OS. Each distinct UI has an associated set of system applications for such tasks as messaging, browsing, telephony, multimedia and contact and calendar management. These applications typically make use of generic application engines provided by Symbian OS. SDKs are required to support the installation of third-party applications written in native C++. There are two UIs for Symbian OS v9, SDKs for which can be found at the websites listed here:

- Nokia S60 (***www.forum.nokia.com/S60***)
- UIQ (***developer.uiq.com/devtools_uiqsdk.html***)

For the independent software developer, the most important thing to know in targeting a particular phone is its associated reference platform. You can then decide which SDK you need to obtain. In most cases, you can target – with a single version of your application – all phones based on the same reference platform and Symbian OS version associated with this SDK. The Symbian OS System Definition papers give further details of possible differences between phones based on a single SDK (***www.symbian.com/phones***).

Getting an IDE for Development

To develop your code, you'll need an Integrated Development Environment (IDE), which contains an editor, compiler and linker. Two IDEs

are recommended – for non-commercial development, use Carbide.c++ Express Edition, which is a free download. Otherwise, Carbide.c++ Developer and Professional Editions are recommended, along with CodeWarrior. Support for Visual Studio .NET 2003 is also available using Carbide.vs.

- Carbide.c++ (www.forum.nokia.com/carbide)
- CodeWarrior (www.forum.nokia.com/codewarrior)

Getting a UID for Your Application

A UID is a 32-bit number. Every graphical application should have its own UID that allows Symbian OS to distinguish files associated with that application from files associated with other applications. UIDs are also used in other circumstances, such as to identify streams within a store and to identify one or more of an application's views.

To get a UID, register with and log on to the Symbian Signed website (www.symbiansigned.com). Request your UIDs from the My Symbian Signed tab – ten UIDs is a reasonable first request.

Don't build different Symbian OS applications with the same application UID – even the same test UID – on your emulator or Symbian OS machine. If you do, the system recognizes only one of them and you will not be able to launch any of the others.

Symbian OS Developer Tools

As well as tools offerings from Symbian partners, Symbian Developer Network (developer.symbian.com/main/tools) provides a number of free, unsupported and open source tools.

Support Forums

Symbian Developer Network offers newsgroups and support forums:

- news://developer.symbian.com
- <http://developer.symbian.com/forum>

Symbian Developer Network partners also offer support for developers.

Sony Ericsson Developer World

As well as tools and SDKs, Sony Ericsson Developer World (***developer.sonyericsson.com***) provides a range of services including newsletters and support packages for developers working with the latest Sony Ericsson products such as the P990.

Forum Nokia

As well as tools and SDKs, Forum Nokia (***forum.nokia.com/main.html***) provides newsletters, the Knowledge Network, fee-based case-solving, a Knowledge Base of resolved support cases, discussion archives and a wide range of C++ and Java-based technical papers of relevance to developers targeting Symbian OS.

Sun Microsystems Developer Services

In addition to providing a range of tools and SDKs, Sun also provides a wide variety of developer support services including free forums, newsletters, and a choice of fee-based support programs.

- Forums: ***forum.java.sun.com***
- Support and newsletters: ***developer.java.sun.com/subscription***

Symbian OS Developer Training

Symbian's Technical Training team and Training Partners offer public and on-site developer courses around the globe. Course dates and availability are listed at ***www.symbian.com/developer/training***.

Symbian normally offers a 20 % discount on all bookings confirmed up to one month before the start of any course. This discount cannot be used in conjunction with any other discounts.

Course	Level	Language
Symbian OS essentials	Introductory	C++
Java on Symbian OS	Introductory	Java
Symbian OS: Application engine development	Intermediate	C++
Symbian OS: Application UI development	Intermediate	C++
Symbian OS: Internals	Advanced	C++
Symbian OS: UI system creation	Advanced	C++

Intermediate and advanced courses require previous attendance at the OS Essentials course. The UI system creation course also requires previous attendance at the Application UI course.

Developer Community

These community websites offer news, reviews, features and forums, and represent a rich alternative source of information that complements the Symbian Developer Network and the development tools publishers. They are good places to keep abreast of new software and to announce the latest releases of your own applications.

My-Symbian

My-Symbian (***my-symbian.com***) is a website based in Poland and dedicated to news and information about Symbian OS phones. This site presents descriptions of new software for Symbian OS, classified by user interface. It also features discussion forums and an online shop.

All About Symbian

All About Symbian (***www.allaboutsymbian.com***) is a website based in the UK and dedicated to news and information about Symbian OS phones. The site features news, reviews, software directories and discussion forums. It has strong OPL coverage.

SymbianOne

SymbianOne (***www.symbianone.com***) features news, in-depth articles, case studies, employment opportunities and event information all focused on Symbian OS. A weekly newsletter provides up-to-date coverage of developments affecting the Symbian OS ecosystem. This initiative is a joint venture with offices in Canada and New Zealand.

NewLC

NewLC (***www.newlc.com***) is a collaborative website based in France and dedicated to Symbian OS C++ development. It aims to be initially valuable to developers just starting writing C++ applications for Symbian OS; in time it will cover more advanced topics.

infoSync World

infoSync World (***symbian.infosyncworld.com***) is a website based in Norway that provides features, news, reviews, comments and a wealth

of other content related to mobile information devices. It features a section dedicated to Symbian OS, covering new phones, software and services – mixed with strong opinions that infoSync is not afraid to share.

TodoSymbian

TodoSymbian (www.todosymbian.com) is a website written in Spanish. It provides news, reviews, software directories, discussion forums, tutorials and a developers' section.

Symbian OS Books

Development

Symbian OS C++ for Mobile Phones, Vol. 1, Harrison *et al.* John Wiley & Sons. ISBN: 0470856114

Symbian OS C++ for Mobile Phones, Vol. 2, Harrison *et al.* John Wiley & Sons. ISBN: 0470871083

Symbian OS Explained, Stichbury. John Wiley & Sons. ISBN: 0470021306

Developing Software for Symbian OS, Babin. John Wiley & Sons. ISBN: 0470018453

Symbian OS Internals, Sales *et al.* John Wiley & Sons. ISBN: 0470025247

Symbian OS Platform Security, Heath *et al.* John Wiley & Sons. ISBN: 0470018828

Symbian OS C++ Architecture, Morris. John Wiley & Sons. ISBN: 0470018461

Programming PC Connectivity Applications for Symbian OS, McDowall. John Wiley & Sons. ISBN: 0470090537

S60 Programming – A Tutorial Guide, Coulton *et al.* John Wiley & Sons. ISBN: 9780470027653

Management

Symbian for Software Leaders, Wood. John Wiley & Sons. ISBN: 0470016833

Symbian Academy

The Accredited Symbian Developer Primer, Stichbury and Jacobs. John Wiley & Sons. ISBN: 0470058277

Smartphone Operating System Concepts with Symbian OS, Jipping. John Wiley & Sons. ISBN: 0470034491

References

- Bluetooth SIG (2006) *Bluetooth Specification v2.0 + EDR*, **www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/**
- British Standards Institute (2005) *The C++ Standard: Incorporating technical corrigendum No. 1*, 2nd Edition, Wiley.
- Furber, S. B. (2000) *ARM System-on-chip Architecture*, Addison Wesley.
- Heath, C. (2006) *Symbian OS Platform Security*. Chichester: John Wiley & Sons.
- Henney, K. (2001) *The Miseducation of C++*, Application Development Advisor.
- IETF (1999) 'MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)', **www.ietf.org/rfc/rfc2557.txt**
- Mery, D. (2000) *Why is a different operating system needed?* CutterIT Journal.
- Morris, B. (2007) *The Symbian OS Architecture Sourcebook: Design and evolution of a mobile phone OS*, Symbian Press.
- Sales, J. (2005) *Symbian OS Internals: Real-time kernel programming*, Symbian Press.
- Stallings, W. (2000) *Operating Systems: Internals and design principles*, Prentice Hall.
- Taligent Inc. (1994) *Taligent's Guide to Designing Programs*, Addison Wesley.
- Tanenbaum, A. S. and Woodhull, A. S. (1997) *Operating Systems: Design and implementation*, 2nd Edition, Prentice Hall.
- Tasker, M., Allin, J. Dixon, J. Forrest, J., Heath, M., Richardson, T. and Shackman, M. (2000) *Professional Symbian Programming: Mobile Solutions on the EPOC Platform*, Wrox.

Index

- & (address-of/reference operator) 50–1, 55–6, 111–14, 692–3
- += (assignment operator) 144
- == (equal comparison operator) 141, 168
- = (assignment operator) 131, 144, 146–7, 189
- >= (greater-than-or-equal-to comparison operator) 141, 168
- <= (less-than-or-equal-to comparison operator) 141, 168
- != (not-equal comparison operator) 141, 168
- > (greater-than comparison operator) 141, 168
- >> (input operator) 123–4, 218–19, 337–8
- < (less-than comparison operator) 141
- << (output operator) 123–4, 218–19, 337–8
- 32-bit CPUs 17–18

- a drive 205
- a (function argument) prefixes 36, 45

- AAC (advanced audio coding) 653
- abld 12–14, 268–9, 383–4
- abstraction principles, object-oriented approaches 43–4, 57–9, 69–70, 88, 225–6, 367–8, 515, 534–5, 558, 561–7, 588–9, 594, 652–3
- Accept 617–19, 623–4
- access modes, files 211–14
- ACID (atomic, consistent, isolated, durable) RDBMS concepts 720–1
- ACS Publisher ID 269–70, 273, 275–6, 277–8
- Activate 327, 404–5, 425–31, 479–80, 507, 511, 520–1, 579–80
- ActivateContext 511–14, 525
- ActivateGc 502–14, 520–1, 524–5
- ActivateViewL 396, 407–8, 420
- activation processes, views 396, 397, 398, 407–11, 425–9
- active objects
 - see also asynchronous operations; events; RunL

- animation 523
- basic pattern 160
- CActive 44, 158–97, 249, 576–82
- cancellations 159–67, 176–81, 184, 186–8, 191–6, 199–200
- client–server architecture 179–81, 183–6, 241–3, 247–54, 259–62
- co-operative event handling 175
- communications 174–5
- concepts 30–4, 60, 151–201, 523–4, 577–82, 682–99, 749
- construction 159–67, 174, 191–4, 196–7
- Deque 175, 187
- destruction 159–67, 188, 189, 196–7, 252–4
- early cancellations 179–80
- error handling 159–67, 177–81, 182–90, 191–200
- error values 190
- event handling 31–4, 153–201, 577–82
- examples 158–67, 190–4, 195–200

- active objects (*continued*)
 - exhausted server resources 180–1
 - Fibonacci program example 195–200
 - GUI framework 32–4, 158
 - ICL 682–99
 - in-depth look 158–67
 - KRequestPending 169–70, 177–81, 182–3, 184–7, 189, 192–3, 198–200
 - late cancellations 179–81
 - leave/trap mechanism 159–67, 177–81, 189–90, 251–2
 - leaving asynchronous methods 190
 - long-running tasks 194–200
 - normal request processing 179
 - overheads 200–1
 - panics 182–7, 188, 252
 - performance issues 200–1
 - priorities 170–5, 194–200, 577–8
 - problems 173–5, 182–90
 - queues 170–5, 184–7
 - request methods 159–67, 177–81
 - RThread::RequestSignal 186
 - second requests 183–4
 - self-completion 191–200
 - ‘Start flashing menu item’ example 158–9, 165–7
 - starvation problems 173–5, 189
 - state machines 190–200
 - stray signal panics 182–7
 - unable-to-start requests 177–8
 - usage styles 159
 - workings 167–71
- active scheduler
 - communications 174–5
 - concepts 31–4, 158–201, 577–82, 698–9
 - event-processing wait loop 158, 170–2, 181
 - roles 170–2
 - starting/stopping 181
 - stray signal panics 182–7
- ActivePageId 472
- adaptive multi-rate (AMR) 653, 666–7, 676–7
- Add 167–72, 175
- AddAddresseeL 642–6
- AddAttachment 640–6
- AddControlLC 357–64
- AddDataSinkL 681–2
- AddDataSourceL 681–2
- AddFileAttachmentL 642–6
- AddFileSystem 208
- AddFriendL 588–9
- AddRecipientL 634–5, 638–41
- AddTextAttachmentL 642–6
- AddToStackL 336–7, 409–10, 426–7, 434–8
- AddView... 322–6, 356, 395–6, 406–10
- advanced audio coding (AAC) 653
- After 160, 162–7
- AIM 588, 596–7
- aknapp.h 314
- aknappui.h 314
- akndoc.h 314
- alarms 559–60
- alerts, dialogs 460–2
- AlertWin 460–2
- All About Symbian community website 758
- AllFiles 265
- Alloc... 46, 82–3, 90, 101–3, 132–3, 138, 140, 388
 - correct usage 132–3
 - HBufC usage 132–3
- AllocReadResourceL 388, 516
- Alt+F2 keys 287
- Alt+F4 keys 287
- AMR (adaptive multi-rate) 653, 666–7, 676–7
- animation 516, 521–3, 694, 696–8
 - see also Flush
- active objects 523
- concepts 521–3, 694, 696–8
- images 694, 696–8
- AnnotationFont 489–92
- ANSI 716
- antennas, Tuner API 707–8
- AOs see active objects
- APIs
 - Camera API 584, 651, 656, 699–706
 - capabilities 29–30, 250, 264–8, 274–80, 648–9, 676–7, 682, 707
 - CGraphicsContext 446, 450–1, 482–4, 486–92, 509–14, 531–56
 - classes 52–3, 58, 576–82
 - clips 653–66
 - concepts 36–40, 51–5, 58, 185, 213–14, 224–6, 283–4, 472–5, 482–3, 576–82, 648–9, 651–713
 - descriptors 109–10
 - DevSound 652, 655, 666–72, 676–8
 - dialogs 472–5
 - file-sharing APIs 213–14
 - groupings 37–9
 - ICL 584, 651, 682–99
 - interface classes 58
 - IPCs 25–6, 215, 241–62, 730
 - multimedia APIs 276, 280, 584, 587, 602, 651–713
 - Platform Security 29–30, 217–18, 250, 257, 263–80, 648–9, 682, 729–31
 - stores 224–40
 - streaming APIs 654–6, 666–71
 - Symbian SQL 729–31
 - Tuner API 584, 651, 706–13
 - types 36–7, 51–2
 - virtual functions 54–5
- APPARC
 - see also application architecture; CApp...
 - concepts 39, 311–13
- AppDllUid 319–20, 333–4
- Append 94–5, 108, 111, 122, 125, 137, 139, 142, 356–7, 431–2
- AppendFill 142
- AppendFormat 140, 142

- AppendFormatList 142
- appendices 755–9
- AppendJustify 142
- AppendLC 357–8
- AppendNum 123, 140, 142
- AppendNumFixedWidth... 142
- AppendNumUC 143
- appendText 111
- app_file 329–30, 378–9
- appinfo.rh 327, 329, 377–9
- Application 239, 334
- application architecture
 - see also* APPARC; system services
 - concepts 39, 234–7, 311–13, 493–5, 583
- application framework
 - see also* graphical user interfaces; UIKON
 - concepts 311–30, 333–66, 576–82
- Application Launcher 2, 279
- application picker, screen layouts 5
- application space, screen layouts 3–5
- application UI
 - see also* CEikAppUi; view server
 - concepts 235–7, 312–13, 320–4, 335–44, 355–60, 367, 395–6, 406–8, 415, 423–9, 515–16, 532–56
 - header files 314
- application views 312–13, 323–7, 333, 336–7, 344–51, 356–64, 391–421, 426–7, 483–6, 493–5, 499–502, 533–56, 566–9
 - see also* CCoeControl
 - classes 312–14, 391–421
 - concepts 312–13, 323–7, 336–7, 344–51, 356–64, 391–421, 423–9, 483–6, 493–5, 499–502, 566–9
 - controls 324–7, 350–1, 356–7, 359–60, 400–6, 412–14, 423–9, 483–6, 499–502, 503–14, 566–9
 - header files 314
 - noughts-and-crosses application 344–51, 356–64, 398–420, 426–9, 499–502, 507–9
 - S60 326–7, 344–51, 393
 - screen-sharing processes 502–14
 - UIQ 325–6, 356–64, 391–420, 568–9
- application-initiated drawing 446–7, 493–5, 502
 - see also* drawing
- applications 17–20, 28–9, 33–4, 37, 206, 224, 227–34, 236–40, 241–62, 263–80, 296, 311–66, 608–9, 755–6
 - see also* programs
- API groupings 37
- authorization processes 264–6, 272–8
- classes 312–30, 333–66, 493–5, 576–82
- communications' requirements 608–9
- concepts 263–4, 268–74, 278–9, 296, 311–30, 608–9
- controls 324–7, 400–6, 412–14, 423–55
- device/size-independent graphics 529–56
- distribution preparations 268–72
- entry points 318–19
- files 203–4
- 'Hello World' example 1–15, 90–9, 111–14, 121–37, 158–67, 190–4, 219, 297–8, 315–30, 482–6, 506–7, 531–56
- icons 263, 329–30, 372–9, 385, 491–2, 546–9, 562
- installation 263–4, 268–74, 278–9, 296
- IPCs 241–62
- kernel architecture 19–20
- MVC (Model–View–Controller) 312–30, 451, 493–5, 557
- noughts-and-crosses application 220–1, 270–1, 331–66, 374–89, 398–420, 426–55, 464–80, 497, 499–502, 605–49
- Platform Security 29–30, 217–18, 250, 257, 263–80, 648–9, 682, 729–31
- publication 263–80
- registration files 328–30, 377–9
- releases 263–4
- resource files 38–9, 268, 270–1, 284, 313, 327–30, 352–5, 359, 361–4, 367–89
- screen-sharing processes 502–14
- self-signed applications 275
- shortcut keys 6, 568–9
- signed applications 263, 264–7, 269–78, 295–6
- test criteria 273–5, 277–8
- third-party applications 17–18, 584–91, 593–604, 682, 755
- UIDs 206, 224, 227–34, 236–40, 255, 267, 317–18, 329, 334–5, 378–9, 392, 408–9, 597–601, 633–7, 660–6, 674–5, 679, 681–2, 684–92, 702, 729–31, 756
- unsigned applications 265–6, 273–6, 279–80
- APP_REGISTRATION_INFO 329–30, 378–9
- Apps 375, 379, 383–4
- AppUi 235–7, 312–14, 320–4, 333, 335–44, 355–60, 367, 395–6, 406–8, 415, 515–16, 532–56
- ARM 10, 18, 20–1, 268–9, 283–4, 306–8, 383–4, 647
 - see also* CPUs
- ARRAY 371, 469–72

- arrays 21, 55–6, 210, 328, 352–3, 369–71, 400–6, 426–9, 469–72, 509, 562–3, 646–8
- arrow keys 6
- ascent, drawing basics 485–6
- AscentInPixels 537–8
- ASCII 33–4, 42, 387
- Assembler 61
- ASSERT... 48–9, 72–3, 114, 162, 165–71, 359–60, 409, 429–30, 436–8, 647–8, 659, 665–6
- assert macros 48–9, 72–3, 114, 162, 165–71, 359–60, 409, 429–30, 436–8, 647–8, 659, 665–6
- Assign... 104, 105–6, 146, 228–34, 237–9, 335
- asynchronous operations 21–2, 30–4, 151–201, 242–54, 608–9, 682–702
 - see also* active objects
 - concepts 151–6, 242–3, 608–9, 682–3
 - definition 151–2
- At 229–30, 235–7
- atomic RDBMS concepts 720–1
- Att 207
- Attach 215, 255–7, 738–9
- attachment operations, SQL 721–2, 738–41
- au files 653
- audio 584, 651–713
 - see also* multimedia...; sounds
 - buffer sizes 669–71
 - clip APIs 654–66
 - controller plug-ins 679–82
 - DevSound 652, 655, 666–72, 676–8
 - input streaming 670–1
 - notifications of resource
 - availability 659–61, 680–2
 - output streaming 667–70
 - playback 654–5, 656–66, 676–8, 680–2, 710–11
 - recording 654–5, 661–6, 676–8, 680–2, 711–12
 - streaming APIs 655–6, 666–71
 - tone API 655, 671–2
 - Tuner API 709–11
 - UIDs 660–6, 681–2
- authorization processes, Platform Security 264–6, 272–8, 648–9
- automatic objects 65–70, 76–86
 - see also* stack
- Avkon 39, 370–1, 469–72, 478
 - see also* CAkn...; S60
- avkon.lib 313, 317–18
- avkon.rh 327
- avkon.rsg 327
- backed-up behind windows 452–5, 501–2, 506–7, 515–21
- backed-up windows 451–5, 501–2, 506–7, 515–21
- BackedUpWindowL 452–5
- bad practices, C++ 59
- BAFL
 - see also* resource files; system services
 - concepts 38–9, 383, 388–9, 583, 586
- barsread.h 388–9
- base APIs 37–9
- BaseConstructL 322, 325–6, 336–7, 406–8, 411–13, 426
- baselines, drawing basics 485–6
- BASIC 722
- basic application framework library 39
 - see also* resource files
- basics, Symbian OS 1–15, 17–86
- batteries 4, 18, 26–7, 701–2
 - see also* power management
- BeginRedraw 501–14, 527–8, 580–1
- Berkeley Sockets API 584, 616
- billing storms 272
- binary data 87–149, 211
 - see also* databases; descriptors
 - concepts 87, 109–10, 122–3
 - descriptors 87, 109–10, 122–3, 124
- streaming APIs 655–6, 666–71
- tone API 655, 671–2
- Tuner API 709–11
- UIDs 660–6, 681–2
- authorization processes, Platform Security 264–6, 272–8, 648–9
- automatic objects 65–70, 76–86
 - see also* stack
- Avkon 39, 370–1, 469–72, 478
 - see also* CAkn...; S60
- avkon.lib 313, 317–18
- avkon.rh 327
- avkon.rsg 327
- backed-up behind windows 452–5, 501–2, 506–7, 515–21
- backed-up windows 451–5, 501–2, 506–7, 515–21
- BackedUpWindowL 452–5
- bad practices, C++ 59
- BAFL
 - see also* resource files; system services
 - concepts 38–9, 383, 388–9, 583, 586
- barsread.h 388–9
- base APIs 37–9
- BaseConstructL 322, 325–6, 336–7, 406–8, 411–13, 426
- baselines, drawing basics 485–6
- BASIC 722
- basic application framework library 39
 - see also* resource files
- basics, Symbian OS 1–15, 17–86
- batteries 4, 18, 26–7, 701–2
 - see also* power management
- BeginRedraw 501–14, 527–8, 580–1
- Berkeley Sockets API 584, 616
- billing storms 272
- binary data 87–149, 211
 - see also* databases; descriptors
 - concepts 87, 109–10, 122–3
 - descriptors 87, 109–10, 122–3, 124
- streaming APIs 655–6, 666–71
- tone API 655, 671–2
- Tuner API 709–11
- UIDs 660–6, 681–2
- LIT operations 124
- binary files 211–13
 - see also* files
- binary objects 28
- Bind 618–19
- binding operations, databases 742–3
- BindInt 744, 746
- BitBlt 492, 547–9
- BITGDI 514, 546–7
- BITMAP 374–7
- bitmap-based viewfinders 702, 703
- BitmapL 703, 705–6
- bitmaps 269–72, 285–6, 328–9, 372–7, 451–5, 491–2, 509–10, 525–8, 534, 546–55, 682–99, 704–6
- backed-up windows 451–5
- bitmap-based viewfinders 702, 703
- blitting 546–9, 553–5
- color depth 375–6, 525–6, 534, 552–5
- concepts 372–7, 451–5, 491–2, 509–10, 525–8, 534, 546–55, 682–99, 704–6
- conversions 372–3, 374–7, 682–99
- drawing 491–2, 509–10, 546–9
- fonts 549–50
- icons 372–7, 385, 491–2, 546–9
- mask bitmaps 373–7, 682–99
- noughts-and-crosses application 374–7
- rotation 683–9
- transformations 698–9
- Windows 372–3, 548–9
- BitmapTransforms 698
- blanket grants 266
- BLD files
 - see also* component definition file
 - concepts 12–14, 383–4
- bldmake 12–14
- blitting 546–9, 553–5

- Bluetooth 18, 265, 269, 274–5, 279, 288–9, 583–4, 586, 603, 605–10, 612, 615–32, 648–9
 - attributes 625–6, 630–2
 - concepts 605–10, 612, 615–32, 648–9
 - listening for incoming connections 621–4
 - published services 624–7
 - remote phones 627–30, 632
 - S60 627–30
 - searches 630–2
 - security issues 622–3, 648–9
 - serial communications 615, 620, 621
 - sockets server 616–17, 620–32
 - two-phone connections 621–32
 - UIQ 627, 629–30
- bmconv 372–3, 376, 548–9
 - see also* bitmaps
- BMP files 372–3, 374–5, 556
- books, Symbian OS 759
- Booleans 42–3
- bootstrap loader 284–5
- boundaries, descriptors 93, 137
- bracketing rules, coding conventions 46–9
- breakpoints, debugging 289, 290
- brush 346, 448–51, 487–92
- brushes 346, 448–51, 484–92, 500–2, 540–2
 - concepts 489, 500–2
 - drawing basics 346, 448–51, 484–92, 500–2
- BTCOMM 612–15, 621
 - see also* Bluetooth
- BUF... 368
- buffer descriptors 88–149
 - see also* descriptors; TBuf...
 - concepts 88–9, 93–6, 121–2, 129
- buffer sizes, audio 669–71
- buffered decoding 691
- BufferToBeEmptied 678
- BufferToBeFilled 677–8
- build 12–14, 268–9, 383–4
- built-in types 50–1, 66–7, 218, 368–9
 - see also* T (data-type) classes
- buttons 3–4, 6, 328, 352–3, 439–42, 461–2, 469–72, 483–4, 497, 505–6, 523–4, 562–82
- BYTE 368–9, 473–5
- C++ 1, 9–11, 14–15, 17, 30–6, 41–60, 216, 268–9, 286, 289–90, 316–17, 318, 371, 381, 384, 476, 730, 756, 757–9
 - bad practices 59
 - best practices 54–5
 - casting concepts 56–7, 498–9
 - comments 381
 - concepts 30–6, 41–60, 371, 587
 - developer resources 757–9
 - object-oriented approaches 8, 41–60
 - Symbian OS 30–6, 41–60, 61–2
 - templates 55–6
 - TPtr... non-conformance 131–2
- C 56–7, 87, 89, 99–100, 368–9, 381, 722
- c drive 18, 203, 205, 208, 284–6, 295
 - see also* RAM
- C (heap-allocated) classes
 - see also* heap
 - concepts 43–4, 66–70, 488
- C32 583–4, 603, 611–12
 - see also* Comms Server
- CActive 44, 158–97, 249, 576–82, 607–49
 - see also* active objects
- CActiveScheduler 576–82
- CActiveScheduler::Add 167–72, 175
- CActiveScheduler::Error 163–4
- CActiveScheduler::Start 165–7, 181
- CActiveScheduler::Stop 181
- CAF (Content Access Framework) 656
- CAimProtocol 590–1
- CAknApplication 238–9, 314–30, 334
- CAknAppUi 314–30, 335–44, 394
- CAknControl 458
- CAknDocument 234–5, 314–30, 334–5
- CAknInformationNote 337–8
- CAknView 345, 394, 398, 403–4, 406, 415
- CAknViewAppUi 394–6
- calendar application 5, 203–4
- callbacks *see* framework functions
- Camera API 584, 651, 656, 699–706
 - background 651, 699–706
 - bitmap-based viewfinders 702, 703
 - capturing the image 705–6
 - direct screen access viewfinders 702–3
 - preparations 704–5
 - resolution factors 704–5
 - selection processes 700
 - setting-up processes 700–2
 - still-image capturing considerations 703–6
 - viewfinders 702–3
- CameraInfo 700–6
- cameras 584, 651, 656, 699–706
 - see also* multimedia...
- CamerasAvailable 700–6
- Cancel 161–2, 164–7, 177–81, 188, 255–7, 613–14
- CancelClippingRect 490, 497
- CancelImageCapture 705–6
- cancellations, active objects 159–67, 176–81, 184, 186–8, 191–6, 199–200
- CancelPlay 671–2
- CancelPrepare 671–2
- CancelWrite 627

- CApaApplication 312–30, 334
- capabilities 29–30, 65–7, 250, 264–8, 274–80, 295–8, 437–8, 648–9, 676–7, 682, 699–700, 707
 - see also individual capabilities*
 - basic level 265
 - concepts 29–30, 250, 264–8, 274–81, 295–8, 317–18, 437, 648–9, 676–7, 682, 699–700, 707
 - extended level 265
 - input capabilities 437–8
 - list 266–7, 275–6, 279–80
 - unsigned-sandboxed capabilities 265–6, 279–80
 - varieties 265, 266–7, 279–80, 648–9
- CAPABILITY 266–7, 317–18
- CApaDocument 312–30, 334
- Capitalize 142
- capture/grab contrasts, pointers 440–2, 573
- CaptureImage 705–6
- CaptureLongKey 581–2
- Carbide.c++ IDE 1, 10–11, 14–15, 268, 286, 289–91, 294–5, 302, 306, 309, 316, 318, 368, 756
 - debugging 14–15, 289–91, 294–5, 302, 306, 309
 - resource files 368
- CArrayFixBase 55–6
- cascaded menus 5–6
- cascading deletes, SQL 720–1
- case 68, 76, 78, 162–3, 192–4, 198, 237, 251, 323–4, 354–5, 407–8, 410, 413–14, 419–20, 435–8, 443–4, 480, 509
- casting concepts 56–7, 498–9
- catch 9, 73
- categories, classes 43–4, 66–70
- CAudioPlayEngine 658–66
- CAudioPlayer 657–66
- CAudioRecordEngine 665–6
- CAudioStreamOutEngine 667–71
- CBA (Control Button Array) 328, 352–3
- CBase 43, 44, 48, 55, 58, 62–70, 81–2, 85, 95, 102, 116–17, 168, 182, 248, 339, 342, 393, 515–18
 - see also heap*
 - cleanup stack 79–80
 - concepts 66–7, 79–80, 85, 102, 116–17
 - virtual destructors 66–7, 79–80, 85
- CBaseMtm 642
- CBitmapContext 492, 547–9
- CBitmapDevice 535, 547–9
- CBitmapRotator 683–99
- CBitmapScaler 683–99
- CBluetoothSocket 616–32
- CBluetoothTransport 607–49
- CBtClientToServer 631–2
- CBTDevice 629–30
- CBufferedImageDecoder 691
- CBufStore 224–6
- CCamera 675, 699–706
- CCameraEngine 701–6
- CClickMaker 576
- CClientMtmRegistry 641–6
- CCnvCharacterSetConverter 124
- CCoeAppUI 393–420, 516–17, 575–82
- CCoeControl 54, 57, 312–30, 344–51, 355–60, 393–420, 423–55, 458–80, 482–3, 495–8, 501–2, 505–21, 531–3, 563–5, 571–2, 578
 - see also controls; dialogs; views*
 - drawing-support summary 514–21
- CCoeControlStack 567–8, 576–82
- CCoeEnv 163, 311–13, 388–9, 515–16, 532–56, 575–82
- CCoeRedrawer 576–82
- CColorList 553
- CConsoleBase 8, 121–2
- CDelayedHello 159–67, 171, 190–4
- CDialogUserPasswordDialog 465–6
- CDictionaryFileStore 238–9
- CDictionaryStore 238–9
- CDirectFileStore 224–6, 227–34
- CEditableText 545–6
- CEikAlignedControl 477
- CEikApplication 238–9, 312–30, 334, 532–56
- CEikAppUi 312–30, 334–44, 351, 393–420, 532–56
 - see also controller*
- CEikArrowHeadScrollBar 477
- CEikBorderedControl 458, 476–7
- CEikButtonBase 477
- CEikChoiceListBase 477
- CEikDialog 57, 458–80, 509, 562–3
 - see also dialogs*
- CEikDocument 234–7, 312–30, 334–5, 532–56
 - see also model*
- CEikEdwin 545
- CEikHorOptionButtonList 477
- CEikImage 477
- CEikInfoDialog 57
- CEikLabel 400–6, 412–13, 476–7
- CEikLabeledButton 477
- CEikListBox 477
- CEikMenuPane 353–4, 418–19
- CEikMenuPaneItem 388–9
- CEikonEnv 159, 237, 311–13
- CEikPrinterSetupDialog 551
- CEikRichTextEditor 545
- CEmailTransport 606–49
- CEmbeddedStore 224–6, 232–4
- CER files 269–72, 277
- cert 295–6
- Certificate Generator 269–70
- Certificate Signing Request (CSR) 277

- CExample...
 - device/size-independent graphics 532–56
- CFbsBitGc 509–14
- CFbsBitmap 547–9, 682–99, 703, 706
- CFibonacciGenerator 196–200
- CFileStore 225–37, 334–5
- CFlashingHello 165–7, 190–4
- CFont 482–92, 534–56
- CFrameImageData 693–4
- CGraphicsContext 446, 450–1, 482–4, 486–92, 495–8, 509–14, 531–56
 - see also* graphics context concepts 482–4, 486–92, 495–6, 497, 509–14, 531–56
 - features 486–7
- CGraphicsDevice 532–56
- ChangeDisplayL 410, 420
- ChangeMode 212–13
- channel identifiers (CIDs) 621–32
- char... 89, 93–5
- Character Converters 602
- CHARACTER_SET 369
 - see also* resource files
- Charconv 602
- CHelloWorldAppView 482–6, 492–8, 506–7
- chunking
 - see also* memory model concepts 24–5, 29, 706
- ChunkL 706
- ChunkOffsetL 706
- CIcqProtocol 590–1
- CIdle 173–4
- CIDs (channel identifiers) 621–32
- CImageDecoder 682–99, 706
- CImageDisplay 683–99
- CImageDisplayPlugin 696–7
- CImageEncoder 682–3, 691–9
- CImProtocol 588–604
- CInfraredTransport 607–49
- CJPEGExifDecoder 685
- classes
 - see also* C...; M...; R...; T...
- APIs 52–3, 58, 576–82
- applications 312–30, 333–66, 493–5, 576–82
- categories 43–4, 66–70
- concepts 43–4, 52–3, 57–9, 66–70, 576–82, 607
- controller 312–13, 320–4, 333, 336–44, 493–5, 606–9
- controls 424–55
- coordinate classes 487–92
- decoding 683–5
- dialogs 458–60, 462–3, 476–7
- documents 312–13, 320–1, 333, 334–5
- encoding 691–4
- engine classes 333, 336–44, 363–4
- graphics context 446, 450–1, 482–92
- interface principles 57–9, 196
- naming conventions 43–4, 60, 102
- relationships 57–9, 64, 160, 494, 516–17
- S60 313–30, 333–55, 394–5
- stores 224–6
- transport concepts 607
- UIQ 313–30, 355–66, 393–4
- views 312–14, 391–421
- cleanup stack 7–11, 35–6, 43–5, 60–2, 75–86, 99, 209, 308–9, 488–9, 576–82
 - see also* exception handling
- CBase 79–80
- CleanupStack::PushL
 - failures 79
- concepts 35–6, 43–5, 60–2, 75–86, 99, 209, 308–9
- examples 35–6, 76–81
- GUI applications 76–7
- pointers 35–6
- R (resource) classes 80–1
- roles 35, 75, 76–86
- unnecessary uses 78–9
- CleanupClosePushL 80–1, 99, 106–8, 146, 209, 211, 213–14
- CleanupDeletePushL 80–1
- CleanupReleasePushL 80–1
- CleanupStack, concepts 76–81, 576–82
- CleanupStack::Pop 133–5, 160–1, 217–18, 322–4, 341–2, 357–8, 406–8, 681
- CleanupStack::
 - PopAndDestroy 7–11, 35–6, 67, 76–83, 84–5, 90, 102–3, 108, 113, 133–5, 209, 211, 213–14, 218, 227–34, 236–9, 404, 639
- CleanupStack::PushL 7–11, 35–6, 45, 67, 76–81, 99, 133–5, 160–1, 217–18, 341–2, 615, 637–9
- Clear 347–8, 405–6, 444–51, 482–6, 499–502
- ClientRect 426
- client–server architecture
 - see also* CServer2; interprocess communications; RSessionBase
- active objects 179–81, 183–6, 241–3, 247–54, 259–62
- clean-up methods 252–4
- concepts 25–6, 33, 179–81, 183–6, 211, 214–16, 241–3, 247–54, 259–60, 261–2, 424, 504–14, 729–31
- critique 259–60, 261–2
- panics 252
- sessions 26, 180–1, 207–9, 211, 214–16, 241–3, 247–54, 259–60, 261–2, 424
- set-up methods 247–8
- start-up methods 248–50
- sub-sessions 253–4
- Symbian SQL 729–31
- using a session 250–2
- window server 504–14
- clipping region, drawing basics 490, 497–8, 511, 514
- clips, MMF 653–66

- Close 44, 68, 80–1, 102–3, 105–6, 146, 210, 214–16, 252–4, 256, 364, 387, 710, 737, 747–50
- close operations, databases 737–8
- CloseWindow 517–18
- CMdaAudioClipUtility 657–66
- CMdaAudioInputStream 655–6, 670–1
- CMdaAudioOutputStream 655–6, 667–71
- CMdaAudioPlayerUtility 654–5, 656–66
- CMdaAudioRecorderUtility 654–5, 711
- CMdaAudioToneUtility 655, 671–2, 677–8
- CMessageTransport 606–49
- CMMFBuffer 677–8
- CMMFController 678–82
- CMMFController-Implementation-Information 662–6
- CMMFControllerPlugin-SelectionParameters 662–6, 674–5
- CMMFDevSound 666–72, 676–8
- CMMFFormatImplementation-Information 662–6
- CMMFFormatSelection-Parameters 662–6
- CMMRdsTunerUtility 711–12
- CMmsClientMtm 641–6
- CMmsSendAsTransport 607–49
- CMmsTransport 607–49
- CMMTunerAudioPlayer-Utility 710–11
- CMMTunerAudioRecorder-Utility 711–12
- CMMTunerUtility 706–13
- CMsvAttachment 645–6
- CMsvEntry 636–7
- CMsvEntrySelection 636–7
- CMsvSession 635–7
- CMsvStore 642–6
- CMultiPartHello 191–4
- CMyAppUI 393–5
- CMyView 393–5
- CnvUtfConverter 124
- co-operative event handling, active objects 175
- CodeWarrior 289, 290–2, 294–5, 307, 309, 384, 756
 - debugging 289, 290–2, 294–5, 307, 309
 - resource files 384
- coding conventions
 - see also* naming...
 - bracketing rules 46–9
 - indentation style rules 46–9
 - line breaks 46–9
 - Symbian OS 17, 36, 43–9, 369–71
- coeau.i.h 569, 577
- coecntrl.h 314, 423, 495–8
- coemain.h 172–3, 516, 577
- Collapse 127–8, 142
- Collate 143
- collation processes, descriptors 138–9
- color depth, bitmaps 375–6, 525–6, 534, 552–5
- Column... 746–7
- columns, SQL 718–53
- COM... 612–13, 615
- COM ports 288–9, 612–13, 615
- command-line build 12–14, 268–9, 383–4
- commands 12–14, 162–3, 166–7, 237, 268–9, 321–6, 335–44, 351–5, 360–6, 380–1, 383–4, 407–8, 414–20, 509–10, 558–82, 680–1, 717–53
 - custom commands 680–1
 - dialogs 321–2
 - menus 351–5, 361–6, 380–1, 407–8, 414–20
 - S60 338–9, 351–5, 414–20
 - SQL 717–53
 - UIQ 325–6, 360–6, 407–8, 414–20
- COMMDB 584
- CommDD capabilities 276, 280
- CommitL 217–18, 228–34, 235–9, 642–6
- comms extension modules (CSYs) 603, 612, 615
- Comms Server 583–4, 603, 605–6, 611–15
- communications
 - see also* connectivity; messaging; serial...; system services
 - active objects 174–5
 - API groups 37–9
 - concepts 25–6, 215, 241–62, 583–4, 586–7, 603, 605–49
 - controller/transport communications 606–9
 - emulator 287–9
 - IPCs 25–6, 215, 241–62
 - Platform Security 648–9
 - requirements 608–9
 - security issues 622–3, 646–9, 682
 - sockets server 605–6, 615–32
 - transport concepts 605–9
- community websites, Symbian OS 758–9
- CompactL 230–1
- Compare 138, 140
- CompareC 139, 140
- CompareF 110, 138, 140
- compilation 27–9, 129–30, 216, 383–7
- Complete 252
- CompleteSelf 191–200
- complexity issues, drawing 509–14
- component definition file 11–14
 - see also* BLD files
- ComponentControl 46, 356–7, 405–6, 428–31, 436–8, 444, 508–9, 513–14, 518–21
- compound controls 426–9, 454–5, 457–80, 483–6, 507–9, 562, 564–6
 - see also* controls; dialogs
- compressed data 389, 652–3, 682–99

- CONARC (Converter Architecture) 602–3
- concrete behaviour,
 - object-oriented approaches 57–8, 88–9, 92, 224–5
- concurrency 30–4, 511–14, 527–8
- CONE (control environment) 38–9, 86, 172–3, 181, 193–4, 207–8, 311–13, 383, 483–4, 496–8, 503–23, 530, 545
 - see also* CCoe. . .
 - windows 503–4
- ConeUtils 44
- configuration, emulator 285–6, 294–5, 306
- Confirm button 6, 566–70
- Connect 617, 619–20, 632
- connectivity 18, 19, 37–9, 269, 278–9, 287–9, 583–4, 605–49
 - see also* communications
- consistent RDBMS concepts 720–1
- console 45
- ConsoleMainL 7–11
- const 50, 55–6, 93–4, 111–20, 124–9, 168–9, 495, 498
- const char. . . 89, 95–7
- const TDesC. . . 111–20, 124–9
- constants 27–9, 45, 66–7, 89, 168–70
- const_cast 57
- constraints, SQL 719–21
- ConstructFromResourceL 478–9
- ConstructL 52, 83–6, 160–7, 191–4, 196–7, 322–6, 336–44, 347–51, 355–6, 363–4, 404, 406–8, 411–13, 425–31, 444, 479, 506–7, 536, 579–80
- constructors
 - see also* New. . .
 - concepts 51–2, 55–6, 67–8, 81–6, 104, 159–67, 174, 191–4, 196–7, 322–6, 336–44, 355–6, 363–4, 406–13, 425–9, 516–17
 - leave functions 83
 - second-phase constructors 52, 81–6, 336–7, 406–8, 428–9, 516–17
 - two-phase construction 52, 81–6, 336–7, 406–8, 428–9, 516–17
- Contacts 584
- containers 404–5, 426–9, 454–5, 457–80, 483–6, 507–9, 517–18, 562, 564–7, 750–1
 - see also* compound controls
- Content Access Framework (CAF) 656
- context switches
 - see also* threads
- concepts 23–4
- inefficiencies 24
- ContinueConvert 689–91
- ContinueOpenL 691
- ContinueProcessingHeaderL 689, 691
- Control Button Array (CBA) 328, 352–3
- control context, special effects 522, 524–5
- control environment
 - see also* CCoe. . . ; CONE
- concepts 38–9, 86, 172–3, 181, 193–4, 207–8, 311–13, 383, 483–4, 496–8, 503–23, 530, 545, 576–82
- control stack, interaction graphics 567–9, 576–82
- ControlCaption 476–7
- ControlEnv 515–16
- controller 312–13, 320–4, 333, 336–44, 347–8, 351, 413–14, 493–5, 606–9, 662–6, 678–82
 - see also* CEikAppUi
- controller plug-ins, MMF 678–82
- ControlOrNull 476–7
- controls 38–9, 86, 172–3, 181, 193–4, 207–8, 311–13, 324–8, 350, 356–7, 359–60, 383, 400–6, 412–14, 423–55, 459–80, 482–6, 492–8, 499–523, 531, 545, 557–82
 - see also* screens; windows
- application views 324–7, 350–1, 356–7, 359–60, 400–6, 412–14, 423–9, 483–6, 499–502, 503–14, 566–9
- backed-up behind windows 452–5, 501–2, 506–7, 515–21
- backed-up windows 451–5, 501–2, 506–7, 515–21
- classes 424–55
- compound controls 426–9, 454–5, 457–80, 483–6, 507–9, 562, 564–6
- concepts 423–55, 482–6, 492–8, 499–521, 557–82
- custom controls 477–80
- definition 423–4, 483, 515
- dialogs 423–4, 427–8, 462–75, 503–4, 505–9, 559–82
- dimmed/invisible controls 409–11, 454–5, 520–1, 561, 563–4
- drawing 424, 427–9, 438, 444–51, 482–6, 492–8, 499–502, 515–21
- general-purpose controls 562–3, 566–7
- interaction graphics 557–82
- keys 423–9, 432–42, 459–60, 469–72, 523–4, 557–82
- layouts 429–32, 519–20
- lodger controls 424, 504–9, 514–15, 516–18
- noughts-and-crosses application 426–55, 464–80, 497, 499–502
- observer interface 442–4, 463, 474–5, 564–6
- pointer events 423–4, 432, 439–42, 459–60, 557–82

- controls (*continued*)
 - redrawn windows 445–51, 492–8, 501–4, 511–14, 526–9, 568, 577–82
 - screen-sharing processes 502–14
 - screens 423–55, 459–60, 502–14
 - simple controls 424–9
- SizeChanged 350, 404–5, 427–32
- stock controls 475–7
- Tuner API 709–10
- types 424–9, 504–9, 516–18
- user-generated events 423–4
- views 324–7, 350–1, 356–7, 359–60, 400–6, 412–14, 423–9, 483–6, 499–502, 567
- window server 424, 432–3, 439–42, 445–51, 496–8, 503–14, 557–82
- window-owning controls 424, 457, 504–9, 514–15, 516–18
- windows 424, 432–3, 439–42, 445–51, 483–6, 496–8, 503–14, 515–21, 557–82
- convenience functions 52
- conversions
 - bitmaps 372–3, 374–7, 682–99
 - descriptors 122–8
 - ICL 682–99
 - wide/narrow descriptors 124–8
- Convert 687, 689–94
- Converter Architecture (CONARC) 602–3
- coordinates 487–92, 503–4, 514
- Copy 124–8, 139, 143, 738
- copy constructors 51, 67
- copy operations, SQL 737–8
- CopyC 143
- CopyCP 143
- CopyF 143
- CopyLC 138, 143
- copyTextL 134–5
- CopyUC 138, 143
- CountComponentControls 356–7, 404–6, 427–9, 436–8, 508–9, 513–14, 518–21
- cp1252 369
- CParser 586
- CPermanentFileStore 224–6, 228–34
- CPersistentStore 225–6, 231–4
- CPicture 546
- CPP files 136, 318–19, 372–3, 384
- CPrinterDevice 535, 551–2
- CPrintPreviewImage 58, 551
- CPrintSetup 551
- CPUs 17–18, 19–22, 189, 200, 283–4, 306–8
 - see also* ARM; x86...
- CQBtUISelectDialog 629–30
- CQikApplication 314–30
- CQikAppUi 314–30, 393, 395–6, 406–8
- CQikColorSelector 555
- CQikCommand 363–4, 407–8
- CQikDocument 237, 314–30
- CQikMultiPageViewBase 355–66
- CQikNumericEditor 477
- CQikSimpleDialog 458–80
 - see also* dialogs
- CQikSlider 477
- CQikSoundSelector 477
- CQikTabScreen 477
- CQikTabScreenPage 477
- CQikTTimeEditor 477
- CQikVertOptionButtonList 477
- CQikViewBase 314–30, 355–66, 393, 399–401
- CQikViewDialog 458–80
 - see also* dialogs
- Create 67–8, 104–5, 110, 146, 211, 215, 698–9, 731–2, 749–51
- CreateAppUiL 320–4, 334–5
- CreateAttachment2L 643–6
- CreateAttachmentL 640–6
- CreateBackedUpWindowL 452–5, 517–18
- CreateContext 535, 545
- CreateCustomControl 474–5, 478
- CreateDocument... 319–20, 333–4
- CreateImplementationL 595–601
- CreateL 46, 102, 104–7, 111–12, 123–4, 146, 228–34, 248–9, 634–5, 638–46
- CreateLC 46, 228–37
- CreateMax... 104–5, 146
- CreateMessage2L 643–6
- CreateNewItemL 410–11, 413–14
- CreatePrivatePath 208
- CreateResourceReaderLC 516
- CreateServiceRecordL 625–6
- CreateSession 247–54
- CreateSubSession 253–4
- CreateTextAttachmentL 642–6
- CreateTile 427–9
- CreateWindowL 326–7, 404–5, 425–9, 431–2, 444–51, 507, 517–18
- creation operations
 - databases 723–9, 730–6, 749–51
 - message queues 245–6, 257–8
- CResolver 600
- CRichText 639
- CRS232Transport 607–49
- cryptographic libraries 587
- CScreenDevice 535
- CSdpAgent 630–2
- CSdpAttrValueDES 626–7
- CSdpSearchPattern 630–2
- CSecureStore 225–6
- CSecurityBase 224
- CSerialTransport 607–49

- CServer2 169, 247–54
 - see also* client–server architecture
 - CSession2 247–54
 - CSheduledTask 585
 - CSmsTransport 607–49
 - CSR (Certificate Signing Request) 277
 - CStreamDictionary 227–34, 236–7, 334–5
 - CStreamStore 224–6, 235–7, 238
 - CSYs (comms extension modules) 603, 612, 615
 - CTextView 543–5
 - CTransport 607–49
 - CTransportInterface 606–49
 - Ctrl+Alt+Shift+F keys 524
 - Ctrl+Alt+Shift+G keys 524
 - Ctrl+Alt+Shift+K keys 524, 567
 - Ctrl+Alt+Shift+M keys 524
 - Ctrl+Alt+Shift+R keys 524, 568
 - Ctrl+Alt+Shift+S keys 292, 376
 - Ctrl+Alt+Shift keys 292, 376, 427
 - CTypefaceStore 549–50
 - curly brackets, coding conventions 47–9
 - current position, drawing basics 489
 - cursors 557–82
 - see also* focus
 - custom commands 680–1
 - custom controls, dialogs 477–80
 - CVideoPlayEngine 673–5
 - CVideoPlayerUtility 655, 672–4
 - CVideoRecorderUtility 655, 674–5
 - CWindowGc 347–9, 405–6, 444–55, 482–6, 490, 499–502, 509–14, 520–1, 541

 - D classes 43
 - d drive 205
 - data 285–6, 294–5
 - data caging, concepts 29, 264, 267–8, 730
 - data model *see* model
 - data names, naming conventions 44–5
 - data section, executable programs 27–9
 - data types
 - SQL 726–7
 - Symbian OS 41–3, 368–9
 - data validation, security issues 646–8
 - database engine, SQL basics 717–18
 - databases 203–4, 225–6, 240, 307, 584–5, 624–9, 715–53
 - see also* RDBMS; SQL
 - attachment operations 721–2, 738–41
 - binding operations 742–3
 - close operations 737–8
 - concepts 715–53
 - configurations 732
 - copy operations 737–8
 - creation operations 723–9, 730–6, 749–51
 - definition 717
 - deletion operations 718, 733–8
 - detachment operations 740–1
 - errors 725–7, 750–3
 - events 722–3
 - indexes 718, 734–53
 - insertion operations 718, 726–9, 732–7, 740, 741–4
 - multiple databases 721–3
 - naming conventions 731–2
 - open operations 731–2, 739
 - placeholders 742–3
 - primary databases 739
 - read operations 731, 744–9
 - read–write permissions 738–9
 - retrieval statements 744–9
 - returned data 741–9
 - statements 740–9
 - stored procedures 722–3
 - triggers 723
 - update operations 718, 733–7
- DataL 706
- DataNewL 684–94
- dates, dialogs 462
- DBMS 31–2, 231, 584–5, 715–53
 - see also* RDBMS
- deactivation processes, views 397, 398, 407–11, 509–11, 521, 524–5
- _DEBUG 46, 72–3, 364, 411, 602, 647–8
- debug keys, special effects 521, 523–4, 568–9
- debugging 11, 14–15, 46, 72–3, 283–309, 364–5, 415–16, 427–8, 521, 523–4, 602, 647–8
- breakpoints 289, 290
- Carbide.c++ IDE 14–15, 289–91, 294–5, 302, 306, 309
- CodeWarrior 289, 290–2, 294–5, 307, 309
- concepts 11, 14–15, 46, 72–3, 283–309, 364–5, 427–8, 523–4, 602, 647–8
- d_exc... 301–4
- drawing keys 293
- emulator 14–15, 283–4, 289–309, 523–4
- heap-management macros 303–5
- HookLogger 305–6
- keys 292–4, 521, 523–4, 568–9
- logs 294, 296–9
- memory tests 302–6
- miscellaneous tools 308–9
- mobile phones 306–8
- on-target debugging 306–8
- optimization issues 306–7
- resource allocation keys 292–3
- SymScan 308–9
- window server logging keys 293–4
- decoding 584, 651, 653, 676–7, 682–91, 706
 - basics 685–8
 - buffered decoding 691
 - classes 683–5
 - concepts 682–91

- decoding (*continued*)
 - options 684–5
 - progressive decoding 688–91
 - subclasses 685
- default heap
 - see also* heap
 - concepts 62–5
- default views 396, 398, 421
 - see also* views
- default_data 597–600
- Deferred Function Calls (DFCs)
 - 22, 33–4
- define 45, 254–7, 371, 382–3, 552–3
- Delete 63–5, 76–8, 90, 100–3, 114, 121–4, 139, 143, 207, 254–7, 323, 336–7, 358–9, 738
- DELETE FROM 737
- DeleteContactL 461–2
- DeleteMenuItem 418–19
- deletion operations, databases
 - 718, 733–8
- DenseFont 489–92
- Deque 175, 187
- derived classes 54–5, 69–70, 161, 490, 492–3, 495
- Des 95, 97–8, 101, 132–4, 145, 147
- descent, drawing basics 485–6
- descriptors
 - see also* HBufC; RBuf; TBuf...; TDes...; TPtr...
 - anatomy 91–3
 - APIs 109–10
 - binary data 87, 109–10, 122–3, 124
 - boundaries 93, 137
 - collation processes 138–9
 - concepts 9, 21, 43, 87–149, 309, 586, 640–1, 662–3
 - conversions 122–8
 - correct usage 128–37, 149
 - definition 87
 - efficiencies 89–90
 - externalization 123–4, 218–19
 - folding processes 138
 - formatting 135, 140
 - hierarchical diagram 88
 - internal structure 91–3
 - internalization 123–4, 218–19
 - literals 88–91, 93
 - manipulation 138–49
 - memory layouts 91–3
 - methods 110–20, 135, 138–49
 - naming conventions 138–9
 - narrow descriptors 108–9, 124–8
 - neutral descriptors 108–9
 - number conversions 122–3
 - operations 121–8, 141, 144, 146–7, 218–19
 - parameters 110–14, 128–9
 - preview example 90–1
 - returns from methods 114–20
 - string conversions 124–8
 - substring methods 140
 - text console class 121–2
 - types 88–91, 110–20
 - uses 87
 - wide descriptors 108–9, 124–8
- design issues
 - object-oriented approaches 59–60
 - plug-ins 595–6
 - Symbian OS 17–40, 54, 59–60, 64, 558, 564–5, 577–8
- DestroyedImplementation 595–6
- destructors, concepts 35–6, 43–5, 52, 62–70, 76–7, 79–80, 159–67, 188–9, 196–7, 252–4, 323, 336–7, 355–6, 395–6, 425, 601
- Detach 215–16, 740–1
- detachment operations, SQL 740–1
- DevCertRequest 275–6
- Developer Certificates 275–7
- Developer Network, Symbian OS 756–8
- developer resources 106, 308–9, 755–9
- developer training, Symbian OS 757–8
- developers, emulator 2, 267–9, 277
- Device 46
- device drivers
 - concepts 18–20, 26–7, 205
 - power management 26–7
- Device Management 263–4
- device-independent graphics 529–56
- devices 14
- DevSound 652, 655, 666–72, 676–8
- DevVideo 652
- d_exc... 301–4
- DFCs (Deferred Function Calls) 22, 33–4
- dial-up networking 37, 621–2
- DIALOG 370–1, 459–60, 464–9, 472–5, 478–9
- dialogs 72, 321–2, 359, 396, 423–4, 427–8, 457–80, 503–4, 505–9, 522–3, 559–82
 - alerts 460–2
 - APIs 472–5
 - classes 458–60, 462–3, 476–7
 - commands 321–2, 558
 - complex dialogs 462–3, 505–6
 - concepts 457–80, 503–4, 505–9, 559–82
 - controls 423–4, 427–8, 462–75, 503–4, 505–9, 559–82
 - creation 459–60, 464–6
 - custom controls 477–80
 - definition 457–8
 - error handling 72
 - examples 459–80
 - focus 462–80
 - framework functions 474–5
 - launching 459–60, 464–6
 - library functions 475–6
 - modal aspects 457, 567
 - modelless aspects 457

- multi-page dialogs 462–4, 466–72
- noughts-and-crosses application 464–80
- query dialogs 461–2
- resource definition 459–60, 464–6, 472–80
- S60 457–80
- simple dialogs 459–62
- single-page dialogs 459–62, 464–6
- standard dialogs 460–2
- state changes 463, 474–5, 561, 564–70
- stock controls 475–7
- UIQ 457–80, 568–9, 581
- waiting/non-waiting aspects 457–8
- Digital Rights Management 263–4
- dimmed/invisible controls 409–11, 454–5, 520–1, 561, 563–4
 - see also controls
- direct memory access (DMA) 19–20
- direct screen access viewfinders 702–3
- directories
 - see also TEntry
 - concepts 209–10, 267–8, 379, 383–4
 - data caging 29, 264, 267–8
 - emulator 285–6
 - files 205, 207–8, 209–10, 267–8
 - timestamps 210
- DiscardBrushPattern 489–92
- DiscardFont 482–6, 489–92
- DiskAccess 265
- DiskAdmin capabilities 276, 280
- DismountFileSystem 208
- DisplayMode 547–9
- displays
 - see also drawing
 - graphics for display 481–556
 - ICL 683–4, 694–9
 - viewfinders 702–3
- distribution preparations, applications 268–72
 - DLG_ITEM 472–5
 - DLG_LINE 370–1, 470–8
 - DLLs see dynamically linked libraries
 - DMA (direct memory access) 19–20
 - DoActivateL 409
 - DoCancel 159–67, 171, 176–81, 184, 188, 191–6, 199–200
- document files
 - see also files
 - concepts 204–5, 334–5
- documents 312–14, 320–1, 333, 334–5, 532–56
 - see also CEikDocument
- DoDeactivateL 398, 411
- DoLaunchRead 613
- DOUBLE 368
- double deletions 64–5
- Draw... 75–6, 347–51, 404–6, 427–9, 444–55, 482–505, 520–1, 541, 544–5, 557
- DrawableWindow 452–5
- DrawBetweenRects 500–2, 541–2
- DrawBitmap 491–2, 547–9
- DrawBy 489–92
- DrawComponents 512–14
- DrawComps 350, 427–9
- DrawDeferred 351, 446–55, 501–2, 520–1, 528
- DrawEllipse 346, 450–1, 491–2
- drawing 75–6, 293, 312–13, 345–51, 356–64, 404–6, 409–11, 424–55, 481–556
 - see also graphic...
 - application-initiated drawing 446–7, 493–5, 502
 - basics 482–92
 - bitmaps 491–2, 509–10, 546–9
 - brushes 346, 448–51, 484–92, 500–2
 - CCoeControl summary 514–21
- CGraphicsContext 446, 450–1, 482–4, 486–92, 495–8, 509–14, 531–56
- clipping region 490, 497–8, 511, 514
- complexity issues 509–14
- concepts 424, 427–9, 438, 444–51, 481–556
- controls 424, 427–9, 438, 444–51, 482–6, 492–8, 499–502, 515–21
- coordinate system 487–92, 503–4, 514
- current position 489
- device/size-independent graphics 529–56
- filled shapes 491–2
- flicker-free drawing 514, 527–9, 546–9
- fonts 482–92, 516–17
- functions 490–2, 520–1
- leave rules 498
- lines 346, 449–51, 476, 484–92
- origin 489–90
- pens 346, 448–51, 484–92
- points 485–92
- rectangles 345–6, 449–51, 482–98, 499–502, 506–9, 513–14, 515–21, 531–56
- redrawn windows 445–51, 492–8, 501–4, 511–14, 526–9, 531–2, 568, 577–82
- region-related classes 487–92
- screen-sharing processes 502–14
- special effects 521–6
- specialized justification settings 490
- text 482–6, 492
- vertical justification 485–6
- window server 445–51, 496–8, 503–14, 526–9
- drawing keys, emulator debugging 293
- DrawInRect 536–8, 540–1, 545
- DrawLine 346, 449–51, 490–2

- DrawNow 364, 409–11, 438, 446–7, 452–5, 496–8, 500–2, 514, 520–1, 525, 527, 696–7
- DrawOneTile 500–2, 510–14
- DrawOneTileNow 502–14
- DrawPie 491–2
- DrawPolygon 491–2
- DrawPolyLine 489–92
- DrawRect 345–6, 449–51, 482–6, 497, 499–502, 536–7, 541–2
- DrawRoundRect 491
- DrawSymbol 345–51, 499–502
- DrawText 482–6, 490, 492, 540–2
- DrawTwoTiles 514
- DrawUtils 448–51, 500–2, 541
- Drive 207–8
- drives 205–8, 267–8, 284–6, 295
 - see also* c...; e...; z...
 - emulator 284–6, 295
 - file system 205–8, 267–8
 - types 284–6
- DRM 265, 656, 682
- DROP 733–4
- DServer 248–50
- DSession 248
- DThread 167–71
- DTMF strings 677–8
- durable RDBMS concepts 720–1
- dynamic objects 62–86
 - see also* heap
- dynamically linked libraries (DLLs)
 - concepts 12, 28–9, 37–9, 49, 52–6, 61–2, 284–6, 292, 302, 523, 576, 583–4, 585–6, 587–604, 729–31
 - naming conventions 50
 - plug-ins 28, 583–4, 587–604
 - Symbian SQL 729–31
 - writable static data optimization 29, 44
- DynInitMenuPanel 338–9, 353–5, 362–4, 415, 418–19
- e drive 18
- E (enumerated constant) prefixes 45, 66–7
- E32 38, 182, 188
 - see also* user library
- e32base.h 7–11, 38, 77, 81, 90, 121, 173
- e32cmn.h 168
- e32cons.h 7–11, 90, 121
- e32def.h 38, 41, 42, 53–4, 109, 136
- e32des8.h 109
- e32des16.h 109
- e32keys.h 122, 433
- E32Main 7–11, 90, 121, 319
- e32std.h 38, 41, 74, 108, 109, 487–9
- EActive... 173, 176–81
- EAllowGeneratedMask 684–5
- EAlphaChannel 687–8
- EAudioEvent 711
- EButton... 439–42
- ECAM API 699–706
 - see also* camera...
- ECDRV 611
 - see also* emulator
- EColor... 554–5
- EColor16MA 526
- ECOM
 - see also* plug-ins; system services
 - abstraction layers 594
 - concepts 583–4, 587, 593–604, 606–9, 651–6, 662–6, 678–82
 - controller plug-ins 678–82
 - ICL 584, 651, 682–99
 - low-level plug-in code 598–9
 - references 601–3
 - resolver uses 597, 599–601
 - resource files 596–600, 678–9
 - step-by-step summary 600–1
 - uses 593–4, 602–4, 606–9, 651–6, 662–6, 678–82
 - writing 596–602
- ECOMM 611–12
 - see also* LDDs
- ECUART 612–13, 614
- EDrag 439–42
- EDrawNow 438
- EDWINS 462, 465, 471
- EEventInteractionRefused 443–4
- EEventKey... 433–8
- EEventPointerBufferReadyL 574
- EEventPrepareFocus-Transition 443–4, 463–4
- EEventRequestCancel 442–4
- EEventRequestExit 442–4
- EEventRequestFocus 443–4
- EEventStateChanged 442–4, 463, 566
- EFalse 42–3
- efficiencies
 - context switches 24
 - descriptors 89–90
 - drawing controls 448
 - lodger controls 504–9, 514
- EFileRead 211–13
- EFileReadAsyncAll 212–13
- EFileShare... 211–13
- EFileStream 211–13
- EFileStreamText 211–13
- EFileWrite 211–13
- EFormat... 704–6
- efsrv 38
 - see also* file server
- EGray... 554–5, 687–8
- EIK_APP_INFO 328, 352, 361–2
- eikdialg.hrh 473
- eikfctry.h 479–80
- eikon.rh 327, 370, 384
- eikon.rsg 327, 384
- EikStart::RunApplication 319
- EImageCaptureSupported 700–6
- EKA2, concepts 19–21, 27, 244–5
- EKeyUp 436
- EKeyWasConsumed 434–8
- EKeyWasNotConsumed 434–8
- ELeave 63–4, 76–7, 83–5, 344, 404, 480
- ELF (Executable & Link Format) 27–9

- emails 269, 278, 583–4, 605, 632–49
 - concepts 638–9
 - MTMs 638–9
 - SMS 639
- embedded stores 231–4
 - see also* stores
- EMMFStatePlaying 677–8
- EMMFStateRecording 677–8
- EMove 439–42
- EMsvEntries... 635–7
- emulator
 - see also* epoc...
 - application launcher 2
 - communications 287–9
 - concepts 1–15, 267–9, 277, 283–309, 376, 379, 523–4, 610–12, 637
 - configuration 285–6, 294–5, 306
 - debugging 14–15, 283–4, 289–309, 523–4
 - developers 2, 267–9, 277
 - directories 285–6
 - drives 284–6, 295
 - ECDRV 611
 - eshell 300–1
 - GUI style 2–7
 - heap-management macros 303–5
 - HookLogger 305–6
 - keys 286–7, 292–4, 523–4
 - launching 2
 - logs 294, 296–9
 - memory tests 302–6
 - menus 3–6
 - on-target debugging contrasts 306–8
 - Platform Security 294, 297–8
 - registration files 379
 - restrictions 283–4
 - S60 2, 4–5, 14, 15, 283–301, 637
 - screen layouts 3–5, 376
 - settings 294–5
 - start-up 284–6
 - testing certificates 295–6
 - time factors 284, 308
 - UIQ 2–6, 13–14, 283–304
 - usage 1–2, 267–9, 277, 283–309
 - EnableBackup 453–5
 - EnableRedrawStore 529
 - encoding 584, 651, 653, 676–7, 682–3, 685, 691–4
 - basics 692–4
 - classes 691–4
 - concepts 682–3, 685, 691–4
 - configuration 693–4
 - query facilities 692
 - END 374–5, 379
 - end-user document files
 - see also* files
 - concepts 204–5
 - EndRedraw 501–2, 511–14, 527–8, 580–1
 - engine classes 333, 336–44, 363–4
 - ENoDrawNow 438
 - Enter key 6
 - Enterprise-grade data handling 264
 - entry points, application framework 318–19
 - ENUM 369, 371, 375–6, 478
 - see also* resource files
 - ENumberOfControls 508–9
 - enumerations 45, 66–7, 369, 371, 375–6, 508–9
 - EOption... 684–5, 695
 - EOrientation... 700–6
 - epoc... 2, 285–6, 294, 307
 - EPOC16 (SIBO) 87, 241–2, 311–12
 - epoc32 10–14, 41, 270–1, 284–6, 295–6, 300, 370, 383–4
 - see also* emulator
 - epocstacksize 65
 - epocwind.out 294, 296–9
 - EPriority... 172–4, 196–200
 - EQikCommandTypeScreen 415–17
 - ERequestPending 169–70, 176–81
 - ERequestWrite 250–4
 - ERgb... 554–5
 - Error 163–4
 - error handling
 - see also* exception...; KErr...
 - active objects 159–67, 177–81, 182–90, 191–200
 - concepts 8–10, 34–6, 61–2, 70–5, 159–67, 177–81
 - flat-spin cycles 189
 - panics 7–11, 65, 72–5, 130, 134–7, 182–8, 252, 296, 298, 301–2, 307, 518–19
 - programming errors 72–3, 182–7
 - RunL 163–4, 171, 183
 - scope 70–3
 - trap harness 8–10, 34–6, 73–5, 77, 80, 85, 177–8, 189, 251–2
 - eshell 300–1
 - ESock
 - see also* sockets server
 - concepts 28, 39, 288, 604
 - ESql... 746–7
 - estor.lib 38, 124
 - ESwitchOn 439–42
 - ETel server 29–30, 39, 603–4
 - Ethernet 289
 - ETrue 42–3
 - ETuner... 707–13
 - EUART 611
 - euser.dll 20–1, 28, 38, 167, 182, 305–6
 - see also* user library
 - euser.lib 11
 - event handling
 - active objects 31–4, 153–201, 577–82
 - concepts 31–4, 153–67, 577–82
 - examples 33–4, 158–67
 - events 26–7, 30–4, 151–201, 557–82
 - see also* active objects
 - concepts 151–201, 557–82, 722–3
 - databases 722–3
 - overview 152–3
 - phases 153–6
 - types 152–3, 557–8

- EVideoCaptureSupported 700–6
- EViewFinder... 702–6
- EWindowBackupAreaBehind 453–5
- EWindowBackupFullScreen 453–5
- exception handling
 - see also* error...
 - concepts 34–6, 46, 61–2, 70–5, 76–7
- EXE files
 - see also* programs
 - concepts 27–9, 65, 267–72, 284–6, 378–9, 384–5, 729–31
- EXEC 732–49
- Executable & Link Format (ELF) 27–9
- executable images, concepts 27–9
- executables
 - see also* programs
 - concepts 27–9, 267–8
 - SID (Secure ID) 267–8, 271
- ExecuteLD 459–60, 465–6
- execution-in-place principles 18
- executive calls, concepts 21
- Exif files 584, 694, 697–8, 704–6
- Exit 354–5
- Expand 125, 127–8, 143
- EXPORT_C 28, 46, 53–5, 60, 163–4, 167–71, 388–9, 511–14, 525, 542–3, 613–14
- exports 21, 28, 46, 53–4, 167–71
- extensibility principles 583–604
- ExtensionInterface 697–8
- extensions, files 205–6
- external representation, streams 216–17
- externalization
 - descriptors 123–4, 218–19
 - streams 216–17, 218–24, 226, 235–7, 335–44
- ExternalizeL 218–24, 235–6, 335–44, 539
- EZLIB
 - see also* EPOC; system services
- concepts 585–6
- F1 key 287
- F5 key 290–1
- F9 key 287
- F10 key 287
- F32 29, 38–9, 205
 - see also* file server
- f32file.h 205
- FadeBehind 454
- fading 455, 580–1
- FALSE 42
- fbs.h 547
- FEPs (front-end processors) 432–3, 438, 568–70, 602–3
- Fibonacci program example, active objects/long-running tasks 195–200
- File 215
- file manager applications 204–5
- file server
 - see also* F32; RFs; servers; system services
 - concepts 29–30, 31–2, 38–9, 205–40, 516–17, 583, 683–4
 - overheads 207
 - sessions 207–9, 211
- file system 19–20, 29, 38, 71–2, 203–40, 253–4, 730
 - concepts 203–40, 730
 - database files 730
 - drives 205–8, 267–8
 - services 205–16
 - VFAT file system 205–6, 210
- FileNewL 684–94
- files
 - see also* databases; documents; resource...; streams
 - access modes 211–16
 - applications 203–4
 - binary files 211–13
 - concepts 19–20, 29, 38, 71–2, 203–40, 253–4, 715–16
 - data caging 29, 264, 267–8
 - directories 205, 207–8, 209–10, 267–8
 - error handling 71–2
 - extensions 205–6
 - modes 211–14
 - names 205–7
 - open operations 211–13
 - pathnames 205–7
 - Platform Security 29, 217–18, 264
 - read operations 211–14
 - sectors 215–16
 - shared access 211–16
 - specifications 205–7
 - stores 224–34, 334–44
 - text files 211–13, 592–3
 - types 204–5, 715–16
 - write operations 211–14
- FileSpec 206
- Fill 143
- filled shapes 491–2
- FillZ 143
- final 54
- FinalClose 594, 601–4
- Find 140
- FindC 138, 140
- FindF 110, 140
- FindProtocol 617–18
- FixedSequenceCount 671–2
- FixedSequenceName 671–2
- flat files, RDBMS 715–16
- flat-spin cycles, error handling 189
- flicker-free drawing 514, 527–9, 546–9
- flipped images 694–9
- floating windows 396
 - see also* dialogs
- floating-point data 41, 307–8, 368, 726
- Flush 516, 522–3
 - see also* animation
- FM radio 706–13
 - see also* radio
- focus 350, 357–60, 364–6, 427–9, 435–8, 449–51, 462–80, 557–82
- FocusChanged 570–1
- FocusedControl 463–4
- Fold 143
- folding processes, descriptors 138
- FONT 550

- fonts 482–92, 509–14, 516–17, 534–56
 - bitmaps 549–50
 - concepts 482–92, 516–17, 534–56
 - drawing basics 482–92
 - memory leaks 486
 - scalable fonts 549–50
- FontShell 486–7, 549–50
- for statements 47–9, 404–5, 413–14, 428, 745
- FORM 470–2, 490, 492
- Format 140, 143, 413–14
- formats, media formats 653, 666–7, 676–80, 688–94, 703–6
- formatting, descriptors 135, 140
- forms, programs 27–8
- Forum Nokia 757
- FrameCount 688, 691
- FrameInfo 688, 691
- frames, images 687–9
- framework APIs, concepts 51, 60
- framework basics, Symbian OS 30–6, 51–2, 570
- framework functions
 - concepts 51–2, 474–5
 - dialogs 474–5
- FreeDeviceDrivers 613–14
- FreeLogicalDevice 611–12, 613–14
- FreePhysicalDevice 611–12, 614
- friend 169
- front-end processors (FEPs) 432–3, 438, 568–70, 602–3
- fscanf 140
- FsSession 516
- Function 250–1
- functions 31–3, 46, 49–55
 - concepts 46, 49–55
 - drawing 490–2, 520–1
 - naming conventions 46
 - prototypes 49–51
 - types 51–2
 - virtual functions 31–3, 49–50, 54–5, 57–60, 163–4, 408–11, 490–2, 508–9, 557, 588–9
- fundamental data types, Symbian OS 41–3
- GameWonBy 342–4
- gc 448–51, 482–6, 493–5, 499–502, 511, 524–5, 541
 - see also graphics context
- GDI (graphical device interface) 38, 486, 489, 490, 514, 531, 534–5, 538–9, 543–53
- gdi.h 486, 490, 538–9, 541–2, 553
- general-purpose controls 562–3, 566–7
- Get 255–7
- GetAttachmentFileL 645–6
- GetByName 619–20
- GetCapabilities 707–13
- Getch 121–2
- GetChannelRange 708–9
- GetDir 210
- GetFileTypesL 692–3
- GetFontById 550
- GetFrequencyBandRange 708–9
- GetImageSubTypesL 692–3
- GetImageTypesL 692–3
- GetInvalidRegion 580–1
- GetNearestFontInPixels 539, 550
- GetNearestFontInTwips 539, 549–50
- GetSupportedAudioTypesL 675
- GetSupportedBitRatesL 665–6
- GetSupportedInputDataTypesL 676–7
- GetSupportedOutputDataTypesL 676–7
- GetSupportedVideoTypesL 675
- getter functions 46
- GetTunerAudioRecorderUtilityL 711–12
- GetZoom 543
- Gif files 556, 587, 694
- GLDEF_C 7–11, 90, 338–9
- global variables 27–8, 45, 243–5, 254–7
- GLREF_C 338–9
- Google chat 588
- grab/capture contrasts, pointers 440–2, 573
- graphical applications 315–30, 481–556
 - device/size-independent graphics 529–56
 - ‘Hello World’ example 315–30, 482–6, 506–7, 531–56
 - noughts-and-crosses application 331–66, 374–89, 398–420, 426–55, 464–80, 497, 605–49
- graphical device interface (GDI) 38, 486, 489, 490, 514, 531, 534–5, 538–9, 543–53
- graphical user interfaces (GUIs) 1–7, 32–4, 37–9, 76–7, 151–2, 207–8, 311–30, 331–66, 391–421, 423–55, 529–56
 - see also application framework; S60; UIQ
- active objects 32–4, 158
- cleanup stack 76–7
- concepts 1–7, 32–4, 37–9, 76–7, 151–2, 207, 311–30, 331–66, 481–556, 557–82
- CONE 38–9, 86, 172–3, 181, 193–4, 207–8, 311–13, 383, 483–4, 496–8, 503–21
- controls 324–7, 400–6, 412–14, 423–55, 481–556, 557–82
- device-independent graphics 529–56
- dialogs 321–2, 359, 396, 423–4, 427–8, 457–80
- emulator 2–7
- interaction graphics 347–9, 440–2, 463, 557–82
- RFs 207–8
- underlying features 2–7

- graphics context 347–9, 405–6, 444–55, 481–556
 - CGraphicsContext 446, 450–1, 482–4, 486–92, 495–8, 509–14, 531–56
 - classes 446, 450–1, 482–92
 - concepts 446, 450–1, 482–92, 509–21, 524–5, 531–56
 - coordinate system 487–92, 503–4, 514
 - default settings 484, 490
 - functions 490–2, 520–1
 - getting processes 483–4
 - regions 487–92, 496–8, 510–14
 - return values 490
 - setting-up processes 489–92
- graphics for display
 - see also* drawing
 - concepts 481–556
- graphics for interaction *see* interaction graphics
- GraphicsShell 546
- Grow 487, 536–7
- GUIs *see* graphical user interfaces

- Handle 675
- HandleAcceptCompleteL 623–4, 627
- HandleCommandL 162–3, 166–7, 237, 321–4, 335–44, 351, 354–5, 363–5, 407–8, 415, 418–19
- HandleConnectComplete 632
- HandleControlEventsL 350–1, 362–4, 427–9, 442–4, 565–6
- HandleControlStateChangeL 463, 474–5, 564–6
- HandleEvent 154–5
- HandleFibonacci-
 - CalculatorResetL 195–200
- HandleFibonacciResultL 195–200
- HandleInteractionRefused 463
- HandleKeyL 74–5
- HandlePointerBufferReadyL 574–5
- HandlePointerEventL 347–9, 440–2, 557–8, 571–2, 575
 - see also* interaction...
- HandleReceiveCompleteL 627–8
- HandleRedrawEvent 446–7
- HandleSessionEventL 635–7
- HandleWsEvent 440–2, 567–8, 575
- handwriting recognition 433, 574
- hardware 17–18, 21–2, 26–7, 34, 276, 280, 609, 652–3, 701–2
 - see also* CPUs; device...; I/O...; memory
 - device plug-ins 652–3
 - power management 21–2, 26–7, 34, 276, 280, 609, 701–2
- has-a class relationships 57, 59, 64, 67, 160, 494, 516–17
- HasPrefixC 141
- HBufC 43, 88–9, 99–102, 107–9, 111–12, 119–20, 123–4, 130, 132–5, 138, 145–6, 149, 219, 222–4, 388, 692–3, 706
 - see also* heap-based descriptors
 - Alloc usage 132–3
 - concepts 88–9, 99–102, 107–9, 111–12, 119–20, 123–4, 130, 132–5, 138, 145–6, 149, 219, 222–4, 388, 692–3
 - correct usage 130, 132–5, 149
 - Des usage 132–4
 - manipulating-data methods 145–6
 - method parameters 111–12, 114, 119–20
 - other descriptors 101–2, 132–3
 - RBuf migration 107–8, 148
 - ReAlloc... usage 133–5
 - size changes 102
- HEADER 374–5
- header files 52–5, 136, 224, 313–14
 - APIs 52–5
 - literals 136
 - S60 313–14
 - streams 224
 - UIQ 313–14
- heap
 - see also* CBase
 - concepts 8–9, 21, 24–5, 34–6, 62–86, 95, 130, 145–7, 303–5
 - creation 24–5, 62–70
 - definition 62–3
 - exception handling 34–6, 70–5
 - heap-management macros 303–5
 - large stack descriptors 130
 - heap marking 8–9, 34–6, 74–5
 - see also* exception handling
 - heap-based descriptors 43, 88–149
 - see also* descriptors; HBufC
 - concepts 88–9, 99–108, 119–20, 138, 145–7
 - ‘Hello World’ example 1–15, 90–9, 111–14, 121–37, 158–67, 190–4, 219, 297–8, 315–30, 482–6, 506–7, 531–56
 - device/size-independent graphics 531–56
 - S60 315–30
 - UIQ 315–30
- HelloS60.cpp 318–19
- HelloS60.hrh 327
- HelloS60.rls 327
- HelloS60.rss 327, 329
- hellotext 7–11
- HelloUIQApplication.cpp 318–19
- HelloUIQ.hrh 327
- HelloUIQ.rls 327
- help icons 562
- high-resolution pointer events 573–4
- history views 400–14, 416–20
 - see also* views

- HitSquareL 340–4, 351
- HookLogger 305–6
- HorizontalPixelsToTwips 542–3
- HorizontalTwipsToPixels 542–5
- HRH files 320, 327–8, 353–5, 371, 384–5, 470–3, 478
- HTML 556
- HTTP 586, 602

- i (member variable) prefixes 36, 45, 68
- I/O devices, background 18, 33–4
- IBM 434, 552, 555
- ICL *see* Image Conversion Library
- icons 263, 329–30, 372–9, 385, 491–2, 546–9, 562
 - color depth 375–6
 - concepts 372–7, 491–2, 546–9
 - masks 373–7
 - sizes 373–4, 385
- ICQ 588
- IDEs (Integrated Development Environments), resources 1, 11, 14–15, 91, 268–9, 306, 383–4, 755–6
- idioms, Symbian OS 17–18, 30–40
- Idle 173–4, 190–200
- IdOfControlWithFocus 435–8
- IdOfFocusControl 359–60, 365–6, 427–9, 435–8, 463–4, 474–5
- if statements 47–9, 77–8, 107–9, 176, 187–90, 198, 209–11, 299, 340, 343–4, 363, 365–6, 405–6, 413–14, 418–19, 429, 447–8, 452, 501–2, 511
- ifdef 379
- IgnoreEventsUntilNext-PointerUp 441–2
- Image Conversion Library (ICL) 584, 651, 682–99
 - decoding 584, 651, 682–91
 - displays 683–4, 694–9
 - encoding 584, 651, 682–3, 685, 691–4
 - overview 682–3
 - rotated images 694–9
 - scaled images 694–9
 - transformations 694–9
 - uses 682–3
- ImageBufferReady 705–6
- imagecodecdata.h 685
- ImageConversion 683–91
- ImageReady 706
- images 206, 374, 556, 584, 587, 651–713
 - see also* multimedia...
 - animation 694, 696–8
 - concepts 682–99
 - decoding 584, 651, 682–91, 706
 - displays 683–4, 694–9
 - encoding 584, 651, 682–3, 685, 691–4
 - flipped images 694–9
 - frames 687–9
 - progressive decoding 688–91
 - rotated images 694–9
 - scaled images 694–9
 - thumbnails 695
 - transformations 694–9
 - type-determination methods 688
- images folders 374
- IMAP4 637–9
 - see also* emails
- ImApp 590–1
- IMEIs 275–6
- IMLib.dll 590–1
- IMPLEMENTATION_INFO 596–8
- IMPORT_C 46, 49–50, 53–4, 60, 110, 168–71, 223, 248–9, 255, 397, 408, 495, 517–21, 538–9, 565–6, 571, 608, 638, 677, 749–50
- include 7–11, 90, 121, 299, 327–8, 370, 374–6, 384, 599
- indentation style rules, coding conventions 46–9
- Index 518–19

- indexes
 - resource files 385–7
 - SQL 718, 734–53
- infoSync World community website 758–9
- infrared 18, 37, 39, 269, 278–9, 287–8, 583–4, 603, 605–11, 614–15, 616–32
 - see also* IrDA
- concepts 605–11, 614–15, 616–32
- serial communications 611, 614–15
- socket-based communications 616–32
- inheritance, object-oriented approaches 58–9, 61–2, 69–70, 606
- INI files 237–9, 294–5, 400
- InitComponentArrayL 357–64
- initialization 153–5
- initialization issues 136–7, 153–4
- InitializeComplete 677–8
- InitializeL 677, 710–11
- inline 49, 55–6, 114, 168–9, 248–9
- input streaming, audio 670–1
- InputCapabilities 437
- Insert 139, 143
- insertion operations, SQL 718, 726–9, 732–7, 740, 741–4
- installation 263–4, 268–72, 278–9, 296
 - see also* SIS files
- Installation File Generator 269–70
- Installation File Signer 269–70
- instant messaging 587–604
- integer data types 41–3, 66–7, 368, 726
- Integrated Development Environments (IDEs), resources 1, 11, 14–15, 91, 268–9, 306, 383–4, 755–6
- interaction graphics
 - abstractions 561–7
 - big issues 557–8

- interaction graphics (*continued*)
 - concepts 347–9, 440–2, 463, 557–82
 - control stack 567–9, 576–82
 - key-event processing 567–71
 - pointer-event processing 347–9, 440–2, 557–8, 571–6
 - programmer requirements 561–2
 - user requirements 559–61
 - window server/control environment APIs 576–82
- interface principles, object-oriented approaches 57–60, 196
- INTERFACE_INFO 596–8
- internalization
 - descriptors 123–4, 218–19
 - streams 216–17, 218–24, 236, 335–44
- InternalizeL 218–24, 236, 335–44, 538–9
- Internet 269, 279, 555–6
 - see also* web...
- interprocess communications (IPCs)
 - see also* client–server...; message queues; publish–subscribe...
 - concepts 25–6, 215, 241–62, 730, 748
 - mechanisms 26, 241–62
- interrupt service routines (ISRs) 33–4
- interrupts, concepts 19–20, 21, 33–4
- Intersection 513–14
- Invalidate 447, 452–5, 501–2, 509–14, 528–9, 580–1
- invisible controls 409–11, 454–5, 520–1, 563–4
 - see also* controls
- IP 37, 39, 287–8, 289
- IPCs *see* interprocess communications
- IRC 588
- IRCOMM 612–13, 614–15, 617
- IrDA 18, 287–8
 - see also* infrared
- irPod 288
- is-a class relationships 57, 59, 160
- IsActivated 521
- IsActive 162–71, 188
- IsAntennaAttached 707–8
- IsBackedUp 452–5
- IsBlank 521
- IsCrossTurn 339–40, 347–8
- IsDimmed 455, 563–4
- IsFocused 438, 449–51, 570–1
- IsHeaderProcessing-Complete 689–91
- IsLoaded 576
- IsNewGame 340
- IsNonFocusing 571
- IsNull 252
- isolated RDBMS concepts 720–1
- IsReadyToDraw 521
- ISRs (interrupt service routines) 33–4
- Jabber protocol 588, 590–8
- Java 9, 54, 73, 484, 585–6, 716, 757
- joining concepts, databases 721–2
- jotter applications, files 203–4
- Jpeg files 206, 556, 584, 587, 685, 693–4, 697–8, 704–6
- Justify 144
- K (constant) prefixes 45
- kernel 18–40, 161–7, 242–62
 - see also* microkernel
 - concepts 18–22, 242–3
 - definition 19
 - EKA2 19–21, 27, 244–5
 - kernel-side/user-side mode 20–3, 25–6, 161–7, 242–62
 - memory model 24–5
 - nanokernel 19–20
 - roles 18–19, 242–3
 - thread concepts 21–4
- kernel-side mode, concepts 19–23, 26, 161–7, 200–1, 242–62
- KErrAlreadyExists 611–12
- KErrArgument 178, 187, 695–6
- KErrCancel 176–81, 188, 669–71
- KErrCorrupt 219
- KErrDied 252–3
- KErrDisconnected 609
- KErrGeneral 178
- KErrNoMemory 104, 693
- KErrNone 35–6, 74, 104–5, 113, 163–4, 170–1, 179, 192, 200, 564, 611–12, 631, 657–8, 675, 687, 690–4, 701, 731–8
- KErrNotFound 211–15, 518–19, 637, 657–8, 693–4
- KErrNotSupported 526, 590–1, 634–5, 657–8, 681, 698
- KErrServerTerminated 181, 253
- KErrUnderflow 669–71, 687, 689–91
- key files 272
- Keyboard 153
- keyboard focus *see* focus
- keyboards 6, 18, 153, 154–5, 286–7, 432–42, 568–9
- KeyEvent 576
- keys 6, 33–4, 286–7, 292–4, 347–8, 352–3, 361–2, 376, 423–9, 432–42, 459–60, 469–72, 523–4, 557–82, 718
- control stack 568–9
- controls 423–9, 432–42, 459–60, 469–72, 523–4, 557–82, 718
- debugging 292–4, 521, 523–4, 568–9
- emulator 286–7, 292–4, 523–4
- shortcut keys 6, 523–4, 568–9

- softkeys 352–3, 361–2, 469–72
- SQL terminology 718
- text cursors 571
- window groups 581–2
- KL2CAPPassiveAutoBind 623
- KMMFEventCategory... 678
- KNullDes... 112, 132
- KNullUid 228, 684–5
- KNumDataLines 400–6, 413–14
- KNumHistoryRecords 400–6
- KRequestPending 169–70, 177–81, 182–3, 184–7, 189, 192–3, 198–200
- KRfcommPassiveAutoBind 623
- KRgb... 552–5
 see also color...
- KSdpAttrIdOffsetService-Name 626–7
- KSqlErr... 751–3
- KStatement 742–3
- KTile... 332–3
- KUIdAppIdentifierStream 236–7
- KUIdAppRegistration-ResourceFile 329–30, 378–9
- KUIdEcamEvent... 702–6
- KUIdMediaTypeAudio 679
- KUIdMediaTypeVideo 674–5, 679
- KUIdMsgType... 633–46
- _L 8, 135–6
- L... (leaving function) suffixes 9–11, 36, 50, 309
- L2CAP 617, 620–32
- LABEL 403
- label controls, views 400–6, 412–14
- LANG 317–18, 379, 381–2
- language systems, API groupings 37
- LanguageBaseAttributeID-List 626–7
- languages
 - collation processes 138–9
 - resource files 379–82
- LaunchRead 613
- layouts, controls 429–32, 519–20
- LBUF 371
- LC descriptor naming convention 138
- LCD displays 514
- LDDs (logical device drivers) 610–15
- leaks, memory 10, 61–2, 64, 75, 99, 112–13, 303–5, 486
- Leave 9–10, 35–6, 63–4, 73–5, 77, 80–1, 85, 187, 636–7
- leave functions 8–9, 35–6, 50, 83–5, 159–67, 187, 190, 221, 251–2, 309, 498
 see also exception handling
 constructors 83
- LeaveIfError 161, 208–11, 213–15, 217–18, 256–7, 428, 461–2, 545, 588, 611–12, 618–19, 634, 686, 694, 707
- LeaveScan 308–9
- LeaveWithInfoMsg 564
- Left 140, 141
- LeftPtr 144
- LegendFont 489–92
- Length 126–8, 137, 139, 141, 142
- LIB files 37–9, 53
- libraries 11–12, 19–21, 27–9, 37–9, 51–3, 304–5, 317–18, 475–6, 583–604, 682–99
 see also dynamically linked
 libraries
 APIs 51, 53, 58, 60
 concepts 27–9
 dialogs 475–6
 functions 51–3
 types 27–8
- LIBRARY 11–12, 304–5, 317–18, 598–9
- line breaks, coding conventions 46–9
- LineChangedL 474–5
- lines
 - see also* pens
 - drawing basics 346, 449–51, 476, 484–92
- LINK 369
- link-by-ordinal constraints 28–9
- Listen 616–32
- ListImplementationsL 597–600
- LIT... 72, 90–109, 111–12, 119–21, 124–8, 133–7, 148, 162, 165–6, 206, 219, 299, 308–9, 407–8, 419–20, 611, 614, 732–4, 741–4
 see also TLitC
- literals
 - concepts 88–91, 93, 135–6, 308–9
 - header files 136
- LLINK 369–70, 386–7, 470–5
- Load 591–2
- LOC files 328–30, 381–2, 384–5
- LOCAL_C 319
- LOCALISABLE_APP_INFO 377
- localisable_resource_file 329–30, 378–9
- localizable files 328–30, 367–8, 377–82
- localizable strings 379–82
- LocalPort 623
- LocalServices capabilities 276, 279, 615, 648–9
- Locate 141
- LocateF 138, 141
- LocateReverse 141
- LocateReverseF 141
- Lock 212
- lodger controls 424, 504–9, 514–15, 516–18
 see also controls
- logical channels 620–32
- logical device drivers (LDDs) 610–15
- LoginL 589–90
- logs, debugging 294, 296–9
- LONG 368–70, 386–9, 473–5
- long-running tasks, active objects 194–200
- Lookup 591–2

- LostConnection 608–9
- Lotus Sametime 588
- LowerCase 144
- LTEXT 368–70, 386–7, 470–5

- M (abstract interface) classes
 - see also* mixins
 - concepts 43–4, 69–70, 81, 558
- macros 28, 46, 48–9, 72–3, 114, 162, 165–71, 359–60, 409, 429–30, 436–8, 647–8, 659, 665–6
 - assert macros 48–9, 72–3, 114, 162, 165–71, 359–60, 409, 429–30, 436–8, 647–8, 659, 665–6
 - naming conventions 46
- MainL 7–10
- MaiscBufferCopied 670–1
- MaiscOpenComplete 670–1
- makekeys 269–72, 275
- MakeLineVisible 476, 563–4
- makesis 269–72
 - see also* installation
- MakeVisible 409–11, 454–5, 520–1
- MakeWholeLineVisible 476
- MAknFadedComponent 458
- malloc 99
- manipulation, descriptors 138–49
- manufacturers, signed applications 274–80
- MaoscBufferCopied 669–71
- MaoscPlayComplete 669–71
- MapcInitComplete 657–66
- MapcPlayComplete 658–66
- maps 44, 531–56
- mask bitmaps 373–7, 682–99
 - see also* bitmaps
- Match 141
- MatchC 141
- MatchF 141
- Math 44
- math functions 21
- MatoPrepareComplete 671–2
- MaxLength 139, 144, 223–4
- MaxSize 144
- MBG files 372–7
- MBluetoothSocketNotifier 623, 627, 632
- MBM files 269–72, 328–9, 372–7, 548–9
 - see also* bitmaps
- MCameraBuffer 703, 705–6
- MCameraObserver... 699–706
- MCoeControlObserver 349–51, 427–9, 442–4, 458, 463, 565–6
- MCoeView 314–30, 392, 397–8, 408–20
- MCoeViewDeactivation-Observer 396
- MContentHandler 586
- MDA (Media Server) 654
- MdaAudioInputStream 656–7, 670–1
- MdaAudioSamplerEditor 661
- MDevSoundObserver 677–8
- MDF (Media Device Framework) 652–713
- Media Device Framework (MDF) 652–713
- media formats 653, 666–7, 676–80, 688–94, 703–6
- Media Server (MDA) 654
- MEikCommandObserver 351
- MEikDialogPageObserver 458
- MEikMenuObserver 44, 351, 566
- Mem 44, 126–8
- memory 10, 18, 19–20, 23–4, 29, 38–9, 62–3, 70–3, 87–8, 104–5, 133–4, 190, 216–40, 302–6, 693
 - see also* RAM; ROM
 - cards 18
 - constraints 18, 29, 71–2, 302–3, 452, 530
 - descriptor layouts 91–3
 - heap-management macros 303–5
 - HookLogger 305–6
 - leaks 10, 61–2, 64, 75, 99, 112–13, 303–5, 486
 - management concepts 19–20, 23–5, 29, 61–86, 99, 237
 - out-of-date locations 133–4
 - out-of-memory errors 62–3, 70–3, 87–8, 104–5, 190, 302–6, 693
 - stores 224–40
 - streams 38–9, 216–24
 - tests 302–6
- memory management unit (MMU) 19–20, 23–4, 29
- memory model
 - see also* chunking
 - concepts 24–5
- menu bars 3–5, 352–5, 568
- MENU_BAR 352–3
- MENU_ITEM 352–3, 380, 386–9, 417–18
- MENU_PANE 380–1, 386–7, 417–18
- menus 3–6, 338–9, 351–5, 361–4, 380–1, 386–9, 407–8, 415, 423–4, 559–60
 - commands 351–5, 361–6, 380–1, 407–8, 414–20
 - controls 423–4
- MENU_TITLE 352–3
- message queues
 - see also* interprocess communications
 - concepts 26, 241, 245–7, 257–9, 261–2
 - creation operations 245–6, 257–8
 - critique 261–2
 - operation types 245–7
 - receiving operations 245–6, 257–9
 - sending operations 245–6, 257–9
 - usage operations 259
- message server, roles 605–6, 632–49
- messaging 203–4, 245–6, 257–9, 583–4, 586–7, 603–4, 605–9, 632–49

- see also* communications; MMS; SMS; system services
- concepts 605–9, 632–49
- emails 638–9
- files 203–4
- receiving messages 635–7, 645–6
- S60 v3 emulator 637
- sending messages 245–6, 257–9, 584, 633–5, 639–44
- transport concepts 605–6
- tree 633
- types 632–3
- messaging-type modules (MTMs) 603–4, 633–46
- metadata, definition 717
- methods, descriptors 110–20, 135, 138–49
- Metrowerks 286, 307
- Metrowerks Target Resident Kernel (MetroTRK) 307
- MFibonacciResultHandler 195–200
- MGraphicsDevice 531–56
- MGraphicsDeviceMap 44, 531–56
- MIclImageDisplayObserver 694–8
- microkernel
 - see also* kernel
 - concepts 19–20, 27, 30–1, 250, 257–8
 - roles 19
 - timers 27
- Microsoft Windows 14, 305–6, 307–8, 367–8, 484, 548–9, 585
- bitmaps 372–3, 548–9
- Explorer 14
- HookLogger 305–6
- resource compiler 367–8
- Mid 140, 141
- middleware APIs 37–9
- MidTPtr 144
- MIF files 377
- MiidoImageReady 696–8
- MIME files 587–8, 598–9, 642–6, 679–80, 684–5, 689–92
- mixins 44, 58, 60, 69–70, 195–200, 558, 654, 667, 671, 677
 - see also* interface. . .; M (abstract interface) classes
- MkDir 207
- MkDirAll 207
- MMdaAudioOutputStream-Callback 667–71
- MMdaAudioPlayerCallback 657–66
- MMdaAudioToneObserver 671–2
- MMdaObjectStateChangeObserver 661–6
- MMF (Multimedia Framework) 584, 602, 651–713
 - basic structure 651–2
 - concepts 651–713
- mmfcontrollerframe-work.lib 662–6
- MMMFAudioController-CustomCommand-Implementor 680–2
- MMMFAudioPlayController-CustomCommand-Implementor 680–2
- MMMFAudioPlayDevice-CustomCommand-Implementor 680–2
- MMMFVideoRecord-Controller-CustomCommand-Implementor 680–2
- MMMFResourceNotification-CustomCommand-Implementor 680–2
- MMMFVideoController-CustomCommand-Implementor 680–2
- MMMFVideoPlayController-CustomCommand-Implementor 680–2
- MMMFVideoRecord-Controller-CustomCommand-Implementor 680–2
- MMMRdsDataObserver 712
- MMMRdsFrameReceiver 712
- MMMSignalStrengthObserver 708
- MMMTunerAudioPlayer-Observer 710–11
- MMMTunerAudioRecorder-Observer 711–12
- MMMTunerChangeObserver 708
- MMMTunerObserver 706–13
- MMMTunerStereoObserver 708
- MMP files
 - see also* project definition files
 - concepts 11–12, 14–15, 65, 124, 254, 266–7, 304–5, 307, 317–18, 374–7, 381–2, 591–2, 598–9, 730–1
- MMS (Multimedia Messaging Service) 583–4, 605–9, 632–49
 - concepts 639–46
 - receiving 645–6
 - sending 639–44
- MmsUtils 640–6
- MMSvAttachmentManager 645–6
- MMSvSessionObserver 635–7
- MMU (memory management unit) 19–20, 23–4, 29
- MNG files 584, 694, 697–8
- mobile phones
 - see also* smartphones
 - background 17–18, 26–7
 - constraints 26–7, 29, 71–2, 242, 302–3, 424, 452, 530
 - debugging 306–8
- modal aspects, dialogs 457, 567
- model 312–13, 320–1, 333, 334–5, 451, 493–5, 557
 - see also* CEikDocument;
 - documents
- MVC (Model–View–Controller) 312–13, 320–1, 451, 493–5, 557
- modeless aspects, dialogs 457

- model-view-controller (MVC)
 - 312–30, 451, 493–5, 557
- modems 39
- Modified 207
- monolithic kernel architecture,
 - concepts 19, 26, 31
- MoscoStateChangeEvent
 - 661–6
- MountFilesystem 208
- Move 487–92, 499–502
- MoveBy 489–92
- MoveFocusTo 350, 359–60, 427–9
- MoveTo 489–92
- MP3 files 206, 653, 666–7
- MPageRegionPrinter 544–5
- Mpeg files 587
- MPluginInterface 595
- MPrintProcessObserver 58, 551
- MQikCommandHandler 458
- MQikCommandListener 458
- MQikContainer 458
- MSaveObserver 237
- MSdpAgentNotifier 630–2
- MSdpAttributeValueVisitor
 - 630–2
- MSN 588
- MTapoInitializeComplete
 - 710–11
- MTapoPlayEvent 710–11
- MTMs (messaging-type modules)
 - 603–4, 633–46
- MToTunerEvent 709
- MTransportObserver 608–9, 614–15
- multi-page dialogs 462–4, 466–72
 - see also* dialogs
- multibitmap (MBM) files 269–72, 328–9, 372–7, 548–9
 - see also* bitmaps
- multicasting 259
 - see also* publish–subscribe mechanism
- multilingual applications, resource
 - files 317–18, 381–2
- Multimedia Framework (MMF)
 - 584, 602, 651–713
- API types 653–6
 - basic structure 651–2
 - clips 653–6
 - concepts 651–713
 - controller plug-ins 678–82
 - custom commands 680–2
 - DevSound 652, 655, 666–72, 676–8
 - streaming APIs 654–6
- Multimedia Messaging Service (MMS)
 - 583–4, 605–9, 632–49
 - concepts 639–46
 - receiving 645–6
 - sending 639–44
- multimedia services 276, 280, 584, 587, 602, 651–713
 - see also* audio; camera...; ECOM; images; radio...; system services; video
- Camera API 584, 651, 656, 699–706
 - concepts 584
- Image Conversion Library (ICL)
 - 584, 651, 682–99
- media formats 653, 666–7, 676–80, 688–94, 703–6
 - Tuner API 584, 651, 706–13
- MultimediaDD capabilities 276, 280, 656–7, 682, 709
- multiple databases 721–3
- multiple inheritance 58–9, 61–2, 69–70
 - see also* inheritance
- multiple resource files 382–3, 516–17
 - see also* resource files
- multiple SDKs 14
- multitasking operating systems,
 - concepts 22–4, 30–4, 156–8, 424, 511–14
- multithreading 18–19, 157
- MUTABLE macros 498
- mutexes 241
- MVC (model-view-controller)
 - 312–30, 451, 493–5, 557
- MVideoPlayerUtility-Observer 672–4
- MVideoRecorderUtility-Observer 674–5
- MViewCmdHandler 345, 349–51, 427–9
- MvruoPrepareComplete 675
- MvruoRecordComplete 675
- My-Symbian community website 758
- myCert 272
- MYCUSTOMCONTROL 478–9
- MySQL 716–53
 - see also* SQL
- NAME 327–8, 369, 372, 382–3
 - see also* resource files
- NAMED_FONT 550
- naming conventions 9–11, 36, 43–50, 60, 102, 138–9, 205–7, 308–9, 369–71, 731–2
 - see also* coding...
- classes 43–4, 60, 102
- data names 44–5
- descriptors 138–9
- DLLs 50
- function names 46
- macros 46
- SQL 731–2
- Symbian OS 8–11, 36, 43–9, 60, 102, 138–9, 308–9, 369–71, 731–2
 - underscores 46
- nanokernel
 - see also* kernel
 - concepts 19–20
- narrow descriptors
 - see also* descriptors
 - concepts 108–9, 124–8
 - conversions 124–8
- Netscape 555
- NetworkControl 265
- networking 19–20, 37–9
- NetworkServices capabilities 266, 276, 280, 644–5, 648–9
- neutral descriptors
 - see also* descriptors
 - concepts 108–9
- New 63–4, 76–7, 83–5, 146, 319, 344, 426, 480

- NewApplication 319–20
- NewL 46, 83–6, 90, 123–4, 132–3, 146, 159–67, 191–4, 196–7, 219, 222–4, 326–7, 336–44, 355–6, 412–13, 428, 506–7, 535–6, 589–90, 598–600, 700–2
- NewLC 46, 83–6, 146, 227–34, 406–8
- NewLC community website 758
- NewMax 146
- NewMaxL 146
- NewMaxLC 146
- NewSessionL 248–9
- newsgroups, Symbian OS 756–7, 758–9
- Next 745–9
- NextRecordRequestL 630–1
- Nokia
 - see also* S60
 - Forum Nokia 757
 - Nseries 2
 - SDKs 2, 287–9, 377, 755
- non-pre-emptive multitasking 157–201, 523
- non-privileged mode
 - see also* user-side...
 - concepts 19–24
- NormalFont 489–92, 516
- notifications of audio resource
 - availability 659–61, 680–2, 708, 712
- NotifyChange 207, 708
- NotifyChangeCancel 207
- NotifyDataAvailable 259
- NotifyRdsDataChange 712
- NotifySignalStrength 708
- NotifySpaceAvailable 259
- NotifyStereoChange 708
- noughts-and-crosses application
 - 220–1, 270–1, 331–66, 374–89, 398–420, 426–55, 464–80, 497, 499–502, 605–49
- bitmaps 374–7
- controls 426–55, 464–80, 497, 499–502
- dialogs 464–80
- S60 333–66, 399–420, 464–80
- transports 605–49
- UIQ 355–66, 399–420, 450, 464–80
- views 344–51, 356–64, 398–420, 426–9, 507–9
- NULL 63, 65, 94, 109–10, 124, 134–5, 206, 319, 480, 538–9, 670–1, 693, 726, 744
- null thread, concepts 21–2
- Num 122–3, 144
- numbers
 - descriptor conversions 122–3
 - templates 56
- numeric keypad, emulator keys 287
- NumFixedWidth 144
- NumFixedWidthUC 144
- NumUC 144
- OBJ files 28
- object-oriented approaches 8, 41–60, 69–70, 88–9, 91, 120, 224–6, 367–8, 515, 534–5, 558, 561–7, 588–9, 594, 652–3
- abstraction principles 43–4, 57–9, 69–70, 88, 225–6, 367–8, 515, 534–5, 558, 561–7, 588–9, 594, 652–3
- concrete behaviour 57–8, 88–9, 91, 224–5
- design patterns 59–60
- fundamental characteristics 62
- inheritance 58–9, 61–2, 69–70, 606
- interface principles 57–60, 196
- polymorphism principles 28–9, 588–93
- re-usable code 54, 59–60, 69–70
- violations 120
- Objective-C 36
- objects
 - active objects 30–4, 60, 151–201
 - concepts 57–9, 61–86, 601–2
 - creation 24–5, 62–70, 159–67, 601–2
 - destruction 35–6, 43–5, 52, 62–70, 76–7, 79–80, 159–67, 188–9, 196–7, 252–4, 323, 336–7, 355–6, 395–6, 601–2
 - lifecycles 601–2
 - ownership 64–5, 119–20
 - persistence 224–40, 334–44, 364–6
 - pick correlation 572–3
- Observer 442–4
- observer interfaces 396, 442–4, 463, 474–5, 564–6, 654
- OfferKeyEvent... 33, 347–50, 360, 427–9, 434–8, 557–8, 562–3, 567–71
 - see also* interaction...
- OfferKeyL 434–8, 567–8
- OK button 72
- OkToExitL 474–7
- on-target debugging 306–8
 - see also* debugging
- On-The-Go (OTG) 610
- one-shot grants 266
- OneTile 502–3
- opaque_data 678–80
- Open... 46, 67–8, 110, 211–12, 215, 229–35, 236–7, 238–9, 253–4, 334–5, 364, 670–1, 691
- Open Font System 549, 602
- open operations
 - files 211–13
 - SQL 731–2, 739
- open platform, Symbian OS 17–18, 28, 272–3
- OpenAsObserverL 635–7
- OpenAsyncL 635–7
- OpenDesL 675
- OpenFileL 234–5, 334–5, 364, 657–66, 675
- OpenIniFileLC 238–9
- OpenLC 46, 229–34, 236–7
- OpenOffice 585–6
- OpenSyncL 635–7
- OpenUrlL 675
- operations
 - see also* entries at start of index

- operations (*continued*)
 - descriptors 121–8, 140–1, 144, 146–7, 218–19
- operator 55–6, 88, 123–4, 131, 141, 144, 146–7, 168–9, 189, 218–19
- OPL 451
- optimization issues
 - backed-up behind windows 452–5
 - debugging 306–7
- origin, drawing basics 489–90
- orphans, SQL 720–1
- out-of-date memory locations 133–4
- out-of-memory errors 62–3, 70–3, 87–8, 104–5, 190, 302–6, 696
 - see also* error handling
- output streaming, audio 667–70
- output.txt 308–9
- overheads
 - active objects 200–1
 - file server 207
 - threads 200–1
- overloaded objects 35–6, 186, 212–13, 219
- overloaded operators 123, 131, 168–9, 212–13, 219
- ownership
 - objects 64–5, 119–20
 - returned heap-based descriptors 119–20
- OwnsWindow 424, 513–14

- P&S *see* publish–subscribe mechanism
- P990 3–4, 450, 659, 757
- package definition files 263, 269–72, 277–8
 - see also* PKG files
- PAGE 469–73
- PageChanged 472
- page_content 467–8
- Panic 8–11, 72–3, 136, 162–7, 302, 323, 659
- panics 8–11, 65, 72–5, 130, 134–7, 162–7, 182–8, 252, 296, 298, 301–2, 307, 518–19, 659
- active objects 182–7, 188, 252
- codes 72–3, 130, 134–5, 182, 188, 252, 296, 298, 302, 307, 731–2
- concepts 72–5, 130, 134–7, 182–8, 252, 296, 298, 301–2
- stray signal panics 182–7
- uses 72–3
- ParameterIndex 746
- parameters 36, 45, 50–1, 67, 110–14, 115–19, 128–9, 135, 692–3
- descriptors 110–14, 128–9
- passing by reference/value 50–1, 67, 115–19, 128–9, 135, 692–3
- passing by reference/value
 - concepts 50–1, 67, 115–19, 128–9, 135, 692–3
- TDes... parameters 128–9
- passwords 272, 462, 464–6
- Path 206
- pathnames, files 205–7
- Pause 658–66
- PC Connectivity software 269, 278–9
- PCM (Pulse Code Modulation) 653, 666–7, 669–71
- PCs 5–6, 18, 26, 28, 71, 269, 278, 283–309, 434, 555, 609
- PDA's 609
- PDDs (physical device drivers) 610–15
- pen 487–92
- pending requests 152–3, 169–70, 177–81, 182–6
- pens 2–6, 18, 152–3, 313, 346, 423–4, 432, 439–42, 448–51, 484–92, 497, 536–7, 540–2, 567
- concepts 489
- drawing basics 346, 448–51, 484–92
- performance issues, active
 - objects/threads 200–1
- persistent stores 224–40, 334–44, 364–6, 715
 - see also* stores
- physical device drivers (PDDs) 610–15
- pick correlation, pointer events 572–3
- PINs 623
- pipes 261
- pixels 487–92, 503–4, 514, 525–6, 534–56
- PKG files 263, 269–72, 277–8
 - see also* package definition files
- placeholders, databases 742–3
- Platform Security 29–30, 217–18, 250, 257, 263–80, 294, 295–6, 297–8, 615, 622–3, 644–5, 648–9, 682, 729–31, 749–51
 - see also* capabilities; security issues
- Play... 657–66, 674, 681–2, 696–7
- playback
 - audio 654–5, 656–66, 676–8, 680–2, 710–11
 - video 655, 672–4, 680–2
- PlayData 677–8
- PlayError 678
- PlayInitL 677–8
- Plot 490–2
- plug-ins 28, 583–4, 587–604, 607–9, 616–17, 633–46, 651–6, 678–82
 - see also* ECOM
- concepts 583–4, 587–604, 607–9, 616–17, 633–46, 651–6, 678–82
- controller plug-ins 678–82
- definition 587–8
- design issues 595–6
- DLLs 28–9, 588–93
- ICL 584, 651, 682–99
- instant messaging 587–604
- MMF 678–82
- MTMs 603–4, 633–46
- object lifecycles 601–2
- packaged binaries 590–1
- polymorphism 588–93

- references 603
- remote communications
 - 607–9, 616–17
- security issues 649, 682
- Symbian OS 603–4, 607–9, 651–6, 678–82
- POINTER 735–6
- pointer descriptors 88–149, 485
 - see also* descriptors; TPtr...
 - concepts 88–9, 96–9, 131–4, 148–49
- PointerEvent 576
- pointers 35–6, 50–1, 64–7, 75, 95–149, 347–9, 361–2, 406–8, 414, 423–4, 432–42, 459–60, 557–82
- cleanup stack 35–6
- controls 423–4, 432, 439–42, 459–60, 557–82
- event-processing sequence 440–4, 574–6
- grab/capture contrasts 440–2, 573
- high-resolution events 573–4
- pick correlation 572–3
- references 50–1, 67
- sequence paradigms 572
- views 406–8, 414
- points, drawing basics 485–92
- polylines 489–92
- polymorphic-interface DLLs
 - see also* dynamically linked libraries
 - concepts 28–9, 588–93
- polymorphism principles 28–9, 588–93
- Pop 133–5, 160–1, 217–18, 322–4, 341–2, 357–8, 406–8, 681
- pop-up windows 457–8, 505–6, 567
 - see also* dialogs
- POP3 637–9
 - see also* emails
- PopAndDestroy 7–11, 35–6, 67, 76–83, 84–5, 90, 102–3, 108, 113, 133–5, 209, 211, 213–14, 218, 227–34, 236–9, 404, 639
- portable views 395
 - see also* views
- PositionChanged 432, 519–20
- PostLayoutDynInitL 474–8
- power management
 - see also* batteries
 - concepts 21–2, 26–7, 34, 276, 280, 609, 701–2, 707–8
 - device drivers 26–7
 - Tuner API 707–8
- PowerMgmt capabilities 276, 280
- PowerOn 701–7
- PowerOnComplete 702–6
- pre-emption concepts 18–19, 21–4, 30–4, 156–8, 173–4, 523
- Preallocate 256
- predictive text input 569
- PreLayoutDynInitL 474–8
- Prepare 673–5, 744, 748
- PrepareForFocusGainL 438, 443–4, 571
- PrepareForFocusLossL 438, 443–4, 463–4, 564, 571
- PrepareForFocus-TransitionL 475
- PrepareImageCaptureL 704–6
- primary databases 739
- PrimeL 681–2
- Print 296
- PrintBandL 544–5, 551
- Printf 7–11, 90, 121–2
- printing 530–1, 535, 543–56, 616, 624–5
- priorities, active objects 170–5, 194–200, 577–8
- private 48–9, 54, 267–8, 288–9
- PrivatePath 208
- privileged mode
 - see also* kernel-side...
 - concepts 19–24
- processes 20–4, 241–62, 308
 - concepts 22–4, 308
 - definition 23
 - IPCs 241–62
- kernel-side/user-side mode 20–3, 25–6, 242–62
- servers 25
- SID (Secure ID) 267–8, 271
- thread contrasts 23–4
- ProcessPointerBuffer-ReadyL 575–6
- ProcessPointerEventL 440–4, 575
- programming errors 72–3, 182–7
- programs 1–15, 18, 20–1, 23–4, 27–9, 46, 72–3, 182–7, 283–309, 529–56, 561–2
 - see also* applications; executables; processes
 - concepts 7–11, 27–9, 72–3, 182, 283–309, 561–2
 - data section 27–9
 - debugging 11, 14–15, 46, 72–3, 283–309
 - device/size-independent graphics 529–56
 - forms 27–8
 - read-only data section 27–9
 - shared libraries 27–9, 37–9
 - SID (Secure ID) 267–8, 271
 - text section 27–9
- progressive decoding, images 688–91
- project definition files 11, 14–15, 266–7, 307, 317–18
 - see also* MMP files
- properties, publish–subscribe mechanism 26, 241, 243–5, 254–7, 260–1
- protected 49–50, 54, 69–70
- Protocol/Service Multiplexers (PSMs) 621, 624–5
- prototypes, functions 49–51
- ProtServ platform security capabilities 250
- PRT 616, 620
- ps 300–1
- Psion 30
- PSMs (Protocol/Service Multiplexers) 621, 624–5
- Ptr 128, 139, 142
- PtrZ 144

- public 48–9, 53–5, 69–70, 82–3
- publication, applications 263–80
- published services, Bluetooth 624–7
- publish–subscribe mechanism
 - see also* interprocess communications;
 - RProperty
 - concepts 26, 241, 243–5, 254–7, 260–1
 - critique 260–1
 - defining operations 243–4, 254–7
 - deletion operations 243–4, 255–7
 - deterministic behaviour 257
 - global variables 243–5
 - operation types 243–5
 - Platform Security 257
 - property ownership 254–5
 - retrieve operations 243, 254–7
 - usage operations 256–7
- Pulse Code Modulation (PCM) 653, 666–7, 669–71
- PushL 7–11, 35–6, 45, 67, 76–81, 99, 106, 133–5, 160–1, 217–18, 341–2, 615, 637–9

- QDialogs 464
- QFileMan 304
- qikapplication.h 314
- qikappui.h 314
- QIK_COMMAND_LIST 325–6, 361–4, 415–17
- QIK_CONTAINER_ITEM 468
- QIK_CONTAINER_SETTINGS 401–2
- QIK_CONTROL 465–8
- QIK_CONTROL_COLLECTION 465–7, 473–5
- qikctrl.lib 313
- QIK_DIALOG... 464–6, 473–5
- qikdocument.h 314
- Qikon 39, 313
 - see also* CQik...; UIQ
- qikon.hrh 327
- qikon.lib 317–18
- qikon.rh 327
- QIK_VIEW... 325–6, 359–64, 401–3, 466–7, 473–5
- qikviewbase.h 314
- QIK_VIEW_CONFIGURATIONS 401, 466–7, 473–5
- quartzkeys.h 433
- query dialogs 461–2
 - see also* dialogs
- QueryWinL 461–2
- Quicktime 587

- R (resource) classes
 - cleanup stack 80–1
 - concepts 43–4, 51, 64, 67–70, 247–54, 308–9
- Radio Data System (RDS) 709–10, 711–12
- radios 584, 651–2, 706–13
 - see also* multimedia...; Tuner API
- RAM 18, 21–2, 24–5, 27–9, 71, 93–5, 148, 203–4, 284–5, 294–5, 375, 550, 715
- RArray 647–8
- rasterizers 510, 546–50, 602
- R_AVKON_SOFTKEYS_OK_CANCEL 370–1, 478–9
- R_AVKON_SOFTKEYS_OPTIONS_BACK 469–72
- raw data, returns from methods 116–18
- raw files 653
- RBackedUpWindow 451–5, 501–2, 516–17, 546–7, 578–9
- RBlankWindow 578–9
- RBuf 88–9, 99–100, 102–9, 111, 112–13, 119–20, 124, 130, 135, 138, 146–7
 - see also* heap-based descriptors
- assignments 105–6
- concepts 88–9, 99–100, 102–9, 111, 112–13, 119–20, 124, 130, 135, 138, 146–7
- construction 103–4
- creation 104–5
- destruction 106
- HBufC migration 107–8, 147
- manipulating-data methods 146–7
- method parameters 111, 112–13, 119–20
- reallocation 106
- RBufReadStream 224
- RBufWriteStream 224
- RComm 583–4, 612–15
- RCommServ 583–4, 611–12
- RDBMS 231, 584–5, 715–53
 - see also* SQL...; system services
 - ACID (atomic, consistent, isolated, durable) concepts 720–1
 - attachment operations 721–2, 738–41
 - background 715–53
 - basics 716–29
 - binding operations 742–3
 - close operations 737–8
 - configurations 732
 - copy operations 737–8
 - creation operations 723–9, 730–6, 749–51
 - deletion operations 718, 733–8
 - detachment operations 740–1
 - errors 725–7, 750–3
 - events 722–3
 - flat files 715–16
 - indexes 718, 734–53
 - insertion operations 718, 726–9, 732–7, 740, 741–4
 - multiple databases 721–3
 - naming conventions 731–2
 - open operations 731–2, 739
 - overview 715–16
 - placeholders 742–3
 - primary databases 739
 - read operations 731, 744–9
 - read–write permissions 738–9
 - retrieval statements 744–9
 - returned data 741–9
 - schemas 716, 725–53
 - statements 740–9
 - stored procedures 722–3

- triggers 723
- update operations 718, 733–7
- RDebug::Print 296
- RDecryptStream 224
- R_DEFAULT_DOCUMENT_NAME 382–3
- RDesReadStream 217–24
- RDesWriteStream 217–24
- R_DIALOGS_USER_PASSWORD_DIALOG 465–6
- RDictionaryReadStream 224, 238–9
- RDictionaryWriteStream 224, 238–9
- RDir 209–10
 - see also* directories
- RDrawableView 452–5, 516–18, 578–81
- RDS (Radio Data System) 709–10, 711–12
- re-usable code, object-oriented
 - approaches 54, 59–60, 69–70
- Read... 209–10, 212–13, 222–4, 250–4, 613–15, 620, 627–8, 670–1
- read operations
 - files 211–14
 - SQL 731, 744–9
- read-only data section, executable
 - programs 27–9
- ReadDeviceData capabilities 276, 280
- ReadDirContentsL 209–10
- ReadFileSection 208
- ReadInt... 218–19, 222–4, 236–7, 388–9
- ReadL 222–4
- Readme.txt files 277–8
- ReadReal... 222–4
- ReadResource... 388, 412–13
- ReadUInt... 220–4
- ReadUserData capabilities 276, 280, 637, 648–9
- real-time systems
 - see also* EKA2
 - concepts 18–19, 242, 244–5
- ReAlloc... 102, 106, 108, 112–14, 120, 133–5, 146–7
- Receive 259
- ReceivedPayload 608–9, 647–8
- ReceiveRdsFrames 712
- REComSession 594–6, 600–4
 - see also* ECOM
- RecordData 678
- RecordError 678
- recording
 - audio 654–5, 661–6, 676–8, 680–2, 711–12
 - video 655, 674–5, 680–2, 699
- RecordInitL 677–8
- rectangles 326–7, 345–6, 347–51, 404–5, 424–9, 444–51, 482–98, 499–502, 506–9, 513–14, 515–21, 531–56
 - drawing 345–6, 449–51, 482–98, 499–502, 506–9, 513–14, 515–21, 531–56
- TRect 345–6, 347–51, 405, 425–30, 444–51, 482–92, 495–502, 506–7, 513–14, 536, 544–5
- redrawn windows
 - see also* controls
 - concepts 445–51, 492–8, 501–4, 511–14, 526–9, 531–2, 568, 577–82
 - storing issues 528–9
 - wasteful redraws 497–8
 - window server 445–51, 496–8, 501–4, 511–14, 526–9
- ReducedSize 686–7
- ReductionFactor 686–7
- reference list 761
- referential integrity, SQL 719–20
- REG files 328–30, 378–9, 385
- Region 487–92
- regions, graphics context 487–92, 496–8, 510–14
- RegisterViewL 395–6, 407
- registration
 - files 328–30, 377–9
 - views 392–6, 407, 421
- REGISTRY_INFO 596–8
- reinterpret_cast 57, 595–6
- relational databases 231, 584–5, 715–53
 - see also* RDBMS
- relationships
 - classes 57–9, 64, 160, 494, 516–17
 - RDBMS 715–16
- release 81, 217–18, 284–6, 300, 383–4
- releases, applications 263–4
- ReleaseTunerControl 709
- removable media 205–6
- RemoveFileSystem 208
- RemoveFromStack 427–9
- RemoveView 395–6
- Rename 207
- REncryptStream 224
- Repeat 145
- Replace 110, 139, 145, 211, 215
- ReplaceL 230
- ReplaceLC 227–34
- ReportEventL 442–4, 565
- ReportWhoseTurn 341–2, 358–9, 407
- ReportWinner 340
- RequestComplete 178, 184, 189–90, 192–3, 195–200
- RequestFocusL 360, 365–6, 463–4
- RequestSignal 186
- RequestTimer 154–5
- Reserve... 701–6
- Reset 196–9, 339–44, 363–4, 490, 497, 511–14, 524–5, 697, 743–4, 747–8
- ResetGc 513–14, 520–1, 524–5
- ResetL 363–4
- ResetStats 420
- ResetView 350–1, 427–9
- Resize 487
- resolution factors, cameras 704–5
- resolver uses, ECOM 597, 599–601

- RESOURCE 268, 270–1, 284, 327–9, 352–5, 359, 361–4, 369, 370–1, 379–80, 386–7, 403, 412–18, 464–9, 478–9, 596–8
 - see also* resource files
 - concepts 369, 370–1, 379–80
- resource allocation keys, emulator debugging 292–3
- resource files 38–9, 268, 270–1, 284, 313, 327–30, 352–5, 359, 361–4, 367–89, 459–80, 583, 596–600, 678–9
 - see also* BAFL
 - application registration files 328–30, 377–9
 - bitmaps 269–72, 285–6, 328–9, 372–7
 - CCoEnv 388–9
 - compilation 383–7
 - compressed data 389
 - concepts 38–9, 313, 327–8, 352–5, 361–4, 367–89, 459–60, 464–6, 472–5, 596–600, 678–9
 - definition 367–8
 - ECOM 596–600, 678–9
 - file structures 385–7
 - icons 263, 329–30, 372–9, 385
 - indexes 385–7
 - languages 379–82
 - localization 328–30, 367–8, 377–82
 - multilingual applications 317–18, 381–2
 - multiple resource files 382–3, 516–17
 - reading 388–9
 - RSG files 327, 382–5
 - statement types 368–72
 - SVG-T (Scalable Vector Graphics-Tiny) 377
 - updates 377–8
 - uses 367–8
 - Windows/Symbian contrasts 367–8
- resource localizable strings (RLS files) 327, 379–82, 385
- Restore... 235–7, 334–44, 365
- retrieval statements, databases 744–9
- return 84–5, 113–14, 115–22
- returned data, databases 741–9
- returns from methods, descriptors 114–20
- Revert 230
- RFCOMM 617, 620–32
- RFile 44, 50, 67–8, 110, 211–18, 239, 253–4
 - see also* file...
 - concepts 211–18, 253–4
 - file-sharing APIs 213–14
- RFileBuf 214–16, 239
- RFileLogger 298–300
- RFileReadStream 217–24
- RFileWriteStream 217–24
- RFs 68, 110, 207–10, 239, 253–4, 334–5, 683–4
 - see also* file server
- RH files 327, 329, 370–1, 377, 384–5, 477–8
- RHandleBase 254
- RHostResolver 584, 619–20
- rich text 539, 545, 639
- Right 140, 142
- RightTPtr 145
- RLibrary 591–3
- RLS files (resource localizable strings) 327, 379–82, 385
- rls_string 380–1, 464
- Rmdir 207
- RMessage2 247–54
- RMessagePtr2 247–54
- RMsgQueue... 257–9
 - see also* message queues
- RNotifier 627–30
- ROM 18, 25, 93–6, 148, 204–5, 284–5, 307–8, 548–50
 - see also* system files
- root streams, stores 225–30
- Rotate 698–9
- rotated images 694–9
- rows, SQL 718–53
- RProperty 254–7
 - see also* publish–subscribe mechanism
- RQikAllocator 304–5
- RReadStream 217–24, 748–9
 - see also* streams
- RResourceFile 388–9
- RS232 serial ports 583–4, 603, 605–6, 607, 609–15
 - see also* serial communications
- RSC files 269–72, 284, 368, 378–9, 384–5
 - see also* resource files
- RScheduler 585
 - see also* Task Scheduler
- RSdpDatabase 624–5
- RSendAs 584, 633–46
 - see also* messaging
- RSendAsMessage 633–5, 638–9, 640–6
- RServer2 247–54
- RSessionBase 185, 247–54, 625–7
 - see also* client–server architecture
- RSG files 327, 382–5
- RSocket 584, 616–32
 - see also* sockets server
- RSocketServ 616–32
- RSoundPlugIn 576
- RSqlColumnReadStream 748–9
- RSqlDatabase 730–53
 - see also* SQL
- RSqlParamWriteStream 748–9
- RSqlSecurityPolicy 749–51
- RSqlStatement 741–9
- RSS files 327, 329, 372–3, 377, 378, 381–5, 401
- RSS_SIGNATURE 328
- RStoreReadStream 224, 229–34, 236–7
- RStoreReclaim 230
- RStoreWriteStream 224, 227–34, 235–7
- RSubSessionBase 247–8, 253–4, 625–6
- RTest 297, 303

- RTest::SetLogged 297
- RThread::RequestSignal 186
 - see also threads
- RTimer 27, 44, 67–8, 159–67
 - see also timers
- RTOS
 - see also real-time systems
 - concepts 18–20
- RunApplication 319
- RunDlgLD 629–30
- RunError 159–67, 171, 189, 191–4, 196–7, 199, 249
- RunL 31–4, 46, 156–201, 248–9, 256–7, 567–8, 575–7, 655, 672, 682–5, 687–94, 698–9
 - see also active objects
 - error handling 163–4, 171, 183
 - priority problems 173–5, 201
- RWindow 44, 67–8, 445–51, 453–5, 501–2, 510–14, 516–18, 526–8, 578–81
- RWindowBase 454, 578–81
- RWindowGroup 576–82
- RWindowTreeNode 517–18, 578–81
- RWriteStream 44, 217–24, 338–44, 748–9
 - see also streams
- RWsBuffer 510–14, 576–82
- RWsSession 492, 576–82

- _S 136
- S classes 43
- s32crypt.h 224
- s32file.h 224
- s32mem.h 224
- s32std.h 224
- s32store.h 224
- S60 2, 4–6, 14–15, 39, 195, 234–9, 270, 283–309, 311–30, 333–66, 370–1, 393, 394–420, 439, 530, 559–61, 566–7, 621, 627–30, 641, 683, 694, 699, 755
 - see also CAkn...; Nokia
 - application views 326–7, 344–51, 392, 394–420
 - background 313–30, 333–66, 560, 566–7, 627–30, 641, 683, 694, 699
 - Bluetooth 627–30
 - cascaded menus 6
 - classes 313–30, 333–55, 394–5
 - commands 338–9, 351–5, 414–20
 - dialogs 457–80
 - displayed images 683, 694
 - emulator 2, 4–5, 14, 15, 283–309, 637
 - header files 313–14
 - ‘Hello World’ example 315–30
 - icon sizes 374
 - INI files 238
 - MMS MTMs 641
 - noughts-and-crosses application 333–66, 399–420, 464–80
 - screens 4–6, 344–51, 432, 439, 566–7, 683, 694
 - SDKs 2, 287–9, 377, 546, 755
 - UIQ code 315, 345, 355–66, 400–20, 468–72, 474–5
 - v3 SDK 2, 288
 - view architecture 391–420
- SaveGameStateL 364–5, 411
- SaveL 237, 354–5, 364–6, 419–20
- SaveMessageL 642–6
- scalable fonts 549–50
 - see also fonts
- Scalable Vector Graphics-Tiny (SVG-T) 377
- Scale 698–9
- scaled images 694–9
- scan codes 433–4
- scanf 140
- Schedule Server see Task Scheduler
- scheduling 19, 21–4, 30–4, 156–201, 523
 - see also active...
 - concepts 19, 21
 - pre-emptive/non-pre-emptive concepts 21–4, 30–4, 156–8, 523
- schemas, RDBMS 716, 725–53
- SCHSVR see Task Scheduler
- Screen 510–14
- screens 3–6, 285–6, 312–13, 324–7, 331, 344–51, 355–66, 376, 414–20, 423–55, 459–80, 481–556, 566–82, 587, 655, 672–713
 - see also controls; drawing; graphic...; windows
 - device/size-independent graphics 529–56
 - direct screen access viewfinders 702–3
- S60 4–6, 344–51, 432, 439, 566–7, 683, 694
- sharing 502–14
- UIQ 3–6, 355–66, 432, 439–42, 450, 566–9, 572, 581
- scrolling, special effects 522, 525, 529
- SDKs (software development kits) 2, 7, 13–14, 43, 57, 99, 122, 124, 208, 231, 266, 283, 287–8, 295, 297, 300–4, 370, 377, 546, 579, 755
- SDP (Service Discovery Protocol) 624, 627–9
- second-phase constructors
 - see also constructors
 - concepts 52, 81–6, 336–7, 406–8, 428–9, 516–17
- sectors, files 215–16
- Secure ID (SID) 267–8
- SECUREID 254, 267, 317–18
- security issues 29–30, 217–18, 250, 257, 263–80, 294, 295–6, 297–8, 622–3, 646–9, 682, 729–31, 749–51
 - see also Symbian Signed
 - authorization processes 264–6, 272–8
 - Bluetooth 622–3, 648–9

- security issues (*continued*)
 - capabilities 29–30, 250, 264–8, 274–80, 295–8, 648–9, 656–7, 682
 - communications 622–3, 646–9, 682
 - concepts 29–30, 217–18, 250, 257, 263–80, 622–3, 646–9, 682, 749–51
 - data caging 29, 264, 267–8, 730
 - data validation 646–8
 - Developer Certificates 275–7
 - emulator 294, 297–8
 - files 29, 217–18, 264
 - one-shot grants 266
 - plug-ins 649, 682
 - publish–subscribe mechanism 257
 - releases 263–4
 - SID (Secure ID) 267–8, 271
 - signed applications 263, 264–7, 269–78, 295–6
 - support issues 266–8
 - threat types 264–6, 272–3, 646–9
 - unsigned applications 265–6, 273–6, 279–80
 - workings 264–6
- Seek 212
- SEikControlInfo 478–80
- selections, SQL 718, 740–9
- self-signed applications 275
- semaphores 31–4, 168–71, 182–90, 241
- semicolons 47
- Send... 250–4, 259, 644–6
- Send As server 584, 633–46
- SendAs 584, 633–46
- SendEventToClient 678
- SendMessage 589–90
- SendReceive 185–7, 250–4
- SentPayload 608–9
- serial communications 18, 583, 603, 605, 607–15, 617, 620–32
 - see also* communications
 - advantages 609–10
 - Bluetooth 615, 620, 621
 - classes 607
 - concepts 605, 607–15
 - data exchange 613–15
 - infrared 611, 614–15
 - limitations 610
 - opening 610–12
 - RFCOMM 617, 620–32
 - SerialUtils 613–14
- servers
 - see also* file...; sockets...; view...; window...
 - client–server architecture 25–6, 33, 179–81, 183–6, 207–9, 214–16, 241–3, 247–54, 259–60, 261–2, 424, 504–14, 729–31
 - concepts 19–20, 25–6, 28, 29, 30–4, 37–9, 207–9, 583, 605–6
 - IPCs 241–62
 - late request cancellations 179–81
 - processes 25
 - Symbian SQL 729–53
 - transports 605–6
- Service Discovery Protocol (SDP) 624, 627–9
- service providers 152–3
- ServiceL 33, 250–4
- SessionPath 208
- sessions
 - client–server architecture 26, 180–1, 207–9, 211, 214–16, 241–3, 247–54, 259–60, 261–2, 424, 729–31
 - file server 207–9, 211
- Set... 46, 98–9, 108, 139, 145, 148, 206, 207, 227, 346–51, 357–8, 363, 404, 431, 487–92, 500–2, 511, 517–20, 526, 668–71
- SetActive 160, 162–98, 613, 618–19, 698–9
- SetAtt 207
- SetAudioEnabledL 675
- SetAudioPropertiesL 668–71
- SetAudioTypeL 675
- SetAutoFlush 492
- SetAvailable 363
- SetBackgroundColor 580–1
- SetBlank 425–6, 444–51, 520–1
- SetBodyTextL 638–9, 641–2
- SetBrushColor 448–51, 489–92, 500–2, 540–2
- SetBrushOrigin 489–92
- SetBrushPattern 489–92
- SetBrushStyle 346, 448–51, 489–92, 500–2, 536–7, 540–2
- SetBusy 237
- SetClippingRect 490, 497, 511, 536–7
- SetComponentToInherit-Visibility 454–5
- SetConfig 612–13
- SetContainerWindowL 404–5, 507, 517–18
- SetCornerAndSize 431, 519–20
- SetDataTypeL 667–71
- SetDbPolicy 750–1
- SetDefault 14
- SetDefaultViewL 396–7, 407
- SetDestinationBitRateL 665–6
- SetDimmed 455, 563
- SetDriveName 207–8
- SetExtent 431, 519–20, 580–1
- SetExtentToWholeScreen 431
- SetFaded 455
- SetFadingParameters 455
- SetFocus 428, 436, 438, 444, 463–4, 571
- SetFocusByIdL 357–60, 364–6
- SetFocusing 360, 438, 571
- SetFullRedraw 535–7
- SetGraphicsDeviceMap 542–3
- SetHeight 487
- SetHello 159–67
- SetImageSource 694–9
- SetIsProportional 550

- SetJustInTime 295
- SetL 625–6
- SetLastBuffer 677–8
- SetLength 139, 145
- SetLineDimmedNow 476
- SetLogged 297
- SetMatchToMimeTypeL 663–6
- SetMax 145
- SetMediaIdsL 674–5
- SetMessageRootL 644–6
- SetModified 207
- SetNonFocusing 571
- SetObserver 442–4, 565
- SetOpt 620
- SetOrigin 489–92
- SetPageDimmedNow 472
- SetPenColor 346, 450–1, 489–92, 497, 536–7, 540–2
- SetPenSize 450–1, 489–92
- SetPenStyle 448–51, 489–92, 500–2, 540–2
- SetPlayWindow 659
- SetPointerCapture 440–2, 573
- SetPort 619–20
- SetPosition 431–2, 519–20, 658–66
- SetPriority 174–5
- SetQualityAlgorithm 699
- SetRect 326–7, 404–5, 425–31, 444–51, 507, 519–20
- SetSecurity 622–3
- SetSessionPath 208
- SetSize 431, 479, 519–20, 580–1
- SetSizeInTwips 548
- SetSizeWithout-Notification 431–2, 519–20
- SetStateL 363–4
- SetStrikethroughStyle 489–92
- SetSubjectL 634–5, 642–6
- setter functions 46, 98–9, 108
- SetTextL 363, 413–14, 507, 535–6
- SetTransparency... 526
- SetTwipToPixelMapping 542–3
- SetUnderlineStyle 489–92
- SetUpL 695–6
- SetVideoTypeL 675
- SetVolumeLabel 207–8
- SetWidth 487
- SetZoomAndDevice-DependentFontL 542–3
- SetZoomFactor 542–3
- SetZoomInL 543
- SetZoomOutL 543
- shared access, files 211–16
- shared chunks
 - see also interprocess communications
 - concepts 26, 29, 706
- shared libraries
 - see also dynamically linked...
 - concepts 27–9, 37–9
- SharedProtected 213–14
- Shift key 292, 376, 427, 440, 524, 567
- shortcut keys 6, 523–4, 568–9
- ShouldUseReceivedMtmUid 636–7
- ShowTurn 350–1, 427–9
- Shrink 346, 482–7, 541
- ShrinkToAspectRatio 686, 696–7
- SIBO (EPOC16) 87, 241–2
- SID (Secure ID) 267–8, 271
- signed applications 263, 264–7, 269–78, 295–6
 - see also Symbian Signed
- signsis 269–72, 277
- simple controls 424–9
 - see also controls
- single-page dialogs 459–62, 464–6
 - see also dialogs
- sinks, controller plug-ins 681–2
- SIS files 263, 269–74, 277–9, 295–6, 382
 - see also installation
- Size 46, 110, 126–8, 137, 139, 142
- size-independent graphics 529–56
- SizeChanged 350, 404–5, 427–32, 519–20
- SizeInPixels 547–9
- SizeInTwips 547–9
- sizes 46, 110, 126–8, 137, 139, 142, 350, 404–5, 427–32, 487–92, 519–20, 529–56
- Skype 588
- smartphones
 - see also mobile phones
 - background 17–18, 26–7
 - constraints 26–7, 29, 71–2, 242, 302–3, 424, 452, 530
 - debugging 306–8
- SMIL files 640–6
- SMS (Short Messaging Service) 26, 265, 605–9, 632–49, 738
 - concepts 639, 738
 - emails 639
- SMTP 634, 638–9
 - see also emails
- sockets server
 - see also Bluetooth; ESock; infrared; servers; TCP/IP
- communications 605–6, 615–32
 - concepts 28, 37–9, 584, 586, 604, 605–6, 615–32
 - processing sequence 616–17
 - roles 605–6, 615–32
 - transport concepts 605–6
- softkeys 352–3, 361–2, 469–72
- software development kits (SDKs) 2, 7, 13–14, 43, 57, 99, 122, 124, 208, 231, 266, 283, 287–8, 295, 297, 300–4, 370, 377, 455, 546, 579, 755
- Sony Ericsson
 - Developer World 757
 - P990 3–4, 450, 659, 706, 757
- sounds 576, 584, 651–713
 - see also audio; multimedia...
- SOURCE 11–12, 317–18, 374–5, 598–9
- source code 7–11, 14, 28–9, 36, 43–9, 72–3, 283–309, 316–30, 715–53
 - see also naming conventions; programs

- source code (*continued*)
 - concepts 7–11, 28–9, 36, 43–9
 - debugging 11, 14–15, 46, 72–3, 283–309
 - device/size-independent graphics 529–56
 - SQL 715–53
- SOURCEPATH 11–12, 317–18, 374–5, 598–9
- sources, controller plug-ins 681–2
- special effects 521–6
 - animation 516, 521–3
 - backed-up behind windows 452–5, 501–2, 506–7, 515–21
 - control context 522, 524–5
 - debug keys 521, 523–4, 568–9
 - scrolling 522, 525, 529
 - transparent windows 522, 525–6, 528
- specialized justification settings, drawing basics 490
- specifications, files 205–7
- splash screens 372
- sprintf 121–2, 140
- SQL 231, 715–53
 - see also* RDBMS
 - ACID (atomic, consistent, isolated, durable) concepts 720–1
 - attachment operations 721–2, 738–41
 - background 716–53
 - basics 716–29
 - binding operations 742–3
 - cascading deletes 720–1
 - close operations 737–8
 - configurations 732
 - constraints 719–21
 - copy operations 737–8
 - creation operations 723–9, 730–6, 749–51
 - data cages 730
 - data types 726–7
 - deletion operations 718, 733–8
 - detachment operations 740–1
 - errors 725–7, 750–3
 - examples 723–49
 - features 716–21
 - indexes 718, 734–53
 - innovations 719–21
 - insertion operations 718, 726–9, 732–7, 740, 741–4
 - multiple databases 721–3
 - naming conventions 731–2
 - open operations 731–2, 739
 - open source implementations 716–17
 - operation basics 723–9
 - orphans 720–1
 - placeholders 742–3
 - Platform Security 729–31, 749–51
 - primary databases 739
 - read operations 731, 744–9
 - read–write permissions 738–9
 - referential integrity 719–20
 - retrieval statements 744–9
 - returned data 741–9
 - schemas 716, 725–53
 - statements 740–9
 - Symbian SQL 715, 729–53
 - syntax 722–9
 - terminology basics 717–18
 - update operations 718, 733–7
- sqldb.h 730–1
- SQLite 716–18, 721–53
- SquareStatus 449–51
- SRLINK 369
- stack 21, 24–5, 65–70, 75, 76–86, 88–9, 95, 129–30, 209, 568–70
 - see also* cleanup stack
 - concepts 21, 24–5, 65–70, 75, 76–86, 88–9, 95, 129–30, 209
 - creation 24–5, 65
 - definition 65
 - file server sessions 209
 - size considerations 129–30
 - threads 21, 24
 - uses 65, 129–30, 209
- stack descriptors 88–149
 - see also* TBuf . . .
 - concepts 88–9, 93–5, 121–2, 129–30, 145–7
 - creation 93–4
 - heap allocations 130
 - large stack descriptors 129–30
 - run-time size-setting dangers 129–30
- standard windows 579–82
- Start . . . 165–7, 181, 191–200, 249, 301–2, 374, 627–30
- START BITMAP 374
- START RESOURCE 317–18, 374, 379, 598–9
- starter_shell.rsc 284
- StartViewFinderBitmapsL 703
- StartViewFinderDirectL 703
- starvation problems, active objects 173–5, 189
- state changes, dialogs 463, 474–5, 561, 564–70
- state machines, active objects 190–200
- statements, SQL 740–9
- static 50, 72, 167–71, 196–7, 342, 515–16
- static binding 128–9
- static const char 89, 93, 100
- static libraries, concepts 28
- static-interface DLLs 28, 598–9
- static_cast 57
- STATICLIBRARY 304–5
- StationSeek 708
- status bars 4
- stock controls, dialogs 475–7
- Stop 181, 658–66, 670–1, 697
- StopViewFinder 702–6
- Storage Manager 279–80
- Store . . . 235–7, 334–44, 365–6
- STORE concepts 38–9
 - see also* streams
- stored procedures, databases 722–3
- stores
 - see also* streams

- application architecture 234–7
- class hierarchy 224–6
- concepts 38–9, 223–40, 334–44
- creation 226–34
- embedded stores 231–4
- files 224–34
- INI files 237–9
- persistence concepts 224–40, 334–44, 364–6
- physical structures 226
- read operations 229–34
- root streams 225–30
- stream dictionaries 225–6, 228–39
- uses 223–6
- stray signal panics, active objects 182–7
- streamed decoding *see* progressive decoding
- streaming APIs 654–6, 666–71
- streams
 - see also* stores
 - base classes 44, 217–18
 - concepts 38–9, 44, 216–40, 335–6, 748–9
 - dictionaries 224, 225–6, 228–39
 - external representation 216–17
 - externalization 216–17, 218–24, 226, 235–7, 335–44
 - header files 224
 - internalization 216–17, 218–24, 236, 335–44
 - noughts-and-crosses application 220–1, 331–66
 - reading/writing functions 221–4
 - types 216, 223–4
- stringLen 125–6
- StringPool 586
- strings 87–149
 - descriptor conversions 124–8
 - localizable strings 327, 379–82
 - RLS files (resource localizable strings) 327, 379–82, 385
 - substring methods 140
 - Stroustrup, Bjarne 35
 - STRUCT
 - see also* resource files
 - concepts 369–70, 386–9, 470–8
 - structs 43, 369–71, 386–7, 470–8
 - Subscribe 256–7
 - substring methods, descriptors 140
 - Sum Microsystems 757
 - supervisor concepts 20–4
 - support forums, Symbian OS 756–7
 - SVC
 - see also* kernel-side mode
 - concepts 20
 - SVG-T (Scalable Vector Graphics-Tiny) 377
 - Swap 145, 147
 - SWI instruction 21
 - switch statements 92, 192–4, 198, 237, 251, 323–4, 354–5, 364, 407, 410, 413–14, 419–20, 435–8, 443–4, 480, 509
 - SwitchFocus 350, 427–9, 443–4
 - SwitchTurn 341–2, 363–4
 - Symbian Audio Controller 653
 - Symbian Libs Adapter 729–30
 - Symbian OS
 - see also* kernel
 - background 1–15, 17–40, 263, 268–80, 284–5, 367–8, 372–4, 378–9, 391–2, 424, 498, 509–14, 525–7, 549–50, 558, 564–5, 570, 577–8, 580–1, 603–4, 651–3, 676, 755
 - bad practices 59
 - basics 1–15, 17–86
 - books 759
 - C++ 30–6, 41–60, 61–2
 - coding conventions 17, 36, 43–9, 369–71
 - community websites 758–9
 - component definition file 11–14
 - constraints 18, 26–9, 71–2, 242, 302–3, 424, 452, 530
 - data types 41–3, 368–9
 - design issues 17–40, 54, 59–60, 64, 558, 564–5, 577–8
 - Developer Network 756–8
 - developer training 757–8
 - drives 205–8, 267–8, 284–6, 295
 - emulator 1–15, 267–9, 277, 283–309, 376, 379, 523–4, 610–12, 637
 - EPOC16 (SIBO) 87, 241–2, 311–12
 - extensibility principles 583–604
 - framework basics 30–6, 51–2, 570
 - historical background 18, 30, 34–5, 87–8, 241–2, 311–12, 558
 - idioms 17–18, 30–40
 - installation 263–4, 268–74, 278–9, 296
 - naming conventions 9–11, 36, 43–9, 60, 102, 138–9, 308–9, 369–71, 731–2
 - newsgroups 756–7, 758–9
 - object-oriented approaches 8, 41–60, 367–8
 - open platform 17–18, 28, 272–3, 549
 - project definition files 11, 14–15, 266–7, 307, 317–18
 - SDKs 2, 6, 13–14, 43, 57, 99, 122, 124, 208, 231, 266, 283, 287–8, 300–4, 370, 377, 455, 546, 579, 755
 - security issues 29–30, 217–18, 250, 257, 263–80, 294, 295–6, 297–8, 622–3, 646–9, 682, 729–31, 749–51
 - support forums 756–7
 - System Definition papers 755

- Symbian OS (*continued*)
 - system introduction 17–40
 - system services 583–7, 603–4
 - templates 55–6, 94
 - tools 1–15, 283–309, 755–9
 - v5.0 109, 555
 - v7.0 526–7, 603, 654
 - v8.0 99, 525, 528, 580
 - v8.1 21, 70, 125, 127, 526
 - v9 2, 263, 266, 267, 270, 273–4, 277, 279, 295, 297, 307, 368, 378, 592–3, 755
 - v9.1 529, 601–2, 651–3, 656, 666–7, 671, 676, 682, 694, 697–9
 - v9.3 610
- Symbian Signed 263, 264–7, 269–78, 295–6
 - benefits 273–4
 - concepts 263, 264–7, 269–78, 295–6
 - Developer Certificates 275–7
 - logo 278
 - overview 272–8
 - procedural steps 273–4, 277–8
 - submission of SIS file 277–9
 - test criteria 273–5, 277–8, 295–6
 - unsigned applications 265–6, 273–6, 279–80
 - VeriSign 278
- Symbian SQL 715, 729–53
 - see also* SQL
- _SYMBIAN32_ 46
- SymbianOne community website 758
- SymbolFont 489–92
- SymScan 308–9
- SynchL 215–16
- synchronization 19, 241
- synchronous operations 31–3, 151–2, 168–71, 176–81
- sys 268
- System Definition papers, Symbian OS 755
- system files
 - see also* files; ROM
 - concepts 204–5, 268, 284–5, 307–8
- system introduction, Symbian OS 17–40
- system services
 - concepts 583–7, 603–4
 - further information 586–7
- system-initiated redraws 493–5, 496–8
- SystemGc 445, 448–51, 482, 483–6, 493–5, 499–502, 520–1, 545
 - see also* graphics context
- SYSTEMINCLUDE 11, 317–18, 598–9
- T (data-type) classes, concepts 43–4, 51, 65, 66–8, 488
- tables, SQL 717–53
- tabs 566–7
- TActivePriority 172–3
- TAknExFormDialog-
 - ControlIds 472
- TAknExFormPageControlIds 470–2
- Taligent coding standard 36
- TAmPm 66–7
- TAny 42–3, 55–6, 81, 601
- TARGET 11–12, 266–7, 317–18, 598–9
- target devices,
 - device/size-independent
 - graphics 529–56
- TARGETPATH 374–5, 379
- TARGETTYPE 11–12, 317–18, 598–9
- Task Scheduler 585
 - see also* system services
- TAudioCaps 668–71
- TBandAttributes 544–5
- TBmpImageData 693–4
- TBool 42–3, 168–9
- TBTDeviceResponseParams-
 - Pckg 628–9
- TBTDeviceSelectionParams-
 - Pckg 628–9
- TBTServiceSecurity 622–3
- TBTSockAddr 622–3
- TBuf... 56, 65, 67, 88–149, 206–7, 219, 353–4, 380–1, 388
 - see also* buffer descriptors
 - concepts 88–9, 93–6, 108–9, 111, 116, 118–19, 121–2, 125–32, 135, 147–9, 206–7, 219, 353–4
 - file names 206–7
 - manipulating-data methods 147–9
 - RAM usage 95
 - run-time size-setting dangers 129–30
 - text console 121–2
- TC TrustCenter 269–70, 273
- TCameraInfo 700–6
- TCardinality 226
- TCleanupItem 80–1
- TCoeEvent 565–6
- TCoeInputCapabilities 437
- TCommConfig... 612–13
- TCP/IP 37, 39, 287–8, 289, 584, 586, 604, 605–6, 616–32
- tcpiip.prt 604
- TDes... 44, 50, 56, 88–149, 206–7, 212, 223–4, 235–7, 255
 - see also* descriptors
- const TDesC... 111–20, 124–9, 132
 - correct usage 128–9, 149
 - declaration dangers 128
 - files 206–7, 212
 - folding processes 138
 - manipulating-data methods 140–5
 - passing by reference/value 128–9
- TDescriptorDataSource 694–9
- TDisplayMode 517–18, 547–8, 553–5
- TEComResolverParams 597, 599–600
- telephony server
 - see also* ETEL
 - concepts 39, 603–4
- telephony stack 19–20

- Temp 110, 211, 215
- templates
 - concepts 55–6, 94
 - numbers 56
 - thin-template patterns 55–6
- TEntry 130, 209–10
 - see also* directories
- TEntryArray 210
- tests
 - see also* Symbian Signed
 - applications 273–5, 277–8
 - emulator certificates 295–6
 - memory 302–6
- TEventCode 427–9
- TEventModifier 433–8
- TEventType 711
- TEXT 368
- text
 - cursors 571
 - drawing 482–6, 492
 - editors 559–62
 - section 27–9
- text console class 8, 121–2
 - see also* CConsoleBase
- text files 211–13, 592–3, 645–6
 - see also* files
- text messages 26, 265, 605–6
 - see also* SMS
- TFileName 44, 130, 206–7
- TFixedArray 400–1, 404, 647–8
- TFontSpec 534–56
- TFontStyle 539, 545–56
- TFormat 704–6
- TFrameDataBlock 693–4
- TFrameInfo 686–8
- TFrequency 708–13
- TFullName 44, 130
- thin-template patterns 55–6
- third-party applications 17–18, 584–91, 593–604, 682, 755
- thread-local storage (TLS) 515–16
- threads 19, 21–4, 30–4, 62–5, 151–201, 241–62, 307–8, 515–16, 651–6, 682–99
 - see also* active objects
 - concepts 21–4, 30–1, 62–5, 151–201, 307–8, 515–16, 651–6, 682–99
 - context switches 23–4
 - definition 23
 - ICL 682–99
 - IPCs 241–62
 - kernel concepts 21–4
 - message queues 26, 241, 245–7, 257–9, 261–2
 - MMF 651–6
 - multithreading 18–19, 157
 - overheads 200–1
 - process contrasts 23–4
 - publish–subscribe mechanism 26, 241, 243–5, 254–7, 260–1
 - stack 21, 24
- threat types, security issues 264–6, 272–3, 646–9
- throw 9–10, 63, 73–4
- throwing 9–10, 34–6
 - see also* error handling
- thrupenny-bitting 574
- thumbnails 695
- Tic-Tac-Toe *see*
 - noughts-and-crosses application
- Tiff files 587
- TileStatus 348–51, 427–9
- TImageDataBlock 693–4
- Timer 153
- timer events 152–5
 - see also* events
- timer thread, concepts 22
- timers 19, 22, 27, 159–67
 - see also* RTimer
 - concepts 27
- times, dialogs 462
- timestamps, directories 210
- TImplementationProxy 599
- TInt... 9–10, 41–2, 45, 48–50, 65, 69–70, 76, 82–3, 90, 97, 105–6, 111–12, 119, 123–8, 168–9, 200, 211, 214–24, 228, 232–4, 255–8, 429–31
- TIpArgs 247–54
- TIrdaSockAddr 618–19
- title bars 3–4, 562–3
- TitleFont 482–6
- TJpegImageData 693–4
- TKeyEvent 427–9, 433–42
- TKeyResponse 347–50, 427–9, 435–8
- TL2CapConfig 621
- TLex 122–3, 140
- TLibraryFunction 592
- TLitC 89–93
 - see also* descriptors; LIT
- TLogicalColor 553–4
- TLS (thread-local storage) 515–16
- TMdaAudioSettings 668–71
- TMdaPackage 667–71
- TMMFCapabilities 676
- TMMFileHandleSource 656
- TMMFileSource 656, 694–9
- TMMSource 656, 694–9
- TName 130
- TNameRecord 619–20
- TodoSymbian community website 759
- toolbars 324–5, 530–1
- tools 1–15, 283–309, 755–9
- TOptions 684–91, 693–4, 700–6
- touch-sensitive screens 2–7, 18, 152–3, 313, 423–4, 432, 439–42
- TParse... 206–7
- TPoint 44, 487–92, 519–20, 536–7
- TPointerEvent 439–42
- TPriority 172–3
- TProtocolDesc 617–18
- TPtr... 67, 88–9, 96–9, 109–11, 113, 115–18, 124, 127–8, 131–4, 135–7, 144–5, 148–9
 - see also* pointer descriptors
 - = (assignment operator) problems 131
- C++ conventions 131–2
- concepts 88–9, 96–9, 109–11, 113, 115–18, 124, 127–8, 131–4, 135–7, 144, 147–9
- correct usage 131–4, 149
- initialization issues 136–7
- manipulating-data methods 148–9

- TPtr... (*continued*)
 - method parameters 111, 113, 115–18
- RBuf migration 107–8
 - types 96–9
- transformations, images 694–9
- transparent windows, special
 - effects 522, 525–6, 528
- transport concepts 605–49
 - see also* communications; messaging
- TRAP... 7–10, 35–6, 73–5, 77, 80, 85, 163, 177–8, 189, 251–2, 498–9
 - see also* error handling
- TReal... 42, 222–4, 307–8
- TRect 345–6, 347–51, 405, 425–30, 444–51, 482–92, 495–502, 506–7, 513–14, 536, 544–5, 686, 696
 - see also* rectangles
- TRequestStatus 162–7, 168–71, 176–81, 183, 185–7, 189, 192–3, 197–8, 215–16, 256–7, 613, 623–4, 698–9, 749
- TResourceReader 308, 388–9, 404–5
- TRgb 552
 - see also* color...
- triggers, databases 723
- Trim 145
- TrimAll 145
- TrimLeft 145
- TrimLetl 145
- TrimRight 145
- TRotationAngle 698–9
- TRUE 42
- try 9, 73
- TryChangeFocusToL 463, 476
- TryHitL 347, 349, 436–8, 441–2, 447–51
- TryHitSquareL 349–51
- TryMakeMove 340–1, 342–4
- TSecurityPolicy 750–1
- TSize 346–51, 430, 450–1, 487–92, 499–502, 519–20, 536–7, 696, 699, 703
- TSockAddr 618–19
- TState 661–6
- TStdScanCode 433–8
- TStreamId 228–37, 334–44
- TText... 42, 110, 117–18
- TTileState 350–1
- TTileStatus 350–1
- TTunerCapabilities 707–8
- TTunerFunctions 707–8
- TTypeface 539
- TUIdType 227–34
- TUint... 41–2, 66–7, 97, 104, 110, 169, 196–8, 222–4, 239, 255, 453
- Tune 708–13
- Tuner API 584, 651, 706–13
 - see also* radios
 - antennas 707–8
 - audio playback 710–11
 - audio recording 711–12
 - background 651, 706–13
 - controls 709–10
 - limited support 706
 - notifications 708, 712
 - power management 707–8
 - RDS 709–10, 711–12
 - uses 706–10
- tuner.h 706
- TunersAvailable 706–13
- TUUIID 625–6
- twips 534–43
- two-phase construction
 - see also* constructors
 - concepts 52, 81–6, 336–7, 406–8, 428–9, 516–17
- typedef 9–10, 43–4, 66–7, 108, 119
- TZoomFactor 531–56
- UC descriptor naming convention 138
- udeb 285–6, 289–90, 300, 306, 383
 - see also* emulator
- _UHEAP_MARK 304
- _UHEAP_MARKENDC 304
- UI *see* user interfaces
- UIDs (unique identifiers) 206, 224, 227–34, 236–40, 255, 267, 317–18, 329, 334–5, 378–9, 392, 408–9, 597–601, 633–7, 660–6, 674–5, 679, 681–2, 684–92, 702, 729–31, 756
- UIKON 39, 57, 311–30, 384, 395, 452, 518, 523–4, 530
 - see also* APPARC; application framework; CEik...; CONE
 - concepts 311–30, 518, 523–4
 - debug keys 523–4
- uikon.hrh 327
- uikon.rh 327, 386–7
- UIQ 2–6, 13–14, 37, 39, 86, 234–9, 279, 283–304, 311–30, 355–66, 391–420, 432, 439–42, 530, 555, 560–1, 566–9, 572, 609, 621, 627–30, 641, 755
 - see also* CQik...
- 3rd Edition SDK 2, 288–9, 304–5, 355–66, 375
- application views 325–6, 355–64, 391–420, 568–9
- background 313–30, 355–66, 391–420, 560, 566–9, 572, 627–30, 641
- Bluetooth 627, 629–30
- classes 313–30, 355–66, 393–4
- commands 325–6, 360–6, 407–8, 414–20
- dialogs 457–80, 568–9, 581
- emulator 2–7, 13–14, 283–309
- FEPs 433, 568–70
- header files 313–14
- ‘Hello World’ example 315–30
- icon sizes 374
- key-event processing 567–9
- MMS MTMs 641

- noughts-and-crosses application
 - 355–66, 399–420, 450, 464–80
- persistence 364–6
- S60 code 315, 345, 355–66, 400–20, 468–72, 474–5
- screens 3–4, 355–66, 432, 439–42, 450, 566–7, 572, 581
- SDKs 2, 288–9, 295, 300–5, 546, 755
- Storage Manager 279–80
- view architecture 391–420, 568–9
- underscores, naming conventions 46
- Unicode 12, 46, 108–9, 124, 206–7, 223, 368, 387, 434, 550
- UNIQUE 736–7
- unique identifiers (UIDs) 206, 224, 227–34, 236–40, 255, 267, 317–18, 329, 334–5, 378–9, 392, 408–9, 597–601, 633–7, 660–6, 674–5, 679, 681–2, 684–92, 702, 729–31, 756
- Unix 585
- UnloadCommModule 613–14
- UnLock 212
- unsigned applications, Platform Security 265–6, 273–6, 279–80
- unsigned char 125–6
- unsigned int 41–2, 126–7
- unsigned short int 126
- unsigned-sandboxed capabilities 265–6, 279–80
- UPDATE 737
- update operations, SQL 718, 733–7
- UpdateAttributeL 625–6
- UpdateCommandsL 363–4, 410–11
- UpperCase 145
- UREL 306
- URLs 654, 656–7
- USB 18, 205–6, 268, 269, 278–9, 609–10
- UseBrushPattern 548–9
- UseFont 482–6, 489–92
- UseL 76–8, 82–3
- User 44
- user interfaces 2, 37–9, 76–7, 158, 195–200, 220–1, 312–30, 331–66, 391–421, 423–55, 493–5, 755
 - see also* graphical...; S60; UIQ
- API groupings 37–9
- controls 324–7, 400–6, 412–14, 423–55
- SDKs 2, 755
- user library
 - see also* euser...
- concepts 20–1, 38–9, 305–6
- user requirements, interaction
 - graphics 559–61
- user-generated events, controls 423–4
- user-side mode, concepts 19–22, 24–6, 167–71, 200–1, 242–5
- user-side servers 19–20, 25–6, 241–3
- User::After 7–11
- User::Alloc... 62–3, 82–3, 90
- UserEnvironment capabilities 274–5, 276, 280, 699–706
 - see also* Bluetooth
- User::FreeLogicalDevice 611–12
- User::FreePhysicalDevice 611–12
- USERINCLUDE 11, 317–18, 598–9
- User::Leave 9–10, 35–6, 73–5, 77, 80–1, 85, 187, 636–7
- User::LeaveIfError 161, 208–11, 213–15, 217–18, 256–7, 428, 461–2, 545, 588, 611–12, 618–19, 634, 686, 694, 707
- User::Panic 7–11, 72–3, 136, 162–7
- User::RequestComplete 178, 184, 189–90, 192–3, 195–200
- User::SetJustInTime 295
- User::WaitForAnyRequest 170–1, 176–81, 183, 186–7
- User::WaitForRequest 176–81, 186–7, 215–16
- uses-a class relationships 57, 64, 494, 516–17
- UTC 210
- UTF... 109, 124–5, 726–7, 732
- utility classes 21
- UUIDs 625–6
- v flag 376
- Val 122–3
- validation 447, 452–5, 501–2, 509–14, 528–9, 563–4
- ValidDecoder 691
- VENDORID 317–18
- VeriSign 278
- vertical justification, drawing basics 485–6
- VerticalPixelsToTwips 542–3
- VerticalTwipsToPixels 542–5
- VFAT file system 205–6, 210
- video 587, 651–713
 - see also* multimedia...
- clip APIs 655
- controller plug-ins 679–82
- playback 655, 672–4, 680–2
- recording 655, 674–5, 680–2, 699
- view architecture 344, 355, 391–421
 - concepts 391–421
- View context area 3–4
- view server
 - see also* application UI
- concepts 392–420
- ViewActivatedL 397, 398, 408–11
- ViewConstructFrom-ResourceL 325–6, 357–64, 412–13, 466–7

- ViewConstructL 325-6, 357-66, 397-8, 408-13
- ViewDeactivatedL 397, 398, 408-11
- ViewFinderFrameReady 703
- viewfinders, cameras 702-3
- ViewId 325-6, 397-8, 408-11
- ViewRecorder 674-5
- views 312-13, 323-7, 333, 336-7, 344-51, 356-64, 391-421, 426-7, 483-6, 493-5, 503-14, 533-56, 566-9
 - see also* CCoeControl
 - activation processes 396, 397, 398, 407-11, 425-9
 - classes 312-14, 391-421
 - concepts 312-13, 323-7, 336-7, 344-51, 356-64, 391-421, 426-7, 483-6, 493-5, 499-502, 566-9
 - construction 325-6, 357-66, 397-8, 408-13, 425-31, 444
 - controls 324-7, 350-1, 356-7, 359-60, 400-6, 412-14, 423-9, 483-6, 499-502, 503-14, 566-9
 - creation 406-8
 - deactivation processes 397, 398, 407-11, 509-11, 521, 524-5
 - default views 396, 398, 421
 - definition 391
 - generic derivation 392-3
 - header files 314
 - history views 400-14, 416-20
 - IDs 325-6, 397-8, 408-11, 420
 - label controls 400-6, 412-14
 - management 406-8
 - MCoeView 314-30, 392, 397-8, 408-20
 - noughts-and-crosses application 344-51, 356-64, 398-420, 426-9, 497, 499-502, 507-9
 - observer interfaces 396, 442-4, 474-5, 564-6
 - pointers 406-8, 414
 - portable views 395
 - registration 392-6, 407, 421
 - screen-sharing processes 502-14
 - virtual destructors, CBase 66-7, 79-80, 85
 - virtual functions 31-3, 49-50, 54-5, 57-60, 163-4, 408-11, 490-2, 508-9, 557, 588-9
 - virtual keyboards 433, 568-9
 - virtual-memory environment, concepts 23, 24-5, 71
 - viruses 272
 - VisitAttributeValueL 631-2
 - void 42, 43, 50, 53-4, 69-70, 76, 90, 114, 123-4, 134-7, 490
 - Volume 207-8
 - volumes 207-8
- w32std.h 38, 439, 576-8
- Waiter 153
- WaitForAnyRequest 170-1, 176-81, 183, 186-7
- WaitForRequest 176-81, 186-7, 215-16
- waiting/non-waiting aspects, dialogs 457-8
- wav files 653
- WBXML 586, 602
- web browsing
 - see also* Internet
 - device/size-independent graphics 555-6
- websites 555-6, 755-9
 - see also* Internet
- while loop 122, 745, 748-9
- Wi-Fi 18
- wide/narrow descriptors
 - see also* descriptors
 - concepts 108-9, 124-8
 - conversions 124-8
- wild cards 206
- Win32 APIs 283-4
 - see also* emulator
- Window 505-6, 509-10
- window groups 576-82
- window server
 - see also* servers; WSERV
 - backed-up windows 451-5, 505-6, 515-21
 - classes provided 576-82
 - client-server architecture 504-14
 - concepts 30-4, 38-9, 173-4, 293-4, 424, 432-3, 439-42, 445-51, 496-8, 503-17, 526-9, 557-82
 - controls 424, 432-3, 439-42, 445-51, 496-8, 503-14, 557-82
 - drawing 445-51, 496-8, 503-14, 526-9
 - features added 526-9
 - flicker-free drawing 514, 527-9, 546-9
 - interaction graphics 557-82
 - redrawn windows 445-51, 496-8, 501-4, 511-14, 526-9
 - screen-sharing processes 503-14
 - special effects 521-6
 - window groups 576-82
- window server logging keys, emulator debugging 293-4
- window-owning controls 424, 457, 504-9, 514-15, 516-18
 - see also* controls
- windows 424, 445-51, 457, 483-6, 492-8, 501-7, 511-14, 515-21, 526-9, 531-2, 568, 577-82
 - see also* controls; screens
 - complexity issues 509-14
 - CONE 503-4
 - overlapping windows 503-15
 - redrawn windows 445-51, 492-8, 501-4, 511-14, 526-9, 531-2, 568, 577-82
 - screen-sharing processes 502-14
 - standard windows 579-82

- transparent windows 522,
525–6, 528
- types 578–82
- wins 295
- winscw 12–14, 46, 284–6,
290–1, 295, 300, 379, 383,
400
- WinSock 289
- WORD 368–9, 387, 470–2, 478
- writable static data optimization,
DLLs 29, 44
- Write 50, 110, 123–4, 212–15,
222–4, 250–4, 613–15, 620,
627–8
- write operations, files 211–14
- WriteDeviceData capabilities
276, 280
- WriteInt... 218–19, 222–4,
236–7, 338–44
- WriteL 215–16, 222–4
- WriteReal... 222–4
- WriteUInt... 220–4
- WriteUserData capabilities 276,
280, 644–5, 648–9
- ws32 38
- WsBuffer 510–14
- WSERV 38
 - see also* window server
- x86 processor 283–4
- XHTML 586
- XML 586, 602, 716
- XScale 18
- Yahoo! 588
- z drive 18, 204–5, 284–6, 295,
307
 - see also* ROM
- Zero 145
- ZeroTerminate 145
- ZIP files
 - EZLIB 585–6
 - Symbian Signed submissions
277–8
- zlib compression library 585–6
 - see also* EZLIB
- ZoomFactor 542–3
- zooming 530–56