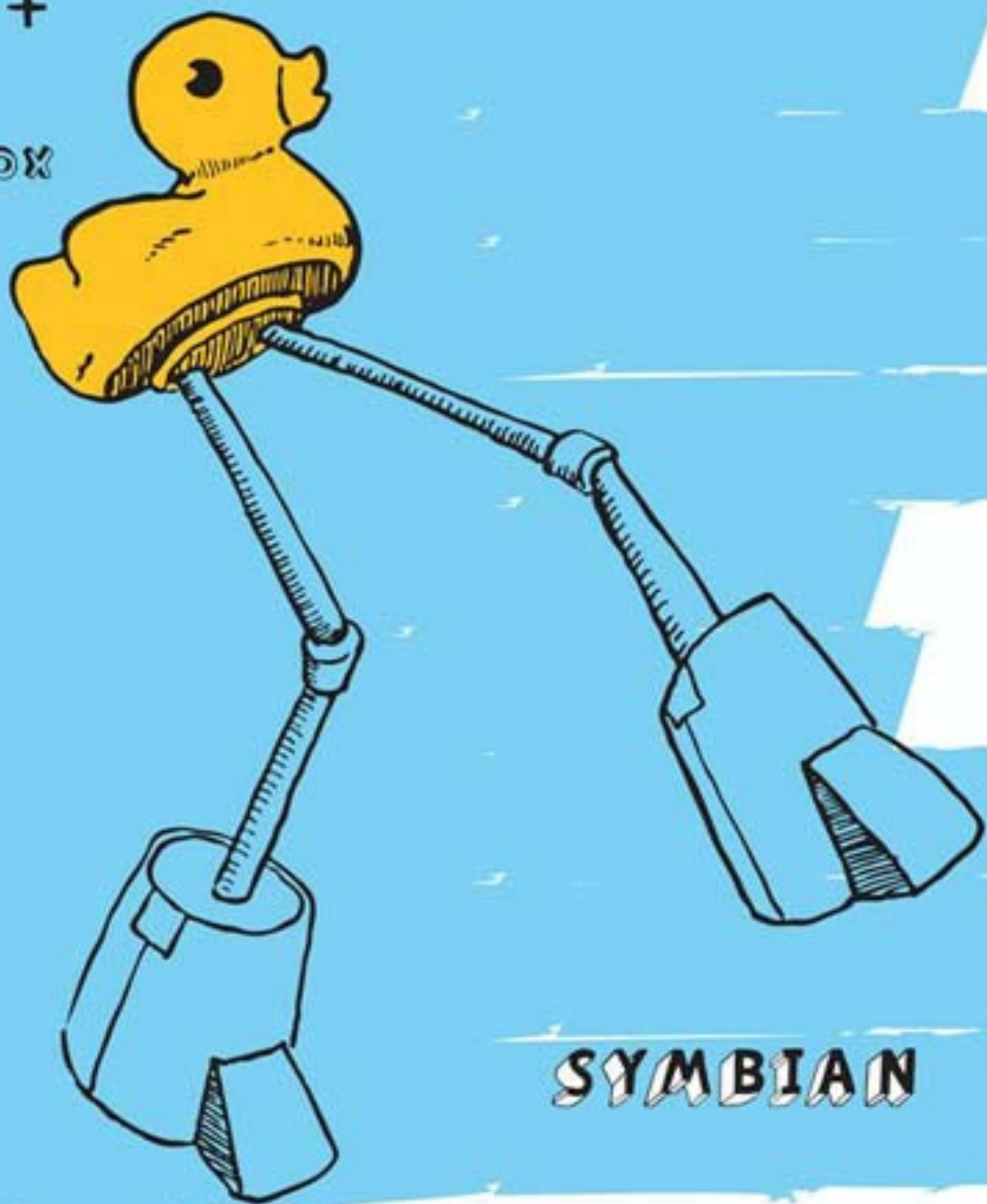


Porting to the Symbian Platform

Open Mobile Development
in C/C++

Mark Wilcox



 WILEY

SYMBIAN

Porting to the Symbian Platform

Open Mobile Development in C/C++

Porting to the Symbian Platform

Open Mobile Development in C/C++

Lead Author
Mark Wilcox

With
**Lauri Aalto, Will Bamberg, David Caabeiro, Ivan Litovski,
Gábor Morvay, Lucian Piros, Jo Stichbury, Paul Todd,
Gábor Torok, Vinod Vijayarajan**

Reviewed by
**Nicholas Addo, Michael Aubert, Serage Bet-el-mat, Madhavan
Bhattathiri, Robert Cliff, David Crookes, Biswajeet Dash, Craig
Heath, John Imhofe, Mark Jacobs, Erik Jacobson, Pekka Kosonen,
Ian McDowall, Lucian Piros, Mangesh Pradhan, Steve Rawlings,
Salvatore Rinaldo, Espen Riskedal, Andy Sizer, Colin Ward**

Head of Technical Communications, Symbian
Jo Stichbury

Managing Editor, Symbian
Satu Dahl

This edition first published 2009
© 2009 John & Wiley Sons Ltd

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ,
United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

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 or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

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

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.

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

ISBN 978-0-470-74419-2

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

Contents

About this Book	ix
Author Biographies	xv
Author's Acknowledgements	xxi
Symbian Acknowledgements	xxiii
1 Introduction	1
1.1 What Is Porting?	2
1.2 What Is Portability?	2
1.3 Why Port to Mobile Platforms?	3
1.4 Why Get Interested Now?	6
1.5 Why Port to the Symbian Platform?	17
2 The Porting Process	23
2.1 Choosing a Project	24
2.2 Analyzing the Code	28
2.3 Re-architecting	29
2.4 Setting Up the Development Environment	31
2.5 Integrating with the Symbian Build System	34
2.6 Compiling	39
2.7 Fixing Problems	41
2.8 Running and Testing	44

2.9	Debugging	45
2.10	Re-integrating	46
2.11	Summary	47
3	Symbian Platform Fundamentals	49
3.1	In the Beginning	49
3.2	Naming Guidelines and Code Conventions	50
3.3	Data Handling	55
3.4	String Handling: Descriptors	59
3.5	Error Handling and Memory Management	71
3.6	Event-Driven Programming	93
3.7	Writeable Static Data	99
3.8	Multiple Inheritance	100
3.9	Summary	102
4	Standard APIs on the Symbian Platform	103
4.1	P.I.P.S. Is POSIX on Symbian OS	104
4.2	Open C	106
4.3	The STLport, uSTL and Open C++	107
4.4	Which Version of Symbian OS?	108
4.5	How to Use the APIs	109
4.6	Examples: SoundTouch and SoundStretch	114
4.7	Known Limitations, Issues and Workarounds	120
4.8	Summary	131
5	Writing Hybrid Code	133
5.1	Popular APIs You Can't Use Directly	134
5.2	How to Create a Hybrid Port	140
5.3	Example: Guitune	145
5.4	Summary	157
6	Other Port Enablers	159
6.1	Real-time Graphics and Audio Libraries	159
6.2	Simple DirectMedia Layer	168
6.3	OpenKODE	169
6.4	Qt	177
6.5	Summary	187
7	Porting from Mobile Linux	189
7.1	Major Players in the Mobile Linux Space	190
7.2	Porting from Linux to Symbian	195
7.3	Summary	206

8	Porting from Microsoft Windows	207
8.1	Architecture Comparison	208
8.2	Application Compatibility	209
8.3	Development Languages and SDKs	210
8.4	SDKs and APIs	214
8.5	Porting an Application	216
8.6	Windows-specific Issues	220
8.7	Signing and Security	233
8.8	Porting from C# and .NET	236
8.9	Summary	240
9	Porting from Other Mobile Platforms	243
9.1	Android	244
9.2	BREW	253
9.3	iPhone OS	254
9.4	Summary	270
10	Porting a Simple Application	271
10.1	Selecting a Project	271
10.2	Analyzing the Code	273
10.3	Setting Up the Development Environment	275
10.4	Integrating with the Symbian Build System	276
10.5	Getting It to Compile	277
10.6	Getting It to Work	277
10.7	Extensions Specific to Mobile Devices	281
10.8	Deploying and Testing on Target Hardware	286
10.9	Re-integrating	287
10.10	Summary	287
11	Porting Middleware	289
11.1	GDAL	290
11.2	Qt	301
11.3	Summary	307
12	Porting a Complex Application	309
12.1	Selecting a Project	310
12.2	Analyzing the Code	311
12.3	Re-architecting	312
12.4	Setting Up the Development Environment	315
12.5	Integrating with the Symbian Build System	315
12.6	Getting It to Compile	318
12.7	Re-writing the User Interface	325

12.8	Testing and Debugging	329
12.9	Re-integrating	330
12.10	Summary	330
13	The P.I.P.S. Architecture	333
13.1	The Glue Code	334
13.2	The Core Libraries	335
13.3	The Backend	336
13.4	Emulator Writeable Static Data Support	343
13.5	Summary	345
14	Security Models	347
14.1	The Capability Model	348
14.2	Process Identity	356
14.3	Data Caging	357
14.4	Code-Signing and Certification	359
14.5	Certification and Platform Security	361
14.6	Development Code	366
14.7	Tool Support	368
14.8	Symbian Platform Security Compared with Other Models	369
15	Writing Portable Code and Maintaining Ports	375
15.1	Recognizing Portable Code	376
15.2	Design Strategies and Patterns	377
15.3	Strategies for Maximizing the Number of Portable Modules	386
15.4	Configuration Management	392
15.5	Summary	395
Appendix A	Techniques for Out-of-Memory Testing	397
References		409
Index		411

About this Book

I originally started writing this book as part of the Symbian Press Technology series back in May 2008. Since that time, a lot has changed in the mobile software world, particularly for those working with Symbian devices. The book you now hold in your hands is published by the Symbian Foundation using a rather more informal approach.

Determining the content of a technical book in a fast-moving industry is never easy but when organizations and strategies are changing faster than the code, it becomes extremely difficult. Because of that, several sections of this book have been revised multiple times in a valiant effort to bring the most relevant and up-to-date information to you, the reader. I sincerely hope we've succeeded. Nevertheless, we anticipate further changes, if only to some of the URLs we reference. For that reason, you can find a wiki page for this book at ***developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform***, which will also be used to host the example code accompanying the book and to record errata reported against this version of the text.

What Is Covered?

The primary topic of this book is, as the title suggests, porting software to the Symbian platform. Further explanation of what exactly we mean by 'porting' and why you'd want to port to the Symbian platform can be found in Chapter 1. Before getting that far though, you might want to know what the Symbian platform is.

Terminology and Version Numbering

In June 2008, Nokia announced its intention to acquire the shares in Symbian Ltd that it didn't already own and create a nonprofit organization, called the Symbian Foundation. By donating the software assets (formerly known as Symbian OS) of Symbian Ltd, as well as its own S60 platform, to this nonprofit entity, an open source and royalty-free mobile software platform was created for wider industry and community collaboration and development.

The acquisition of Symbian Ltd was approved in December 2008 and the first release of the Symbian platform is expected in the second quarter of 2009, shortly after this manuscript goes to the publisher.

For that reason, most of the material in this book applies to versions of Symbian code released before the creation of the Symbian Foundation. Table 1 maps the naming and numbering of the releases of the Symbian platform, Symbian OS and S60 covered in this book. Symbian^1 is a renaming of S60 5th Edition and Symbian OS v9.4 to denote the starting point for the Symbian platform releases. The first version of the platform released by the Symbian Foundation independently is Symbian^2 (pronounced 'Symbian two').

Table 1 Symbian Platform Releases

Symbian platform	S60 platform	Symbian OS
n/a	3rd Edition	v9.1
n/a	3rd Edition, Feature Pack 1	v9.2
n/a	3rd Edition, Feature Pack 2	v9.3
Symbian^1	5th Edition	v9.4
Symbian^2	5th Edition, Feature Pack 1	v9.4 (+ some features back-ported from v9.5)
Symbian^3	n/a ^a	v9.5 (and community contributions)
Symbian^4	n/a	n/a ^b

^a When the Symbian Foundation is fully operational, Nokia is expected to stop marketing S60 as an independent brand.

^b Although Symbian Ltd had a roadmap beyond Symbian OS v9.5, the content of future releases is unlikely to resemble the previous plans very closely.

Core Material

The core material of this book explains how to go about porting your code to the Symbian platform. Descriptions of the various technologies and APIs available to help you can be found in Chapters 2 to 6:

- Chapter 2 explains how to increase the chances of your porting projects succeeding. It includes details of common issues that will be faced when porting to the Symbian platform.
- Chapter 3 describes key features of the native programming environment on the Symbian platform, Symbian C++, and compares them with some standard C/C++ alternatives that can be used.
- Chapter 4 provides details of the APIs that are available for standard C/C++ programming on the platform and how you can use them.
- Chapter 5 discusses some popular APIs you might be using on other platforms that aren't available the Symbian platform and what alternatives are available. In many cases, this can mean that you need to write some native Symbian C++ code, so I also explain how you can write hybrid code, mixing Symbian C++ with standard C/C++.
- Chapter 6 explains the other APIs that are available on, or coming soon to, the Symbian platform that could make the task of porting easier. These APIs include cross-platform standards being developed by industry consortiums and the popular Qt application framework.

Platform-Specific Porting Guides

Following on from the core material, Chapters 7 to 9 focus on the details of porting from some other common platforms. Chapter 7 describes a variety of mobile Linux variants, the APIs they provide, and how to port code from them to the Symbian platform. Chapter 8 focuses on porting from Microsoft Windows platforms, both desktop and mobile, and also includes a section on porting code written for the .NET Compact Framework. Chapter 9 covers three mobile platforms which have less in common with Symbian: Android, BREW and iPhone. The chapter attempts to provide a mental map for migrating from these platforms as well as advising on the most appropriate target APIs for porting projects in each case.

Porting Examples

Code examples are provided throughout the book wherever appropriate, but Chapters 10 to 12 focus on larger porting projects for which full source code is available. Chapter 10 describes an extremely simple application written using Qt that can be ported with virtually no code changes at all. Chapter 11 describes two middleware ports that were in progress at the time of writing, the Geospatial Data Abstraction Library (GDAL) and the Qt cross-platform application framework itself. The chapter explains the high-level technical details of the ports and also discusses the issues that the ports highlight which should be common to most

middleware porting efforts. Chapter 12 introduces a more complex open source mobile application that is being ported from Windows Mobile to the Symbian platform, targeting the Qt APIs as far as possible.

Advanced Topics

The final three chapters of the book add useful information for those who have become comfortable with the basic porting process and would like some more advanced material. Chapter 13 explains the architecture of the P.I.P.S. layer which provides POSIX and standard C support on the Symbian platform. Chapter 14 provides a comprehensive guide to Symbian platform security and compares it with the security models in place on other mobile platforms. Finally, Chapter 15 is intended to provide advice and best practice for those attempting to write and maintain a common code base across multiple platforms.

Who Is this Book For?

Chapter 1 of this book is intended to be readable by a non-technical audience; it provides an introduction for those trying to understand the current state of available programming environments on the Symbian platform in comparison with those available on other mobile platforms.

The rest of the book was written with two primary audiences in mind:

- Developers familiar with other platforms wanting to port their code to the Symbian platform
- Existing Symbian developers wanting to port code written for other platforms to include in their projects.

However, there is a third audience who may also find this book extremely useful: experienced standard C/C++ and POSIX developers who would like to start developing for the Symbian platform. There are many open source developers, particularly for Linux-based platforms, who are interested in mobile software. These developers should find that the programming environment on the Symbian platform is becoming increasingly familiar and this book provides a guide that should enable them to leverage their existing knowledge to target Symbian-based devices.

The Future of the Symbian Platform

Just before the final draft of this book went to the publisher, Nokia submitted a proposal to the Symbian Foundation about their plans for the

future of the platform's user interface and programming environment. The essence of this proposal was that the current application framework for the platform should be replaced with Qt and a new user interface built on top of it. This proposal had not yet been accepted at the time of writing but, assuming it is, this implies that the material covered in most of this book is more important than ever, not just for porting but also for those wanting to write new C++ applications for Symbian. At the same time, some of the material focused on the creation of user interfaces with the current application framework is likely to lose relevance over time. You are advised to check the latest platform roadmaps at ***developer.symbian.org*** and select the appropriate technologies from this book accordingly.

Author Biographies

Mark Wilcox

Mark has been involved in handset development projects for Ericsson, Fujitsu, Nokia, Panasonic and Samsung. He has worked on everything from a GPRS stack, device drivers and a power management server to messaging applications and the code that draws the soft keys. While working as a software architect for Nokia's Multimedia business unit, developing their flagship Nseries products, Mark developed his interest in the multimedia capabilities of Symbian OS. He drew on this experience when co-authoring *Multimedia on Symbian OS: Inside the Convergence Device* for Symbian Press in 2008. He became interested in the open source development model while working on Linux-based, set-top-box products and has been actively attempting to improve the environment for free and open source software on the Symbian platform ever since. He has successfully ported a number of application and middleware projects to the Symbian platform, both open and closed source.

Mark became an Accredited Symbian Developer and a Forum Nokia Champion in 2007. In 2009, he joined the technical services team at the Symbian Foundation, dedicated to helping developers get the most out of the platform.

Lauri Aalto

Lauri started programming well before the age of 10. For the past 15 years, he has been programming professionally. His projects have included S60 platform development, independent product development, open source

middleware, telecommunications operator services, software engineering tools, academic research and teaching. Currently he is a project manager and partner at Boogie Software, working as a 'hired gun' for Nokia. In his work roles, he makes sure he is not stuck in management only but can also keep his hands dirty writing pragmatic production software. Lauri is a Forum Nokia Champion founding member. He holds a degree in mathematics, computer science and software engineering from the University of Oulu.

Will Bamberg

Will joined Symbian in 1997 and worked there for almost six years, for most of that time in the security team as an engineer and then as an architect. He had some involvement in the initial design and deployment of platform security during that time. Between 2003 and 2005, he worked for Nokia in Vancouver, mostly on telephony. Since then he has worked on Symbian OS for an independent contractor in a variety of domains including text, internationalization and graphics.

David Caabeiro

David has been working with Symbian OS since 2002. Before moving into mobile technologies, he spent some years working on different platforms, mainly using C and C++. He currently provides assistance and services to different companies, working on a range of technologies including telephony, networking, multimedia and porting. He can usually be found lurking around Symbian-related discussion boards.

In his spare time, David enjoys music, photography and simply escaping to some remote place in search of peace.

Ivan Litovski

Ivan joined Symbian in 2002, working first in the Java Team and later with the System Libraries, Persistent Data Services and Product Creation Tools teams. Ivan has 11 years of professional experience with Java, embedded software, information security, networking, scalable servers and Internet protocols. He enjoys solving difficult problems and research and development. He has authored several patents and papers in peer-reviewed, international journals. He holds a BSc in electronics and telecommunications from the University of Nis, Serbia and is an Accredited Symbian Developer.

Gábor Morvay

In 1999, after graduating as a mathematics and physics teacher from Eötvös Loránd University of Science and then receiving a BSc in computer science, Gábor joined a consultancy company, where he had the chance to become acquainted with EPOC and was involved in the development of several applications that made it to the ROM of early phones based on Symbian OS. He had several roles starting as a developer and becoming a technical lead and project manager of software projects targeting Symbian OS devices. At Agil Eight, his present company, while remaining a Symbian C++ enthusiast, he has had the opportunity to learn Objective-C and be part of an iPhone application development team. During his career, he has worked in several technological areas, such as networking, security, telephony and image processing. When not working, he can often be found running on Margaret Island in Budapest.

Lucian Piros

Lucian Piros has been working in telecommunications and embedded and mobile computing for over 10 years, after completing a mathematics and computer science degree at Babeş-Bolyai University. He has worked in the Symbian ecosystem since 2006, in the PIM team of Symbian as well as for the Contacts team in Nokia. He provides extensive technical support to Symbian developers worldwide, through SDN/SDN++ forums, and contributed to *Symbian OS C++ for Mobile Phones Volume 3*. Lucian became an SDN Ambassador in 2008.

Jo Stichbury

Jo has worked with the Symbian ecosystem for over a decade, for Symbian, Nokia, Sony Ericsson and Advansys. She is the author of *Symbian OS Explained* (2004), co-author of the *ASD Primer* (2006) and lead author of *Games on Symbian OS* (2008). Noting a biennial pattern to her publications, she eagerly awaits a commission for publication in 2010, but in the meantime is spending her time working for the Symbian Foundation as leader of the Technical Communications team.

Paul Todd

Paul's first experience with Symbian was the very old Psion LZ64, which he used to interface to surveying instruments where the collected data

was exported to Autocad and used to generate maps for quantity surveys. He emigrated to England in the early 1990s, where he did some work on developing quantity-surveying tools until he got involved with the first enterprise synchronization tool for PDAs (ASL Connect).

After a number of years of work on Windows Server and Windows Mobile/CE, he was finally dragged back into the world of Symbian development. Currently, he designs and develops enterprise class software for PIM Data Management (Onebridge) and device management (Afaria) for Sybase. He is an active Forum Nokia champion and contributor of a number of Symbian papers.

Paul's areas of interest include ways that development can be made both faster and more reliable so that end-user products (especially when developed by teams) can provide a better experience out of the box. He is a keen advocate of .NET and co-founder of two startups, next-mobileweb.com and seqpoint.com.

Gábor Torok

Gábor has been involved in mobile software development since 2000, when he started to contribute to Nokia S60 platform development. He filled many roles, such as software engineer, test and asset manager and software architect, which eventually gave him a good overview of development in this environment.

His instinct to share knowledge with the community inspired him to actively contribute to Symbian discussion forums, which was rewarded by Nokia by electing him as one of the founder members of the Forum Nokia Champion program in 2006. His further contributions were so valuable that he's been re-elected three times since then.

Nevertheless, Gábor is interested in other parts of the mobile industry, too. He started his own blog in 2007 and got lots of positive feedback from people commenting on his articles. He also decided to explore other mobile platforms, such as Android, so that his view is not limited to a single ecosystem, which eventually resulted in his involvement in writing some parts of this book.

Vinod Vijayarajan

Vinod is the Technology Architect of Generic OS Services, a module in S60 that includes P.I.P.S. and Open C/C++. He has been associated with P.I.P.S. since its conception and helped it grow from a fragile, proof-of-concept port to a solid, stable foundation for technologies such as Qt and Python on S60. As lead and architect, he designed and implemented several key components in the libraries, drove releases,

ran port-assistance programs, engaged with end users and application developers and helped permeate first-hand knowledge of working with Open Source code, communities and practices in Symbian (now Nokia).

Before he joined Symbian, he was employed by two startups, working on Bluetooth profiles and multimedia frameworks for embedded systems.

When not working, he loves to read, travel and dabble in Python.

Author's Acknowledgements

There are a lot of people involved in the creation of a book but none so important as those that support the author and make sure he eats and sleeps at the right times as deadlines approach. So I'd like to start by thanking my wife Lara, who tirelessly supported me through a second book without a break after the first – that's it for now, I promise!

Before I was even aware of this book project there were people with the foresight to plan and commission it, so thanks are due to Antony Edwards and Martin Tasker for their part in the original vision for the book and to Jo Stichbury for giving me the opportunity to write it. Jo also served as co-author, editor and mentor, all of which she did with her usual high levels of skill, efficiency and good humor; without her there would be no book. I was also fortunate enough to work with an extremely talented team of authors, so thanks to David, the two Gábors, Ivan, Lauri, Lucian, Paul, Vinod and Will. Special mentions are due to David, Ivan and both Gábors for stepping in to fill the places of those whose other commitments had to take priority with very tight deadlines. I owe an extra thanks to Lucian for all his help with the example code projects too.

A lot of people reviewed drafts of various chapters, too many to name individually here again, but I thank them all. The reviewer that deserves singling out is Lauri Aalto. Lauri is one of the true Symbian gurus and I was very pleased when he agreed to review multiple chapters as well as contribute an appendix. His natural perfectionist tendencies have significantly improved the quality of the text in several places and we should all thank him for it.

Finally, writing a book is only half the story. An amazing amount of work goes into coordinating authors and reviewers and then turning the hundreds of pages of electronically stored text and graphics into an actual book. For this great effort, thanks are due to Satu, formerly of Symbian Press and now a colleague at the Symbian Foundation, and all of the team at Wiley.

Symbian Acknowledgements

Symbian would like to thank Mark for delivering his excellent manuscript to us on time and in a great shape. We'd also like to thank all the co-authors and reviewers of this book for their hard work and dedication – this project has been a real pleasure to work on. Thanks also to the ever-supportive team at Wiley and to Shena Deuchars, our copy editor, for her light touch and diligent edits.

1

Introduction

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

This is a book about making software work on the most widely used open mobile device operating system on the planet, in other words, it is about porting to the Symbian platform. It is also a book that describes a new paradigm for the development of native applications¹ on mobile devices, creating and using code that is much more portable than has been possible historically. If that doesn't already sound like something that's worth investing your time and effort in, then hopefully this chapter will convince you.

Before discussing this new era of mobile software development we need to answer a few basic questions:

- What is porting?
- What is portability?
- Why port to mobile platforms?
- Why get interested now?
- Why port to the Symbian platform?

All of this and more will be revealed in the following sections. Along the way, I'll introduce some of the core technologies that will enable the next generation of native mobile applications. Using these technologies is the subject of the bulk of this book. In answering these questions, I'll also look to the future to give a hint at the directions your projects could go in after you've successfully ported them to the Symbian platform.

¹ Native applications are delivered in a binary format which the hardware understands directly, rather than being interpreted by some intermediate software layer.

1.1 What Is Porting?

Taking the established convention of the Internet age, and consulting Wikipedia as our starting point, we define porting as follows:

*In computer science, **porting** is the process of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed (e.g. different CPU, operating system, or third party library). The term is also used in a general way to refer to the changing of software/hardware to make them usable in different environments.*

en.wikipedia.org/wiki/Porting

In other words, ‘porting’ is making software run on different hardware or operating systems, or using different libraries, or any combination of these. There are several examples of porting projects in this book which cover all of these differences. For example, when porting a typical application developed for the Microsoft Windows desktop to the Symbian platform, it will need to run on a different processor (ARM rather than x86 architecture) and take into account differences between the operating systems and user interface libraries, in order to work on a Symbian phone.

1.2 What Is Portability?

Closely tied to the process and practice of porting software is the concept of software portability. The Wikipedia page also states:

Software is portable when the cost of porting it to a new platform is less than the cost of writing it from scratch. The lower the cost of porting software, relative to its implementation cost, the more portable it is said to be.

How portable a piece of code is will determine how much effort is involved in porting it to another environment. This is not a fixed quantity for a given piece of code but rather a function of the similarity between the original environment and any desired target environment. For example, an application written for an Apple Macintosh may be fairly easy to port to an iPhone but very difficult to port to a Windows Mobile device. However, it is possible for code to be written in such a way that it is portable to a wide range of platforms. Some excellent advice for this practice can be found in Chapter 15. Additionally, the ease of porting to a particular platform can change significantly if the software environment on that platform is modified. This is the case with the Symbian platform, thanks to the addition of several industry standard and other cross-platform programming interfaces in the last couple of

years, as described in Chapters 4 and 6. Such a change to the software environment can make a massive difference to the case for investing in porting your software, whether that's a financial investment or the investment of your time.

1.3 Why Port to Mobile Platforms?

Having an understanding of what porting is, we turn our attention to why you'd want to port to mobile platforms at all. To address this issue, I have a few different and complementary answers. Motivation is a very individual thing so there are different reasons for targeting mobile devices depending on your perspective. What follows is an attempt to categorize a few common perspectives.

1.3.1 The Marketing Angle

Consider that the global installed base of PCs is just over one billion units² and that more mobile phones than that are now sold every single year. In fact, in early 2008 the number of mobile subscriptions exceeded half the global population of 6.7 billion. Although this figure included a fairly large number of people with multiple subscriptions, as you read this it's almost certain that more than half of the people on Earth have a mobile phone (see Figures 1.1–1.3³). What's more, that percentage is growing rapidly.

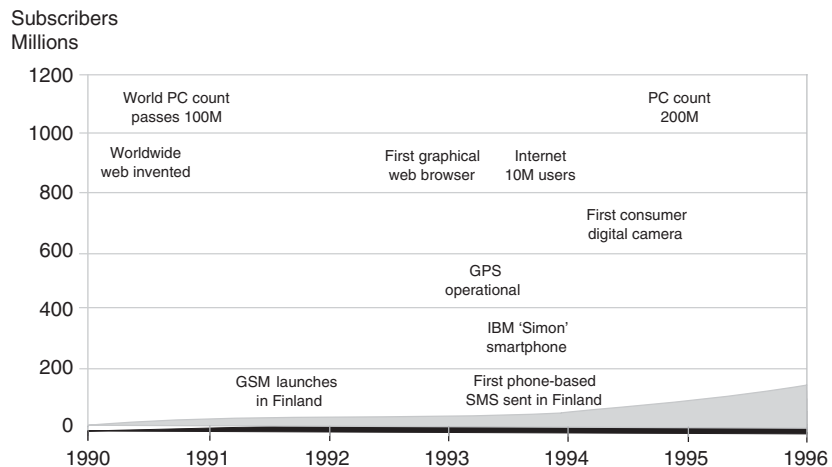


Figure 1.1 Growth of mobile phone adoption and related technology 1990–96

² Source: Gartner, www.gartner.com/DisplayDocument?ref=g_search&id=644708.

³ Diagrams adapted and reprinted with permission from Tomi Ahonen Consulting.

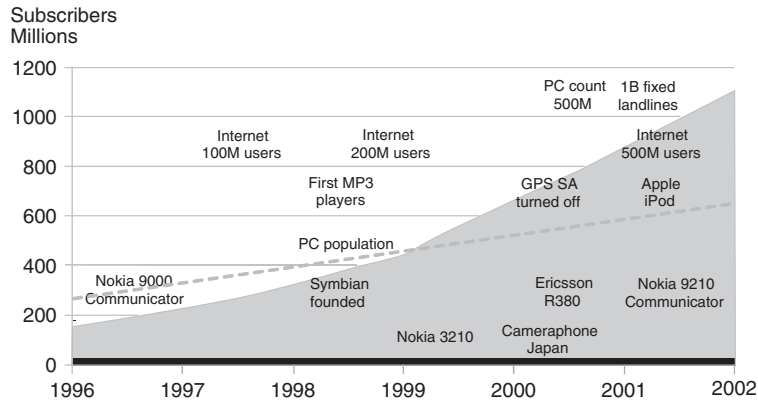


Figure 1.2 Growth of mobile phone adoption and related technology 1996–02

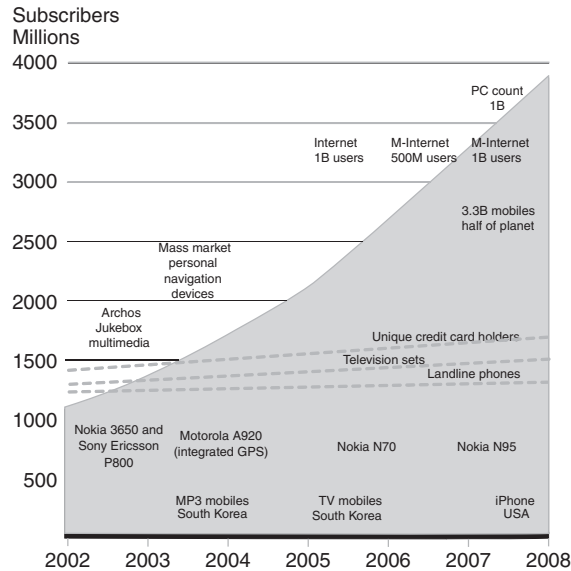


Figure 1.3 Growth of mobile phone adoption and related technology 2002–08

Estimates from Wireless Intelligence⁴ (which have historically been far too conservative) suggest that, by the end of 2010, 70% of the global population will be carrying a mobile phone. Although a lot of the recent growth has been in low-end handsets that can't run complex third-party software, reducing hardware costs enable device manufacturers to push more advanced operating systems into mid-range models. Market analysts

⁴ www.wirelessintelligence.com.

Gartner predicted smartphone sales of 190 million units in 2008 alone, with a forecast for over 700 million units in 2012, accounting for 65% of all handset sales.⁵ Even though the crisis in the global financial system is likely to reduce the demand from the previously expected level over the next few years, the long-term picture is still intact. A significant fraction of all human beings will soon have a powerful personal computing and communications device with them at all times.

1.3.2 The Hacker's Reason

Just to be clear, I'm not talking about people who crack security systems here but those who 'delight in having an intimate understanding of the internal workings of a system, computers and computer networks in particular'.⁶ In this sense, the word 'hacker' is generally used as a mark of respect for a particularly skillful and knowledgeable software engineer or programmer. There are a lot of reasons to get excited about mobile technology, which has a much greater level of variety and rate of development than desktop systems. Specific to porting software to mobile platforms, the following points are worthy of consideration:

- Porting can be a good way to start learning a new platform.
- Existing projects can reach thousands or even millions more users.
- There's a great opportunity for new developers to build a reputation.
- New avenues for innovation are opened by combining existing code with mobile-specific features and data.

Also, porting can often have an excellent reward-to-effort ratio, often requiring surprisingly little effort to get some useful results. The recent shift towards open source platforms in the mobile industry, including the recently formed Symbian Foundation, provides an excellent environment for the hacker mentality to thrive.

1.3.3 The Geek's Reason

Many (but not all) hackers are also geeks. However, the geek's motivation for porting software to a mobile platform is so that they can use it on that platform, rather than the enjoyment or understanding they gain from the porting task itself. For some people, the idea of having certain favorite applications or services with them at all times is just going to be too tempting to resist. In a world where an increasing number of people are becoming addicted to text messaging, often sending more than

⁵ www.cellular-news.com/story/33277.php.

⁶ tools.ietf.org/html/rfc1392#page-21.

100 messages in a day, who knows what this trend may spread to next? Some mobile network operators are already offering unlimited access to mainstream social networking clients such as Facebook and MySpace on pre-paid tariffs. There's even a mobile client for the virtual world, Second Life. Geeks with a taste for slightly less popular applications, social networks or virtual worlds may not want to wait for someone else to give them access from their mobile. With ever-growing levels of Internet and technology addiction in various forms, the opportunity to have 24/7 access may not be viewed by everyone as a good thing. Then again, it might just be the case that having permanent availability of access reduces the compulsion to over-indulge and enables a better balance. Being able to carry their virtual worlds with them might just encourage people to spend more time in the real world and perhaps even combine the best of both.

1.4 Why Get Interested Now?

Similar arguments to those in the previous section could have been made at any time in the last few years. My assertion is that the smartphone concept has now reached the point in its technical evolution where it's ready for mass-market adoption by both the device creation industry and the consumer. Ever since we first started recording and processing information in a digital form there has been an ongoing trend towards a globally connected ubiquitous computing environment. Computing technology started out in room-sized isolated units, shared by large numbers of researchers. Since that time it has become increasingly personal, portable and connected. We'll consider the consequences of this long-term shift at the end of this section, but first a little history.

1.4.1 A Brief History of Digital Convergence

Every story has to start somewhere; we'll start this one just over 30 years ago. In 1978, there were some important firsts that, with hindsight, could be viewed as the beginning of the current era of global digital convergence. In the telecommunications world, the first TCP/IP networks were being built and tested and at the same time Bell Labs launched a trial of the first commercial cellular network.⁷ This was also the year that the first GPS satellite was launched (although it wasn't much use on its own) and the first entirely digitally recorded popular music album⁸ was produced. A few years later, in 1981, Sony released the Mavica, the world's first commercial electronic still camera, and IBM released their first PC.

⁷ Although Finland had the ARP network from 1971, this is not considered a true cellular network as there was no automatic handover between cells.

⁸ For the curious, it was Ry Cooder's *Bop Till You Drop*.

It is often said that we tend to overestimate the short-term impact of a new technology and underestimate the long-term implications. This tendency of individuals and society as a whole is sometimes referred to as macro-myopia. I believe this occurs because technological innovation opens up the potential for new markets but also inevitably has teething problems. The market potential encourages several competing solutions to the early problems, resulting in a fragmented market and consumer confusion. It's not until winners emerge, or competitors collaborate to develop a common standard, that consumer adoption increases and mass-market pricing can be achieved. In many cases, further improvements in technology are required before a compelling end-user experience is possible. The converging technologies provide examples of this process.

The original Sony Mavica was not a true digital camera but really just a video camera that could take freeze frames. The product demonstrated the potential for the 'filmless' camera but it then took a further five years for Kodak to develop the first megapixel sensor, capable of producing a standard 5"x7" photo-quality print. From there, Kodak collaborated with others, although it was still a further five years before the first professional digital camera (the Nikon F-3) was released, closely followed by the Photo CD standard. The first digital cameras for the consumer market didn't arrive for another three years, finally hitting the shops in 1994. Fifteen years later, traditional film is hardly used at all.

Similarly, in the digital audio market, the first portable player prototype was developed in 1979. Unfortunately it could only store about three and a half minutes of music and so it was never produced commercially. Consumers had to wait until 1998 for the first mass-produced digital audio players. The inventions of cheaper non-volatile storage devices and digital audio compression technologies were necessary steps to make the original idea a commercial reality. The industry also had to agree on a relatively small set of audio compression techniques and storage formats to make it possible for users to play their content on multiple devices.

Meanwhile in computing, the first IBM PC launched in 1981 was fairly expensive (\$3000) and technologically inferior to the competition that came from Apple over the following years. The Lisa and then the rather more famous Macintosh introduced the first graphical user interfaces. In this case, the industry standardized on the PC because IBM had used off-the-shelf components in their design, plus a third-party operating system, in order to save time and money; this opened the way to much cheaper clones from other manufacturers. However, it took until 1990 for Microsoft to develop a graphical user interface for the PC, version 3.0 of the now ubiquitous Windows operating system, comparable to that on the early Macintosh. The combination of fairly standard hardware and a common operating system with an interface that anyone could learn

to use made the PC an attractive target for a wide range of third-party software.

One such piece of software was the web browser. Although TCP/IP allowed computers all over the world to communicate, there was no convenient standard way for non-technical users to publish and access information on the Internet. Then Tim Berners-Lee, working at CERN, invented the Web and built the first web server and text-based browser in 1990. Three years later, CERN announced that the Web would be free to everyone and in the same year the first graphical browser, Mosaic, was released. With the addition of popular search engines, AltaVista and Yahoo, adoption was rapid. The leader of the team that built Mosaic formed his own company, which became Netscape Communications Corporation. Netscape had early dominance in the market but soon faced strong competition from Microsoft's Internet Explorer. The war between the two browsers resulted in proprietary innovations on both sides, seeking to gain competitive advantage but creating fragmentation for developers and end users. Microsoft eventually ended the war (and browser innovation for several years) by abusing its operating-system monopoly to force Netscape out of the market by 1998.⁹ Monopolies can be very bad for innovation!

In the case of GPS, the situation is rather different since the system was built for the US military and controlled by their government. However, the timeline is very similar. The 24th GPS satellite was launched in 1993, bringing the system to what was called 'initial operational capacity'. In the same year, the decision was taken to authorize worldwide civilian use of the system, free of charge. Unfortunately, until 2000, the system included a special feature called 'selective availability' which deliberately introduced errors to the signal for civilian users, providing a position to only 100 m accuracy rather than the sub-20 m accuracy available to the military. Since the feature was removed, there has been a significant increase in the use of the technology, particularly for vehicle navigation systems.

The specification and creation of cellular phone systems across the world was to some extent government controlled, like GPS (exceptions being the United States and Japan). As a result, several countries developed different standards to build their own first generation (analog) systems. An alternative approach was taken in the Nordic region; Denmark, Finland, Norway and Sweden co-operated to specify a common system called NMT. The specifications were free and open, allowing many companies to produce NMT hardware, pushing prices down. The open system and lower cost of hardware encouraged adoption across parts of Europe, the Middle East and Asia. In fact, the first commercial NMT service was introduced in Saudi Arabia in 1981, just ahead of the Swedish network. The success of NMT and the experience

⁹ www.usdoj.gov/atr/cases/f3800/msjudgex.htm.

gained was a significant advantage to Nokia and Ericsson when the second-generation GSM system was developed following a similar open standards approach. The first GSM network was launched in 1991 by Radiolinja in Finland. The standard is now in use by over 80% of the world's mobile subscribers.¹⁰ The standards have since been extended to enable high-speed data communication as well as the original voice and basic messaging features.

The early evolution of mobile phones was closely tied to that of the networks they operated on. The first cellular phones were generally too large for anything but vehicle installation, although some models were designed to be carried in backpacks. The first handheld mobile, the DynaTAC 8000X, was produced by Motorola in 1983. Over the following years technological developments allowed phones to get smaller, lighter and cheaper. Apart from the shift to digital communications, the basic functionality remained the same for a long time. In early 1999, a major shift began with the launch of the Nokia 3210. The device included predictive text input, allowing faster text entry via the numeric keypad; three games, ensuring popularity with younger consumers; changeable covers; and an internal antenna, making it more customizable, attractive and pocketable. The result was a huge commercial success (approximately 160 million units sold) and massively increased the popularity of mobile phones for teenagers and young professionals. Having reached mass-market appeal and pricing, devices started to evolve in two separate directions: some models continued the cost reductions to reach a wider market and other models increased in features and functionality, rapidly becoming more powerful computing and communications devices.

As the various digital technologies discussed above found their way into mass-market consumer electronics devices it became almost inevitable that they would converge. With common requirements for a general-purpose processor, memory, display and user input methods, there was a strong economic incentive to combine them into a single device. Although there have been, and probably will continue to be, niche-market portable convergence products, from the early personal digital assistants (PDA) to small-form-factor, Internet-enabled devices such as the modern netbook; it seems most probable that the only convergence device to be adopted by a significant fraction of the world's population will be an evolution of the mobile phone. There are two important reasons for this: always-on network connectivity is best achieved through the cellular network (even if other bearers are also used when available) and the device that people reliably carry, regardless of culture, is their phone.¹¹

¹⁰ www.gsmworld.com/newsroom/market-data/market_data_summary.htm.

¹¹ Extensive research has shown that keys, money and a phone are the only things that most people (who own a mobile phone) almost always carry. See www.janchipchase.com/blog/archives/2005/11/mobile_essentia.html if you're interested in finding out why.

1.4.2 Birth of the Mobile Convergence Device

The potential for a powerful combination of computing and communications devices was recognized very early. The first ever ‘smartphone’, called Simon, was released by IBM in 1993. Nokia’s first smartphone (Nokia 9000), launched in 1996, was an early ‘Communicator’ product that ran GEOS on an x86 CPU. The IBM Simon and the Nokia 9000 were both extremely bulky, expensive and struggled with poor battery life, issues that have plagued many advanced mobile devices since then. It became clear to the leading mobile device manufacturers of the time that what was needed was an advanced operating system that was purpose built for battery-powered, resource-constrained devices which were seldom, if ever, rebooted. A new operating system developed by Psion to power their latest range of personal organizers, called EPOC32, was identified as the most suitable candidate. A joint venture was formed between Ericsson, Motorola, Nokia and Psion to continue the development to meet the needs of the emerging mobile device market. So, in 1998, Symbian was founded and the operating system was renamed Symbian OS.

The first year of the new millennium was also the year the first ever Symbian-based phone went on sale. The Ericsson R380, as it was called, was the size of a standard phone but the keyboard folded back to reveal a touchscreen. It combined a mobile phone with a number of personal organization tools, including an address book, a calendar, email, voice memo and a note pad. However, the R380 was ‘closed’: no additional applications could be installed to the phone. The operating system supported the installation of aftermarket software but it was not permitted on this device. This naturally raises the question of what exactly is a smartphone? The answer has to be that the definition has evolved over time as technology has advanced. Originally the term was applied to any device that combined the features of a phone and a PDA. More recently, the term can only legitimately be applied to phones with an operating system supporting the installation of native applications after a device has been purchased. As such, some might say that the Nokia 9210 Communicator, which launched a year later in 2001 as the first open Symbian phone, was the first genuine smartphone.

1.4.3 Inevitable Fragmentation

In the first few years of open smartphone development, the device manufacturers that were collaborating around Symbian OS decided that they could add value and differentiate their offerings from one another at the user interface layer. To this end, several user interfaces and application frameworks were created. At one point Nokia had three of their own: Series 60 (now referred to as S60), Series 80 and Series 90. Sony Ericsson and Motorola used another user interface, called UIQ,

and NTT DoCoMo in Japan created yet another, called MOAP(S). The Symbian platform became badly fragmented for developers at the UI layer. In addition, Microsoft entered the mobile-device operating system market with a version of their Windows CE platform named Windows Mobile¹² and other manufacturers shipped devices based on Windows Mobile or the competing Palm OS from the handheld computer market. Around the same time, Research In Motion (RIM) also started developing their range of pagers and corporate messaging devices into fully featured convergence devices.¹³

There was a multitude of mobile device options for both consumers and developers. To a large extent, developers tended to prefer Palm OS and Windows Mobile (partly due to familiarity from earlier handheld computing platforms) while consumers tended to select Symbian-based devices (probably due to their more phone-like form factors and functionality). Between 2001 and 2005, the device volumes were still relatively low and the technology not yet mature enough to provide really compelling convergence devices. By the middle of 2006, things were starting to change; Nokia had introduced their extremely successful Nseries range¹⁴ of 'multimedia computers' and demonstrated the potential for a single model, the N70, to sell several million units. At the same time, technology was evolving to make larger displays a reality at a mass-market price point. A larger display made mobile Internet use a more realistic proposition and that, combined with the high profitability of premium phones, renewed interest from outside the industry. Internet giant Google and luxury device maker Apple started investing heavily to create their own offerings, Android and the iPhone range. Simultaneously, mobile network operators and service providers called for an end to fragmentation so that they could develop applications and services that would work across the whole range of devices.

Many companies saw the free and open nature of Linux as a potential solution. It promised to be an operating system that could provide the basis for a platform that the industry could collaborate to develop. However, initially Linux's only real qualifications for this role were that it was robust, free and had an enthusiastic developer community. Having been designed and developed for desktop computers, its size, start-up time, power consumption and lack of support for typical mobile device hardware meant that a lot of work was needed to create a serious candidate. To meet this challenge, groups such as the Mobile Linux Initiative and the CE Linux Forum were started. As their efforts started to pay off, other groups began building on their work to create more complete platforms. Unfortunately, rather too many groups were all trying

¹² The original Microsoft Pocket PC and Smartphone brands were unified as Windows Mobile.

¹³ Although initially open to native applications, RIM's BlackBerry devices now only allow Java ME applications, like most feature phones.

¹⁴ www.nseries.com.

to do roughly the same thing with slightly different motivations; there was the Linux Phone Standards (LiPS) Forum, GNOME Mobile and Embedded, the LiMo Foundation, Ubuntu Mobile, and the Mobile and Internet Linux Project (Moblin). There were also multiple companies and communities trying to leverage these efforts to create similar but not entirely compatible device platforms such as Openmoko and Nokia's Maemo. By the time Google announced the Linux-based Android platform, and yet another group to develop it (the Open Handset Alliance), the situation was starting to look rather farcical.

1.4.4 Reunification

While the various mobile Linux variants were busy creating more fragmentation, Nokia were starting to reduce it in the Symbian world. Initially they reduced their user interface (UI) platforms from three (Series 60, Series 80 and Series 90) to just the most successful one, re-named as S60 and intended to be flexible enough that it could target all the segments previously covered by the others. Then in June 2008, Nokia made an extremely bold move to unify the Symbian platform, announcing their intention to buy all of the shares that they didn't already own in Symbian Ltd and donate the Symbian and S60 software assets to a new non-profit organization called the Symbian Foundation. At the same time, Motorola and Sony Ericsson pledged their UIQ assets and NTT DoCoMo and Fujitsu did the same for MOAP(S). The intention of this was to create a single royalty-free unified UI platform, using the best of all the existing platforms but maintaining backwards compatibility with S60. The acquisition of Symbian was approved at the beginning of December 2008, with the first release of the Symbian platform occurring in the first half of 2009. At launch, the Symbian Foundation made selected components available as open source (under the Eclipse Public Licence) and it is working to establish the most complete mobile software offering available in open source. In the interim, all the contributed software is available to members of the Foundation under a separate royalty-free license.¹⁵ However, this is just while third-party technology is disentangled from the platform and the code base is sanitized; the stated plan is to release the entire platform as open source by the middle of 2010, two years after the initial announcement.

Almost immediately after the Symbian Foundation was announced, two of the Linux standards groups joined forces with LiPS, and folded into the LiMo Foundation. Somewhat ironically for a platform based on an open operating system, the LiMo Foundation platform is not open source, nor is there a public commitment to provide anything other than a binary SDK to non-members at the time of writing.¹⁶ Additionally, many

¹⁵ Membership of the Symbian Foundation is open to all at minimal cost (see www.symbian.org/members for more details).

¹⁶ See www.limofoundation.org/images/stories/pdf/limo_foundation_ipr_guide.pdf.

of the devices developed using the LiMo Foundation platform may well not allow the installation of third-party, aftermarket native software; even where this is permitted, there are only source compatibility, not binary compatibility, guarantees so applications may have to be recompiled for each manufacturer's devices. Even so, the LiMo Foundation is going a long way towards consolidating the efforts of the major players in mobile Linux. Also, it has been noted by many industry commentators that the membership of the LiMo Foundation has a significant overlap with that of the Open Handset Alliance. It seems unlikely that manufacturers will ship devices based on both platforms for long, so further consolidation is probable in the future.

Another point of interest is that, just prior to being acquired by Nokia, Trolltech (creators of the Qt cross-platform application framework) joined the LiMo Foundation. Originally most of the Linux platforms were using GTK+ (part of the GNOME project), the most popular open alternative to Qt, for their application frameworks. Now both Maemo and Openmoko have started supporting Qt applications. With Trolltech also having released a Qt port for Windows Mobile in late 2007, it seems that there just might eventually be a single native application framework to target the vast majority of mobile devices. This topic is explored further in Chapter 6 but it's time to return to the question of why now is a good time to get interested in mobile software.

1.4.5 A Tipping Point

So, what was the purpose of this journey through history? In fact there were two reasons for providing all of this background information:

- The mobile industry has recently passed a turning point – although mobile device sales continue to grow rapidly the fragmentation problem should now be getting better, not worse.
- Seen in its historical context, the converged mobile device as a product category has been waiting for a number of technological breakthroughs to make it truly compelling as a mass-market product. Those breakthroughs have now been made.

In support of the first point, there is the fact that the Symbian ecosystem is now aggressively moving to support cross-platform standards. In the main, these are standards that are already available for embedded Linux platforms and, to a certain extent, Microsoft's Windows Mobile. Between them, these platforms cover the vast majority of mobile devices for which it's possible to write native aftermarket applications. This makes the case for investment in software development much more compelling.

To defend my second assertion, I could simply point to the statistics in Section 1.3.1. However, somewhat strangely for a book about the

Symbian platform, I'd like to point to the success of Apple's iPhone, particularly in the North American market, as proof of the statement. Unencumbered by the need to support existing features and standards that early converged device adopters have come to expect, Apple has been able to create a device which is optimized for media consumption on the move that also just about passes for a phone and textual communicator. They have done this from scratch in a relatively short timescale and made it attractive enough to sell significant volumes in a market that had previously seen very low uptake of mobile converged devices. In doing so, they've prompted a renewed focus on usability¹⁷ throughout the industry which should help boost all advanced mobile device sales. One final point on the subject of the iPhone is that the Apple App Store has also helped make a convincing case for native mobile applications, making sales revenues of approximately \$1 million per day in its first three months of operation.

From a pure technology perspective, we now have genuine mobile broadband Internet with high-speed packet access (HSPA) and screens of sufficient size and resolution to browse standard websites and watch full-length films. Many advanced mobile devices also incorporate personal navigation via GPS, alternative (and often free) local connectivity via WiFi, and a camera of suitable quality for anyone other than a professional or serious photography enthusiast. The next generation of mobile device hardware will feature multiple processor cores to provide significantly enhanced computing power without compromising battery life when that power isn't required. It will also bring more advanced 2D- and 3D-graphics acceleration, with the potential to further improve the user interface and the gaming experience. If we're not already past it, I believe the wider adoption of these features will take us beyond the tipping point where the inevitability of mobile software and services becoming mass-market on a global scale is assured.

Before we look any further into the future, just for a moment, imagine a person from 30 years ago transported in time to today (or waking from a coma, being rescued from a desert island – whatever you find more believable). The digital technologies we take for granted were only just being invented in their world and someone is showing them the latest high-end mobile device. It's a small wireless phone – amazing. It's also a camera and it can store and play their entire music collection. They can send messages and pictures almost anywhere in the world – that's quite mind blowing. Wait, it can play last night's TV show or live video from the other side of the world. That's not all; it can also find the answer to almost any question, show them where they are on a map and give verbal instructions to get to any other place they want. All of this, they're told, is only just scratching the surface of what this mobile device can do. It really must seem like a little magic box. In just 30 years, technology

¹⁷ And, of course, a string of attempted clones from other manufacturers.

has advanced so far as to be almost beyond comprehension. How will technology have changed the world in another 30 years?

1.4.6 Mobile Devices of the Future

While the current case for mobile software development is interesting, the future is exciting. Where will mobile technology be in 30 years? It's almost impossible to predict but many clues are available now. Tomi Ahonen, top industry consultant and author, makes the case that mobile is becoming a new and superior mass media channel.¹⁸ The central thesis of the book is that mobile, connected devices can replicate the features of all of the existing mass media channels and also have seven additional unique benefits (paraphrased):

- truly personal mass-media channel
- permanently carried
- always on
- built-in payment system
- available at point of creative impulse, enabling user-generated content
- near-perfect audience data
- context of media consumption.

Many of these unique benefits from a marketing perspective are potential avenues for further enhancement in the future. Devices could become ever more personal if they are worn rather than carried. In early 2008, Nokia showed a Morph concept¹⁹ phone that makes use of nanotechnology in new materials to create a flexible device that can be molded to the desired use but also worn like a bracelet or watch. The advanced materials can even help with the always-on aspect by harvesting solar energy across the entire surface of the device. Even in the much nearer term, nanotechnology promises to help free mobile devices from some of the power-usage constraints imposed by the battery. One of the many new battery technologies that will be available soon uses silicon nanowires to increase the charge density to around 10 times that of existing lithium-ion batteries;²⁰ it also has the advantage of significantly reduced degradation across charge cycles, giving the battery a much longer life.

With the battery life no longer restricting use of the mobile device, it's likely to be put to many more uses. While it's already possible to pay

¹⁸ Ahonen, T. (2008) *Mobile as 7th of the Mass Media*. Futuretext.

¹⁹ See www.nokia.com/A4852062.

²⁰ news-service.stanford.edu/news/2008/january9/nanowire-010908.html.

for digital content downloads with your phone, that may be extended much further. The keys, money and phone that most people carry with them everywhere could merge into just the phone. Like the early trials of computer networks, which eventually led to a global Internet, there is already a very successful contactless payment system in Hong Kong called Octopus, which is accepted on all public transport systems and in a wide range of shops. It is also commonly used in place of keys for building access. Octopus is based on RFID technology and has already been embedded into mobile phone covers. Another system currently being trialed around the world is PayPass from MasterCard, using very similar technology.²¹ It is also being made available in mobile phones. Currently contactless payment systems are only available for low-value transactions but in the future it may be possible to authorize higher value payments via a biometric sensor in a suitably equipped mobile device, allowing the whole process to be seamless.

As mobile devices become increasingly powerful, capable, less constrained by available power and a standard method of payment, they are also likely to become the most common portal for accessing information and services via the Internet. If this seems improbable, consider that it is already the case in India, where wireless technology has leapfrogged fixed lines, but also in Japan and Korea,²² two of the most advanced digital societies in the world, where standard broadband availability is extremely high. At this point, it should be made clear that mobile devices won't completely replace PCs and mobile will not replace the other media, just that the dominant paradigm will shift. If your application or service isn't available on mobile platforms then it will probably be left behind.

However, the unique benefit that will make developing applications and services for mobile platforms really interesting is 'context'. At present, this is largely limited to some social context (gathered via a social network) and possibly the location of the device. In the future, it is likely to be an area for major expansion with access to a vast array of contextual information from sensors and other systems in the device, elsewhere on the user and also in the environment. This information could be used to make digital services behave in more intelligent and helpful ways. Beyond the obvious and widely discussed medical monitoring and safety applications, there are also enormous opportunities to make our interactions with an increasingly technological world more natural and simple, reducing information overload as the amount of information available continues to increase. It's far too large and complex a topic to discuss here but fortunately an extremely deeply considered and well

²¹ Visa also has a contactless payment system, called payWaves, and Visa was one of the early members of the Symbian Foundation, so further developments can be expected there too.

²² According to data from the national regulators in the respective countries.

written account is already available in a book by Adam Greenfield;²³ I highly recommend it. Whatever form ubiquitous computing eventually takes, it seems likely that it will be based around advanced mobile devices for at least the next decade or two. By porting your code to mobile platforms now, you'll be ready to take it in all kinds of new directions that won't be available on the desktop.

1.5 Why Port to the Symbian Platform?

If you're convinced by the argument in the previous sections and have decided that porting your software to a mobile device platform is an excellent plan then the next question I need to answer is: Why the Symbian platform? Again, there are several answers:

- market share
- open source and royalty-free
- maturity
- range of hardware
- cross-platform APIs.

We look at each of these factors in the following sub-sections but first, Table 1.1 gives a comparison of the Symbian platform with some of its major competitors at the time of writing.²⁴

1.5.1 Market Share

Unsurprisingly for such a promising market, there has been fierce competition from the outset. It has been both surprising and impressive that Symbian was consistently able to maintain a market share well above 50% until the third quarter of 2008, with its nearest competitors somewhere in the teens (see Figure 1.4).

Although RIM has a reasonable and growing market share, BlackBerry products don't currently allow native aftermarket applications, only offering a Java-based environment, so we can't really consider them in the context of this book. All Linux-based devices are grouped together, including LiMo and Android devices, although exactly what the breakdown is

²³ Greenfield, A. (2006) *Everyware: The dawning age of ubiquitous computing*. New Riders. It's worth noting in support of the case presented here that, shortly after publishing this book, Adam Greenfield became head of design direction for service and user-interface design at Nokia.

²⁴ For informed updates on the mobile platform landscape, read the Vision Mobile blog at www.visionmobile.com/blog.

Table 1.1 Comparison of Mobile Platforms

	Symbian Foundation	RIM BlackBerry	Apple iPhone	Windows Mobile	LiMo	Android
Market share	High	Medium	Medium	Medium	Low	Low
Source code licensing	Open and free	Closed, not externally licensed	Closed, not externally licensed	Shared, ^a per-unit royalties	Shared, ^b moderate entry cost	Open (gated) and free
Maturity	High	High	Medium	High	Medium	Low
Range of hardware	High	Medium	Very low	High	Medium	Low
Cross-platform APIs	Very wide support	Java only	Limited support	Wide support	Moderate support	Java only

^aMicrosoft has shared source licenses under which it makes some of the platform code available to partners.

^bLiMo source code is open to consortium members.

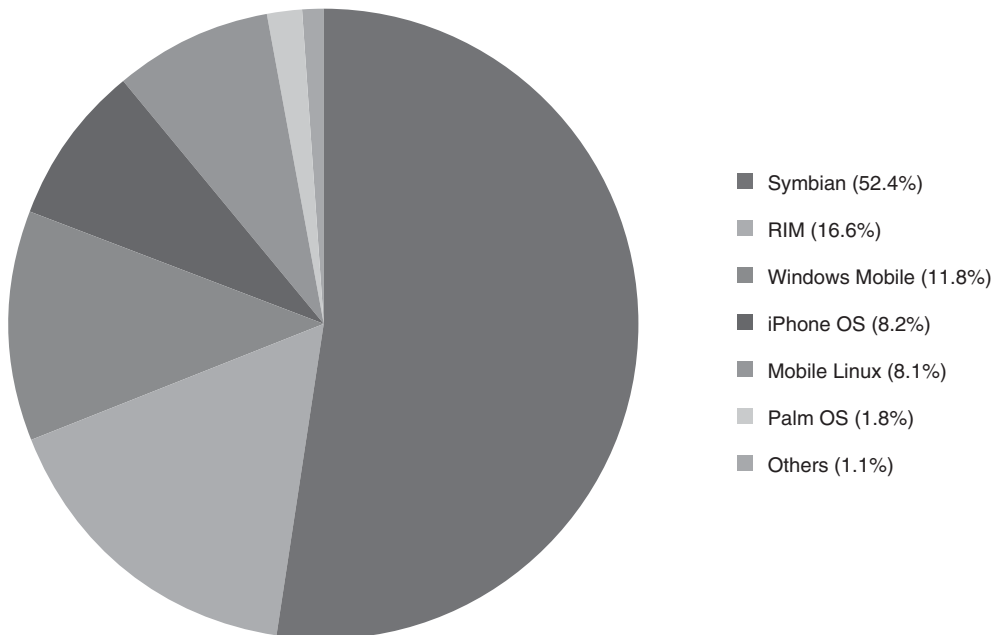


Figure 1.4 Smartphone operating system market share in 2008, clockwise from Symbian (at 52.4%)²⁵

²⁵Source: Gartner, www.gartner.com.

and how many of these can be targeted with native software is not clear at the time of writing. Microsoft, whether fairly or not, has never really been trusted by the mobile industry as there have always been fears that its embedded offerings would suffer from similar reliability issues to many of its desktop operating systems. This wasn't helped by the fact that the early Windows Mobile devices did actually display the infamous 'blue screen of death'²⁶ to end users. Finally Apple, with its much praised iPhone, has historically been a luxury device maker, focusing on high margins at the higher end of the market. It seems unlikely, although far from impossible, that it will either license the platform to others or make a successful push into the mid-range devices that Symbian has been working hard to reach in the last few years. With the unification of the platform under the Symbian Foundation, it seems extremely probable that you'll be able to target the greater number of devices with a single version of an application on the Symbian platform for many years to come.

1.5.2 Open Source and Royalty-Free

The Symbian Foundation plans to make the Symbian platform both open source and royalty-free by June 2010. Having worked on open source systems and on closed systems, both with privileged access to the source and without, I can honestly say that I wouldn't want to work on a time-critical development project without access to the source code for the underlying system libraries if I had any alternative. As mobile platforms become increasingly complex, it seems impossible for documentation and examples to keep up with the development of the system. No matter how good the documentation (and a lot of software documentation really isn't at all), there will always be gaps. When you fall into one of those gaps, then you need to be able to look at the source and work it out for yourself. If you can't do this, you're just left guessing and blindly searching for a solution.

From the platform development side, mobile device software is becoming too large and complex for any single organization to manage and maintain on its own. By opening the platform up to external contributions and shared development, it's possible to take advantage of the best solutions the industry has to offer. Of the platforms in Table 1.1, the Symbian Foundation is the only one with a declared intention to be fully open, although LiMo and Android are also open to contributions from consortium members. With a royalty-free license on the code, two additional options become available. First, smaller OEMs can use the platform to create devices without significant up-front investment or commitment to device volumes, while taking advantage of the full range of third-party software created for the platform. Second, if you've written an application that's dependent on a platform component then you may be able

²⁶ en.wikipedia.org/wiki/Blue_Screen_of_Death.

to port that component to another platform and distribute it with your application, easing future porting to other platforms.²⁷

1.5.3 Maturity

The Symbian platform has been designed from the ground up for the needs of mobile devices. Unlike some of the alternatives, it hasn't been adapted or converted from a desktop platform. The system has been proven in the market for almost a decade. By contrast, some of the competing operating systems have only just started shipping products in the last year or two. The step from single products to a fully-featured platform capable of shipping tens of differentiated device models every year whilst retaining compatibility and satisfying network operator requirements for security and standards compliance is a painful one. It has to be admitted that in the past some of that pain has reached third-party Symbian developers, but a lot of lessons have been learned. It remains to be seen whether other platforms will make the transition smoothly.

1.5.4 Range of Hardware

If you're planning to make your software work across a wide range of mobile devices, then you get the greatest diversity of device hardware platforms on the Symbian platform. Symbian have put a lot of effort into features, such as demand paging²⁸ and symmetric multi-processing,²⁹ that enable the platform to scale from mid-range devices right up to the latest high-performance hardware. If your software works on the most resource-constrained, mid-range devices but also scales to take advantage of more advanced hardware when available then you can be confident that performance won't be an issue when porting it to other mobile platforms. This makes the Symbian platform a great choice for your first porting target. However, it should be noted that there are time-to-market benefits involved in targeting a subset of Symbian devices or another platform with a smaller range of hardware for the first release of an application. On the other hand, discovering that your design or architecture doesn't scale after you've got the product in the market could be worse than delaying the launch.

1.5.5 Cross-Platform APIs

There are two key factors to consider here: the ease of porting to the Symbian platform and the ease of porting to other platforms from

²⁷ Obviously this will depend on the licensing for your software and any further dependencies of the system component.

²⁸ developer.symbian.org/wiki/index.php/Demand_Paging_on_Symbian.

²⁹ developer.symbian.org/wiki/index.php/Roadmap_for_OS_Base_Services.

Symbian. As stated in Section 1.2, a significant effort has gone into creating industry-standard and other cross-platform APIs for Symbian devices in recent history. It is now possible to run most standard C and C++ code on Symbian devices, including a lot of code that conforms to POSIX standards – this is enabled by software packages called P.I.P.S. and Open C/C++, which are described in Chapter 4. However, it is still necessary to write the code in a way that respects the limitations and reliability requirements of mobile devices – excellent advice for this can be found in Chapter 3.

Perhaps the most appealing aspect of the Symbian platform from a porting perspective is Nokia's purchase of Trolltech and the subsequent porting of the Qt application framework to S60. This enables the porting of applications already written for Qt to the Symbian platform with relative ease but also porting to Qt on Symbian, which then makes it possible to port to other platforms supported by Qt with a significantly reduced level of effort. Perhaps the greatest potential benefit here comes from Nokia's stated intention to use Qt across most of its device portfolio, including Series 40, the feature phone platform. It is still unknown if or when a port to Series 40 will be available for third-party developers. You'll find more about Qt in Chapter 6, along with details of some other cross-platform APIs that are also available on other embedded platforms, including games consoles, set-top boxes and various mobile Internet devices.

Before you jump into the details of the various APIs, please have a look through Chapter 2, which describes the process for successful porting projects on the Symbian platform. Hopefully it will save you many wasted hours. Good luck and happy porting!

2

The Porting Process

For every expert there is an equal and opposite expert.

Anon

This chapter discusses various strategies for porting code to the Symbian platform. There are many different types of software project and a number of different ways of porting the code for any specific project. My view is that there is no single ‘right’ way to approach a port although there are several ‘wrong’ ones! Most successful efforts will follow a very similar high-level process. I present that process with descriptions of the options available at each stage so that developers can make informed choices and understand the potential consequences of following a particular strategy. A lot of the decisions involved are about trade-offs between maintenance issues, such as keeping a common code-base, and other factors, such as porting effort, performance, usability, native look-and-feel, and so on. Often it is tempting to create a ‘fork’¹ of the original project for speed or simplicity but it’s important to bear in mind the lifetime cost of this decision, including maintaining your own version of the software you’ve branched from. This is such an important topic that we dedicate Chapter 15 to portability and maintenance issues. While this chapter presents several options that could result in a ‘fork’ being created, please consider the alternatives very carefully before making the choice.

There are usually a lot of unknowns at the beginning of a porting project and the top-down approach, analyzing all of the code in fine detail before starting to work on it, is unlikely to be the most efficient. The bottom-up approach of just trying to compile the project unmodified and

¹ A fork happens when developers take a copy of source code from a project and start independent development on it, creating a distinct piece of software. This is considered a bad thing in free and open source software development and is strongly discouraged because it tends to cause a large duplication of effort. Forking of proprietary code is more common, in my experience, but the consequences are the same.

seeing what happens can also be sub-optimal, particularly if the project turns out to be inappropriate for the platform. The key to success, as with most engineering, is finding the right balance between top-down and bottom-up strategies for the specific project.

There are several steps involved in a successful port and some of those that follow will be optional, depending on your goals:

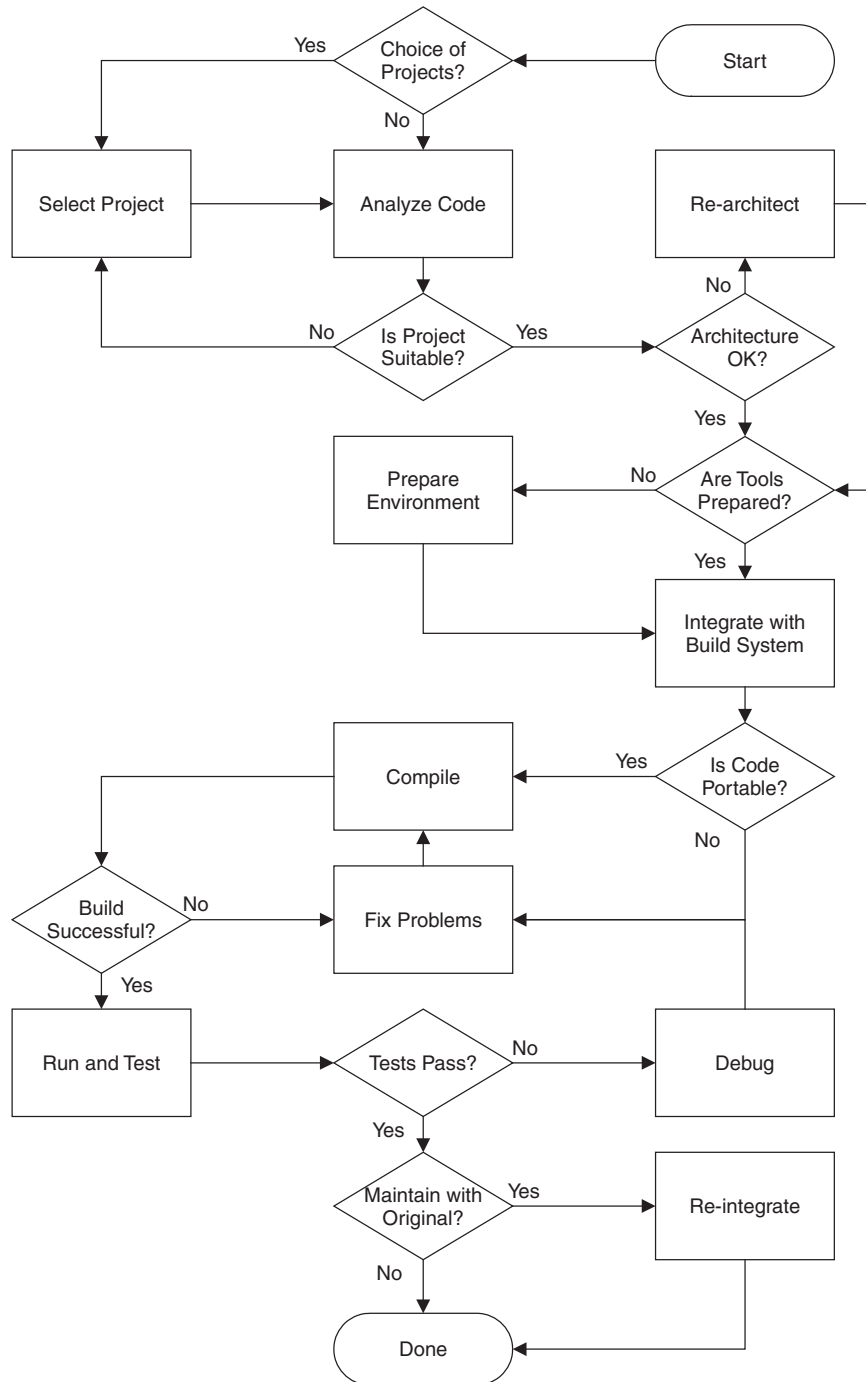
- choose a project to port
- analyze the code
- re-architect (if necessary)
- set up the development environment
- integrate your project with the Symbian build system
- compile
- fix problems
- run and test
- debug
- re-integrate with the original code (if desirable).

This is not a linear sequence and you would expect to iterate through parts of the sequence several times (see Figure 2.1). For example, you may select a project to port because it's widely used but then, after analyzing the code, decide to go back and select a different one! Similarly, it's unlikely that you'll fix all the compilation problems or bugs in the project at the first attempt. It can often make sense to have a first pass through the code and modify or comment out parts that obviously won't work before trying to compile it for the first time.

2.1 Choosing a Project

In many cases, you will have a choice of several projects to port, but on some occasions you may not have to make a choice at all – the project may be selected for you or perhaps you're the author of an original project that you want to port. Where there are multiple projects available, it is often possible to make an initial selection purely by examining project websites and documentation.

Making a selection is not an exact science and depends on a number of personal and technical preferences, such as whether you are familiar with the software as a user, or are more comfortable with C than with C++.

**Figure 2.1** The porting process

Here are some other things to bear in mind when making a selection:

- *Is the project actively maintained?*
There is a lot of 'abandonware' out there, software which is unfinished or at best only beta quality. Trying to fix bugs in the original code could be very difficult unless it is well documented (which is very rare) or easy to understand.
- *How many people were involved in the project?*
The vast majority of open source projects are developed by one or two people. Projects with multiple contributors tend to be better documented (or at least commented) and follow better engineering practices.
- *Will you have co-operation from the original authors?*
If the original authors or current maintainers of the project are supportive of your port, they may be able to give advice or even do some of the work for you. For larger projects it is always worth making contact at an early stage, particularly if the aim is to re-integrate any changes.
- *Has the project already been ported to other platforms?*
The more platforms a project has already been ported to, the easier it is likely to be to port to another one. This is because the platform-specific code should have been separated from portable code during previous ports.
- *Is the project suitable for your target hardware?*
Factors to consider here are both practical (e.g. screen size and input method) and technical (e.g. required processing power and RAM and use of floating-point arithmetic).
- *Are there any dependencies?*
Open source projects often build upon other open source libraries. Proprietary projects might use third-party libraries with commercial licenses. Are the libraries used in the project already available on the Symbian platform? If not, will you have to port them too and can you legally do so? Many commonly used libraries are available on the Symbian platform (see Chapter 4).
- *What license is the project distributed under?*
If you (or your company) own the copyright to the code then this is not an issue: you can use the code in any way you wish. For open source projects, there are hundreds of different licenses available and you need to be careful not to violate the terms of use. One of the main issues is the strength of any 'copyleft' clause, which broadly divides the licenses into three categories:
 - unrestricted (you can keep any changes to yourself), e.g. Apache and BSD licenses

- weak copyleft (you must release the original code and, in many cases, any changes under the same license but you can add or link with proprietary code), e.g. LGPL and Eclipse licenses
- strong copyleft (you must release the source for any project which uses this code under the same license, typically using one of the GPL versions).

If you are using open source code in a commercial project then you need to check the license details very carefully and it would be wise to have your company's legal advisors give approval to proceed. However, even open source developers need to be careful as there can be incompatibilities between licenses which prevent the combination of source code licensed under different terms.

Symbian Foundation Open Source Licensing

A specific license worth looking at in more detail is the Eclipse Public License (EPL)² selected for the Symbian Foundation code-base. This is a weak copyleft license that specifically allows proprietary extensions. Any changes to EPL-licensed code must be released under the same license. Additional modules, plug-ins or applications can be distributed with EPL-licensed code under any licensing scheme that is compatible as long as the additions or modifications would not be considered a 'derivative work' under US Copyright Law. Guidance from the Eclipse Foundation is that copying EPL-licensed code with a few minor modifications creates a 'derivative work', while writing a new module or plug-in with your own code that merely interfaces with EPL-licensed APIs does not. In scenarios between those two extremes, it would be wise to seek legal advice if there is any uncertainty as to what constitutes a 'derivative work'.

The GNU Public License (GPL)³ is very popular in the free software community and, in fact, the Linux kernel is licensed under version 2. Code licensed under the GPL cannot legally be combined with EPL-licensed code. So Symbian Foundation code cannot be included in a GPL-licensed project. A GPL-licensed project can only be ported to the Symbian platform if all of its dependencies fall under a 'system library' exception clause⁴ in the appropriate version of the GPL. Essentially, this means that if you are developing purely against a public SDK and don't need to distribute any additional EPL-licensed components with your executable then it should be fine.

² The EPL is available from www.eclipse.org/legal/epl-v10.html.

³ The GPL is available from www.gnu.org/copyleft/gpl.html.

⁴ See www.gnu.org/licenses/gpl-faq.html#GPLIncompatibleLibs for the exact details.

2.2 Analyzing the Code

Having selected a candidate project, it's time to have a look at the code. It is definitely worth spending some time on this to avoid nasty surprises later. The goal here isn't to understand how everything works but to get an idea of how the project is structured. You want to work out roughly how much of the code will work unchanged, what will need modifying and which parts will need to be re-written. This should give you a rough idea of how much work is involved, which is useful to know before you start, even if you haven't got a manager breathing down your neck demanding estimates!

If you're exceptionally fortunate, your project will be neatly divided into portable and platform-specific code, perhaps even with a porting guide explaining which parts need to be re-written and how to go about it. In most cases, you won't be anywhere near this lucky and will have to scan through individual files or folders to assess the scale and difficulty of the task. If you're dealing with an open source project, please consider documenting this process for those that follow you, porting to other platforms. It might even be you that benefits from this documentation, if you need to port your entire project to another platform, or if you come back to this project later after changing your mind and attempting to port another one (I've actually been in this situation – all I can say is that documentation for academic projects sometimes rather exaggerates the actual quality of the code!). However good the documentation and code appear to be, it is always useful to compile and run the code on a supported platform. Doing this can not only give you a feel for the maturity and stability of the project but also provide additional information about configuration and the level of compiler warnings to expect.

If your code is pure C or C++ (i.e., it does not call external libraries) then it will probably run on the Symbian platform unchanged. Most standard C/C++ and POSIX libraries are also supported but user interface and multimedia functionality, or anything that directly accesses other hardware, is probably not. For details of which libraries are supported and how to use them on the Symbian platform, see Chapter 4. Examples of libraries that are not currently available on the Symbian platform and how to port software that uses them is explained in Chapter 5.

If you have more than one suitable candidate project, it would be a good idea to confirm some of your assumptions from scanning the project documentation or website. How well documented or commented is the code? Check the included header files to see if there are any unexpected dependencies. Are there likely to be problems with limited resources? In some cases, it may be worth analyzing the code for several projects before making a final selection.

2.3 Re-architecting

Having analyzed the code, you should consider any general architectural changes required in order to make the porting process (and future maintenance) easier. Where some of the code is portable and other parts need to be re-written, it usually helps to have a modular architecture in which the portable parts are separated from the rest of the project. For applications with a graphical interface, you want to find a nice clean separation between the user interface (UI) and the engine⁵ – if that isn't the case you might consider re-factoring the code on the original platform before you start the port. For small applications, it may not matter but for a large application it may be a sign that you should go back and select another project to port, assuming this is an option. If you are re-factoring a project to increase portability, it's advisable to discuss this with developers working on the project on other platforms if you want to re-integrate your changes when you are done. This kind of major re-factoring is likely to affect everyone using the code-base and integration of the changes will have to be managed carefully.

A popular way of splitting the UI and the engine for Symbian applications is to have them in separate binaries with the UI executable interfacing to a portable DLL. If this split doesn't already exist for the project you are porting then it may not always be sensible to create it artificially. However, you can still split the UI from the engine at a source code (file) level and compile to a single hybrid executable. Historically, one of the major reasons for the split into separate binaries on Symbian OS has been the desire to support multiple UI platforms with a single application engine. The single Symbian platform will reduce the benefit of separating at a binary level as device manufacturers make the transition to a single UI framework. There are still benefits to designing an application with such a split on any platform; for example, it makes it easy to exercise the engine from a test harness and also allows easy stubbing of the engine functions to test the UI. It is debatable whether introducing such a split to a ported project is worth the extra effort unless the core functionality is likely to be re-used in other applications. These alternative application architectures are illustrated in Figure 2.2.

On several platforms, applications often use a polling paradigm: they check the status of various input devices or ongoing operations each time they are scheduled. Depending on the type of application, this may not be very power efficient. For example, it's fine for an arcade-style game⁶ where the user is constantly interacting anyway but perhaps not so good

⁵ Options for modularizing Symbian applications are discussed in Willee, H. (2008) developer.symbian.org/wiki/index.php/Modularizing_Symbian_C++_Applications.

⁶ Chapter 2 of Stichbury, J. et al. (2008) *Games on Symbian OS* covers implementation of a main loop and polling for input (available from developer.symbian.com/gamesbook).

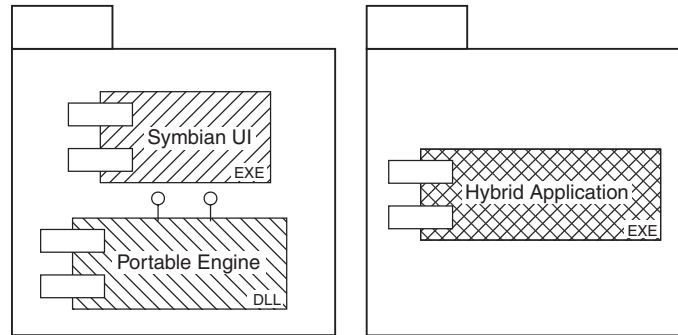


Figure 2.2 Application architectures

for a chess game or a document viewer where the user only interacts occasionally. On the Symbian platform, the preferred model is event-driven programming (with active objects – see Chapter 3 for details), in which an application's process is suspended, waiting to be notified by the system that an event in which it is interested has occurred. The event-driven paradigm is also used for multi-tasking on the Symbian platform in preference to multi-threading, for most use cases. However, for a ported application it is usually best to change as little as possible, since fewer changes mean fewer opportunities to introduce bugs. For that reason, it is generally recommended to keep existing polling and multi-threading designs where possible. Get the code working first and then determine whether performance or power consumption is a problem. Unfortunately, the lack of standard API availability in UI and multimedia areas forces design changes in some cases. Chapter 5 provides some examples.

Apart from major architectural changes that may be required to make the port possible or maintainable, you may also need minor changes to deal with quirks of the platform or performance issues. Some of these changes will require very little modification to code, for example by moving some functionality into another thread or having it run in a callback rather than polling a device. In these cases, you need to think carefully about how the change impacts the rest of the code and what you need to test to make sure it works.

At this stage, you also need to decide what to do with missing dependencies. There are three main options:

- port the dependency
- write a porting layer
- re-write the parts that use the dependency.

A porting layer involves writing wrappers with the original function names from the missing dependency and implementing the underlying

functionality in native Symbian C++ code. This can be a good strategy if your project uses only a small subset of the functionality exposed by a missing library. Your choice also depends on how the project is going to be maintained. If it is to be maintained as part of the original project with conditional compilation, rather than as a separate project, then you will probably want to port the dependency or write a porting layer to minimize changes to the common core of the application.

If you need to use some native Symbian functionality to replace a missing dependency, it may also require some changes to the architecture. A common example is an executable that runs a main loop and needs to use Symbian functions that depend on active objects. Active objects require an active scheduler and that blocks the thread in which it is running. The obvious solution here is to run the active scheduler and associated active objects in a separate thread, making the request from the main thread and then polling for the result. However, if you are planning to wait for the result of the operation (i.e. you want a synchronous call), then you can simply start the active scheduler when you make the request and stop it again when you get the response, running the whole thing within one cycle of the main loop. This may not be a great idea if the main loop is running your user interface.

2.4 Setting Up the Development Environment

If you are new to Symbian development, then there are a number of tools and resources you will need to set up before you can get started with your port.

2.4.1 System Requirements

The system requirements for C/C++ development on Symbian are as follows:

- Microsoft Windows 2000 Professional with Service Pack 3 or Microsoft Windows XP Professional with Service Pack 2⁷
- at least 512 MB of RAM (as with all software development, more is better – 2 GB is recommended when using the IDE)
- a Pentium-class, 1 GHz or faster processor (again, faster is better)
- at least 1 GB of free disk space (more required for multiple SDKs)

⁷ Windows Vista is not officially supported by older Symbian SDKs (S60 3rd Edition and Feature Pack 1 in particular): although it can be made to work, the installation process is rather more complex. There is an extensive wiki page on the issues involved in getting S60 SDKs to install on Windows Vista at wiki.forum.nokia.com/index.php/Moving_to_Windows_Vista.

- a 16-bit color display capable of 1024x768 resolution.
- Java Runtime Environment (JRE) (available from java.sun.com; the correct version will be linked from or bundled with the SDK)
- ActivePerl 5.6.1 (available from activestate.com; the correct version will be linked from or bundled with the SDK)
- local administrator rights for installation and removal of software.

Other Desktop Operating Systems

There is no support for Symbian development on Mac OS X or any other Apple operating system – nor is there likely to be in future. If you have an x86-based Mac then dual-boot Windows XP, if possible, or use virtualization software such as VMware Fusion or Parallels.

The bad news for Linux developers is that there is currently no official support for Symbian development under Linux either. A Windows installation is highly recommended but if you are fervent about this sort of thing, then there are some unofficial patches⁸ to the toolchain that allow you to build and deploy applications natively on Linux. You can even run the Symbian emulator under Wine⁹ – just bear in mind that it isn't widely used, so you're pretty much on your own if things don't work.

2.4.2 Integrated Development Environment

The recommended integrated development environment (IDE) for Symbian C++ development is called Carbide.c++. It is based on the popular open source Eclipse IDE and comes in three editions: Developer, Professional and OEM. Since late 2008, with the release of Carbide.c++ v2.0, all of the editions are free and I have listed them in order of increasing feature set. All editions have support for on-device debugging, which can be extremely useful for some types of application. There are additional features in the Professional edition, such as a code performance analysis tool, that you might as well have, since they are free, unless disk space is a serious issue. The OEM edition is only intended for developers working on device firmware; its extra features are not usable by most third-party developers. All of these IDE versions are available as part of the Application Development Toolkit (ADT) from developer.symbian.org/main/tools_and_kits. Historically, the main alternative was Carbide.vs, a plug-in for Microsoft Visual Studio, but this has now been withdrawn.

⁸ www.martin.st/symbian.

⁹ www.winehq.org.

Of course you don't have to use an IDE at all; it is possible to do all of your Symbian development using command line tools and a simple text editor but this is unnecessarily masochistic in the 21st century. You would miss out on lots of useful automation of the build and deployment process as well as excellent debugging features. Throughout the examples in this book, we assume you are using Carbide.c++. However, for large projects, there are distinct speed advantages when building from the command line and only using the IDE for debugging.

2.4.3 SDKs and Plug-ins

Prior to the formation of the Symbian Foundation, Symbian devices were divided between three user interface (UI) platforms: S60, UIQ and MOAP(S). S60 was developed by Nokia, UIQ was originally developed by Symbian and then spun off into a separate company (later owned by Sony Ericsson and Motorola) and MOAP(S) is used by NTT DoCoMo in Japan but it doesn't allow the installation of third-party applications. Within the Symbian Foundation, technologies from all three UI platforms will be combined into a single platform. However, it will only provide backward compatibility with S60 3rd Edition and later versions. For this reason, all of the examples in this book are developed for S60 only, although UIQ-based devices will still be in use for some time, with the first devices based on the unified Symbian platform not expected to start shipping until 2010.¹⁰

Download and install S60 SDKs from: **www.forum.nokia.com/Tools_Docs_and_Code/Tools/Platforms/S60_Platform_SDKs**. The only extra advice I would give, having seen lots of people get this wrong, is not to assume that the latest SDK is the one you want. For the widest possible device compatibility, you'll need either the S60 3rd Edition (Maintenance Release) or UIQ 3.0 SDKs. If you use a more recent SDK than this, it may contain new features not compatible with older devices. Only use newer SDKs if you want to target specific devices and features, or you discover a compatibility break in a newer device (they do happen sometimes despite best efforts to prevent them). The S60 SDKs advance the edition for a major version change and add 'feature packs' for minor version updates, so you have S60 3rd Edition FP1 and FP2, for example. The UIQ SDKs simply used major and minor version numbers, such as UIQ 3.0 and UIQ 3.1. However, UIQ has closed down and there is no longer an official developer portal that distributes the SDKs. You can discover what version a specific device is running by checking **developer.symbian.org/wiki/index.php/Symbian_Devices**.

The exception to the rule about not using newer features is where they are directly relevant for a port. Several standard APIs have been added

¹⁰ Visit **www.symbian.org** for the latest information about Symbian platform releases.

since the initial releases of S60 3rd Edition and UIQ 3.0. The libraries containing these APIs are made available as plug-ins to the relevant SDKs and SIS files for installation on devices that were shipped without them. Further details of these libraries and the related plug-ins, known as P.I.P.S. and Open C/C++, can be found in Chapter 4. Additionally, the port of the Qt application framework to S60 is only supported from S60 3rd Edition FP1 onwards.

2.4.4 Compilers

There are three compilers commonly used in Symbian C++ development:

- Nokia x86 compiler for the Symbian platform emulator on Microsoft Windows (referenced as `WINSCW` within the build system)
- the open source GCC-E compiler, used to cross-compile for target hardware based on the ARM processor (referenced as `GCCE` within the build system)
- ARM RealView Compilation Tools (RVCT), the high-performance commercial alternative to GCC-E, used by device manufacturers to build device firmware (referenced as `ARMV5` within the build system).

The first two of these are free and come bundled with the IDE and SDK respectively; the third should produce higher performance code and smaller binaries for target hardware but must be licensed separately from ARM at a cost.

It is recommended that you accept the default paths and settings while installing the SDK and compilers. If you need to change installation paths for some reason, then ensure that the path to the GCC-E compiler doesn't contain any space characters or your builds will fail with 'missing separator' errors.

2.5 Integrating with the Symbian Build System

Prior to the introduction of Raptor¹¹ in the latest version of the platform, the Symbian build system had been in roughly the same form since the early days of the operating system. This meant it was overdue a complete overhaul. Whatever platform you are porting from, you will need to adapt

¹¹ developer.symbian.org/wiki/index.php/Introduction_to_RAPTOR.

to the build system. This can be a difficult area for larger projects and a major headache if you want to add Symbian to the platforms already supported by a project under a common build system.¹²

However, both Symbian build systems work with the same input files. In order to port build files to the Symbian system, you will need some understanding of how the builds work on both the original platform and the Symbian platform.

We can't cover all available build systems in this book, although this isn't a very difficult task and most build systems are very similar at an abstract level. The majority of open source software projects use `make` and all but the simplest also have some kind of `configure` script¹³ to generate makefiles from templates in order to adapt the build to the host platform.

Symbian has a similar makefile generation system in order to produce makefiles for different host (Windows-based emulator) and target (GCC-E or ARM RVCT cross-compilation) builds. The primary inputs for the Symbian build system are a `bld.inf` file and MMP files. The `bld.inf` file is very simple: it specifies the target platforms, any exported files (headers, test data, configuration files, etc.) and one or more MMP files. The MMP files have a fairly simple syntax¹⁴ that lists the build target, target type and path as well as the source files, libraries and include paths used to build it. Many of these can be copied from the (possibly generated) makefile on the original platform, although note that all executables from Symbian OS v9 onwards must be installed to `\sys\bin`.

2.5.1 Build Process

I hope you already understand the build system for the platform you are porting from; if not, you need to consult the relevant documentation. Let's take a look at the workings of the Symbian build system. I'll use the command-line build steps for simplicity but an IDE will either call the command-line utilities in the background or use internal routines that have the same effect.

¹² Forum Nokia Champion Harry Li is attempting to port the Mozilla platform to Symbian. This requires the Symbian version to be built using Autoconf and GNU Make. He has documented his efforts to port the build files at wiki.mozilla.org/Mobile/Symbian/Build.

¹³ A `configure` script is often created by Autoconf, available from www.gnu.org/software/autoconf.

¹⁴ You can find the complete syntax in the Symbian Developer Library supplied with your SDK or at developer.symbian.com/main/documentation/sdl/symbian94/sdk/doc_source/ToolsAndUtilities94/Build-ref/Mmp-ref/.

1. Run `bldmake bldfiles`. This calls `bldmake.bat`, which simply passes any command-line arguments to a Perl script, `bldmake.pl`. This Perl script creates an appropriate `abld.bat` file for the project.
2. Invoke `abld build`, with optional arguments to specify a target and debug or release variant (otherwise it builds all targets and variants). This runs the batch file generated by Step 1, which executes `abld.pl` to perform several stages of the build process. The most important of these are the 'makefile' stage, which executes `makmake.pl` to generate makefiles from the MMP files, and the 'target' stage, which uses a version of `make` and other tools to build the final binaries.

Curious readers will find all of the scripts mentioned above in the `\epoc32\tools\` sub-directory in their SDK.

2.5.2 Build Files

Since the `bld.inf` and MMP files are the ones you actually have to work with, it is worth examining them in a little more detail. Here is a simple example of a `bld.inf` file:

```
PRJ_PLATFORMS
DEFAULT
PRJ_EXPORTS
..\inc\SoundTouch.h
..\inc\FIFOSamplePipe.h
..\inc\FIFOSampleBuffer.h
..\inc\STTypes.h
PRJ_MMPFILES
SoundTouch.mmp
```

This specifies that the project should generate makefiles for the default platforms (currently WINSCW, ARMV5 and GCCE), export four header files and build a single component which is specified in `SoundTouch.mmp`. The corresponding MMP file looks like this:

```
TARGET          SoundTouch.dll
TARGETTYPE      dll
UID             0x1000008D 0x0839739D
CAPABILITY      None
USERINCLUDE     ..\inc
SYSTEMINCLUDE   \epoc32\include \epoc32\include\stdapis
SYSTEMINCLUDE   \epoc32\include\stdapis\sys
SYSTEMINCLUDE   \epoc32\include\stdapis\stlport
SOURCEPATH      ..\src
SOURCE          SoundTouch.cpp AAFilter.cpp
```

```
SOURCE      FIFOSampleBuffer.cpp FIRFilter.cpp
SOURCE      RateTransposer.cpp TDStretch.cpp
LIBRARY      euser.lib
LIBRARY      libc.lib libm.lib libstdc++.lib
```

The first two lines specify that we want to build a DLL called `Sound-Touch.dll`. From the fifth line onwards we define user and system include paths, the path where the source is located, names of the source files to compile and any libraries to link against. These should be easily recognizable and familiar from almost any other platform. Note that you can repeat the keywords here as many times as you like with one or more files or paths after each, separated by spaces and terminated by a new line.

If this is your first Symbian project then the third and fourth lines, `UID` and `CAPABILITY`, won't be familiar. Every executable on a Symbian device has three unique identifiers (UIDs), one which identifies the target type (e.g. exe or dll), another which identifies a specific sub-type (e.g. a plug-in for a specific framework) and a third which is globally unique for the executable. The two hexadecimal numbers after the `UID` keyword in the MMP file define the second and third UIDs (commonly known as `UID2` and `UID3` respectively). The `CAPABILITY` keyword is related to application signing and the platform security model. You can find more details about Symbian's platform security in Chapter 14. By default, applications have no special capabilities. Certain APIs on the Symbian platform require executables be granted special capabilities in order to use them. The capabilities required for each API are listed in the relevant SDK documentation. This is the basis for a secure platform which limits the harm that can be done by badly written or malicious code. Some capabilities, such as reading or writing user data and recording from the microphone, are not considered particularly sensitive and can be granted by the end user at installation time. Other capabilities, such as direct access to device drivers or unencrypted DRM content, can only be granted by Symbian Signed¹⁵ and in some cases only with the approval of the device manufacturer.

The main alternatives for creating your `bld.inf` and MMP files are to write them from scratch, modify them from an existing application or use a wizard in the IDE to create a skeleton project and add your source files and necessary libraries to the MMP files as you go through the port. I'd strongly recommend this last option if your application has a graphical user interface since it also creates all the necessary boilerplate code for you to interact with the application framework. If you create your project with a wizard, it is automatically assigned a random UID from a testing

¹⁵ See www.symbiansigned.com for details of UID ranges and application signing.

range. If you distribute the project publicly as a SIS file, the UID should be replaced with one assigned by Symbian Signed. If you copy the MMP or write it from scratch, you'll need to select your own UID from the test range or request one from Symbian Signed.

Once you have your build files, you're almost ready to start porting the code. If you didn't create the project with a wizard then you need to 'import' from the `bld.inf` file to create a new project. Refer to the help in your chosen IDE if it isn't obvious how to proceed. In Carbide.c++ you simply select **File, Import** from the menu and then select 'Symbian OS bld.inf file' from the dialog that appears.

2.5.3 Packaging for Device Installation

Having created the build files and a project in your IDE, you should have everything you need to start working on the emulator. To get your code running on a real device, it also needs to be appropriately packaged and signed.

For packaging, you need a PKG (extension `.pkg`) file, which specifies where to get the executables and resource files generated by your build and where to put them in the device's file system. It also includes information on any dependencies that must already be present on the device and a UID which the installer can use for upgrading. Your options for creating the PKG file are the same as for the MMP file and the syntax¹⁶ is again fairly simple. Here is the content of the simple PKG file that Carbide.c++ generates for the SoundTouch project in the build file examples above:

```
#{"SoundTouch DLL"}, (0x0839739D), 1, 0, 0
;Localised Vendor name
%{"Vendor-EN"}
;Unique Vendor name
:"Vendor"
"$ (EPOCROOT) Epoc32\release\$ (PLATFORM) \$ (TARGET) \SoundTouch.dll"
- "!\system\libs\SoundTouch.dll"
```

This is an extremely simple example and the full package file syntax allows for fine control of the installation process with conditional file installation based on language variants, user selection or other testable conditions. The first line is the package header; it specifies the component name, UID (often, but not necessarily, the same as UID3 in the MMP) and version numbers for the component. The localized vendor name is displayed in installation dialogs; it should be replaced with the name of the person, group or company that owns or develops the package. The

¹⁶ developer.symbian.com/main/documentation/sdl/symbian94/sdk/doc_source/ToolsAndUtilities94/Installing-ref/PKG_format/.

unique vendor name is not localized and is used for matching package upgrades. Note that the last two lines should be typed on a single line. Also, this package file is specific to Carbide.c++: `$(EPOCROOT)`, `$(PLATFORM)` and `$(TARGET)` should be replaced with the appropriate values, if you want to use it with command-line tools.

The IDE should ‘self-sign’ SIS files for you by default; that is, it should generate its own certificate (which doesn’t grant any capabilities) and sign the installation file with it. This is fine as long as your project only uses user-grantable capabilities, otherwise you need to request an appropriate certificate from Symbian Signed and let your IDE know where to find it, or sign your SIS file online using Open Signed (from the Symbian Signed website) before you install it.

2.5.4 Incremental Integration

As a final tip for this section, if your source files are not too interdependent, you may find it easier to add them to the project incrementally and get it to compile a piece at a time. You should be able to simply add your source and header files in the IDE and find them automatically added to the MMP. If your port is primarily a learning exercise then you can start with the simplest parts to build confidence.

In a commercial environment, it may make more sense to begin with the most complex parts of the port so that you find any critical issues early, improve timescale estimates or even abandon the port before too much effort has been invested. The things you learn while getting the first parts to compile may enable you to make fixes to other parts of the project before you’ve even attempted to compile them.

2.6 Compiling

This is the easy part! Simply press the Build button in your IDE and wait for it to spit out a great mass of errors. Unless you aren’t expecting any problems or the project is very small and simple then it can be a very good idea to comment or `#ifdef` out large sections of the code and focus on getting some core functionality working. Also, make sure you try building with both the emulator and target compilers regularly to ensure you aren’t introducing anything that doesn’t work in both environments.

It’s important to recognize that build errors generated during a port may be caused by various factors:

- Compiler and linker differences: C and C++ may have published standards but they are not interpreted or enforced in the same way by all compilers and linkers. Open source projects have usually

been developed with the GNU toolchain so you shouldn't find many new errors from this source with GCC-E but there may be some for emulator builds or the ARM RVCT compiler. GCC-E tends to enforce ANSI standards more strictly than other compilers but the linker seems to be more forgiving. Also, if the project uses compiler-specific features, such as `#pragma` directives or inline assembler, then be ready for problems.

- **Missing dependencies:** Being unable to open a header file is a classic sign of a missing dependency. If you're lucky, this is just because the necessary include path isn't specified in the MMP. Sometimes a build system may add some include paths or libraries by default; this kind of implicit dependency won't usually show itself until the build fails. Often a necessary header or function isn't available on the Symbian platform. General strategies for dealing with missing dependencies are discussed in Section 2.3.
- **Mixing C and C++:** Where a project uses both C and C++ on the Symbian platform but was all C code on the original platform, there are often a lot of linker errors. Usually these appear to state that a function is not defined when you can see that it is. The solution here is almost always to wrap `extern "C"` around the declarations and definitions of functions written in C. This prevents the C++ compiler 'mangling' the names and gives them standard C linkage instead. To retain portability, you can use `__cplusplus`, which should be defined by all C++ compilers, like this:

```
#ifdef __cplusplus
extern "C" {
#endif
<your C code here>
#ifdef __cplusplus
}
#endif
```

- **Exports from DLLs:** On the Symbian platform, a function is only exported from a DLL if this is explicitly stated by specifying `EXPORT_C` for the definition and `IMPORT_C` for the declaration. This requirement is removed for the special target type `STDDL` from Symbian OS v9.3 onwards. Alternatively, macros for `EXPORT_C` and `IMPORT_C` can simply be defined with no value on non-Symbian platforms¹⁷ to maintain portability. Static data is never exported from a Symbian DLL and must be wrapped with an appropriate function. This can be hidden from calling code by defining a corresponding macro that

¹⁷ Microsoft Windows platforms have very similar conventions for function exports; projects that already support Microsoft Windows usually have appropriate defines in place.

replaces the variable name with a function that retrieves a reference to it (see Chapter 4 for an example).

- Other platform-specific differences: Data types, macros and functions may be defined in slightly different ways on different platforms. Examine the relevant definitions on both platforms and work out how to proceed. This requires careful judgment for each individual case, particularly where size or quantity limits are involved.

Another thing to be aware of is that some platform or compiler differences may not become apparent until run-time errors occur. A common error of this type is where code makes assumptions about byte positions or word alignment of elements within a structure. This is not portable because padding within a data structure is platform dependent. Any code like this should also be fixed on the original platform. Chapter 15 contains more tips and guidelines for writing portable code.

2.7 Fixing Problems

Here's where it gets interesting. Unless you are porting a purely computational library or an application or utility that only uses standard APIs that are available on the Symbian platform (see Chapter 4), then you need some knowledge of both the original platform and Symbian platform APIs in order to complete this stage. If you've already got plenty of both then this isn't a problem. If you're porting from one of the other mobile platforms covered in this book, then turn to the relevant chapter (Chapters 7 to 9) for further advice. Otherwise you'll need to consult the available documentation on each platform for any problem areas. As a good start, the first place to look for replacements for missing function calls is the Symbian Developer Library supplied with your SDK. The search function works reasonably well and problems only tend to arise where similar functionality isn't available or has a different name. There are far too many cases like this to list (and I'm sure I'm not aware of them all) but the various discussion forums¹⁸ are packed with questions and answers from thousands of developers that have trodden the same path before you. If you can't find what you're looking for, just post a new question.

Common problems when moving code to a mobile device platform from a desktop environment are resource limitations and handling the variation in device specifications. Screen size and the available input methods are also limited on mobile devices. While this can cause design issues, it is not as significant as the variation in these features across devices.

¹⁸ In particular, see developer.symbian.org/forum, Forum Nokia at discussion.forum.nokia.com and NewLC at www.newlc.com/forum.

2.7.1 RAM

By default, a process on the Symbian platform is limited to 1 MB of RAM for the heap and 8 KB for the stack. These can be increased via the `epocheapsize` and `epocstacksize` statements in the MMP file, although note that the maximum stack size is 80 KB and there are still lots of Symbian phones in circulation where you are very unlikely to find 10 MB free for the heap. You may find that you need to change the design or architecture in order to reduce heap usage, particularly for image or video manipulation. While 80 KB should be sufficient stack for all but the most deviously recursive algorithms or those that incorrectly allocate large objects on the stack, the default 8 KB is fairly easy to exceed and overflows aren't checked at run time. If you see data corruption or random crashes (particularly of the KERN EXEC 3 variety) then it's worth increasing the stack size to see if this fixes the error. Projects originating from desktop platforms also often leak memory and fail to handle out-of-memory situations gracefully; this requires extra testing and fixes (see Appendix A).

2.7.2 Storage Space (Flash Memory)

The main problem here is not size but availability. Most devices running Symbian OS v9 and later versions support expandable storage up to multiple gigabytes via SD cards or similar. High-end devices may also have several gigabytes of built-in storage but older or mid-range models are likely to have only a few tens of megabytes in total. The key is to handle out-of-memory situations gracefully and, preferably, give the user the option of storing resources on different drives.

2.7.3 Processing Power

The first Symbian OS v9.x devices ran on around 200 MHz ARM 9 processors with no floating-point unit. That gives performance fairly similar to the PC I was using in 1996. Popular high-end devices at the time of writing have around 300 MHz ARM 11 processors with hardware 3D graphics acceleration. The near future will bring devices with 400–600 MHz processors, featuring multiple cores at the high end. Often, performance has not been optimized at all on desktop platforms and it may be possible to improve slow-running routines by orders of magnitude in some cases, for example by converting floating-point arithmetic to fixed-point arithmetic (see Section 6.3.2). General performance optimization for C/C++ code is beyond the scope of this book.

2.7.4 Battery Life

Mobile device users expect the battery life of devices to be of the order of days, not hours. Be particularly careful with any application designed to run as a background service. If it is essential to poll some device or network connection then make sure the timeout used for polling is as long as possible without compromising usability.

2.7.5 Screen Resolution

Devices vary quite widely in their specifications. For example, QVGA (240 pixels by 320 pixels) was the most popular screen resolution for mobile devices in 2008, with higher resolutions just starting to appear at the high end. S60 3rd Edition phones have also been shipped with 176x208, 208x208, 352x416 and 800x352 resolution displays (measured in pixels). Additionally, very few mobile devices have a full QWERTY keyboard (although there is a trend to include full QWERTY or mini-QWERTY keyboards in some of the latest devices). Most have a touchscreen, a standard ITU-T keyboard and some softkeys, or both. Any code that uses direct key mappings must be reconsidered for usability.

There are several common approaches to dealing with this:

- developing for the lowest common denominator requires the minimum of maintenance and release work but fails to exploit the full capabilities of high-end devices
- developing multiple versions for different classes of device enables greater exploitation of device capabilities at the cost of additional maintenance and release work
- dynamically adapting to device capabilities adds complexity to the ported project and increases the differences from the original but allows a full range of device capabilities to be targeted while retaining a single version to maintain and release.

None of the schemes listed above is ideal and this kind of fragmentation is one of the major complexities of mobile software development. It is very common for two or even all three of these techniques to be used on the same project for different areas of variation. For example, a 3D game may require two versions, one for devices with hardware graphics acceleration and one for those with only a software renderer; however, both versions can dynamically adapt to the screen size of the device at run time.

Although it might be tempting to select different algorithms dynamically, based on the processor speed, this isn't advisable since the function

which queries the speed simply returns a constant value which has not always been accurately updated in device firmware builds.

2.8 Running and Testing

Once you have a subset of your code compiling and linking to produce an executable then you can run it in the emulator or install the generated SIS file to your device. We recommend that you do both as early as possible to ensure you aren't relying on features of the emulator (such as the fact that it is running on an x86 architecture processor rather than an ARM-based one) or using APIs that are not supported in your target device. How early you can sensibly test on the device depends on what you are developing and whether you have access to on-device debugging. The ability to set breakpoints, examine variable values and step through code on a real device is extremely valuable when porting code that doesn't produce many visible (or audible) results.

The subject of testing is too big and contentious to cover in detail in this book. However, there are a couple of questions worth addressing that are very relevant to porting. The first question is, 'How do I know when I've finished the port?' If you're working on a small personal project then the answer is probably, 'Whenever you're happy with it or have had enough!' For larger projects, those that you plan to release or re-integrate with the original code base and, particularly, for commercial projects, you will want some more objective criteria. I believe an excellent approach is the one championed in the Symbian ecosystem by Charles Weir of Penrillian¹⁹ (devise a set of tests on the original platform and make sure they pass there; port the tests as part of the project; when they all pass, you're done). Automated testing is preferred where possible – it allows you to apply test-driven development (TDD) practices to legacy code.²⁰ If your project already has a test suite then you're all set. If not, then you either write one or define a set of manual test cases depending on the scale and type of project.

The second question is, 'What do I need to test?' The easy answer to that question is, 'Enough to make sure it works' although that isn't very useful. Optimists (or ostriches sticking their heads in the sand, depending on your point of view) will say, 'If I haven't changed a piece of code it doesn't need testing.' This isn't necessarily true since any number of differences to the environment in which the code is running could cause it to break: timing issues, resource constraints, differences in the compiler and platform can all be the source of new bugs in previously working code. Pessimists (or software quality engineers) will point to the issues I've

¹⁹ www.penrillian.com/porting.

²⁰ TDD practices are described and explained fully in Feathers, M.C. (2004) *Working Effectively with Legacy Code*, Prentice Hall Pearson Education.

just listed and say, ‘Everything must be tested again!’ In practice, it may not be feasible to do a complete re-test of a ported project, particularly if it doesn’t already have an automated test suite. As usual, the real answer for each individual project lies somewhere between the extremes and you have to use your judgment.

2.9 Debugging

The process of debugging ported code on the Symbian platform is not significantly different from debugging on other platforms. A key difference from a standard POSIX environment is the lack of `stdout` and `stderr` by default (although you can have a console which provides them, it would be in the way of a graphical application). Instead, you can write debugging output to a file or include `e32debug.h` and use the native Symbian `RDebug::Printf()` as a direct replacement.²¹

Another important difference is what happens to errors from the system. Native Symbian C++ functions with names that end L, LC or LD may ‘leave’.²² A leave that is not caught in an appropriate `TRAP` macro causes an application to ‘panic’ – its thread or process is terminated and an error message and code are displayed in a dialog box on the screen. Incorrect use of some system servers may also cause them to panic a thread. There is a system panic reference²³ in the developer library to help you interpret the cause.

A common source of trouble at this stage is an `ALLOC` panic when you exit the application. A dialog pops up after your application closes that says ‘Panic: `ALLOC`’ followed by two hexadecimal numbers (the second one is usually zero). This means you have a memory leak and the first of the two numbers specifies the address of the heap cell that has been leaked. Other platforms tend to be much more forgiving about memory leaks than Symbian, so they may go undetected for years. Fortunately, thanks to each Symbian process having its own virtual address space, if the program makes the same allocations the same heap cell is leaked on subsequent runs.²⁴ This allows for a simple method of debugging these leaks:²⁵

²¹ If you’re developing a simple console application or standalone server with P.I.P.S. or Open C (see Chapter 4), writes to `stderr` are mapped to `RDebug::Printf()` by default. On some SDKs, you have to enable `RDebug` output for the emulator in `epoc.ini` before you see anything in the console of your debugger or the `epocwind.out` file.

²² A ‘leave’ is the Symbian version of throwing an exception (and from Symbian OS v9, is implemented with a standard C++ exception – see Chapter 3).

²³ developer.symbian.com/main/documentation/sdl/symbian94/sdk/doc_source/reference/SystemPanics/.

²⁴ On the emulator, only the least significant four hexadecimal digits of the address are guaranteed to be the same due to address space layout randomization on Microsoft Windows. In practice, I have not found this to be a problem.

²⁵ Adapted from Mika Raento’s Symbian programming pages at mikiie.iki.fi/symbian/leaks.html.

1. Note the hexadecimal address you get with the ALLOC panic.
2. Put a breakpoint somewhere in your application before the memory is allocated (this might require some guessing).
3. After hitting this breakpoint, add a data breakpoint for the address of the leaked cell. If you are unable to set the breakpoint then the memory region is not in use yet; move your original breakpoint further down the application and try again.
4. After successfully setting the breakpoint it should be hit when you allocate the leaked memory. The memory may then be released and re-used. Keep going until you find the code that makes the last allocation of this memory before the application exits with the panic.

There are slightly more complex and scientific methods of tracking down leaks but this one usually suffices. If your leaked memory address changes because it relates to allocations that depend on remote content or user interaction then you may find the HookLogger an excellent tool to help you track them down.²⁶

Another point worth noting here is that, while the details of a panic are provided in the emulator by default, they are not provided on target devices. To get this information when working on target hardware it is necessary to enable it, or use a third-party crash monitoring tool. For further information on this, see wiki.forum.nokia.com/index.php/Extended_panic_code.

2.10 Re-integrating

When you've got it all working, you may want to submit your changes back to the original project. This may not be practical if you've had to make significant changes. Where the changes are only minor, it's important to make sure that you can still compile and run on the original platform (and any others supported) by using the appropriate conditional compilation flags (`__SYMBIAN32__` is defined for all Symbian targets). Even if you're not planning to re-integrate your code, you may still want to upgrade your port with a future version of the original. Bear this in mind while porting and keep your changes localized (small, neat and separate from the main code whenever possible).

If you're porting C++ code then a good strategy for coping with areas of unavoidable variation is to encapsulate them in a class with a common abstract interface (this could be an abstract base class for two classes, one for each platform). Platform-specific versions of the class can be kept in separate files and the appropriate one selected by the build system. If

²⁶ developer.symbian.org/wiki/index.php/Using_Hooklogger.

you can get this refactoring of the original code accepted back into the project then it will be much easier to maintain, with very little chance of merges required for the platform-specific files.

When you do intend to re-integrate your changes, the most important thing is communication with the original developers. It is difficult to give general advice other than that you should follow whatever processes and procedures the project already has in place. Further strategies for writing code to ease future maintenance are discussed in Chapter 15.

2.11 Summary

In this chapter, I've explained a process that you can follow to port code to the Symbian platform. General advice for the various steps of the process is provided along with options for different approaches that you can tailor to the specific needs of your project. Along the way, I've covered some common issues you might face while porting but general advice will only take you so far. To make your porting projects successful, you need to get into the details. The next three chapters provide the core knowledge you need to work with the Symbian platform. Chapter 3 introduces Symbian idioms and compares them with those used in standard C/C++ development. It also explains how to write safe standard C/C++ code for a resource-constrained device. Chapter 4 goes on to introduce the standard libraries that are supported on the Symbian platform and how you can use them to make porting easier. In contrast, Chapter 5 explains which popular libraries are not available to Symbian developers and what you can do about it.

Throughout this book, particularly in Chapters 10–12, there are examples of open source projects ported to the Symbian platform. The process described in this chapter has been used for those ports and I describe them in that context. Feel free to refer back here to refresh your memory while working through the examples.

3

Symbian Platform Fundamentals

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off.

Bjarne Stroustrup

The aim of this chapter is to introduce you to the fundamentals of Symbian C++ and to show you basic guidelines to make your existing C++ code safe on this platform. Chapter 4 describes how you can use standard C and C++ libraries on the Symbian platform, including libc, STL and the Boost libraries, and this chapter discusses native Symbian C++ idioms. The aim is to cover the fundamentals that you need to work effectively on the platform and make you familiar with the terminology used by Symbian programmers, so you can both ‘talk the talk’ and ‘walk the walk.’

These are the parts of Symbian C++ that you can’t live without, or won’t want to, if you aim to write efficient and memory-safe C++ code for Symbian devices. There are entire books about Symbian C++, so this chapter is necessarily a digest of the topics essential for getting up to speed to port code to the Symbian platform. If you need or want to know more about each topic, we recommend you check out the Fundamentals of Symbian C++ wiki pages on developer.symbian.org/wiki/index.php.

3.1 In the Beginning

When Symbian OS was designed in the mid-1990s, the compiler and tool chain available was not fully up to date with the evolving C++ standard. For example, exceptions were not supported initially and later, when the compiler was updated, the code generated when exception handling was

enabled was found to add substantially to the size of the binaries and to the run-time RAM overheads, regardless of whether or not exceptions were actually thrown.

The API interfaces in Symbian C++ are based around standard Symbian notation; casing, prefixes, suffixes and names all convey information to the developer about what the function or variable does and how it performs. This design decision was taken because, when Symbian C++ was being developed, the C++ standard was not finalized and many of the features, such as exceptions, had a high implementation cost at run time, making it unsuitable for mobile development. Existing operating systems also had a high performance penalty when translated to mobile devices so additional constructs, such as active objects and the cleanup stack, were added to make the operating system perform in a more efficient manner and improve the battery life on the device.

This chapter helps developers understand where Symbian C++ differs from ANSI C++ and where Symbian C++ data structures and classes differ from the C++ library classes and structures. The features covered include resource management, strings, error handling and Symbian-specific features. This helps developers to understand how to develop and build applications that are well behaved and easily maintainable.

3.2 Naming Guidelines and Code Conventions

Symbian has very specific coding conventions that are context aware and help developers quickly understand what the function's, variable's or class's behavior will be at run time. This helps code be maintainable and understandable; failing to adhere to these conventions make it hard for developers and Symbian tools, such as LeaveScan and CodeScanner, to identify at compilation time issues such as resource leaks which can lead to serious code defects. The Symbian C++ coding standards are mandatory for submissions to the Symbian Foundation codeline.

3.2.1 Capitalization Guidelines

The first letter of class names is capitalized:

```
class TColor;
```

The words making up class or function names are adjoining, with the first letter of each word capitalized. Classes and functions have their initial letter capitalized. Function parameters and local, global and member variables have a lower-case first letter. Apart from the first letter

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

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

3.2.2 Naming Guidelines: Prefixes and Suffixes

Member variables are prefixed with a lower case *i* which stands for *instance*:

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

Parameters are prefixed with a lower case *a* which stands for *argument*. We do not use *an* for arguments that start with a vowel:

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

(Note that we write `TBool aExampleBool` rather than `TBool anExampleBool`.)

Local variables have no prefix:

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

Constants are prefixed with *k*:

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

A trailing *L* at the end of a function name indicates that the function may leave. This is discussed further in Section 3.5.1.

```
void AllocL();
```

A trailing C on a function name indicates that the function returns a pointer that has been pushed onto the cleanup stack, which is described in more detail in Section 3.5.2:

```
CShapeShifter* NewLC();
```

3.2.3 Class Names

The class-naming guidelines make the creation, use and destruction of objects more straightforward so that, when writing code, the required behavior of a class should be matched to the Symbian class characteristics. Programmers using an unfamiliar class can be confident in how to instantiate an object, use it and destroy it without untoward side effects, such as leaking memory.

To enable the types to be easily distinguished, Symbian uses a simple naming convention that prefixes the class name with a letter (usually T, C, R or M).

T Classes

- T classes are simple classes that behave much like the C++ built-in types.
- T classes do not need an explicit destructor because they do not contain any member data that requires cleanup or which itself has a destructor.
- T classes contain all their data internally and have no pointers, references or handles to data, unless that data is owned by another object responsible for its cleanup.
- Individually declared T class objects are usually stack-based but they can also be created on the heap.
- Although the data contained in these classes is simple, some T classes can themselves have fairly complex APIs, such as the descriptor base classes TDesC and TDes (see Section 3.4.4).
- In other cases, a T class is simply a C-style struct consisting only of public data. Likewise, enumerations are simple types and so are also prefixed with T. Enumeration members are prefixed with E. For example:

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

C Classes

- C classes are only ever allocated on the heap.
- C classes may contain and own objects and pointers to other objects.
- Unlike T classes, C classes have an explicit destructor to clean up member variables.
- For Symbian memory management to work correctly, C classes must ultimately derive from the class `CBase` (defined in the Symbian header file `e32base.h`). This class has three characteristics that are inherited by every C class:
 - Safe destruction: `CBase` has a virtual destructor, so a `CBase`-derived object is destroyed through the overloaded destructor.
 - Zero initialization: `CBase` overloads operator `new` to zero-initialize an object when it is allocated on the heap. This means that all member data in a `CBase`-derived object is zero-filled when it is created (so this does not need to be done explicitly in the constructor).
 - Private copy constructor and assignment operators: `CBase` classes declare these to prevent calling code from accidentally performing invalid copy operations.
- On instantiation, a C class typically needs to call code which may fail, for example, it may try to allocate memory when insufficient memory is available. There is an inherent risk of a memory leak at this point, so C classes use an idiom called *two-phase construction* to avoid it. I'll give more details on this in Section 3.5.4.

R Classes

- An R class holds an external resource handle, for example a handle to a server session.
- R classes are often small and usually contain no other member data besides the resource handle.
- There is no `RBase` class similar to the `CBase` class.
- A typical R class has a simple constructor and an initialization method that must be called after construction to set up the associated class and store its handle as a member variable of the R class object.
- While a C class directly allocates resources (for example, memory), R classes cause the indirect allocation of resources. For example, in

order to open a file, `RFile::Open()` does not itself open the file; instead it has a handle to a resource opened by the file server.

- R classes may exist as class members or as local stack-based variables. They may be placed on the heap, but it is unconventional to do so, and creates additional responsibilities if they need to be made ‘leave safe’, which is something we describe in Section 3.5.4.
- An R class also has a method that must be called on clean up to release the resource associated with the handle. Although in theory the cleanup function can be named anything, by convention it is almost always called `Close()`. A common mistake when using R classes is to forget to call `Close()` or to assume that there is a destructor that cleans up the resource.

M Classes

- M classes are used to define interface classes. The `M` prefix stands for *mixin*.
- On the Symbian platform, M classes are often used to define callback interfaces or observer classes.
- The only form of multiple inheritance traditionally allowed on the Symbian platform is that involving multiple M classes. The reason for this is covered in more detail in Section 3.8.
- An M class is an abstract interface class that declares pure virtual functions and has no member data.
- Since an M class is never instantiated and has no member data, there is no need for an M class to have a constructor.
- An M class may occasionally have non-pure virtual functions.
- When a class inherits from a `CBase` class (or one derived from it) and one or more M classes it is always necessary to put the `CBase`-derived class first to emphasize the primary inheritance tree. This is because of the way the cleanup stack works (see Section 3.5.2). A detailed discussion can be found in the May 2008 Code Clinic article at developer.symbian.org/wiki/index.php/Mixin_Inheritance_and_the_Cleanup_Stack.

```
class CCat : public CBase, public MDomesticAnimal
```

and not

```
class CCat : public MDomesticAnimal, public CBase
```


Static Classes

Some Symbian classes contain only static member functions. The classes themselves cannot be instantiated; their functions must instead be called using the scope-resolution operator. For example:

```
// Suspend the current thread for 1000 microseconds.  
User::After(1000);
```

Static classes typically provide utility functionality where the functions are collected together within a class for convenience, in a similar way to the way in which a namespace might be used (these classes were created before the use of namespaces was standardized).

The naming convention for these classes is not to prefix with a significant letter; examples are the `User`, `Math` and `Mem` classes.

3.3 Data Handling

3.3.1 Simple Data Types

The Symbian platform defines a set of fundamental types which, for compiler independence, should be used instead of the built-in C++ types when calling native Symbian APIs. They are provided as a set of typedefs (within the file `e32def.h`) as shown in Table 3.1.

3.3.2 Collection Classes

Data collections such as dynamic arrays are fraught with complication in Symbian C++. There have been a number of different generations of container class provided over the years, and the addition of Open C/C++ has at last provided the ability to use STL collection classes.

For instance, you can choose to use `std::vector` to hold a Symbian platform class, provided you handle any possible exception accordingly:

```
std::vector<TTime> table;  
try {  
    TTime t = table.at(1);  
}  
catch (std::out_of_range e)  
{ // Handle exception }
```

Table 3.1 Fundamental Symbian platform types

Symbian typedef	C++ type	Description
TInt	signed int	In general, the non-specific TInt or TUint types should be used, corresponding to signed and unsigned 32-bit integers respectively, unless there is a specific reason to use an 8- or 16-bit variant.
TInt8	signed char	
TInt16	short int	
TInt32	long int	
TUint	unsigned int	
TUint8	unsigned char	
TUint16	unsigned short int	
TUint32	unsigned long int	
TInt64	long long	These use the available native 64-bit support.
TUint64	unsigned long long	
TReal32	float	Use of floating-point numbers should generally be avoided unless it is a natural part of the problem specification. Many Symbian devices do not have a hardware floating-point unit and their floating-point performance is much slower than integer performance.
TReal64	double	Most serious floating-point calculations require double precision. All standard math functions (see the <code>Math</code> class) take double-precision arguments. Single precision should only be used where space and performance are at a premium and when their limited precision is acceptable.
TReal	double	
TAny	void	TAny* is a pointer to something – type unspecified – and is used in preference to void*. TAny is not equivalent to void. ¹

¹ Always use void when a method has no return type.

Table 3.1 (continued)

Symbian typedef	C++ type	Description
		Thus, we write <pre>void TypicalFunction(TAny* aPointerParameter);</pre> not <pre>void TypicalFunction(void* aPointerParameter);</pre> <pre>TAny TypicalFunction(TAny* aPointerParameter);</pre>
TBool	int	This type should be used for Booleans. For historical reasons, the Symbian TBool type is equivalent to int (ETrue = 1 and EFalse = 0). Since C++ interprets any non-zero value as true, direct comparison with ETrue should not be made.

However, if you want or need to use native container classes on the Symbian platform, you should use `RArray<class T>` and `RPointerArray<class T>`, found in `e32cmn.h`. These classes are the most efficient native dynamic containers and provide type safety and bounds-checked access, plus a range of methods for sorting and searching.

`RArray<class T>` comprises a simple array of elements of type `T`, which must be of the same type. The class is a thin template specialization of class `RArrayBase` (see `e32cmn.h`). The elements stored in `RArray<class T>` are typically `T` class objects² or simple types, enums or constants. `RPointerArray<class T>` is a thin template class deriving from `RPointerArrayBase`. It comprises a simple array of pointer elements that address objects, such as `C` class or `M` class objects, stored on the heap.

The API methods for adding elements to and removing elements from these arrays are relatively straightforward. They are documented in the Symbian Developer Library and in the Fundamentals of Symbian C++ wiki pages on developer.symbian.org. However, the following limitations should be noted:

- The size of an array element is bounded. The current implementation for `RArray` imposes an upper limit of 640 bytes; a `USER 129` panic arises if an element is larger than 640 bytes (or is zero or negative in size). In cases where the element size is greater than

² Note that the use of 'class `T`' in the class name is coincidental – it is the standard notation for a templated class.

640 bytes, it is better to store the elements on the heap and use an `RPointerArray<class T>`.

- There is no segmented-memory implementation for `RArray<class T>` or `RPointerArray<class T>`; both classes use a contiguous flat buffer rather than segmented memory layout. Contiguous memory blocks are typically used when high-speed pointer lookup is an important consideration and when resizing is expected to be infrequent. Segmented buffers are preferable for large amounts of data and if the container is expected to resize frequently, or where a number of elements may be inserted into or deleted. This is because, if a single flat buffer is used, numerous reallocations may result in heap thrashing and copying, because each allocation takes a new, larger memory block and then copies the data from the original location to the new one. If you think you need a segmented buffer array, you need to consider using one of the `CArrayXSeg` classes.
- When porting existing code, you should be aware of using `RArray` with classes that rely on non-trivial copy constructors and assignment operators, as `RArray` performs shallow copy over its items.
- For performance reasons, `RArray<class T>` is optimized to make best use of the ARM processor and stores objects with word (four-byte) alignment. This means that, even if the object is less than four bytes in size, it is stored in a four-byte element. Some member functions do not work when `RArray<class T>` is used for classes that are not word-aligned and an unhandled exception may occur on hardware that enforces strict alignment. The functions affected are:
 - the constructor `RArray(TInt, T*, TInt)`
 - `Append(const T&)`
 - `Insert(const T&, TInt)`
 - `operator[]`, if the returned pointer is used to iterate through the container as for a C array.

For a thorough discussion of `RArray`, please refer to ***developer.symbian.org/wiki/index.php/Advanced_RArray***.

The Symbian platform provides a number of legacy dynamic array classes with names prefixed by `CArray`, such as `CArrayFixFlat<class T>`, `CArrayFixSeg<class T>` and `CArrayVarSeg<class T>` (found in `e32base.h`) but you should not use these unless you have a fair degree of Symbian C++ expertise and understand why you need to use them. To attain this level of knowledge, we recommend you study an intermediate or advanced level title on Symbian C++.³

³ We can recommend Stichbury, J. (2004) *Symbian OS Explained*. John Wiley & Sons.

3.3.3 Native Data Collection Classes

Linked Lists

Class `TDbLQue<class T>` can be used to create a double-linked list of objects of type `T`. Objects of type `T` must define a member variable of type `TDbLQueLink` (to contain the forward and backward linked-list pointers). When the `TDbLQue` is constructed, you must specify the offset of the `TDbLQueLink` member variable in the constructor using the `_FOFF` macro. Please see the Symbian Developer Library documentation for further information and examples of the use of these classes.

Circular Buffers

`CCirBuf<class T>` can be used to create a circular buffer of objects of type `T`. Before adding anything to the circular buffer, `SetLengthL()` must be called to set the maximum length of the buffer. If the buffer fills up, the next addition returns 0, indicating that the data cannot be added.

Maps and Sets

`RHashSet` is a templated class that implements an unordered extensional set of objects of type `T` using a probe-sequence hash table. `RHashMap` is a templated class that implements an associative array with key type `K` and value type `V`, using a probe-sequence hash table. See the Symbian Developer Library documentation for further information and examples of how to use these classes.⁴

3.4 String Handling: Descriptors

In Symbian C++, the string classes are known as ‘descriptors’ because objects of each class are self-describing. A descriptor object holds information that describes it fully in terms of its length and layout. That is, it holds the length of the string of data it represents as well as its type, which identifies the underlying memory layout of the descriptor data. The rationale is that descriptors are thus protected against buffer overrun and don’t rely on `NULL` terminators, which reduces the risk of off-by-one errors. This is all very sensible, but descriptors have something of a bad reputation among Symbian programmers because they take some time to get used to. There are a lot of different classes to become familiar with and their use is somewhat contorted by memory management issues.

Most Symbian C++ APIs use descriptors, so it’s impossible to avoid using them completely if you intend to use the native APIs directly.

⁴ Such classes have only been publicly available since S60 3rd Edition FP2.

However, with the advent of Open C/C++, RGA and Qt for S60, as described in Chapters 4 and 6, there are alternatives to descriptors if you don't need to use the native APIs, and you can skip ahead to Section 3.5. However, some knowledge of descriptors is handy should you need to discuss them seriously with a Symbian C++ evangelist or debug code that uses them.

3.4.1 Memory Management

A descriptor object does not dynamically manage the memory used to store its data. Any descriptor method that causes data modification first checks that the maximum length allocated for it can accommodate the modified data. If it can not, it does not re-allocate memory for the operation but instead panics (USER 11) to indicate that a programming error has occurred (see Section 3.5.5 for more about panics). In the event of such a panic, it can be assumed that no descriptor data was modified. Before calling a modifiable descriptor method that may expand the descriptor, you must ensure that there is sufficient memory available for it to succeed.

3.4.2 Character Size

The Symbian platform is built by default to support wide character sets with 16-bit characters (UTF-16 strings). The character width of descriptor classes can be identified from their names. If the class name ends in 8 (e.g. `TPtr8`) it has narrow (8-bit) characters, while a descriptor class name ending with 16 (e.g. `TPtr16`) manipulates 16-bit character strings.

There is also a set of 'neutral' classes that have no number in their name (e.g. `TPtr`). The neutral classes are implicitly wide, 16-bit strings. The neutral classes were defined for source compatibility purposes to ease a previous switch between narrow and wide builds of the Symbian platform and, although today the Symbian platform is always built with wide characters, it is recommended practice to use the neutral descriptor classes where the character width does not need to be stated explicitly. If nothing else, it makes your code more readable.

However, there are occasions when you are dealing with text of a specific character size. For example, you may be working with 8-bit text (e.g. an ASCII text file or Internet email) or simply using the `RFile::Read()` and `RFile::Write()` methods, which are specified for binary data and require explicit use of the 8-bit descriptor classes. When working with 16-bit text, you should indicate the fact explicitly by using the wide strings.

3.4.3 Descriptor Data: Text or Binary?

Descriptors are strings and can contain text data. However, they can also be used to manipulate binary data, because they don't rely on a `NULL` terminating character to determine their length, since it is instead built into the descriptor object. To work with binary data, the 8-bit descriptor classes should be used, with `TInt8` and `TUInt8` to reference individual bytes.

The unification of binary and string-handling APIs makes their use easier for programmers – for example, the APIs to read from and write to a file all take an 8-bit descriptor, regardless of whether the file contains human-readable strings or binary data.

3.4.4 Symbian C++ Descriptor Classes

Figure 3.1 shows the inheritance hierarchy of the descriptor classes.

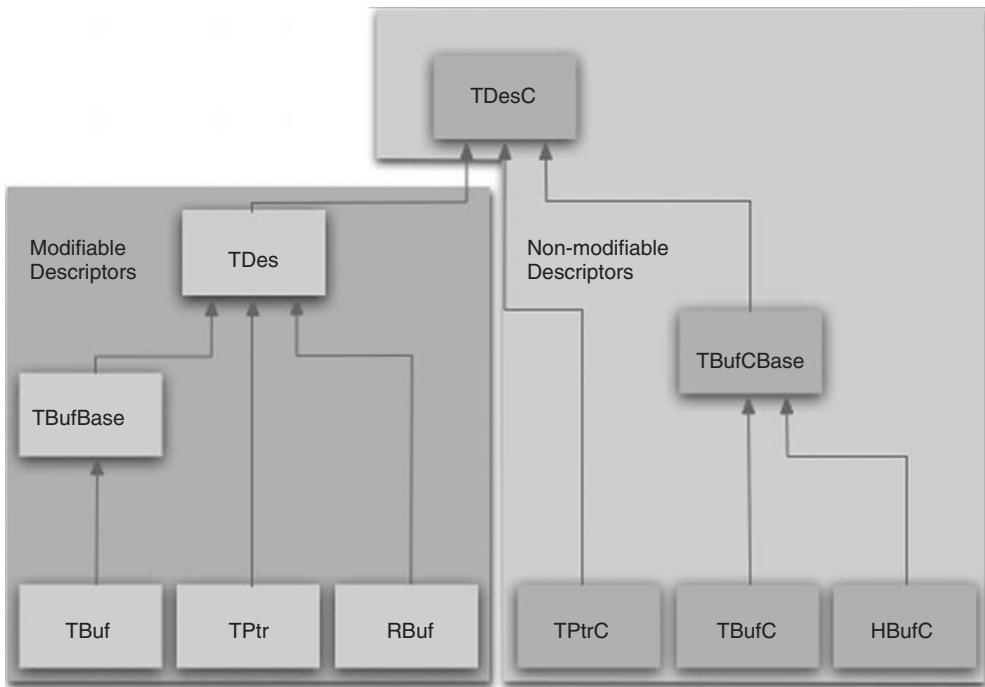


Figure 3.1 Descriptor class inheritance hierarchy showing modifiable and non-modifiable descriptors

`TDesC` and `TDes` are 'abstract' descriptor base classes. They are not abstract in the strictest C++ meaning of the word, in that they do not

consist solely of pure virtual functions (in fact, they have no virtual functions at all), but they are abstract in the sense that they are not meant to be instantiated, having only protected constructors.

Concrete descriptors come in two basic layouts: pointer descriptors, in which the descriptor holds a pointer to the location of a character string stored elsewhere; and buffer descriptors, where the data forms part of the descriptor.

Base Classes: *TDesC* and *TDes*

Apart from the literal descriptors, all of the descriptor classes derive from the base class *TDesC*. The class determines the fundamental layout of every descriptor type: the first 4 bytes always hold the length of the data the descriptor currently contains. In fact, only 28 of the available 32 bits are used to hold the length of the descriptor data (the other four bits are reserved for another purpose, which we'll discuss shortly). This means that the maximum length of a descriptor is limited to 2^{28} bytes (256 MB) which should still be more than sufficient for a single string.

Modifiable descriptor types all derive from the base class *TDes*, which is itself a subclass of *TDesC*. *TDes* has an additional member variable to store the maximum length of data allowed for the current memory allocated to the descriptor. Figure 3.2 shows the memory layouts of *TDesC* and *TDes*.

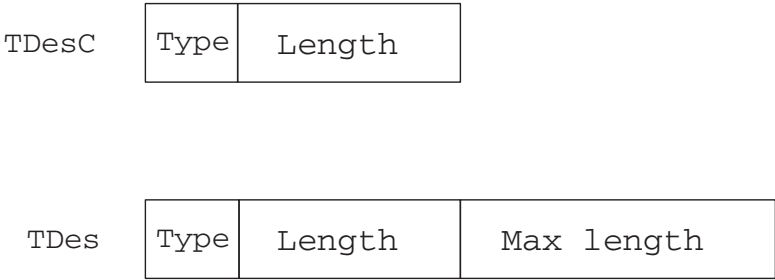


Figure 3.2 Memory layouts of *TDesC* and *TDes*

The `Length()` method of *TDesC* returns the length of the descriptor. This method is never overridden by its subclasses since it is equally valid for all types of descriptor. The `MaxLength()` method of *TDes* returns the maximum length the data value can occupy. Like the `Length()` method of *TDesC*, it is not overridden by derived classes.

Access to descriptor data is different depending on the implementation of the derived descriptor classes. The top 4 bits of the 4 bytes that store the length of the descriptor object are used to indicate the type of descriptor.

When a descriptor operation needs the correct address in memory for the beginning of the descriptor data, it uses the `Ptr()` method of the base class, `TDesC`, which looks up the type of descriptor and returns the correct address for the beginning of the data.

`TDesC::Ptr()` uses a `switch` statement to identify the type of descriptor and thus establish where the data is located. This means that the `TDesC` base class has hard-coded knowledge of the memory layout of all its subclasses and you can't create your own descriptor class deriving from `TDesC` without having access to the base class code. The number of possible different types is limited to 2^4 (16) since there are only four bits in which to store the descriptor type. There are currently five memory layout types and it seems unlikely that Symbian will need to extend the range beyond the limit of 16.

Using the `Length()` and `Ptr()` methods, the `TDesC` base class can implement the operations required to manipulate a constant string object, such as data access, comparison and search (all of which are documented in full in the Symbian Developer Library found in your chosen SDK or online at developer.symbian.org). The derived classes all inherit these methods and, in consequence, all constant descriptor manipulation is implemented by `TDesC`, regardless of the type of the descriptor.

`TDes` defines a range of methods to manipulate modifiable string data, including those to append, fill and format the descriptor data. Again, all the manipulation code is implemented by `TDes` and inherited by the derived classes.

Pointer Descriptors: `TPtrC` and `TPtr`

The string data of a pointer descriptor is separate from the descriptor object and can be stored in ROM, on the heap or on the stack, as long as it is in memory that can be addressed by the process in which the pointer descriptor is declared. The memory that holds the data is not 'owned' by the descriptor and pointer descriptors are agnostic about where the memory they point to is actually stored.

In a non-modifiable pointer descriptor (`TPtrC`), the data can be accessed but not modified: that is, the data in the descriptor is constant. All the non-modifying operations defined in the `TDesC` base class are accessible to objects of type `TPtrC`.

The modifiable `TPtr` class can be used for access to and modification of a character string or binary data. All the modifiable and non-modifiable base-class operations of `TDes` and `TDesC`, respectively, may be performed on a `TPtr`.

Stack-based Buffer Descriptors: `TBufC` and `TBuf`

Stack-based buffer descriptors may be modifiable or non-modifiable. The string data forms part of the descriptor object, located after the length word

in a non-modifiable `TBufC` descriptor and after the maximum-length word in a modifiable `TBuf` buffer descriptor.

These descriptors are useful for fixed-size, relatively small strings, since they are stack-based. They may be considered equivalent to `const char[]` or `char[]` in C, but with the benefit of overflow checking.

`TBufC<n>` is a thin template class which uses an integer value to determine the size of the data area allocated for the buffer descriptor object. The class is non-modifiable: because it derives from `TDesC`, it supplies no methods for modification of the descriptor data. However, the contents of a `TBufC<n>` descriptor are, indirectly, modifiable. We'll discuss how to do this shortly.

The `TBuf<n>` class for modifiable buffer data is a thin template class, the integer value determining the maximum allowed length of the buffer. It derives from `TBufBase`, which itself derives from `TDes`, thus inheriting the full range of descriptor operations in `TDes` and `TDesC`.

Non-modifiable Dynamic Descriptors: `HBuFC`

Heap-based descriptors can be used for string data that cannot be placed on the stack because it is too big or because its size is not known at compile time. This descriptor class can be used where data allocated by `malloc()` would be used in C.

The `HBuFC8` and `HBuFC16` classes (and the neutral version `HBuFC`, which is typedef'd to `HBuFC16`) export a number of static factory functions to create the buffer on the heap. These methods follow the two-phase construction model described in Section 3.5.4. There are no public constructors and all heap buffers must be constructed using one of the `HBuFC` factory methods (or by using one of the `Alloc()` or `AllocL()` methods of the `TDesC` class, which spawn an `HBuFC` copy of any existing descriptor).

As the 'C' suffixed to the class name indicates, these descriptors are not modifiable, although, in common with the stack-based non-modifiable buffer descriptor class, `TBufC`, the contents of the descriptor can be modified indirectly.

`HBuFC` descriptors can be created dynamically at the size needed, but they are *not* automatically resized if additional memory is required. The buffer must have sufficient memory available for a modification operation, such as an append, to succeed or a `USER 11` panic will occur.

Modifiable Dynamic Descriptors: `RBuF`

Class `RBuF` is another dynamic descriptor class. It behaves like `HBuFC` in that the maximum length required can be specified dynamically, but `RBuF` descriptors hide the storage of the descriptor data from the developer.

Upon instantiation, which is usually on the stack, an `RBuf` object can allocate its own buffer (on the heap) or take ownership of pre-allocated heap memory or a pre-existing heap descriptor. The behavior is transparent and there is no need to know whether a specific `RBuf` object is represented internally as a type 2 or a type 4 pointer descriptor.

`RBuf` is derived from `TDes`, so an `RBuf` object can easily be modified and passed to any function where a `TDesC` or `TDes` parameter is specified. Calling the descriptor operations is straightforward because they correspond to the usual base-class methods of `TDes` and `TDesC16`.

The class is not named `HBuf` because, unlike `HBufC`, objects of this type are not themselves directly created on the heap. It is instead an `R` class, because it manages a heap-based resource and is responsible for freeing the memory at clean-up time. `RBuf` is a simpler class to use than `HBufC` if you need a dynamically allocated buffer to hold data that changes frequently. `HBufC` is preferable when a dynamically allocated descriptor is needed to hold data that doesn't change, that is, if no modifiable access to the data is required.

Literal Descriptors

Literal descriptors are somewhat different from the other descriptor types. They are equivalent to `static const char[]` in C and can be built into program binaries in ROM because they are constant. There is a set of macros defined in `e32def.h` which can be used to define Symbian platform literals of two types: `_LIT` and `_L`. Neither literal type derives from the `TDesC` base class, however they are designed to make conversion between literal and non-literal descriptors easy.

_LIT macro

The `_LIT` macro is preferred for Symbian platform literals, since it is more efficient. It must be declared in advance of its use. For example, the Symbian platform defines *null descriptor* literals to represent a blank string. Three variants of the null descriptor are defined in `e32std.h`, as follows:

```
// Build independent:
_LIT(KNullDesC, " ");

// 8-bit for narrow strings:
_LIT8(KNullDesC8, " ");

// 16-bit for UTF-16 strings:
_LIT16(KNullDesC16, " ");
```

`KNullDesC` and its cohorts can be used as a constant descriptor.

When creating your own literals, try not to declare them in a header file, unless they need to be shared. This is to avoid unnecessary bloat: all CPP files that include the header generate a copy of the literal regardless of whether it is used.

`_L` macro

Use of the `_L` macro is now deprecated in production code, though it may still be used in test code (where memory use is less critical). The advantage of using `_L` (or the explicit forms `_L8` and `_L16`) is that it can be used in place of a `TPtrC` without having to declare it separately from where it is used:

```
User::Panic(_L("telephony.dll"), KErrNotSupported);
```

In the example above, the string `("telephony.dll")` is built into the program binary as a basic, `NULL`-terminated string. When the code executes, each instance of `_L` results in the construction of a temporary `TPtrC`, which requires setting the pointer, the length and the descriptor type; this is an overhead in terms of inline constructor code which may bloat binaries where many string literals are used. This is why the `_LIT` macro is considered preferable.

3.4.5 Descriptor Indigestion?

Table 3.2 summarizes the information this chapter has provided about the Symbian C++ descriptor types, and gives the C string equivalent, where appropriate.

3.4.6 Descriptors as Function Parameters

As I described earlier, the `TDesC` and `TDes` descriptor base classes provide and implement the APIs for all descriptor operations. This means that they can be used as function arguments and return types, allowing descriptors to be passed around in code without forcing a dependency on a particular type. An API client shouldn't be constrained to using a `TBuf` just because a particular function requires it, and function providers remain agnostic to the type of descriptor passed to them. Unless a function takes or returns ownership, it shouldn't even need to specify whether a descriptor is stack- or heap-based. When defining functions, the abstract base classes should always be used as parameters or return values.

For efficiency, descriptor parameters should be passed by reference, either as `const TDesC&`, for constant descriptors, or as `TDes&`, when

Table 3.2 Symbian C++ descriptor classes

Name ⁵	Modifiable	Approximate C string equivalent	Type	Notes
TDesC	No	n/a	Not instantiable	Base class for all descriptors (except literals)
TDes	Yes	n/a	Not instantiable	Base class for all modifiable descriptors
TPtrC	No	const char* (doesn't own the data)	Pointer	Data stored separately from the descriptor object, which is agnostic to its location
TPtr	Yes	char* (doesn't own the data)	Pointer	Data stored separately from the descriptor object, which is agnostic to its location
TBufC	Indirectly	const char []	Stack buffer	Thin template with size fixed at compile time
TBuf	Yes	char []	Stack buffer	Thin template with size fixed at compile time
HBufC	Indirectly	const char* (owns the data)	Heap buffer	Used for dynamic data storage where modification is infrequent
RBuf	Yes	char* (owns the data)	Heap buffer	Used for modifiable dynamic data storage
TLitC	No	Static const char []	Literal	Built into ROM

modifiable. You should not define functions that take descriptor parameters by value (which is what happens if you omit the '&' symbol after the class name). This is because the TDes and TDesC base classes do not contain any string data. If you pass them as parameters by value, you statically bind the base class descriptor types at compile time. The code compiles but no valid data is associated with the descriptor parameter because the base class objects do not have storage for string data. For example:

```
void Foo(const TDesC aString); // error, should be const TDesC&
_LIT(KHello, "Hello");
TBuf<5> data(KHello());
```

⁵ Note that use of C-suffixed classes on Symbian does not enforce them to be const in C++; you still need to specify the const modifier, for example, const TDesC&.

```
Foo(data);

void Foo(const TDesC aString)
{
    TBufC<10> buffer(aString); // buffer contains random data
    ...
}
```

As an example of how to use descriptor parameters, consider the system class `RFile`, which defines straightforward file read and write methods as follows:

```
IMPORT_C TInt Write(const TDesC8& aDes);
IMPORT_C TInt Read(TDes8& aDes) const;
```

For both methods, the input descriptor is explicitly 8 bit to allow for both string and binary data to be used within a file. The descriptor to write to the file is a constant reference to a non-modifiable descriptor, while to read from the file requires a modifiable descriptor. The maximum length of the modifiable descriptor determines how much file data can be read into it, so the file server doesn't need to be passed a separate parameter to describe the length of the available space. The file server fills the descriptor unless the content of the file is shorter than the maximum length, in which case it writes what is available into the descriptor. The resultant length of the descriptor thus reflects the amount of data written into it, so the caller does not need to pass a separate parameter to determine the length of data returned.

When calling a function which receives a modifiable descriptor, you don't need to know whether your descriptor has sufficient memory allocated to it, since the descriptor methods themselves perform bounds checking and panic if the operation would overflow the descriptor. Of course, you may not always want a panic, which is somewhat terminal. Check the documentation for any API you use to determine what happens when you pass a descriptor to receive data, in case what you pass is not large enough.

3.4.7 Using the Descriptor APIs

The descriptor API methods supplied by `TDesC` and `TDes` are fully documented in the Symbian Developer Library and in the Symbian Cookbook article found on the Symbian Developer wiki at ***developer.symbian.org/wiki/index.php/Descriptors_Cookbook***.

Remember however that objects of type `TDesC` and `TDes` cannot be instantiated directly because their constructors are protected. It is the derived descriptor types that are actually instantiated and used.

Which Descriptor Class Should I Use?

Figure 3.3 summarizes the factors to bear in mind when deciding on the type of descriptor to use. If you need a constant descriptor then your choice is between a literal, a `TPtrC`, a `TBufC` or an `HBufC`. If the descriptor must be modifiable, the choice is between a `TPtr`, a `TBuf` and an `RBuf`.

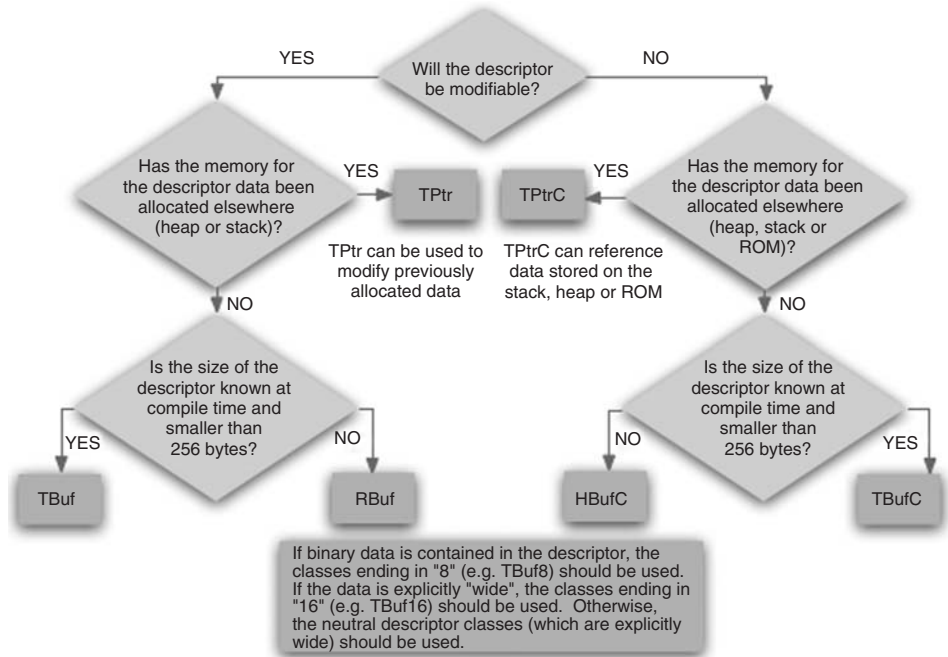


Figure 3.3 Choosing a descriptor class

To conclude this section on descriptors, the following discussion concentrates on some of the trickier areas of descriptor manipulation.

The Difference Between `Length()` and `Size()`

The `Size()` method returns the size of the descriptor in bytes. The `Length()` method returns the number of characters it contains. For 8-bit descriptors, the length is the same as the size, but when the descriptor has 16-bit-wide characters, `Size()` returns a value double that of `Length()` for neutral and explicitly wide descriptors.

`MaxLength()` and Length Modification Methods

The `MaxLength()` method of `TDes` returns the maximum length of a modifiable descriptor value. Like the `Length()` method of `TDesC`, it is

not overridden by the derived classes. The `SetMax()` method doesn't change the maximum length of the descriptor (as one might expect); instead, it sets the current length of the descriptor to the maximum length allowed.

The `SetLength()` method can be used to adjust the descriptor length to any value between zero and its maximum length. The `Zero()` method sets the descriptor's length to zero.

Des() Method

The stack- and heap-based constant buffer descriptor classes, `TBufC` and `HBufC`, provide a method that returns a modifiable pointer descriptor that references the data represented by the buffer. So, while the content of a non-modifiable buffer descriptor cannot be altered directly, except by using the assignment operator to replace the data completely, it is possible to modify the data indirectly by calling `Des()` and then operating on the data via the pointer it returns. When the data is modified via the return value of `Des()`, the length members of both the pointer descriptor and the constant buffer descriptor are updated if necessary.

For `TBufC`:

```
_LIT8(KPalindrome, "Satan, oscillate my metallic sonatas");
TBufC<40> buf(KPalindrome); // Constructed from literal descriptor
TPtr8 ptr(buf.Des()); // data is the string in buf, max length = 40
// Use ptr to replace contents of buf
ptr = (TText8*)"Do Geese see God?";
ASSERT(ptr.Length() == buf.Length());

_LIT8(KPalindrome2,
      "Are we not drawn onward, we few, drawn onward to new era?");
ptr = KPalindrome2; // Panic! KPalindrome2 exceeds max buf length
```

For `HBufC`:

```
// Allocate a heap descriptor of max length 20
HBufC* heapBuf = HBufC::NewLC(20);
TInt length = heapBuf->Length(); // Current length = 0
TPtr ptr(heapBuf->Des()); // Modification of the heap descriptor

_LIT(KPalindrome, "Do Geese see God?");
TBufC<20> stackBuf(KPalindrome);
ptr = stackBuf; // Copies stackBuf contents into heapBuf
length = heapBuf->Length(); // new heapBuf length = 17
```

There is an overhead to calling `Des()` to create a `TPtr` object around a buffer descriptor's data, and it should be avoided when not strictly necessary. For example, if a modifiable dynamic descriptor is needed, it

is better to use class `RBuf`. Another common inefficiency when working with `HBufC` is to use `Des()` to return a `TDesC` to pass into a function. The `HBufC` class derives from `TDesC` and an `HBufC*` pointer can simply be dereferenced when a reference to a non-modifiable descriptor (`TDesC&`) is required:

```
void Function(const TDesC& aText)
{ // Some code }
_LIT(KText, "Hello");
HBufC* text = KText().AllocL(); // Allocate HBufC from descriptor
Function(*text);                // Prefer this way ...
Function(text->Des());           // ... instead of this
```

3.4.8 Overview

The discussion in this section should give you a good idea about the requirements at which you should aim when dealing with standard C/C++ code. When it comes to string-handling functions, you should be aware of code that can potentially introduce overflow and similar problems due to the lack of proper boundary checking. Examples of these functions are `strcmp()`, `strcat()`, and `memcpy()`. Classes that handle these issues transparently should be always preferred, such as `std::string` or Qt's `QString`.

The Symbian platform is unforgiving when it comes to resource leaking as well. A good way to ensure your code is sufficiently robust is extensive use of appropriate testing tools⁶ in the original platform before attempting the port. This can improve the reliability of your code and thus make the port easier to work with on the Symbian platform.

3.5 Error Handling and Memory Management

3.5.1 Leaves and Exceptions

When Symbian OS was designed in the mid-1990s, the compiler and tool chain available did not support C++ exceptions, since they had only recently become part of the C++ standard. An alternative to conventional, but rather awkward, error-checking of function return values was needed. This is why ‘leaves’⁷ were developed – as a simple, effective and lightweight exception-handling mechanism. You’ll encounter lots of ‘leaving code’ when working on the Symbian platform. You need to know how to recognize code that leaves and how to use it efficiently and safely, since it’s possible to leak memory inadvertently in the event of a leave.

⁶ On Linux, one such tool is Valgrind (see valgrind.org).

⁷ ‘Leaves’ is as in the verb ‘to leave’. A function that contains code which may leave is called a ‘leaving function’ while code that has executed along the path of a leave (say, as the result of an exceptional condition) can be said to have ‘left’.

What Is a Leave?

A leave is used to throw an error result back up the call stack to a point at which it can be caught and handled. In Symbian terminology, the `catch` block is known as a trap handler and is created by using a `TRAP()` macro. Like a standard C++ exception, the stack is unwound by the leave, and, as long as the leave is trapped, it does not terminate the thread. A leave propagates a single integer value, a 'leave code', to the trap handler and it represents the error that has occurred. I talk a little more about the implementation of leaves after I discuss the cleanup stack in Section 3.5.2.

A typical leaving function is one that performs an operation that is not guaranteed to succeed, such as a memory allocation that may fail if there is insufficient memory available (low-memory conditions are a reasonably common occurrence on a mobile device, although memory is not as limited these days as it was when Symbian OS was originally designed). When testing any code that may leave, you should test both paths of execution, that is, for a successful call and for a call that leaves as a result of each of the exceptional conditions you expect to handle (low-memory conditions, failure to write to a file, etc.). The Symbian platform provides macros, such as `__UHEAP_SETFAIL` to simulate out-of-memory (OOM) conditions, which are described further in the Symbian Developer Library.

How to Identify a Leaving Function

As I mentioned in Section 3.2.2, if a function may leave, its name must be suffixed with 'L'. You *must* use this rule: of all Symbian C++ naming guidelines it is probably the most important. If you don't name a leaving function accordingly, callers of your code may not defend themselves against a leave and may potentially leak memory.

The L suffix is not checked during compilation, so occasionally you may forget to append it to a function name or you may later add leaving code to a previously non-leaving function. Fortunately, Carbide.c++ provides a useful plug-in called LeaveScan, which you should run regularly against your source code. It checks that all functions that have the potential to leave are named according to the naming guidelines.

Since leaving functions have a leave code, they do not also need to return an error value. Every error that occurs in a leaving function should be passed out as a leave; if the function does not leave it is deemed to have succeeded and returns normally. Generally, leaving functions should just return `void` unless they use the return value for a pointer or reference to a resource allocated by the function.

Some examples of leaving function declarations are as follows:

```
void InitializeL();
static CTestClass* NewL();
RClangerHandle& CloneHandleL();
```

What Causes a Leave?

A function may leave if it:

- Calls one of the system functions that initiate a leave, such as `User::LeaveIfError()` or `User::Leave()`. `User::LeaveIfError()` tests an integer parameter passed into it and causes a leave (using the integer value as a leave code) if the value is less than zero, for example, one of the `KErrXXX` error constants defined in `e32err.h`.
`User::LeaveIfError()` is useful for turning a traditional C-style function that returns a standard error result into one that leaves with that value. `User::Leave()` doesn't carry out any value checking and simply leaves with the integer value passed into it as a leave code.
- Uses the overloaded form of operator `new` which takes `ELeave` as a parameter and leaves if the memory allocation fails. This is discussed shortly.
- Calls other functions that may leave because the code uses one of the previous two approaches or calls code that does so. When you call a function with the potential to leave, your code also then has the potential to leave, unless you catch the leave by surrounding any calls to leaving code with the `TRAP()` macro, which we discuss later.
The following example shows a function with the potential to leave in three locations:

```
/*static*/ CClanger* CClanger::NewL()
{
    CClanger* self = new(ELeave) CClanger; // Leaves if OOM
    CleanupStack::PushL(self);           // May leave
    self->ConstructL();                   // May leave
    CleanupStack::Pop(self);
    return self;
}
```

Note that the code uses the Symbian cleanup stack, which makes heap-based objects leave-safe. It is discussed in more detail in Section

3.5.2. The code is also an example of the two-phase construction idiom discussed in Section 3.5.4.

Symbian Leaves and C++ Exceptions

Although standard C++ exception handling could not be used in early versions of Symbian OS, support for C++ exception handling was introduced in Symbian OS v9.1. From Symbian OS v9.1, the operating system defines a flag called `__LEAVE_EQUALS_THROW__`. This means that `User::Leave()` and `TRAP` are implemented in terms of C++ exceptions. When `User::Leave()` is called, an exception of type `XLeaveException` is thrown (this is a simple type that wraps a single `TInt`, which is the leave code that was passed to `User::Leave()`).

Whenever a C++ exception is thrown, memory must be allocated for the associated exception object. *Nested* exceptions occur when an exception is thrown while another is already being handled. For the second exception, a separate exception object is required, and if a further exception occurs while handling that one, another exception is required, and so on. If the number of levels of nesting of exception was unbounded, so would be the number of allocations of exception objects. This is unacceptable on the Symbian platform, given that memory resources are finite and exceptions are likely to occur specifically because of out-of-memory conditions. Therefore, nested exceptions are not supported by the Symbian platform, and only a single exception can be thrown at any one time. On hardware builds, if another exception occurs while the first is being handled, `abort()` is called to terminate the thread.⁸

By restricting the memory requirement to a single `XLeaveException` exception object, memory can be pre-allocated to ensure that the object can always be created. In fact, when an exception occurs, if there is enough space on the heap, the exception object is allocated there. If not, the `XLeaveException` object is created using pre-allocated space on the stack.

Legacy Implementation of `User::Leave()`

If you are working on a version of Symbian OS pre-v9.1, using a build of the Symbian platform where `__LEAVE_EQUALS_THROW__` is not defined, or on a platform where exception support is not available, then the implementation of `User::Leave()` does not use exceptions.

⁸ Note that nested exceptions are supported on the Microsoft Windows emulator, as exceptions are implemented using Win32 structured exception handling. Therefore code that appears correct on the emulator may not function as intended when executed on target hardware.

When a leave occurs, objects on the stack are simply de-allocated and their destructors are not called because the leaving mechanism does not emulate standard C++ throw semantics.⁹ This legacy implementation is why classes intended to be used on the stack (T classes) are not expected to define an explicit destructor nor own resources that need to be cleaned up.

Heap Allocation Using `new(ELeave)`

Let's take a closer look at the use of `new(ELeave)` to allocate an object on the heap, which leaves if the memory is unavailable. If the allocation occurs successfully, no leave occurs and the returned pointer may be used without a further test that the allocation was successful as would otherwise be required. We have already seen it used in the code for `CClanger::NewL()`, shown previously:

```
/*static*/ CClanger* CClanger::NewL()
{
    CClanger* self = new(ELeave) CClanger; // Leaves if OOM
    // ...
}
```

The above code is preferable to the following code, which requires an additional check to verify that the `clanger` pointer has been initialized:

```
/*static*/ CClanger* CClanger::NewL()
{
    CClanger* self = new CClanger;
    if (self)
    {
        CleanupStack::PushL(self);
        self->ConstructL();
        CleanupStack::Pop(self);
    }
    else
        User::Leave(KErrNoMemory);

    return self;
}
```

The `new(ELeave)` expression allocates the necessary memory by means of the `new` operator, passing in the size of the memory required. It then initializes the object accordingly.

⁹ If you experiment and write a T class which has a destructor, create an object on the stack and then force the code to leave, you'll notice that the class destructor is called on Symbian OS v9.x platforms but not on Symbian OS v7.0s, for example.

The Symbian platform overloads the global operator `new` to take a `TLeave` parameter in addition to the size parameter, but the `TLeave` parameter is only used to differentiate this form of operator `new` from the non-leaving version. The overload calls a heap allocation function that leaves if there is insufficient heap memory available:

```
// From e32const.h
enum TLeave {ELeave};

// From e32std.h
inline TAny* operator new(TUint aSize, TLeave);

// e32std.inl
inline TAny* operator new(TUint aSize, TLeave)
    // Call an allocation method that
    // leaves if memory available < aSize
{ return User::AllocL(aSize); }
```

Note that the standard C++ operator `new` is as follows:

```
void* operator new(std::size_t size) throw(std::bad_alloc);
```

The exception specifier shows that it throws a `std::bad_alloc` exception if there is insufficient memory for an allocation. Since the Symbian platform does not do this by default, it makes it impossible to mix calls that rely on the behavior of the Symbian operator `new` with calls to the standard C++ operator `new` in the same link unit. This is because the linker has to pick a single definition of the symbol to resolve against. If it picks the Symbian version, it breaks code that relies on the behavior of the standard C++ operator `new` and, if it picks the C++ version, it breaks Symbian code. You must separate code that uses the Symbian operator `new` into different link units from code which relies on more standard semantics.

Trapping a Leave Using `TRAP` and `TRAPD`

The Symbian platform provides two macros, `TRAP` and `TRAPD`, to trap a leave. The macros differ only in that `TRAPD` declares a variable in which the leave error code is returned, while the program code itself must declare a variable before calling `TRAP`. Thus the following statement:

```
TRAPD(result, MayLeaveL());
if (KErrNone != result)
{ // Handle error }
```

is equivalent to:

```
TInt result;
TRAP(result, MayLeaveL());
if (KErrNone != result)
{ // Handle error }
```

If a leave occurs inside the `MayLeaveL()` function, which is executed inside the harness, the program control returns immediately to the trap harness macro. The variable `result` contains the error code associated with the leave (i.e. that passed as a parameter to the `User::Leave()` system function) or is `KErrNone` if no leave occurred.

Any functions called by `MayLeaveL()` are executed within the trap harness, and so on recursively, and any leave that occurs during the execution of `MayLeaveL()` is trapped, returning the error code into `result`. Alternatively, `TRAP` macros can be nested to catch and handle leaves at different levels of the code, where they can best be dealt with. I'll discuss the run-time cost of using trap harnesses shortly, but if you find yourself using `TRAP` macros several times in one function, or nesting a series of them, you may want to consider whether you can omit trapping all the leaving functions except at the top level or change the layout of the code. Also note that you must not call a function with an `LC` suffix from inside a trap harness. The following code would leave an object on the cleanup stack if no failure arises, causing an `E32USER-CBase 71`:

```
TRAPD(error, NewLC());
```

For example, there may be a good reason why the following function must not leave but needs to call a number of functions which may leave. At first sight, it might seem straightforward enough simply to put each call in a trap harness.

```
TInt MyNonLeavingFunction()
{
    TRAPD(result, FunctionMayLeaveL());
    if (KErrNone == result)
        TRAP(result, AnotherFunctionWhichMayLeaveL());
    if (KErrNone == result)
    {
        TRAP(result, PotentialLeaverL());
        // Handle any error if necessary
    }
    return (result);
}
```

Prior to Symbian OS v9.1, use of the TRAP macro had an impact in terms of executable size and execution speed, and over-use of the trap handler was considered to be an expensive way of managing leaves. Since Symbian OS v9.1, the run-time overhead is much reduced, but you should still aim for simple code, rather than nesting multiple trap handlers ‘just because you can’. It often makes sense to allow leaves to propagate to higher-level code or to combine leaving functions, as follows:

```
TInt MyNonLeavingFunction()
{
    TRAPD(error,
        FunctionMayLeaveL();
        AnotherFunctionWhichMayLeaveL();
        PotentialLeaverL();
    );
    return error;
}
```

Every program (even a simple ‘hello world’ application) must have at least one TRAP, if only at the topmost level, to catch any leaves that are not trapped elsewhere. If you are a GUI application developer you don’t need to worry about this, though, because the Symbian application framework provides a TRAP.

3.5.2 Cleanup Stack

Consider the following example of a call to a leaving function:

```
void UnsafeFunctionL()
{
    // Allocates test on the heap
    CTestClass* test = CTestClass::NewL();
    test->FunctionMayLeaveL(); // Unsafe - a potential memory leak!
    delete test;
}
```

Memory is allocated on the heap in the call to `CTestClass::NewL()` but the subsequent call to `UnsafeFunctionL()` may leave. Should a leave occur, the memory pointed to by `test` is not deallocated (it is said to be ‘orphaned’) and the function has the potential to leak memory.

The Symbian cleanup stack is designed to address this problem and make heap-based objects referenced by local pointers safe against being

orphaned in the event of a leave. This is more simply described as making them ‘leave-safe’.

Which Class Types Risk Becoming Orphaned?

- **T class objects** are usually stack-based and stack objects cannot be orphaned. However, if you create a T class object on the heap, the pointer that references it must be leave-safe.
- **C class objects** are always created on the heap, so they are never leave-safe unless they are accessible for destruction after a leave occurs, for example, if they are stored as the member variables of an object that can be accessed after the leave.
- **R class objects** are generally not leave-safe since the resources they own must be freed in the event of a leave (through a call to the appropriate `Close()` or `Release()` function). If there is no accessible object to make this call after a leave, the resource is orphaned.

Using the Cleanup Stack

The cleanup stack is accessed through the static member functions of class `CleanupStack`, defined in `e32base.h`:

```
class CleanupStack
{
public:
    IMPORT_C static void PushL(TAny* aPtr);
    IMPORT_C static void PushL(CBase* aPtr);
    IMPORT_C static void PushL(TCleanupItem anItem);
    IMPORT_C static void Pop();
    IMPORT_C static void Pop(TInt aCount);
    IMPORT_C static void PopAndDestroy();
    IMPORT_C static void PopAndDestroy(TInt aCount);
    IMPORT_C static void Check(TAny* aExpectedItem);
    inline static void Pop(TAny* aExpectedItem);
    inline static void Pop(TInt aCount, TAny* aLastExpectedItem);
    inline static void PopAndDestroy(TAny* aExpectedItem);
    inline static void PopAndDestroy(TInt aCount,
                                     TAny* aLastExpectedItem);
};
```

Objects that are not otherwise leave-safe should be placed on the cleanup stack before calling code that may leave. This ensures they are destroyed correctly. If a leave occurs, the cleanup stack manages the deallocation of all objects that have been placed upon it (within the current enclosing TRAP).

The following code illustrates a leave-safe version of the `UnsafeFunctionL()` example shown previously:

```
void SafeFunctionL()
{
    CTestClass* test = CTestClass::NewL();
    // Push onto the cleanup stack before calling leaving function
    CleanupStack::PushL(test);
    test->FunctionMayLeaveL();
    // Pop from cleanup stack
    CleanupStack::Pop(test);
    delete test;
}
```

If `FunctionMayLeaveL()` leaves, the `test` object is destroyed by the cleanup stack as part of leave processing, which we'll discuss shortly. If `FunctionMayLeaveL()` does not leave, the code continues, pops the pointer from the cleanup stack and calls `delete` on it. (This code could equally well be replaced by a single call to `CleanupStack::PopAndDestroy(test)` which performs both the pop and a call to the destructor in one step.)

Why Is `PushL()` a Leaving Function?

`PushL()` is a leaving function because it may allocate memory to store the cleanup pointer and this allocation may fail in low-memory situations. However, you don't need to worry that the object you are pushing onto the cleanup stack will be orphaned if a leave does occur. The cleanup stack is created with at least one spare slot. When you call `PushL()`, the pointer you pass is first added to the vacant slot and, if there are no more vacant slots available, the cleanup stack code then attempts to allocate a slot for the next `PushL()` operation. If that allocation fails, a leave occurs, but your pointer has already been stored safely and the object it refers to is safely cleaned up.

In fact, the cleanup stack code is more efficient than my simplistic explanation; it allocates four slots at a time, by default. In addition, it doesn't release slots when you `Pop()` objects out of them, so a `PushL()` frequently does not make any allocation. This can be useful in circumstances where you acquire more than one pointer that is not leave-safe to push onto the cleanup stack. You can use this knowledge to expand the cleanup stack to contain at least the number of slots you need, then create the objects and safely push them onto the vacant slots.

Popping Items off the Cleanup Stack

Objects are pushed onto and popped off the cleanup stack in strict order: a series of `Pop()` calls must occur in the reverse order of the `PushL()` calls. The following example illustrates this:

```
void ContrivedExampleL()
{
    // Note that each object is pushed onto the cleanup stack
    // immediately it is allocated, in case the succeeding
    // allocation leaves.
    CSiamese* sealPoint = NewL(ESeal);
    CleanupStack::PushL(sealPoint);
    CSiamese* chocolatePoint = NewL(EChocolate);
    CleanupStack::PushL(chocolatePoint);
    CSiamese* violetPoint = NewL(EViolet);
    CleanupStack::PushL(violetPoint);
    CSiamese* bluePoint = NewL(EBLue);
    CleanupStack::PushL(bluePoint);
    sealPoint->CatchMouseL();

    // Other leaving function calls,
    // some of which use the cleanup stack
    ...

    // All calls have successfully completed:
    CleanupStack::PopAndDestroy(bluePoint);
    CleanupStack::PopAndDestroy(violetPoint);
    CleanupStack::PopAndDestroy(chocolatePoint);
    CleanupStack::PopAndDestroy(sealPoint);
}
```

It is possible to `Pop()` or `PopAndDestroy()` one or more objects without naming them, but it's a good idea to name the objects as they are popped off. This makes your code clearer and helps you guard against a 'cleanup stack imbalance' bug, which can be difficult to find because it causes unexpected panics in code quite unrelated to where the error has occurred. If you explicitly pass the name of the pointer when you pop it off the cleanup stack, in debug builds a check is made to confirm that the pointer being popped off is indeed the one you expect it to be. Because it only occurs in debug builds, it has no impact on the speed of released code, although, if you do want to perform checking in a release build, you can use the `CleanupStack::Check()` function.

There are alternative overloads to `Pop()` and `PopAndDestroy()`. For the previous example, the preferred way to destroy the four objects altogether would be this:

```
// Destroy all, naming the last object
CleanupStack::PopAndDestroy(4, sealPoint);
```

If an object is pushed onto the cleanup stack in a function and remains on it when the function returns, we introduce a further naming convention: the function name must have the additional suffix 'C'. This indicates to the caller that, when the function returns successfully, the cleanup stack has additional objects upon it. This idiom is typically used by CBase-derived classes that define static functions to instantiate an instance of the class and leave it on the cleanup stack. The C suffix indicates to the caller of a function that it is not necessary to push any objects allocated by the function onto the cleanup stack.

The following code creates an object of type CSiamese and leaves it on the cleanup stack. This function is useful because the caller can instantiate CSiamese and immediately call a leaving function, without needing to push the allocated object back onto the cleanup stack:

```
CSiamese* CSiamese::NewLC(TPointColor aPointColour)
{
    CSiamese* self = new(ELeave) CSiamese(aPointColour);
    CleanupStack::PushL(self); // Make this leave-safe...
    self->ConstructL();
    return (self) // Leave self on the cleanup stack for
                // the caller to Pop()
}
```

Member Variables and the Cleanup Stack

While heap variables referred to only by local variables may be orphaned, member variables do not suffer a similar fate (unless their destructor neglects to delete them when it is called at some later point). Thus the following code is safe:

```
void CTestClass::SafeFunctionL()
{
    iMember = CClangerClass::NewL(); // Allocates a heap member
    FunctionMayLeaveL();              // Safe
}
```

Even if a leave occurs in FunctionMayLeaveL(), iMember is stored safely as a pointer member variable, to be deleted later through the class destructor. If iMember was pushed onto the cleanup stack, it would be destroyed if a leave occurred, but later the CTestClass destructor would also attempt to destroy it. This could lead to a USER-42 double-deletion panic. It should never be possible for an object to be cleaned up more than once, so you should never push class member variables (objects prefixed by 'i', if the Symbian C++ naming convention is followed) onto the cleanup stack. That's where the naming convention comes in useful – if you ever find yourself writing or reviewing code that uses PushL(iSomething), your inner alarm bell should ring.

Using the Cleanup Stack with Non-CBase Classes

Up to this point, we've only really considered one of the overloads of the `CleanupStack::PushL()` function, `PushL(CBase* aPtr)`, which takes a pointer to a CBase-derived object. When `CleanupStack::PopAndDestroy()` is called for that object, or if leave processing occurs, the object is destroyed by invoking `delete` on the pointer, which calls the virtual destructor of the CBase-derived object.¹⁰

If an object does not derive from CBase, there are two alternative overloads of `PushL()` that may be used.

`CleanupStack::PushL(TAny*)`

The simplest overload, `CleanupStack::PushL(TAny*)`, can be used to push any item onto the cleanup stack. Using this overload means that if the cleanup stack later comes to destroy the object, its heap memory is simply deallocated (by invoking `User::Free()`) and no destructors are called on it. For simple heap-based objects, such as T classes with no destructors, this is fine. But otherwise you may find the presence of this overload leads to unexpected behavior. If a class does not inherit from CBase, directly or indirectly, and you push a pointer to a heap-based object of the type onto the cleanup stack, then no destructors are called if a leave occurs. The object may not be cleaned up as you expect.

A common error is to define a C class and forget to derive it from CBase, causing the incorrect overload to be used. A memory leak occurs if the cleanup stack destroys it, as the destructor is not called and any resources owned are not freed.

So what's the use of the `CleanupStack::PushL(TAny*)` overload? As I've already implied, it is used when you push onto the cleanup stack a pointer to a heap-based object which does not have a destructor, for example, a T class object or a struct which has been allocated on the heap. Recall that, until Symbian OS v9.1, T classes were forbidden from having destructors, to make them leave-safe on the stack. They thus they had no requirement for cleanup beyond deallocation of the heap memory they occupy. You can give a T class a destructor these days and it is still leave-safe on the stack. But, as the previous discussion should have warned you, you can't expect it to be leave-safe on the heap if you use the cleanup stack and require a destructor call for correct cleanup.

When you push heap descriptor objects (of class `HBufC`, described in Chapters 5 and 6) onto the cleanup stack, the `CleanupStack::PushL(TAny*)` overload is used too. This is because `HBufC` objects are, in effect, heap-based T class objects. Objects of type `HBufC` require no

¹⁰ As specified by C++, the object is destroyed by calling destructor code in the most-derived class first, moving up the inheritance hierarchy calling destructors in turn and eventually calling the empty CBase destructor. It is to this end that CBase has a virtual destructor – so that C class objects can be placed on the cleanup stack and destroyed safely if a leave occurs.

destructor invocation, because they contain only plain, built-in-type data. The only cleanup necessary is that required to free the heap cells for the descriptor object.

CleanupStack::PushL(TCleanupItem)

`CleanupStack::PushL(TCleanupItem)` takes an object of type `TCleanupItem`, which allows for customized cleanup routines. A `TCleanupItem` object encapsulates a pointer to the object to be stored on the cleanup stack and a pointer to a function that provides cleanup for that object. The cleanup function can be a local function or a static method of a class. `LeaveProcessing()`, or a call to `PopAndDestroy()`, removes the object from the cleanup stack and calls the cleanup function provided.

You may be wondering how the cleanup stack discriminates between a `TCleanupItem` and objects pushed onto the cleanup stack using one of the other `PushL()` overloads. The answer is that it doesn't – all objects stored on the cleanup stack are actually of type `TCleanupItem`. The `PushL()` overloads that take `CBase` or `TAny` pointers construct a `TCleanupItem` implicitly and store it on the cleanup stack.

As I've already described, the cleanup function for the `CBase*` overload deletes the `CBase`-derived object through its virtual destructor. For the `TAny*` overload, the cleanup function calls `User::Free()`, which simply frees the allocated memory:

```
void CCleanup::PushL(TAny *aPtr)
{
    PushL(TCleanupItem(User::Free, aPtr));
}
```

The Symbian platform also provides three utility functions, each of which generates an object of type `TCleanupItem` and pushes it onto the cleanup stack. The functions use templates so the cleanup methods do not have to be static. These methods are particularly useful when making R classes leave-safe, since they call cleanup methods `Release()`, `Delete()` and `Close()`. For further information about `CleanupReleasePushL()`, `CleanupDeletePushL()` and `CleanupClosePushL()`, please see the Symbian Developer Library documentation or the Fundamentals of Symbian C++ wiki pages on developer.symbian.org.

3.5.3 Leaves and the Cleanup Stack

We can now tie together the implementation of leaves (in terms of standard C++ exceptions) and the cleanup stack.¹¹ When a leave occurs:

¹¹ See [developer.symbian.org/wiki/index.php/Leaves_&_The_Cleanup_Stack_\(Fundamentals_of_Symbian_C++\)](http://developer.symbian.org/wiki/index.php/Leaves_&_The_Cleanup_Stack_(Fundamentals_of_Symbian_C++)).

1. The cleanup stack ‘unwinds’ and each item on it is cleaned up (e.g. destructors are called for C classes, `Close()` is called on any R class object made leave-safe by a call to `CleanupClosePushL()`, etc.).
2. An `XLeaveException` is created and thrown to represent the leave, resulting in the unwinding of the stack.
3. The exception is caught by the trap handler.

During Step 1, the cleanup stack takes responsibility for cleaning up, for example, heap-based C class objects and calling `Close()` on R class objects. At this point, no exception has been raised for the leave and, in the cleanup initiated by the cleanup stack, it is acceptable to leave or otherwise throw an exception, because it won’t be nested.

During Step 2, the normal stack unwinds and the destructors of stack-based objects are called. If a leave were to occur in one of those destructors, it would generate an exception that would be nested within the `XLeaveException` exception thrown for the original leave. On mobile device hardware, this would cause the thread to be terminated.

At Step 3, exception handling is complete. Another leave or exception may occur without being nested in the exception thrown in Step 2.

If you write a destructor that leaves, it does not terminate the thread if the destructor is called by the cleanup stack (for example, for C class objects). However, if a destructor is called as the normal stack unwinds, you must not allow leaves or exceptions, because they have the potential to cause a nested exception that terminates the thread when running on target hardware.

On the Symbian platform, it is actually quite rare for any stack-based object to have a destructor since, until Symbian OS v9.1, T classes could not guarantee that their destructor would be called (leaves did not unwind the stack before Symbian OS v9.1). Likewise, R classes typically perform cleanup in their `Close()` method. However, if you do have a stack-based object with a destructor, or use an `auto_ptr` style of programming, you must be aware of the limitation and avoid code that leaves or otherwise throws exceptions. For example, if a stack-based `auto_ptr` destructor cleans up a heap-based object, the destructor code it calls must not leave or throw an exception either.

Of course, it is questionable whether it is ever sensible to leave in a destructor, since this suggests that the cleanup code may not completely execute, with the potential to cause memory or resource leaks. Instead of leaving in a destructor, it’s preferable to have a separate leaving function, which can be called before destruction, to perform actions that may fail and provide the caller an opportunity to deal with the problem before finally destroying the object. Typically these would be functions such as `CommitL()` or `FreeResourceL()` and they effectively form a ‘two-phase destruction’ pattern, analogous with two-phase construction. If

you use the cleanup stack to destroy an object which requires two-phase destruction, you must ensure that you accommodate the separate cleanup function in the event of a leave.

3.5.4 Two-Phase Construction

Let's examine what happens when you create an object on the heap, by considering the following line of code that allocates an object of type `CExample`, and sets the value of the `foo` pointer accordingly:

```
CExample* foo = new CExample;
```

The code calls the `new` operator, which allocates a `CExample` object on the heap if there is sufficient memory available. Having done so, it calls the `CExample` constructor to initialize the object. You can use the cleanup stack to ensure that the heap object is correctly cleaned up in the event of a leave. But it is not possible to do so inside the `new` operator between allocation of the object and invocation of its constructor.

If the `CExample` constructor itself leaves, a C++ exception (`XLeaveException`) is thrown, as described in Section 3.5.1. The semantics of C++ exception handling ensure that the memory allocated for the `CExample` object is cleaned up, so no memory leak occurs. However, since the exception occurs in the constructor, the destructor of `CExample` is not itself called. So any memory the constructor may have allocated before the leave occurred is orphaned, unless it is made leave safe.

Furthermore, consider the following:

```
CExample* foo = new(ELeave) CExample;
```

If a `CExample` object is successfully allocated but its constructor leaves, throwing an exception, because the Symbian platform doesn't define a placement delete to pair with, there is no compiler-generated cleanup code. So, when the exception occurs, there is no de-allocation code to free the `CExample` object back to the heap. If the object was instead allocated by a call to the non-leaving `new`, the same object is cleaned up. Take a look at the following code for illustration:

```
class CWidget : public CBase
{
public:
```



```

CWidget() { throw std::exception(); }
~CWidget() {}
...
};

// Calling code
LOCAL_C void MainL()
{
    CWidget* widgetSafe = new CWidget;           // Memory cleaned up
    CWidget* widgetLeak = new(ELeave) CWidget;    // Memory orphaned
}

```

The memory allocated for `widgetSafe` is freed back to the heap when `CWidget()` leaves. In contrast, `widgetLeak` is *not* cleaned up when `CWidget()` leaves.

This is very important to keep in mind when porting code which uses smart pointers:

```

boost::scoped_ptr<CWidget> ptr(new CWidget);      // OK
boost::scoped_ptr<CWidget> ptr(new(ELeave) CWidget); // Leaks

```

This example shows an artificial case, where an exception was thrown deliberately. But to initialize an object of a C class, it is quite common to need to write code that throws, say to allocate memory or read a configuration file, which may be missing or corrupt. It is under these circumstances that, to avoid the potential for memory leaks when construction fails, the two-phase construction idiom is used.

Two-phase construction of C class objects¹² is an idiom that you'll see used extensively in Symbian C++ code. It breaks the construction code into two parts, or phases :

- A basic constructor that cannot leave, which is called by the `new` operator. The constructor may invoke functions that cannot leave or initialize member variables with default values or those supplied as arguments.
- A class method (typically called `ConstructL()`). This method may be called separately once the object has been allocated and the constructor has executed. The object should first be pushed onto the cleanup stack to make it safe in the event of a leave. If a leave occurs, the cleanup stack calls the class destructor to free any resources that

¹² Two-phase construction is typically only used for C classes, since T classes do not usually require complex construction code (because they do not contain heap-based member data) and R classes are usually created uninitialized, requiring their callers to call `Connect()` or `Open()` to associate the R class object with a particular resource.

have been successfully allocated, and the memory allocated for the object is freed back to the heap.

The following code shows a simple, but not perfect, example of two-phase construction:

```
class CExample : public CBase
{
public:
    CExample();    // Guaranteed not to leave
    ~CExample();   // Must cope with partially constructed objects
    void ConstructL(); // Second-phase construction code may leave
    ...
};
```

The simple implementation shown here expects clients to call the second-phase construction function themselves, but a caller may forget to call the `ConstructL()` method after instantiating the object and, at the least, will find it a burden since it's not a standard C++ requirement. For this reason, it is preferable to make the call to the second-phase construction function within the class itself. A commonly used pattern in Symbian C++ code is to provide a static function that wraps both phases of construction, providing a simple and easily identifiable means to instantiate objects on the heap. The function is typically called `NewL()` and a `NewLC()` function is often provided too, which is identical except that it leaves the constructed object on the cleanup stack for convenience.¹³

```
class CExample : public CBase
{
public:
    static CExample* NewL();
    static CExample* NewLC();
    ~CExample();   // Must cope with partially constructed objects
private:
    CExample();    // Guaranteed not to leave
    void ConstructL(); // Second-phase construction code may leave
    ...
};
```

Note that the `NewL()` function is static, so you can call it without first having an existing instance of the class. The non-leaving constructors and second-phase `ConstructL()` functions have been made `private`¹⁴ so a caller cannot instantiate objects of the class except through `NewL()`.

¹³ This is also known as the 'named constructor' idiom.

¹⁴ If you intend your class to be subclassed, you should make the default constructor protected rather than `private` so the compiler may construct the deriving classes. The `ConstructL()` method should be `private` (or `protected` if it is to be called by derived classes) to prevent clients of the class from mistakenly calling it on an object which has already been fully constructed.

Typical implementations of `NewL()` and `NewLC()` may be as follows:

```
CExample* CExample::NewLC()
{
    CExample* self = new(ELeave) CExample; // First-phase construction
    CleanupStack::PushL(self);
    self->ConstructL();                    // Second-phase construction
    return (self);
}

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

Note that the `NewL()` function is implemented in terms of the `NewLC()` function rather than the other way around (which would be slightly less efficient since this would make an extra `PushL()` call on the cleanup stack). As a rule of thumb, `NewL()` should be called when initializing member variables, whereas `NewLC()` should be called when initializing local variables.

Each factory function returns a fully constructed object, or leaves if there is insufficient memory to allocate the object or if the second phase `ConstructL()` function leaves. If second-phase construction fails, the cleanup stack ensures both that the partially constructed object is destroyed and that the memory it occupies is returned to the heap.

If your class derives from a base class which also implements `ConstructL()`, you need to ensure that it is also called, if necessary, when objects of your class are constructed (C++ ensures that the simple first-phase constructors of your base classes are called). You should call the `ConstructL()` method of any base class explicitly (using the scope operator) in your own `ConstructL()` method, to ensure the base class object is fully constructed before proceeding to initialize your derived object.

If it is possible to `PushL()` a partially constructed object onto the cleanup stack in a `NewL()` function, why not do so at the beginning of a standard constructor, calling `Pop()` when construction is complete?

```
CWidget::CWidget()
{
    CleanupStack::PushL(this);
    ConstructL();                    // Second-phase construction
    CleanupStack::Pop(this);
}
```

At first sight, this may be tempting, since the single-phase construction I described as unsafe at the beginning of the chapter would now be leave-safe, as long as the object was pushed onto the cleanup stack before any leaving operations were called. But there are several arguments against this approach:

- If the class has a deep inheritance hierarchy, each constructor with potential to leave would call `PushL()` and `Pop()`, which is less efficient than the single use of the cleanup stack in a separate factory function.
- Another, more trivial, argument is that a C++ constructor cannot be marked with a suffixed `L` to indicate its potential to leave unless the class is itself named as such.
- The most significant reason for preferring two-phase construction was described in the June 2008 Code Clinic article at [**developer.symbian.org/wiki/index.php/The_Implications_of_Leaving_in_a_Constructor**](http://developer.symbian.org/wiki/index.php/The_Implications_of_Leaving_in_a_Constructor) and is summarized in the following paragraphs.

If `ConstructL()` leaves, the following sequence occurs:

1. The cleanup stack ‘unwinds’ and destroys the object, explicitly calling `CWidget::~~CWidget()`, and then frees the memory allocated for the object.
2. An `XLeaveException` exception is created and thrown to represent the leave, the normal stack is unwound and the exception is caught by the trap handler.

When an exception is thrown in a constructor, the destructor of that object is not called since the object has not yet been constructed. However, C++ guarantees to call the destructor of any fully-constructed superclasses in the event of an exception. Since the superclass parts of the object *have* been constructed, their destructors are called and the memory allocated for them is freed. So, if `CWidget` derives from `CGadget`, as shown below, the `CGadget` superclass destructor is called twice – first by the cleanup stack and then as a result of the exception. The second time the destructor is invoked, it is on something that has already been destroyed and freed back to the heap. If it attempts to access any member variables, it causes a memory fault, which results in a `USER 42 panic`.

```
class CGadget : public CBase
{
public:
    CGadget();
    ~CGadget() {delete iBuf};
```

```
private:
    HBufC* iBuf;
};

class CWidget : public CGadget
{
public:
    CWidget();
    ~CWidget() {};
private:
    void InitializeL();
};
```

The lesson to learn is that, if a class derives from other classes that have explicit and implemented destructors, you *must* defer calling any leaving code on construction until *after* the standard C++ constructor has completed. Or, put simply, just keep using two-phase construction to ensure leaving code is called *after* the constructor has fully executed. Double-deletion wasn't an issue before leaves were implemented in terms of C++ exceptions in Symbian OS v9.1, but fortunately the two-phase construction idiom has always been employed and encouraged for Symbian C classes, so construction code didn't need to be updated to account for the change in implementation.

3.5.5 Panics

On the Symbian platform, panics are used to highlight a programming error in the most noticeable way possible. When a thread is panicked, the code in it stops running to ensure that the developer is aware of the programming error and fixes the problem. You can't continue running the code until you fix the problem and stop the panic from occurring. Unlike a leave, a panic can't be trapped. The code terminates and there is no recovery.

When a panic occurs in the main thread of a process, the entire process in which the thread is running terminates. On phone hardware, a panic in an application's main thread unceremoniously closes the application without warning.¹⁵

¹⁵ When a panic occurs in a secondary thread, it is only that thread which closes, unless the thread has been set as process critical or process permanent, in which case the process also panics. A call to `User::SetCritical()` is necessary to set a thread to process critical or permanent. `User::SetCritical()` can also be used to set a thread as system critical or system permanent so that, when code runs on phone hardware, a panic in the thread deliberately reboots the phone. See the developer library documentation for further information about this call.

On the emulator, in a debug build, you can choose to break into the code to debug the cause of the panic – this is known as just-in-time debugging. If just-in-time debugging is enabled, a panic stops the emulator and enters the debugger to allow you to diagnose what caused the panic. The debugger is launched within the context of the function that called the panic, using `__asm int 3`. You can use the debugger to look through the call stack to see where the panic arose and examine the problem code.

A call to the static function `User::Panic()` panics the currently running thread. A thread may panic any thread in the *same* process by calling `RThread::Panic()`, but cannot panic threads in any *other* process. If a thread tries to panic a thread in another process, it is itself panicked with `KERN-EXEC 46`.¹⁶

When calling a `Panic()` method, you are required to pass in a descriptor to specify the ‘category’ of the panic and a `TInt` value to identify the panic (known as the panic ‘reason’). You can use any length of panic category string that you wish, but it is truncated to 16 characters when it’s reported. The panic category and reason can be useful for tracking down the cause of a panic, for example, if you don’t have access to the code for debugging. Symbian publishes a set of panic values, listed by category, in the reference section of the Symbian Developer Library documentation. You may encounter the following typical values:

- `KERN-EXEC 3` is raised by an unhandled exception, such as an access violation, caused, for example, by dereferencing `NULL` or an invalid pointer. For example:

```
TInt* myIntegerPtr; // Not initialized
*myIntegerPtr = 5; // Panics with KERN-EXEC 3
```

Another reason for this panic is stack overflow. By default, the stack size assigned to a thread is 8 KB (which can be extended by means of `epocstacksize` up to 80 KB). This is a place where behavior differs between emulator and device. As a rule of thumb, if you get this panic on the device but it’s not reproducible on the emulator, the stack has probably overflowed.

- `E32USER-CBASE 46` is raised by the active scheduler as a result of a stray signal (described further in Section 3.6.2).

¹⁶ There is an exception to this rule where a server thread uses the `RMessagePtr2` API to panic a misbehaving client thread, for example, when a client passes a badly-formed request. Rather than attempt to read or write to a bad descriptor, the server protects itself and flags the problem by causing a panic in the client thread. Generally, a malformed client request occurs because of a programming error in client code, but this strategy also protects a server against more malicious ‘denial-of-service’ attacks in which a client may deliberately pass a badly-formed or unrecognized request to a server to try to crash it.

- `E32USER-CBASE 90` is raised by the cleanup stack when the object specified to be popped off is not the next object on the stack (as discussed in Section 3.5.2).
- `USER 11` is raised when an operation to modify a 16-bit descriptor fails because the operation would cause the descriptor's data area to exceed its maximum allocated length (as described in Section 3.4.4).
- `ALLOC xxxxxxxxx` is raised in debug builds when the heap-checking macros (i.e. `__UHEAP_MARK` and `__UHEAP_MARKEND`) detect that more heap cells have been allocated than freed. This is discussed in Chapter 2.

This section is based on an extended discussion of panics, which can be found in the November 2008 Code Clinic article at ***developer.symbian.org/wiki/index.php/Symbian_Panics_Explained***.

3.6 Event-Driven Programming

Consider what happens when program code makes a function call to request a service. The service can be performed either synchronously or asynchronously:

- A *synchronous* function performs the service to completion and only then returns to its caller with an indication of its success or failure as a return value or as a leave.
- An *asynchronous* function submits a request and returns to its caller, but completion of that request occurs some time later. On the Symbian platform, asynchronous functions can be identified as those taking a `TRequestStatus` reference parameter, which is used to receive the result (an error code) of the request when it completes.

Symbian OS, like other operating systems, uses event-driven code extensively, both at a high level (e.g., for user interaction) and at a lower, system level (e.g., for asynchronous communications input and output). To be responsive, the operating system must have an efficient event-handling model to handle each event as soon as possible after it occurs and, if more than one event occurs, in the most appropriate order. It is particularly important that user-generated events are handled rapidly to give feedback and a good user experience. Between events, the system should wait in a low-power state. This avoids constant polling, which can lead to significant power drain and should be avoided on a battery-powered device. Instead the software should allow the operating system to move to an idle mode while it waits for the next event to occur.

3.6.1 Threads and Active Objects

On the Symbian platform, threads are scheduled by the kernel, switching between them to give the impression that multiple threads are running concurrently. The kernel runs the highest-priority thread eligible and controls thread scheduling *preemptively*. This means that it allows threads to share system resources by time-slice division, but it preempts the running thread if another, higher-priority, thread becomes eligible to run.

A *context switch* occurs whenever the current thread is suspended (for example, if it becomes blocked, has reached the end of its time-slice, or a higher priority thread becomes ready to run) and another thread is made eligible by the kernel scheduler. The context switch incurs a run-time overhead in terms of the kernel scheduler and, if the original and replacing threads are executing in different processes, the memory management unit and hardware caches.

On the Symbian platform, an alternative to using multiple threads is to use active objects. A number of active objects can run in a single thread but, within that single thread, they exhibit *non-preemptive* multitasking: once an active object is running, it cannot be replaced until it has finished.¹⁷

A Symbian application or server often consists of a single, main event-handling thread. A set of active objects run in the thread. Each active object encapsulates an asynchronous service provider. To that provider, it requests an asynchronous service, waits while it is serviced, then handles the completion event. On the Symbian platform, the use of multiple active objects is often preferable to multiple threads:

- A switch between active objects that run in the same thread incurs a lower overhead than a thread context switch. The difference in speed between a context switch between threads and transfer of control between active objects in the same thread can be a factor of 10 in favor of active objects. In addition, the space overhead for a thread can be around 4 KB kernel-side and 8 KB user-side for the program stack, while the size of an active object may be only a few hundred bytes, or less.
- Apart from the run-time expense of a context switch, using preemptive multithreading for event handling can be inconvenient because of the need to protect shared objects with synchronization primitives such as mutexes or semaphores. Since active objects are non-preemptive within a single thread, there is no need for synchronization protection of shared resources.

¹⁷ Although the active objects within a thread are non-preemptive, the thread in which they run is scheduled preemptively.

- Resource ownership is thread-relative by default on the Symbian platform. If a file is opened by the main thread it is not possible for a different thread in the process to use it without the handle being explicitly shared through a call to `RSessionBase::Share()` (and some Symbian platform servers do not support session sharing at all). Because of this restriction, it may be very difficult to use multiple threads as a viable method for event-driven multitasking. Active objects run in the same thread, which means that memory and resources may be shared more readily.

3.6.2 Working with Active Objects

A Symbian platform thread that uses active objects for event handling also has an ‘active scheduler’ (see `CActiveScheduler` in `e32base.h`) that passes event completion signals to their corresponding active objects. Each active object requests an asynchronous service, which generates an event signal some time after the request. The active scheduler detects those event signals, determines which active object is associated with each, and calls it to handle the event. While an active object is handling an event, other signals may be received by the scheduler, but it cannot schedule another active object to run until the currently running active object has finished its event-handling routine.

Construction

Classes deriving from `CActive` must call the protected constructor of the base class, passing in a value to set the priority of the active object. The priority value is used by the active scheduler to determine the order in which event signals are handled if multiple active objects are ready to be ‘scheduled’ at once. That is, while one event is being handled, it is quite possible that a number of additional events may complete. The scheduler must resolve which active object gets to run next. It does this by ordering the active objects using their priority values, handling them sequentially in order of the highest active object priority rather than in order of their completion.

This makes sense if the system is to remain responsive – otherwise, an event of low priority that completed just before a more important one would lock out the higher-priority event for an undefined period, depending on how much code the low-priority active object executed to handle its associated event.

As you can see in `e32base.h`, a set of priority values is defined in the `TPriority` enumeration of class `CActive`. You use the priority value `EPriorityStandard` (`=0`) unless you have a good reason to do otherwise.

As part of construction, the active object code should call a static function on the active scheduler, `CActiveScheduler::Add()`. This

adds the object to a list maintained by the active scheduler of event-handling active objects on that thread.

```
CMyActive::CMyActive()  
: CActive(CActive::EPriorityStandard)  
{  
    CActiveScheduler::Add(this);  
}
```

An active object typically owns an object to which it issues requests that complete asynchronously, generating an event. An object implementing such methods is usually known as an asynchronous service provider and may need to be initialized as part of construction. If the initialization can fail, you should perform it as part of the second-phase construction, as described before.

```
class CMyActive : public CActive  
{  
private:  
    CProvider* iProvider;  
    // ...  
};
```

Submitting Requests

An active object class supplies public methods for callers to initiate requests. They submit requests to the asynchronous service provider associated with the active object using a well-established pattern, as follows:

1. Request methods should check that there is no request already submitted before attempting to submit another. Depending on the implementation, the code may:
 - panic if a request has already been issued (if this scenario could only occur because of a programming error)
 - refuse to submit another request (if it is legitimate to attempt to make more than one request but once one is outstanding all others are ignored)
 - cancel the outstanding request and submit the new one.
2. The active object should then issue the request to the asynchronous service provider, passing in its `iStatus` member variable as the `TRequestStatus&` parameter.

3. If the request is submitted successfully, the request method then calls the `SetActive()` method of the `CActive` base class to indicate to the active scheduler that a request has been submitted and is currently outstanding. This call is not made until after the request has been submitted.

Event Handling

Each active object class must implement the pure virtual `RunL()` member method of the `CActive` base class. When a completion event occurs from the associated asynchronous service provider and the active scheduler selects the active object to handle the event, it calls `RunL()` on the active object. The `RunL()` function has a somewhat misleading name because the asynchronous function has already run. Perhaps a clearer name would be `HandleEventL()` or `HandleCompletionL()`.

Typical implementations of `RunL()` determine whether the asynchronous request succeeded by inspecting the completion code in the `TRequestStatus` object (`iStatus`) of the active object, a 32-bit integer value. Depending on the result, `RunL()` usually either issues another request or notifies other objects in the system of the event's completion; however, the degree of complexity of code in `RunL()` can vary considerably.

```
void CMyActive::RunL()
{
    // check the iStatus for possible errors and
    // resubmit the request if necessary
}
```

Whatever it does, once `RunL()` is executing, it cannot be preempted by other active objects' event handlers. For this reason, the code should do what it needs to in as short a time as possible so that other events can be handled without delay.

Error Handling

`CActive` provides a virtual `RunError()` method which the active scheduler calls if a leave occurs in the `RunL()` method of the active object. The method takes the leave code as a parameter and returns an error code to indicate whether the leave has been handled. The default implementation does not handle the leave and simply returns the leave code passed to it. If the active object can handle any leaves occurring in `RunL()`, it should do so by overriding the default implementation and returning `KErrNone` to indicate that the leave has been handled.

If `RunError()` returns a value other than `KErrNone`, indicating that the leave has yet to be dealt with, the active scheduler calls its `Error()` function to handle it. The active scheduler does not have any contextual information about the active object with which to perform error handling. For this reason, it is generally preferable to manage error recovery within the `RunError()` method of the associated active object, where more context is usually available.

Cancellation

An active object must be able to cancel any outstanding asynchronous requests it has issued. The `CActive` base class implements a `Cancel()` method which calls the pure virtual `DoCancel()` method (which every derived active object class must implement) and waits for the request's early completion. Any implementation of `DoCancel()` should call the appropriate cancellation method on the asynchronous service provider. `DoCancel()` can also include other processing but should not carry out any lengthy operations. It's a good rule to restrict the method to cancellation and any necessary cleanup associated with the request, rather than implementing any sophisticated functionality. This is because a destructor should call `Cancel()` but it may have already cleaned up resources that `DoCancel()` requires. As was already explained, you should not allow code to leave in a destructor, so `DoCancel()` should not leave either.

```
void CMyActive::DoCancel()
{
    iProvider->CancelAll();
}
```

It isn't necessary to check whether a request is outstanding for the active object before calling `Cancel()`, because it is safe to do so even if it isn't currently active.

Destruction

The destructor of a `CActive`-derived class should always call `Cancel()` to terminate any outstanding requests as part of cleanup code. The `CActive` base class destructor is virtual and its implementation checks that the active object is not currently active. It panics if any request is outstanding, that is, if `Cancel()` has not been called. This catches any programming errors which could lead to the situation where a request completes after the active object to handle it has been destroyed. This would otherwise result in a 'stray signal', where the active scheduler cannot locate an active object to handle the event.

Having verified that the active object has no issued requests outstanding, the `CActive` destructor removes the active object from the active scheduler. The destructor code should also free all resources owned by the object, as usual, including any handle to the asynchronous service provider.

3.7 Writeable Static Data

When porting an existing project, you'll probably find yourself working with DLLs. Symbian OS – like most operating systems – supports static (LIB target) and dynamic (DLL target) libraries, but some extra care should be taken when using global variables in DLLs, as it can be expensive in terms of memory usage. This is because writeable static data (WSD) in a DLL typically consumes 4 KB of memory for *each* process into which the DLL is loaded.

When a process loads its first DLL containing WSD, it creates a single chunk to store the data. The minimum size of a chunk is 4 KB (the size of a page frame) and that amount of memory is consumed irrespective of how much static data is actually required. Any memory not used for WSD is wasted (however, if subsequent DLLs are loaded into the process that also use WSD, the same chunk can be used rather than separate 4 KB chunks for every DLL that uses WSD).

Since the memory is per process, the potential memory wastage is:

$$(4\text{KB} - \text{WSD bytes}) \times \text{number of client processes}$$

If, for example, a DLL is used by four processes, that's potentially an 'invisible' cost of 16 KB (minus the memory occupied by the WSD itself).

There are a number of occasions where the benefits of using WSD in a DLL outweigh the disadvantages (for example, when porting code that uses WSD significantly, and for DLLs that are only ever loaded into one process). Note, however, that the current supported version of the GCC-E compiler has a defect such that DLLs with static data may cause a panic during loading.

3.7.1 Usage

There are occasions where you may find you use WSD inadvertently. Some Symbian C++ classes have a non-trivial constructor, which means that the objects must be constructed at run time. You may not think you are using WSD but, because an object isn't initialized until the constructor for the class has executed, it counts as modifiable rather than constant. Here are a few examples:

```
static const TPoint KGlobalStartingPoint(50, 50);
static const TChar KExclamation('!');
static const TRgb KDefaultColour(0, 0, 0);
```

On most versions of the Symbian platform, if you attempt to build DLL code that uses WSD, deliberately or inadvertently, you get a error when you build the DLL for phone hardware. The error looks something like this:

```
ERROR: Dll 'TASKMANAGER[1000C001].DLL' has uninitialised data
```

A DLL that uses WSD will always build for the Microsoft Windows emulator. It is only when code is compiled for phone hardware that the use of WSD is flagged as an error. If you are sure you want to use WSD, you can explicitly enable it by adding the keyword `EPOCALLOWDLLDATA` to the DLL's MMP file.

However, if you want to avoid the potential of extra memory consumption and emulator testing limitations, there is an alternative mechanism available: thread local storage (TLS). TLS can be used in the implementation of Singleton in DLLs on all versions of the Symbian platform (it can also be used in EXEs if desired). TLS is a single, per-thread storage area, one machine word in size (32 bits on Symbian OS v9).¹⁸

3.7.2 Global Destructors

If your code uses global objects, you should be aware that current Symbian platform implementations do not invoke destructors of global objects upon program termination.¹⁹

This is something to keep in mind specially when owning global resources, though you should also be very careful when dealing with global constructors and destructors and the order in which they're invoked.

3.8 Multiple Inheritance

On the Symbian platform, the use of multiple interface class (M class) inheritance is the only form of multiple inheritance that is encouraged.

The following example illustrates a class which inherits from `CBase` and two mixin interfaces, `MRadio` and `MClock`. In this case, `MClock` is

¹⁸For a thorough discussion of this topic applied to the Singleton pattern, please refer to the April 2008 Code Clinic article at developer.symbian.org/wiki/index.php/WSD_and_the_Singleton_Pattern.

¹⁹See wiki.forum.nokia.com/index.php/KIS001019_-_Open_C++:_Destructor_of_a_global_object_is_not_called.

not a specialization of `MRadio` and the concrete class instead inherits from them separately and implements both interfaces:

```
class MRadio
{
public:
    virtual void TuneL() = 0;
};

class MClock
{
public:
    virtual void CurrentTimeL(TTime& aTime) = 0;
};

class CClockRadio : public CBase, public MRadio, public MClock
{
public:
    // From MRadio
    void TuneL();
public:
    // From MClock
    void CurrentTimeL(TTime& aTime);
    // Other functions omitted for clarity
};
```

For M class inheritance, various combinations of this sort are possible. However, other forms of multiple inheritance can introduce significant levels of complexity if the standard `CBase` class is used.²⁰ Multiple inheritance from two `CBase`-derived classes will cause problems of ambiguity at compile time, which can only be solved by using virtual inheritance:

```
class CClock : public CBase
{...};

class CRadio : public CBase
{...};

class CClockRadio : public CClock, public CRadio
{...};

void TestMultipleInheritance()
{
    // Does not compile, CClockRadio::new is ambiguous
    // Should it call CBase::new from CClock or CRadio?
    CDerived* derived = new (ELeave) CDerived;
}
```

²⁰ See the May 2008 Code Clinic article at developer.symbian.org/wiki/index.php/Mixin_Inheritance_and_the_Cleanup_Stack.

3.9 Summary

This chapter has served as a basic introduction to the Symbian platform, describing the essential class types and their lifetimes, as well as fundamental idioms to be aware of when using the Symbian APIs. Then the use of descriptors was discussed; together with the cleanup stack and active objects, they are one of the fundamental cornerstones of the native API.

Once you get familiar with the above topics, you'll be able to make use of the extensive services offered by the OS and start writing your own service providers.

Finally, and no less importantly, writeable static data (WSD) and multiple inheritance were covered. WSD has its place, particularly when porting code that makes use of global objects (in the Singleton pattern, for example) whilst multiple inheritance is required for one of the most widely used patterns in the Symbian platform: the Observer–Listener pattern.

Note that the information provided in this chapter has the sole aim of giving you a bird's-eye view of the platform. The next chapter focuses entirely on standard C/C++ programming and POSIX APIs within the Symbian platform.

4

Standard APIs on the Symbian Platform

The nice thing about standards is that there are so many of them to choose from.

Andrew S. Tanenbaum

This chapter is about the standard C and C++ APIs that are available on the Symbian platform. I start by describing the APIs that are available and the versions of the operating system on which you can use them. I then go on to explain how you can use them with some simple examples. I finish the chapter by describing some of the limitations of the API implementations and ways to work around them. The standard APIs described in this chapter are the most important aid to porting available on the Symbian platform; they can save a significant amount of time compared to porting direct to native Symbian C++ APIs and, in many cases, make the difference between a porting project being feasible or otherwise. However, it's important to note that the APIs discussed in this chapter don't allow you to create a graphical user interface; for that, you need to write hybrid code (Chapter 5) or use Qt (Section 6.4).

Whatever your preferred development platform, there are a number of APIs that you consider 'standard' on that platform. As this is a book about porting, we are only concerned with *cross-platform* standards. Since we are primarily talking about porting from one operating system to another, we are most interested in the Portable Operating System Interface (POSIX) defined by IEEE standard 1003.1. POSIX was originally designed to unify the various flavors of UNIX from a programmer's perspective. It has since been supported to some extent on most major operating systems. If you've read Chapter 3, then you should understand how Symbian OS is different from other operating systems and the very good reasons why those differences exist. POSIX conformance was not a design goal for the early versions of the Symbian platform. However, as mobile device hardware evolved, with greater computing power and memory, it became

increasingly desirable to add support for common POSIX APIs. This was because adding POSIX support not only improved the portability for lots of code but also made the platform accessible to a wider developer pool. In addition, there are a number of other free libraries with wide usage across platforms that were also perceived as useful for development on the Symbian platform, particularly for porting existing software. Projects to provide these standard APIs were started within Symbian and Nokia. The results of these projects were first made public in early 2007 as P.I.P.S. and Open C, respectively.

4.1 P.I.P.S. Is POSIX on Symbian OS

P.I.P.S. is one of those recursive acronyms that are very popular with computer scientists.¹ Despite the name it does not actually provide a fully compliant POSIX-standard interface (see Section 4.7 for known limitations), although the aim is to be as compliant as possible given the limitations imposed by the underlying operating system. The team working on P.I.P.S. at Nokia are continuously improving the level of support for various standards. Table 4.1 shows the libraries that form the core of P.I.P.S. along with the approximate percentage of functions within those libraries that are supported.

Table 4.1 Core P.I.P.S. libraries

Library	Functionality	Function coverage (%)
libc	Standard C and POSIX APIs	47
libm	Mathematical and arithmetic APIs	42
libpthread	Thread creation and synchronization	60
libdl	Dynamic loading and symbol lookup	100

Note that while the function coverage percentages given in Table 4.1 are quite low in some cases, the functions implemented have been chosen following analysis of a large number of open source projects. You should find that almost all of the functions that are commonly used are available. The figures given above may be out of date by the time you read this, since the open source nature of the Symbian platform allows for continuous improvement of the libraries, which is important for a range of device manufacturer projects. It's also worth mentioning that the system APIs in `libc` have been mapped to native Symbian

¹ en.wikipedia.org/wiki/Recursive_acronym.

C++ APIs for better performance. Additionally, the POSIX-style threading support in `libpthread` has been implemented in terms of Symbian's native threading model, which is the source of limitations discussed in Section 4.7.2. The first three libraries in Table 4.1 were all seeded from the FreeBSD project.² Figure 4.1 shows the basic architecture of P.I.P.S. and the applications or libraries which use it. You can find out more about the underlying architecture and internals of P.I.P.S. in Chapter 13.

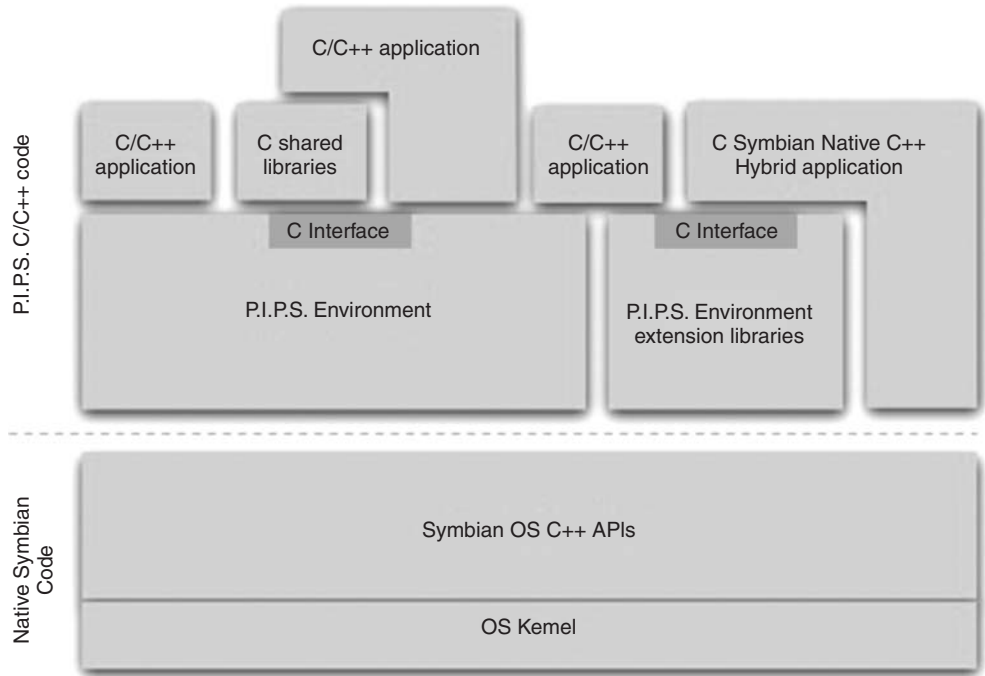


Figure 4.1 P.I.P.S. environment architecture

P.I.P.S. is designed to provide support for standard C programming in a POSIX environment with the dual goals of improving the productivity of existing developers porting software from other platforms and making the Symbian platform more attractive to other developers with C/C++ skills. The P.I.P.S. APIs also enable the addition of a number of extension libraries to the platform, adding support for further standard APIs. The first of these extension libraries was shipped by Nokia along with the core P.I.P.S. libraries in a package called Open C, very shortly after the initial release of P.I.P.S.

² www.freebsd.org.

4.2 Open C

It's worth making clear that Nokia's Open C is a superset of P.I.P.S. – you don't need both. The P.I.P.S. libraries are included as part of the Open C distribution, which adds support for a further five libraries (see Table 4.2).

Table 4.2 Open C libraries

Library	Functionality	Source	Function coverage (%)
libz	In-memory compression and decompression functions	Zlib	100
libcrypt	Cryptography functions	OpenSSL	100
libcrypto	Cryptographic services for SSL, TLS, S/MIME, SSH, OpenPGP, etc.	OpenSSL	77
libssl	OpenSSL secure sockets layer	OpenSSL	86
libglib	General-purpose utility library	GNOME	77

These libraries all originate from open source projects and are widely used in other open source projects, hence the inspiration for the name Open C. The source code for the port of `glib` has been available (as required by the terms of the LGPL, under which it is developed) since the first release of Open C. The source code for the ports of the other libraries has only recently been made available as part of the Symbian platform.

The key difference between P.I.P.S. and Open C is that while the P.I.P.S. libraries are available for all S60 and UIQ devices running Symbian OS v9.1 and later (S60 3rd Edition and later and UIQ 3), Open C is only available for S60 devices. However, Symbian also added support for the `libz` library on UIQ 3.0 and 3.1.

There is an older implementation of the standard C library on the Symbian platform that was created in order to get a Java virtual machine running. The implementation is in `estlib.lib` and the header files are in `\epoc32\include\libc`. This is not compatible with P.I.P.S. and Open C. You should not mix the two in any way as the header files and function names clash, causing unexpected errors at build time or sometimes not until run time. The intention is to deprecate `estlib.lib` as soon as the P.I.P.S. implementation is considered sufficiently stable. In practice, it is already far superior for any porting project.

Both P.I.P.S. and Open C provide standard interfaces for C programming on the Symbian platform. The native programming language is Symbian C++, which has its own special idioms and limitations, as described in Chapter 3.

4.3 The STLport, uSTL and Open C++

Historically, one of the main things that C++ programmers moving to the Symbian platform have missed is the Standard Template Library (STL). In response to this, there have been two Symbian ports of open source STL implementations available for some time. First there was uSTL, a partial implementation of the STL, selected by Penrillian³ as most suitable for a mobile platform due to the small size of the generated object code. Then there's the STLport, one of the most popular and standards-compliant distributions.⁴ Having recently released mobile devices with a built-in storage capacity of 16 GB, Nokia probably didn't see the object code size issue as being anywhere near as significant as it has been in recent history. They selected the STLport for inclusion in Open C++, an extension to Open C that contains the additional libraries for C++ programming shown in Table 4.3.

Table 4.3 STLport libraries

Library	Functionality
IOStreams	Standard C++ input/output streams
STL	C++ Standard Template Library
Boost	Smart pointers, extra containers and math-template functions

The IOStreams and STL implementations come from the STLport while the Boost libraries are part of a separate open source project. Some of the Boost libraries are being added to the C++ standards.⁵ All of the functions in these libraries are supported by Open C++ although the implementations are not 100% compliant with the standards. Open C/C++ is distributed as a single package which supersedes Open C (see Figure 4.2). Again, Open C/C++ is only available for S60 3rd Edition and later, but STLport and uSTL are also separately available for UIQ 3 and standard C++ support is part of the first release of the Symbian platform.

³ See www.pendrillian.com/content/view/82/73 for the download.

⁴ It was ported to the Symbian platform by Marco Jez and released via his blog at marcoplusplus.blogspot.com/2007/05/stlport-for-symbian-os-released.html.

⁵ Ten Boost libraries have already been accepted to the C++ standards committee's TR1, see www.boost.org for details.

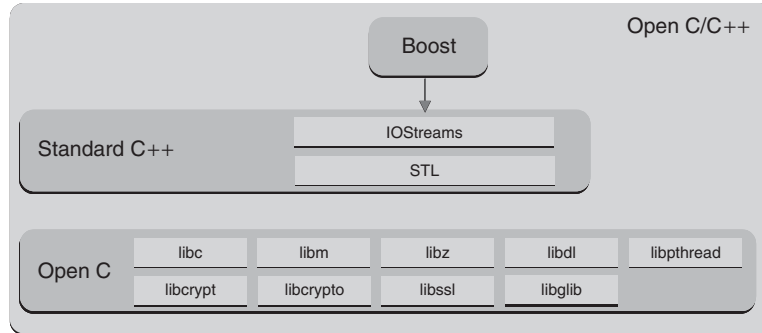


Figure 4.2 Open C++ includes Open C

4.4 Which Version of Symbian OS?

P.I.P.S. and Open C/C++ are available on devices running Symbian OS v9.1 or later. However, the Open C libraries do not ship as part of the firmware on any devices until S60 3rd Edition FP2, which uses Symbian OS v9.3, and Open C++ is not part of the firmware until S60 5th Edition, using Symbian OS v9.4. This means that there are millions of devices running S60 3rd Edition (Symbian OS v9.1) and 3rd Edition FP1 (v9.2) as well as UIQ 3.x, which don't provide these standard C or C++ environments out of the box. The simple solution is that the libraries are provided as SIS files for installation onto older devices and can be bundled with the installation package of any after-market code that uses them. The SIS files are distributed as part of Open C/C++ for S60, which is available for download from Forum Nokia at www.forum.nokia.com/main/resources/technologies/openc_cpp. For UIQ, the appropriate SIS files are available from the Symbian developer wiki page at developer.symbian.org/wiki/index.php/P.I.P.S._is_POSIX_on_Symbian. The necessary extensions to the SDKs to enable development using these libraries (such as header files and documentation) are available from the same locations.

Open C++ is a pragmatic, best-effort implementation of the standards. It cannot be fully compliant with the C++ standard (ISO 14882) since there are various features which depend on updates to both the kernel and toolchain. These updates should be released in Symbian^3. Some high-level features of Symbian's forthcoming standard C++ support include:

- full support for dynamically linked STL – based on STLport version 5.1.4 (Open C++ is based on version 4)
- support for streams and locales

- standard C++ features, such as throwing operator `new`
- better, though still limited, support for mixing standard C++ and Symbian C++ patterns
- support for global destructors (a corner case which affects DLL usage).

So, for most of the standard C/C++ support provided, it is possible to install new versions of the libraries to older devices. However, there are some features which are only available on newer devices. Depending on the set of devices you want to target with your application, you may not be able to use some of the newest features. You can check which models run which version of S60 or UIQ (and Symbian OS) at developer.symbian.org/wiki/index.php/Symbian_Devices.⁶

Unfortunately, you may have to visit the individual manufacturers' websites for full specifications in some cases, as the profiles don't always list the Symbian OS version.

4.5 How to Use the APIs

As you may have guessed from the previous section, how you use the standard APIs also depends in part on which version of Symbian OS you are targeting. The other determining factors are the architecture and target type⁷ for your project.

4.5.1 Before Symbian OS v9.3

Before Symbian OS v9.3, there are no special target types for standard C/C++ projects. You either create a standard Symbian OS executable (EXE), a dynamic link library (DLL) or a static library (LIB) then include headers and link with libraries from P.I.P.S. or Open C/C++. This results in some fairly common problems with porting. The first and most obvious of these is that the standard entry point for a Symbian executable is `E32Main()`; on almost all other platforms, it is called `main()`. There are also differences in the way command-line arguments are dealt with. Fortunately, a simple solution is provided! If the application you are porting has a `main()` function as the entry point then you can link with a static library called `libcrt0.lib` (known as 'glue code') and the problem magically goes away – your `main()` function is called when the application starts. The following parameters should be specified in your MMP file to use the standard C APIs and entry point:

⁶ S60 and UIQ are no longer marketed as separate brands and their websites have closed.

⁷ The target type represents the type of binary that is built and is specified in the MMP file (see Chapter 2).

```

STATICLIBRARY  libcrt0.lib // Note: this must be the first
                  // library specified.
LIBRARY        libc.lib   // Assuming you want to use the
                  // standard C library
LIBRARY        euser.lib  // Needed to use Symbian services
SYSTEMINCLUDE  \epoc32\include\stdapis

```

Note that `euser.lib` is used by `libcrt0.lib` (to set up a top-level TRAP harness amongst other things) so it must be specified regardless of whether you want to use other Symbian OS services. You can also add as many other libraries as you require but they must be listed after `libcrt0.lib` in the MMP file. Obviously, if you don't have `main()` as an entry point (which is always the case for a DLL or a LIB), then you don't need to include `libcrt0.lib` in your MMP file.

For code that uses standard C++ and the STL, you also⁸ need to add the following include paths:

```

SYSTEMINCLUDE  \epoc32\include\stdapis\sys
SYSTEMINCLUDE  \epoc32\include\stdapis\stlport

```

And you need to add this option in your MMP file:

```

//This is required even if the wchar type is not used.
OPTION CW -wchar_t on

```

Most of the STL is implemented, as the name suggests, using templates; thus, the code is generated by the compiler when you use them. Even so, there are some functions in the STLport that make use of pre-compiled code (check the Open C++ documentation or wait for a linker error to find out which ones) that can be found in `libstdc++.lib`. To use this on both the target and the emulator, your MMP file must also contain these lines:

```

#ifdef EPOC32
LIBRARY  libstdc++.lib
#else
FIRSTLIB ../udeb/libstdc++.lib
STATICLIBRARY  exe.lib // For an EXE; replace with edll.lib for a DLL
#endif

```

⁸ The STLport depends on P.I.P.S. and Open C. That is, the standard C++ support is built on top of the standard C support. You must include both if you want to use standard C++.

The other common issue with porting relates to exported functions and data from libraries. In most environments, particularly POSIX-based ones, all functions and data with external linkage (i.e. visible from other source files – usually specified `extern` in the case of data and not specified `static` in the case of a C-style function) are exported from a library by default. In the early days of Symbian OS, RAM and ROM were extremely scarce and unused exports waste space in the export tables for the DLLs. It makes perfect sense to save this space by having a developer specify which functions they want to export explicitly. This is achieved by adding the `IMPORT_C` macro to the declaration of any function that is to be exported and the `EXPORT_C` macro to its definition. So in your header file, you would have:

```
IMPORT_C void MyPortedFunctionToExport(int argument);
```

In the source file, you need:

```
EXPORT_C void MyPortedFunctionToExport(int argument)
{
    // function contents
}
```

For exported data, the situation is more complex because writeable static data (WSD), typically meaning global variables, is not supported by default in Symbian DLLs. It can be enabled by specifying `EPOC-ALLOWDLLDATA` in the MMP file for the DLL. Unfortunately, even when this is enabled, there is a bug in the GCC-E toolchain shipped with all S60 SDKs available at the time of writing, which means that WSD in a DLL doesn't work on the target device in many common scenarios.⁹ Additionally, if your DLL containing WSD needs to be loaded by more than one process at a time, then it does not work on the emulator because separate Symbian processes are emulated within a single Windows process so only one copy of the DLL is actually ever loaded, meaning that all the separate Symbian processes share the same copy of the data, producing unexpected results. To prevent this, the emulator does not permit DLLs with WSD to be loaded into more than one process at a time (you get `KErrNotSupported` if you try). If you have a problem with WSD in a DLL then I'd recommend reading the paper¹⁰ on the Symbian Developer Network that explains the situation in full. You can use WSD

⁹ See the FAQ entry at developer.symbian.org/wiki/index.php/Symbian_C++_Knowledgebase_Q&As for a more complete explanation (or search the Symbian Developer Network website for FAQ-1574).

¹⁰ developer.symbian.org/wiki/index.php/Symbian_Platform_Support_for_Writeable_Static_Data_in_DLLs.

in DLLs on target builds with the ARM RVCT toolchain but if this isn't an option then there are workarounds to avoid using WSD (explained in Section 3.7).

If, after considering all of the issues above, you still want to export WSD from a DLL, or your exported data is constant¹¹ then it is necessary to wrap the data with an accessor function and export that instead, since the Symbian toolchain doesn't allow data to be imported. To make the change transparent to client code you can define a macro to hide the details. Here's a simple example:

```
/* mymathlib.c */
const double PI = 3.1415927;
EXPORT_C double& getPi()
{
    return PI;
}
/* mymathlib.h */
IMPORT_C double& getPi();
#define PI getPi();
```

If you are using C rather than C++ then an alternative to returning a reference is to return a pointer to the data and dereference it as part of the macro.

Thankfully most of the issues discussed in this section have been resolved in later versions of the platform.

4.5.2 Symbian OS v9.3 and Later Versions

From Symbian OS v9.3 (which includes all versions of the Symbian platform), new target types `STDEXE`, `STDDL` and `STDLIB` have been added, specifically for use with P.I.P.S. and Open C/C++. Some of the necessary libraries and include paths (see Table 4.4) are automatically included when you use these target types; they do not have to be added to the MMP file separately.

New UID2 values are required for `STDEXE` and `STDDL` target types to distinguish them from the original `EXE` and `DLL` types. For a `STDEXE`, the UID2 value is `0x20004C45`; for `STDDL`, it depends on whether you are creating a shared library DLL (also `0x20004C45`) or a polymorphic interface DLL (a new UID must be defined for the plug-in type – i.e. don't share the UID for a `STDDL` plug-in with an existing DLL plug-in).

The other key feature of these new target types is that all functions and data with external linkage are exported by default, without any requirement for the `IMPORT_C` and `EXPORT_C` declarations. They also

¹¹ Note that even some data declared as constant can require the use of WSD if it requires non-trivial construction.

Table 4.4 P.I.P.S. libraries in Symbian OS v9.3 and later

Included Library/Path	STDEXE	STDDL	STDLIB
libcrt0.lib	✓ ^a	—	—
libwcr0.lib	✓ ^a	—	—
euser.lib	✓	✓	✓
backend.lib	✓	✓	✓
\epoc32\include\stdapis	✓	✓	✓

^aEither `libcrt0.lib` or `libwcr0.lib` is linked to the executable depending on whether the new MMP file keyword `WCHARENTRYPOINT` is present. Use `libwcr0.lib` when the `main()` entry point for the executable expects wide characters. This new keyword is ignored for all other target types.

support lookup by name as well as lookup by ordinal, unlike the original target types (for more on this, see Section 4.7.5).

Even with these new target types, it is still necessary to add the extra system include paths and compiler option to access the standard C++ environment (see Section 4.5.1), in order to use the STL. You also still need to include any additional libraries from P.I.P.S. or Open C/C++ in the MMP file as normal before you can use them in your code.

4.5.3 Stack Size for Standard C++ and OpenSSL

For all existing versions of the Symbian platform, the default stack size is 8 KB, which is significantly smaller than for most other platforms. Some of the functions in the OpenSSL libraries need more than this default amount of stack. Additionally, standard C++ code often requires more stack space than native Symbian C++. If you are using either of these then the recommended stack size is 64 KB. The default stack size for your project can be set to 64 KB by adding the following line to the MMP file:

```
EPOCSTACKSIZE 0x10000
```

However, it should be noted that there is a limit on the total stack space that can be allocated by a process. The details can vary between devices but there is typically a maximum of 1–2 MB available for stack usage. Assuming only 1 MB available, if you use 64 KB stacks you can only have $1024 \div 64 = 16$ threads, whereas with the default 8 KB stack size you can have $1024 \div 8 = 128$ threads. If you are porting a large application or server and this thread limit is an issue then you could consider compromising on a smaller stack size and only allocating larger stacks for threads that really need them. To set the stack size for an individual

thread using `libpthread`, call `pthread_attr_setstacksize()` before calling `pthread_create()`.

4.6 Examples: SoundTouch and SoundStretch

In this section, I present a (near) textbook example of a dynamic library and a test application that uses it which can be ported using Open C/C++ with very little effort. The full source code for this example can be found at [***developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform***](http://developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform).

4.6.1 Selecting the Project

If a project uses only pure C or C++ code, or libraries that are already available for the Symbian platform, it is trivial to port.

SoundTouch¹² is an audio-processing library for changing the tempo, pitch and playback rate of audio streams or files:

- Tempo (time-stretch): changes the sound to play at a faster or slower speed than the original, without affecting the sound pitch
- Pitch (key): changes the sound pitch or key without affecting the tempo
- Playback Rate: changes both the sound tempo and pitch (to get an effect like a vinyl record played at the wrong speed).

Immediately this sounds like a good candidate because it is basically doing mathematical manipulation of audio data, so there should be very few platform dependencies. The web page for the project also lists the following features:

- high-performance, object-oriented C++ implementation
- clear and easy-to-use programming interface via a single C++ class
- 16-bit integer or 32-bit floating-point PCM mono/stereo supported
- real-time audio stream processing on a 133 MHz Intel Pentium
- platform-independence – runs on Windows, Mac OS X, Linux and AIX.

Between them, these features suggest it should be very easy to port and is probably suitable for running in real time on my target device

¹² The project is hosted at [***www.surina.net/soundtouch***](http://www.surina.net/soundtouch).

(an N95, which has a 330 MHz ARM11 processor). The fact that it has already been ported across different operating systems and processor architectures (AIX usually runs on PowerPCs) is a particularly good sign. Also, 16-bit integer PCM is the standard audio format on devices based on the Symbian platform.

The library comes with a test application, SoundStretch, which is a simple command-line utility that can read and write WAV files, modifying them using all the features of the library. The test application can also count the beats-per-minute (BPM) for a specific file and use the library to modify that to some user-defined value.

4.6.2 Analyzing the Code

A quick look at the code shows that the only dependencies for both the library and the test application are `libc`, `libm` and some standard exception handling (`std::runtime_error`) from STLport. The library doesn't use any writeable static data either. There should be no need for any changes, particularly not architectural ones, to get this to run on the Symbian platform.

4.6.3 Porting the Build Files

There was a choice here – I could have created a single project with two MMP files, one each for the library and the test application with a single `bld.inf` file pointing to both. However, longer term, I'm not going to want to use a command-line utility on the Symbian platform, so I decided to create separate projects with their own `bld.inf` files.

In this case, creating projects from templates generates some unnecessary extras – a global `Panic()` function and a DLL entry function for EKA1-based¹³ builds in one file, and a skeleton example implementation of a class for the DLL in another source file and header file; there is also a file for defining panic codes. None of these files are needed, nor are any of the files in the source or include directories generated by a basic Symbian OS executable template for the test application.

SoundTouch includes several files that can't be used on the Symbian platform, including build files for other platforms and platform-specific optimizations (with inline assembler) for various desktop processors. I decided to copy only the files that would be used into my workspace and follow the standard Symbian directory structure to keep the example code for this book clear, simple and consistent. If I was planning to submit changes back to the project, it would have been better to create the MMP

¹³ EKA1 stands for EPOC Kernel Architecture 1, the name for the old kernel from before Symbian OS v8.1b, which doesn't have a real-time scheduler. It is not covered at all in this book since none of the standard C and C++ APIs discussed here are available on versions before Symbian OS v9.1.

file in the same directory as all the other build files and set the paths up relative to that location. All the unused files could also be copied to the workspace and just not mentioned in the MMP.

The `bld.inf` file for SoundTouch was shown in Chapter 2 but here it is again:

```
PRJ_PLATFORMS
DEFAULT
PRJ_EXPORTS
..\inc\SoundTouch.h
..\inc\FIFOSamplePipe.h
..\inc\FIFOSampleBuffer.h
..\inc\STTypes.h
PRJ_MMPFILES
SoundTouch.mmp
```

The reason for exporting the headers is that it may be desirable to share the library between multiple projects, possibly in different workspaces. Adding these headers as exports causes the build system to copy them to the `\epoc32\include` directory in your SDK. It is not absolutely necessary to do this: instead, the `inc` directory in the SoundTouch project can be added as a `SYSTEMINCLUDE` path in the MMP file of any project that uses it.

With no headers to export, the `bld.inf` file for SoundStretch is even simpler:

```
PRJ_PLATFORMS
DEFAULT
PRJ_MMPFILES
SoundStretch.mmp
```

Again, we've already seen the MMP file for SoundTouch:

```
TARGET          SoundTouch.dll
TARGETTYPE      dll
UID             0x1000008D 0x0839739D
CAPABILITY      None
USERINCLUDE     ..\inc
SYSTEMINCLUDE   \epoc32\include \epoc32\include\stdapis
SYSTEMINCLUDE   \epoc32\include\stdapis\sys
SYSTEMINCLUDE   \epoc32\include\stdapis\stlport
SOURCEPATH      ..\src
SOURCE          SoundTouch.cpp AAFilter.cpp
SOURCE          FIFOSampleBuffer.cpp FIRFilter.cpp
SOURCE          RateTransposer.cpp TDStretch.cpp
LIBRARY         euser.lib
LIBRARY         libc.lib libm.lib libstdcpp.lib
```

The project builds a native Symbian DLL (I'm targeting a Nokia N95 which runs Symbian OS v9.2 to ensure compatibility with older devices, so I can't use the `STDDL` type, although this would make the port even simpler on newer devices). The first value on the `UID` line defines `UID2` as `0x1000008D`, which is the value used for all shared library DLLs on the Symbian platform. These values can be looked up in the Symbian Developer Library that ships with your SDK but most of them relate to native plug-in types that are not relevant while porting. The `UID3` value above has been randomly selected by the IDE from the protected development range¹⁴ and must be replaced with one allocated by Symbian Signed if the project is released publicly in binary form.

Platform Security Considerations

No capabilities are specified for the DLL above because the test application only reads and writes WAV files from public directories in the file system and doesn't require any special capabilities to do so. However, it is important to note that an executable can only load a DLL that has at least the same capabilities. Although the DLL code itself doesn't use any APIs that require special capabilities, if it were to be distributed in binary form then the ideal situation would be that it had an extremely broad set of capabilities (e.g. `All-Tcb`: everything but the `TCB` capability) so that it can be loaded by any application. Unfortunately, this can cause issues with the signing process, since access to some of the most sensitive capabilities requires a legal agreement with the device manufacturer as well as technical and commercial justification. Instead, we recommend signing the DLL with all of the capabilities in the basic and extended sets (no manufacturer-approved capabilities). A test application which exercises all the functionality of the DLL is required for the signing process.¹⁵ In the case of a small open source library which isn't likely to be widely used, like this one, the simplest solution is to distribute the source code only and let users of the library supply their own UIDs and sign the DLL as part of their application. Refer to Section 4.7.6 for more details about how platform security affects compliance of the various P.I.P.S./Open C APIs. See Chapter 14 for more details on platform security and Symbian Signing. Symbian Signed is under review at the time of writing and there may be changes to the signing scheme to provide options that aren't discussed here; please consult the online documentation.

¹⁴ For an explanation of the UID ranges, refer to developer.symbian.org/wiki/index.php/Complete_Guide_To_Symbian_Signed.

¹⁵ See www.symbiansigned.com/Symbian_Signed_DLL_Handling_1.0.pdf for more details.

The MMP file for SoundStretch is as follows:

```
TARGET          SoundStretch.exe
TARGETTYPE      exe
UID             0 0xED2EB6DD
CAPABILITY      None
USERINCLUDE     ..\inc
SYSTEMINCLUDE   \epoc32\include
SYSTEMINCLUDE   \epoc32\include\stdapis
SYSTEMINCLUDE   \epoc32\include\stdapis\sys
SYSTEMINCLUDE   \epoc32\include\stdapis\stlport
STATICLIBRARY   libcert0.lib // Note: this must be the first
                        // library specified

LIBRARY         libc.lib
LIBRARY         libm.lib
LIBRARY         euser.lib
LIBRARY         SoundTouch.lib
#ifdef EPOC32
LIBRARY         libstdcpp.lib
#else
FIRSTLIB        ../udeb/libstdcpp.lib
STATICLIBRARY   eexe.lib
#endif
OPTION CW -wchar_t on
SOURCEPATH      ..\src
SOURCE          BPMDetect.cpp PeakFinder.cpp RunParameters.cpp
                        WavFile.cpp main.cpp
```

There shouldn't be any surprises here. Note that UID2 for a native Symbian EXE type is not generally used by the system and can be set to zero. UID3 is another auto-generated value from the IDE, this time from the unprotected development range, intended for unsigned (or self-signed) applications. SoundTouch.lib is required in order to use the DLL we are about to create from the test application, however, it does not exist at this stage. It is created as part of the build process for the DLL.

4.6.4 Building the Projects

The first attempt at compiling the SoundTouch library produced several errors, all of which had the same cause. Code such as the following should compile if the standard header `<stdexcept>` is included:

```
throw std::runtime_error("Illegal number of channels");
```

However, the argument is not correctly converted to a string object unless the `<string>` header is also included. This second header should be included by `<stdexcept>` internally but it seems there is a minor bug in the current Symbian version of the STLport. As a quick workaround,

I simply included `<string>` in all the files that throw exceptions in that way. With this fix in place, the code compiles for all targets.

At this point, no import library (`.lib` file) is generated. In order for the build system to generate an import library, it is necessary to ‘freeze’ the exports from the project.¹⁶ This can be achieved via the Project menu in the Carbide.c++ IDE or with the command ‘`abld freeze`’ from the command line. This process creates a `.def` file which specifies which functions are exported from the project (and the order in which they are exported). Since I was working on Symbian OS v9.2, it was necessary to add the `IMPORT_C` and `EXPORT_C` declarations to all of the public methods of the `SoundTouch` class and those in its inheritance hierarchy (`FIFOSamplePipe` and `FIFOSampleBuffer`) before performing the freeze operation, otherwise there would be no functions to export!

Once the import library generated successfully, it was possible to compile the test executable on the first attempt without any errors at all! That’s about as easy as porting gets.

4.6.5 Running and Testing

Most Symbian platform devices are not ideally suited to the use of command-line utilities; the screen is usually on the small side and the input capabilities are rarely appropriate for that kind of user interaction. For this reason, initial testing of such programs is much easier on the emulator. To test `SoundStretch`, I used `eshell`, a basic text shell environment for the Symbian platform much like a UNIX/Linux shell. The `eshell` program is provided for the emulator as part of Symbian SDKs. Although `eshell` can also be used on a target device it does not ship as part of released firmware builds and is currently not publicly available as an aftermarket installation. An alternative for device-based testing is `Zsh`, recently made available with the P.I.P.S. distribution.¹⁷ It provides a more fully functional text shell and connects to a remote PC via Telnet so you can still use your PC keyboard while testing on the device.

To run `eshell` in the emulator, you can simply type ‘`eshell`’ from the command line. In order to use the debugger for an executable launched via `eshell`, it is necessary to modify the debug configuration so that it launches `eshell.exe` rather than `epoc.exe` as its emulator. This configuration option can be found under **Run, Debug** in Carbide.c++.

This debugging method is far from ideal for this particular application since, if it throws a `runtime_error`, it should produce output on `stderr`. Normally in a text shell environment on UNIX or Linux, this

¹⁶ The build system also generates an import library if the `EXPORTUNFROZEN` keyword is added to the MMP file. This is intended for use during the early development stages of a new library but may also be appropriate during a larger porting project.

¹⁷ Latest versions of `Zsh` and the Telnet daemon are available from the P.I.P.S. wiki page at developer.symbian.org/wiki/index.php/P.I.P.S._is_POSIX_on_Symbian.

output would appear in the shell. Unfortunately, with `eshell` all that happens is that the application exits with no meaningful message. In theory, `stderr` should be mapped to RDebug output and error messages should appear in the console of the debugger (with the correct configuration of the emulator) but this did not appear to work in my setup.

Debug output to `stdout` via `printf()` statements is correctly captured via the `stdioserver` provided by P.I.P.S.¹⁸ and this can be redirected to a file (which works but is a rather slow way to debug) or a console that appears over the shell.¹⁹ Sadly, in the case of a short-running command-line utility, the debug console disappears when the application exits, before you have a chance to read it! This can often be worked around by adding a call to `getchar()` just before the application exits so that it waits for a key press before closing. Fortunately, it wasn't necessary to do any debugging in this case because copying some WAV files to the C: drive of the emulator and issuing various commands to the test application resulted in appropriately lengthened, shortened or pitch-shifted versions of the originals. Basically, it worked first time!

I'd taken the project far enough for the purposes of this example at this point, although more should still be done for the port to be considered complete. At the very least it would be wise to perform some out-of-memory resiliency testing (as described in Appendix A) to ensure that failures due to lack of resources are handled gracefully. Additionally, it would be good to do some performance testing to see if real-time operation can be achieved while playing back from a file or recording from the microphone. This would require new test applications to be written and is beyond the scope of this book.

Of course, not all porting projects go this smoothly, even when the only dependencies are in libraries provided by Symbian. One reason for this is that, so far, it has not been possible to provide a fully compliant implementation of the standards.

4.7 Known Limitations, Issues and Workarounds

P.I.P.S. and Open C have a number of known limitations and issues, largely due to the restrictions imposed by the underlying OS. Many of these are listed below and, where they are considered serious issues for developers, various workarounds are proposed. In many cases, the P.I.P.S. team is still actively working to improve or correct these limitations.

¹⁸For details of the `stdioserver`, see the P.I.P.S. booklet, which is available from developer.symbian.org/wiki/index.php/A_Guide_To_P.I.P.S..

¹⁹At the time of writing, the default configuration for the `stdioserver` is to output to a console of size `(-1, -1)` – i.e., no output.

4.7.1 Files and File Attributes

There are several constraints on the number of files you can have open, their size and the length of their names. Some of these are imposed by the P.I.P.S. implementation and others are inherited from the underlying file system:

- There is a limit of 256 open file descriptors (including those reserved for `stdin`, `stdout` and `stderr`) per process. This limit applies irrespective of whether you use system calls (such as `open()`) or `stdio` APIs (such as `fopen()`) to open files. The limit of 20 open files implied by `FOPEN_MAX` in `stdio.h` is misleading.
- Outside of these enforced constraints, the size of your heap (and the size of the P.I.P.S. private heap) limits the number of files you can have open.
- File and directory names can be up to 256 bytes in length.
- Files can be up to 2 GB in size.

Timestamps

P.I.P.S. only supports the time of last data modification (`st_mtime`) on files. The time of last access and the time of last status change (`st_atime` and `st_ctime`) are not supported and are always set to `st_mtime`.

Users and Groups

P.I.P.S. does not support users, groups, process groups, sessions or pseudo-terminals; `setgid()`, `setuid()`, `setpgid()` and related functions return 0, but do nothing. Likewise, `getgid()`, `getuid()`, and so on, return 0, but that return value means nothing.

Execute Permissions

The Symbian platform has no concept of execute permissions, and therefore the execute bit in file permissions is ignored in P.I.P.S. If that is the only bit set in the `perm` parameter, `open()`, `create()`, `mkdir()` and similar functions fail with `EINVAL`. On a related note, the `PROT_EXEC` flag is not supported and specifying that alone as the access type in `mmap()` results in an `EINVAL`.

Buffering

File input and output in P.I.P.S. `libc` is buffered. This usually implies a faster than native performance for small reads and writes, at the small

risk of data loss in the event of battery removal or other emergency power-down situations. You can use the `setvbuf()` API to disable buffering on a particular `FILE`.

Link Files

P.I.P.S. emulates symbolic links on the Symbian platform; both `link()` and `symlink()` create symbolic links. Hard links are not supported.

Memory-Mapped Files

There is limited support for memory maps in P.I.P.S. `PROT_NONE`, `PROT_EXEC` and `MAP_FIXED` flags are not supported. Code that depends on memory maps with these properties cannot be ported to the Symbian platform.

Sparse Files

The Symbian platform does not support the notion of sparse files. If you seek beyond the end-of-file marker and write some data, the gaps in the file are filled with `NULL` values that contribute to the file size.

Devices

On UNIX and Linux platforms, direct access to hardware devices is available via special files in the file system. Equivalent files do not exist on the Symbian platform. Access to the hardware is currently only available via alternative APIs (see Chapters 5 and 6 for details).

4.7.2 Memory Handling

Memory management in an emulated POSIX environment on a mobile device has a few issues that are worth being aware of, particularly if you plan to mix standard C/C++ code with native Symbian C++.

Heaps

Heaps on the Symbian platform can be specified per thread, unlike in UNIX systems where a heap is always process-wide. The native class for threads on the Symbian platform is `RThread`, and it is documented in the Symbian Developer Library found in your SDK. Threads on the Symbian platform can be created with their own heap, can share the heap of the creator thread, or can use a separate heap through an `RHeap` object. If you use an `RThread` with a separate heap in your application, you cannot `malloc()` some memory in one thread and simply pass the pointer to

another. The latter thread crashes when it attempts to access that pointer or free the memory. This constraint is abstracted away when you use `pthread`s in your application. On the Symbian platform, `pthread`s are implemented as wrappers over `RThreads`, but are configured to share the same heap (the heap of the main or creator thread²⁰) in a process, thereby ensuring that the `malloc`-and-share-pointer scenario works as expected. If you wish to share dynamically allocated memory between two threads that use different heaps, you must create a new shared heap object (an instance of Symbian's native `RHeap` class), and allocate memory on that heap using `RHeap::Alloc()`. The `malloc()` and `free()` functions supplied by P.I.P.S. implicitly use the current thread's heap. If you wish to use these functions with a custom `RHeap`, you need to make that the default heap of your thread by using `User::SwitchHeap()`.

Internal Allocation

Internal objects and buffers allocated by P.I.P.S. APIs are on a private heap and do not affect the heaps available to user code. This heap is automatically cleaned up on application exit.

Out-of-Memory Errors

Ported code must be aware that it is running in a memory-constrained environment. Out-of-memory situations can occur and need to be handled gracefully. By default, a process is restricted to a maximum of 1 MB of heap. You may increase this limit by using the `EPOCHEAPSIZE` directive in your MMP file (see Section 2.7 for more on heap memory limits on the Symbian platform). In very low memory conditions, your P.I.P.S. application may crash at launch with a `STDLIBS-INIT` panic. This indicates that the P.I.P.S. subsystem was unable to initialize its core components and cannot function. For similar reasons, hybrid applications may encounter this panic when they invoke a P.I.P.S. API.

Stack Size

As I described in Section 4.5.3, the default stack size per thread on the Symbian platform is 8 KB. This default imposes a maximum of 128 threads per process. These limits apply to the total number of `pthread`s and `RThreads`. You can use `pthread_attr_setstacksize()` to set a custom stack size before calling `pthread_create()`, but remember that this proportionally reduces the number of threads you can create in that process. For example, with stack sizes of 16 KB each, you can create only 64 threads per process.

²⁰ If you create an `RThread` with a separate heap and invoke `pthread_create()` from that thread, the new `pthread` shares a heap with the `RThread` you created and not with the main thread of the process.

Stack Overflows

Some P.I.P.S. APIs in `libc's stdio.h` use a large number of stack variables. If you use these APIs in your application and encounter panics that you cannot isolate, try increasing the default stack size using the `EPOCSTACKSIZE` directive in the MMP file (see Section 2.7 for more on stack overflows on the Symbian platform). Also note the recommended stack sizes for applications that use SSL library cryptographic functions or the STL, in Section 4.5.3.

4.7.3 Process Creation and Inter-Process Communication

The Symbian platform does not support the separate fork-and-exec model of process creation. Consequently, P.I.P.S. does not provide either `fork()` or `exec()`. If your project uses the return value from `fork()` to distinguish between code that runs as parent and code that runs as child, you must refactor that section of code. To create child processes in P.I.P.S., use one of `popen()`, `popen3()`, `system()` or `posix_spawn()`. Between them, they cover most typical process-creation semantics:

- `popen()` and `popen3()` allow you to set up `stdio` pipes between parent and child
- `system()` performs a no-frills process launch
- `posix_spawn()` enables you to perform operations (`open()`, `close()` and `dup()`) on the child's file table before it 'starts'
- both `popen3()` and `posix_spawn()` allow you to specify a custom environment for the child.

If the child processes are P.I.P.S. applications (using the `STDEXE` target type or explicitly including the glue code), they inherit file descriptors and environment variables from the parent. You can use `wait()` and `waitpid()` to await termination of child processes and reap their exit status.

File Descriptor Inheritance

P.I.P.S. applications that were launched by use of a P.I.P.S. process creation API inherit certain open file descriptors from the parent. These include all regular files and pipes. Named pipes (FIFOs), sockets and regular files in the private directory of the parent process (see Section 4.7.6 for details of application's private directories) are not inherited.

IPC Mechanisms

P.I.P.S. supports pipes (both named and unnamed), message queues, semaphores, shared memory and local file sockets (AF_UNIX sockets). All IPC mechanisms in P.I.P.S. (except for pipes and local file sockets) are implemented using a secure server (in its own separate process), and may incur a slight memory and run-time speed overhead.

While set up of the shared memory area involves the server, the actual reads and writes are direct to memory and remain fast. Shared memory areas are always read–write, irrespective of permissions used during creation. This is a constraint of the native interface used to implement them.

Other noteworthy constraints and limitations related to the IPC mechanisms are as follows:

- Pipes created with P.I.P.S. APIs are 512 bytes in size. Therefore, writes that are larger than 512 bytes are blocked until a reader reads some of the data.
- The `key_t` type in P.I.P.S. is a signed integer. When specifying `key_t` variables as IPC identifiers (in `msgget()`, `semget()` and `shmget()`), ensure that their values are less than `INT_MAX`.
- IPC resources (for example, message queues, shared memory and semaphores) are created on the heap of the IPC server. Therefore this heap is, in effect, shared by all processes that use the IPC facilities provided by P.I.P.S. The system-wide number of IPC objects you can have open is limited by the size of the server's heap.

Signals

At the time of writing, P.I.P.S. does not support signals. When porting code that uses signals, you need to modify the code to use some native asynchronous construct that approximates signaling in context. For example, you could extend the concept described in the timer workaround, provided in Section 4.7.5, to all signals.

To enable signaling from other processes, the handler thread must set up an IPC facility that sender processes can use to deliver the signal. The facility needs a name that associates it with the recipient process and it also needs to be policed in order to:

- ensure that arbitrary processes cannot send 'terminal' signals (SIGKILL/SIGQUIT) to other processes
- verify that the sender process is who it claims to be

- guard against malicious programs tapping into the facility from outside the framework and sending spurious signals.

The support for signals in P.I.P.S. is planned for future versions (see Chapter 13). It is important to be aware that this support is not present in existing devices where Open C is delivered as part of the firmware and must be added as an upgrade that is bundled with your application.

Access Permissions for IPC Objects

The P.I.P.S. IPC Server enforces some access restrictions for IPC objects. The permissions you specify when creating an IPC object must contain at least the `USER` bits set. Otherwise, even the creator cannot operate on the object. If you wish to allow access from other processes (as is usual), ensure that you set the `OTHER` bits in the `perm` flag. The `GROUP` bits are ignored.

RPipe

Pipes, FIFO special files and `AF_LOCAL` sockets in P.I.P.S. are implemented using the native Symbian C++ `RPipe` interface. `RPipe` is available on Symbian OS v9.3 onwards although, at the time of writing, the `RPipe` interface is not public. User code should not use `RPipe` APIs directly. For v9.1 and v9.2, P.I.P.S. and Open C installers bundle an embedded SIS that installs `RPipe` on these platforms.

4.7.4 Communications and Networking

The average mobile device has much more complex connectivity than the average desktop PC. This causes P.I.P.S. to have limitations on various networking APIs but also extensions to the standard POSIX APIs in order to expose more of the native functionality.

COM Ports

When opening COM ports with `open()`, use `COM:x` as the path, where `x` ranges from 1–n. `COM:1` refers to Symbian's serial port 0. COM ports can only be accessed from the thread that opened the port. Attempting to access the port from another thread results in a panic. If the COM port must be shared between threads, you must write a proxy server in a separate thread that opens the COM port and reads or writes to it on request from other threads. You may also use the `stdio` server as this proxy server by specifying a COM port as media for `stdin` and `stdout` and using `stdio` API for COM port reads and writes. This restriction on access to COM ports from multiple threads is under review and is likely to be removed in a future version of P.I.P.S (probably before you read this) so that the proxy is provided transparently in the back end.

Socket Options

P.I.P.S. supports a subset of standard socket options and `IOCTL` flags. Please refer to the API documentation for the specifics. The functions `setsockopt()`, `getsockopt()` and `ioctl()` fail with `ENOSUP` for unsupported options or flags.

P.I.P.S. also supports a number of ‘non-standard’ socket options that enable access to native networking features. You can use these options to enumerate and query network interfaces or Internet Access Points (IAPs); choose a particular IAP; add, edit or delete routes; start or stop interfaces; configure multicasting, and so on. Here again, details are available in the relevant API documentation.

Future Bluetooth Sockets

At the time of writing, there are plans (and code) for P.I.P.S. to support Bluetooth `RFCOMM` sockets. In the `socket` API, you can specify `AF_BLUETOOTH` as the address family or domain and `BTPROTO_RFCOMM` as the protocol. Utility functions `strtoba()` and `batostr()` convert Bluetooth addresses from string to native format and vice versa.

Internet Access Point Name Restrictions

For reasons of backward compatibility, IAP names in P.I.P.S. are restricted to 49 bytes in ASCII. For names in languages with a multi-byte character set, you are restricted to 24 characters for 2-byte and 16 characters for 3-byte representations. All lengths exclude the terminal `NULL` character.

4.7.5 Miscellaneous

There are a few remaining issues worth mentioning that don’t fit into the categories above.

Timers

P.I.P.S. does not support timer APIs or `SIGALRM`. One possible work-around is as follows:

- 1 Where you would register a timer, create a thread (`TimerThread`) with a higher-than-standard priority. In the thread function, create an `RTimer` and set an alarm using `RTimer::After()` and a `TRequestStatus` variable.
- 2 The `TimerThread` waits on the `TRequestStatus` variable while the main thread continues.

- 3 When the timer triggers, the `TimerThread` exits the wait. It then issues a `Suspend()` on the main and other threads in the process and invokes the registered handler. Once the handler returns, it resumes all threads and exits itself.

Please note that there are some dangers in this approach:

- Context switches to and from the `TimerThread` may introduce some latency. Code that relies on timer resolutions of this magnitude cannot use this solution.
- The `TimerThread` executes the handler in its context. This means that the handler cannot access resources owned by other threads (including the one that registered the handler) without prior and explicit sharing. Even with sharing enabled, deadlocks may ensue when the handler attempts to access resources locked by one of the suspended threads. If the threads are `RThreads` using separate heaps, then the handler cannot access memory allocated by other threads.

If you need to set a number of timers or a periodic timer in your code, and do not want the overhead of creating a thread every time, create a server thread when the process starts up and establish a session to it from your main thread. Then, all timer registrations are messages sent to the server which creates a timer, as in Step 2 above, and acts on the timeout, as in Step 3.

Error Handling

P.I.P.S. maps errors from native APIs to their nearest POSIX equivalents. In cases where such translation would be meaningless, `errno` is set to outside the usual range: greater than `_EMAXERRNO` (124). To retrieve the Symbian C++ error code, use the formula:

```
Symbian Error Code = - (errno - _EMAXERRNO)
```

The `atexit()` Function

The `atexit()` function does not work on the emulator. It accepts function registrations but does not invoke them on process exit. This is due to a constraint with the build toolchain. However, `atexit()` functions correctly on device hardware.

System Configuration

P.I.P.S. does not currently provide a `sysconf()` function to query various system parameters and limits. You need to look up relevant header files or peruse the documentation to determine resource limits. This facility may be provided in a future release.

Build Support APIs

Certain APIs in P.I.P.S. are stubs, provided only to enable builds of existing code. They either return `ENOSUP` or 0 but do nothing.

Hardware Floating-Point Support in libm

The `libm` APIs do not use floating-point hardware on devices that possess it (many devices do not have such hardware anyway).

Symbol Lookup via libdl

Symbian OS versions prior to v9.3 support symbol lookup only by ordinal number (e.g. `dl_sym("1")`). It is necessary to look up the ordinal number manually (i.e. in the DEF file) in order to use this functionality.

Interleaving P.I.P.S. and Native APIs

Interleaving calls to P.I.P.S. APIs with functionally equivalent native APIs, when operating on the same resource, may result in undefined behavior. P.I.P.S. APIs may maintain internal state across functions, which is not updated if you call a native API, resulting in incorrect or unexpected behavior on a subsequent invocation. For example, mixing `RThread` APIs with `pthread`s is usually a bad idea, as is interleaving `libc`'s `stdio` APIs with native `RFile` methods on the same file.

4.7.6 How Platform Security Affects Compliance

As has been mentioned previously, the Symbian platform has a platform security system (see Chapter 14 for a full explanation) that restricts access to the use of certain APIs via the granting of special capabilities. These restrictions also apply to applications created using P.I.P.S. and Open C/C++. Generally, the standard APIs inherit the capability requirements of the native APIs that have been used to implement them. For the majority of standard APIs, there aren't any capability requirements and most of the APIs that do require capabilities only require ones from the user-grantable set:

- `NetworkServices` capability is required for all network access.
- `LocalServices` capability is required for Bluetooth access.
- `ReadUserData` and `WriteUserData` capabilities are required for access to public file system directories.

In connection with this last point there is another aspect of the platform security system that developers using standard APIs need to be aware of – data caging.

Table 4.5 Capability access rules

Directory Path	Capability Required To:	
	Read	Write
\sys	AllFiles	Tcb
\resource	none	Tcb
\private\<ownSID>	none	None
\private\<othersSID>	AllFiles	AllFiles
\<other>	none	None

Data caging is used to protect important files, whether system files or user files. The same system is used to protect the functioning of the system, the user’s personal data and applications’ private data. This is achieved by providing special directories that ‘lock away’ files in private areas; these directories are in addition to the public parts of the filing system. There are three special top-level paths: `\sys`, `\resource` and `\private`. Access restrictions apply to these directories and all subdirectories within them. Table 4.5 summarizes the access rules in terms of which capabilities are required to access specific paths.

You’ll notice that access to subdirectories of the `\private` directory is dependent on the Secure ID (SID) of the process that the code is running in. This SID is equal to UID3 for an executable unless it is explicitly specified using the `secureid` keyword in the MMP file. The path `\private\<ownSID>` is commonly referred to as the application’s private directory, where `<ownSID>` is the UID3/`secureid` value in hexadecimal (without a leading 0x).

On the Symbian platform, all executables must be stored in the `\sys\bin` directory. This means that only the most trusted code can create new executables and the `AllFiles` capability (which also needs manufacturer approval) is required just to read an executable directly. Searching the file system to see if a certain executable is available is not an option. Another consequence of this structure is that resource files that need to be written as well as read should not be stored in `\resource`.

Finally, the default working directory for an application on the Symbian platform is its private directory. Note that other applications cannot share files in this directory. To share files, either use a common server to access the contents (if they must be secure) or simply store them in a public area of the file system.

If You Have a Problem Using P.I.P.S. or Open C

P.I.P.S. APIs may reflect issues and limitations with the native interfaces used to implement them; for instance, limits are usually a native

constraint. Before reporting a problem with P.I.P.S., we suggest that you attempt the same task natively. If the code succeeds, it implies an issue with the P.I.P.S. implementation or your use of it, so please feel free to post a query on the forums at ***developer.symbian.org/forum***.

4.8 Summary

In this chapter, I have described the support for standard C and C++ programming in a POSIX environment on the Symbian platform. P.I.P.S. and Open C/C++ have been introduced with an explanation of how to use them followed by a simple example to show how they can significantly reduce the effort required to port code from other platforms. Then finally I've presented a number of known issues relating to the level of standards compliance to be aware of when porting.

As stated at the start of the chapter, the libraries presented here don't provide any means of creating a user interface. To achieve this, you need to write hybrid code, which is explained in Chapter 5, or use an alternative library, such as Nokia's RGA or Qt, which are described in Chapter 6. Similarly, access to hardware, sensors, GPS data and multimedia functionality are not covered by these standard APIs and alternative solutions must be sought. The next couple of chapters explain current and future options for porting such code to the Symbian platform.

5

Writing Hybrid Code

Consistency is the last refuge of the unimaginative.

Oscar Wilde

Wouldn't it be dull if every platform had exactly the same software environment and API set? There would be no porting required and once someone had written a good application for a specific task, it would just work on all platforms everywhere. Perhaps this is the future? Computing hardware could become so cheap, powerful and efficient that it makes sense to run everything in a standard virtual environment, abstracted from the actual hardware implementation. Perhaps this will be taken to the extreme that everything runs in 'the cloud' with a simple browser-based interface on the device. Alternatively, storage space may be considered sufficient to provide most, if not all, popular libraries on every platform. We are nowhere near this 'ideal' and the future is likely to include web and native hybrids as well as pure web applications. I suspect that getting the software development community to agree on a single common standard is likely to take far longer than it will take for issues of computing resources and battery life to become irrelevant.

The current situation is that there are three popular platforms for desktop computers: Microsoft Windows (with a massive majority of users), Mac OS X, and Linux (although there is a lot of fragmentation with different distributions of Linux). It is not generally possible to target all of these with a single native application,¹ despite very similar (or, in some cases, identical) hardware, although POSIX-compliant code runs on all of them. In the mobile space, the situation is made much more complex by the wide variety of hardware available. On top of

¹ A native application is one that's compiled to the native binary format of the processor that it runs upon, rather than being interpreted or executed in a virtual machine. Native applications are typically written in C/C++, rather than Java ME, Flash Lite, Python, and so on.

the hardware variations, there are five popular mobile device platforms where it is possible to write native code: Symbian (which has by far the largest market share), Windows Mobile, iPhone, BREW and various mobile flavors of Linux. For details of porting native code for these mobile platforms to the Symbian platform, see Chapters 7–9. The iPhone runs a cut-down version of Mac OS X which is a POSIX-based operating system, as is Linux. It is possible to get third-party POSIX libraries² for Windows Mobile although not, as far as I’m aware, for BREW. This means that it is possible to port code based on standard C/C++ and POSIX between nearly all of these platforms. However, unless that is the only code in the project (for example, it is a simple command-line utility or background daemon) then it is likely to require at least some platform-specific code or use additional dependencies that are only available on a subset of the platforms.

In this chapter, I explain when you need to use native Symbian C++ and how you can mix it safely with your ported code. I start by describing some popular APIs that are not available on the Symbian platform along with the alternatives that you can use. This is followed by a section which describes strategies for porting projects that include both platform-specific and portable code. I then present a simple example porting project in which the user interface (UI) and multimedia functionality is converted to use Symbian APIs.

5.1 Popular APIs You Can’t Use Directly

The APIs from other platforms that are not currently available on the Symbian platform can be categorized into types. For projects ported from a desktop environment, the most common are user interface and multimedia APIs. When moving code to the Symbian platform from another mobile platform, it is also possible for there to be dependencies on functionality such as Bluetooth, Personal Information Management (PIM – contacts and calendar), telephony, messaging, device status and other hardware access.

5.1.1 User Interface APIs

There are several UI (or application) frameworks that are widely used in application development. On Microsoft platforms, there’s the Windows API (Win32, a C language interface) and the C++ interfaces provided by the Microsoft Foundation Classes (MFC) and the Active Template Library (ATL). There are no direct equivalents of these APIs on the Symbian

² Generally these are commercial libraries that I won’t list here. However, a basic free library to ease porting of POSIX code is available from wcelibcex.sourceforge.net.

platform, although Chapter 8 explains the similarities between them and the native Symbian application frameworks, giving advice on how to go about porting from one to the other.

Similarly, on Mac OS X there is a general C language interface, called Carbon, and a newer development environment that uses Objective-C, called Cocoa. In Cocoa, there is an AppKit framework for the desktop and a UIKit for the iPhone. Also, the 2D-rendering subsystem and composition engine are known as Quartz. These interfaces provide native UI programming functionality that also doesn't have a direct equivalent on the Symbian platform. Section 9.3 explains how to port code written for the iPhone to the Symbian platform.

On Linux, the most popular application frameworks, GTK+ and Qt,³ have also been ported to the other desktop platforms, making them viable options for cross-platform development. GTK+ is often criticized for not having a native look and feel on non-Linux platforms, while Qt tries very hard to reproduce the native UI style on all platforms. GTK+ is part of the GNOME project and is free, open source software. Qt is also open source and has historically been dual-licensed: free for open source projects but requiring a license for inclusion in proprietary products or applications. During the writing of this book, Nokia acquired Trolltech (the creators of Qt) and moved to the more permissive GNU LGPL license, making Qt free for open and proprietary applications; Symbian is also now a supported platform. As this book went to press, Nokia was proposing to make Qt the primary application framework for development on the Symbian platform from Symbian^4 onwards. In contrast, GTK+ is currently used in several mobile Linux initiatives such as Nokia's Maemo, Intel's Moblin and Openmoko;⁴ both Maemo and Openmoko also have support for development using Qt. GTK+ is written in C and Qt is written in C++, although both have bindings for many other languages. GTK+ is not available on the Symbian platform at the time of writing, although two of the main dependencies GLib (part of Open C) and Cairo (a vector graphics library) are already ported, so it seems feasible that it may be available in the future. You can read more about porting from various mobile Linux variants in Chapter 7.

There is one further cross-platform UI toolkit that is worth mentioning, since ports are being made for several embedded platforms. It's called wxWidgets⁵ and it is well established on many platforms as well as having very popular bindings for programming in Python. Critics suggest that it is rather heavyweight, making projects that use it large and slow to compile. According to the project website, no port for the Symbian platform is currently planned.

³ Available from www.gtk.org and qt.nokia.com, respectively.

⁴ See www.maemo.org, www.moblin.org and www.openmoko.org, respectively.

⁵ More information is available at www.wxwidgets.org.

A special class of applications also has very different UI requirements – games. Most games have full-screen animated graphics and tend to use low-level drawing primitives rather than UI frameworks. The 2D and 3D graphics support in OpenGL ES may be useful here, as may Nokia’s Real-time Graphics and Audio (RGA) APIs or frameworks such as Ideaworks 3D’s AirPlay. OpenGL ES and RGA are covered further in Chapter 6. Several other UI toolkits include support for OpenGL ES and, with the increasing popularity of 3D effects in UIs, there may be an increase in the use of UI toolkits, such as OpenedHand’s Clutter,⁶ built on top of OpenGL ES. However, at the time of writing, use of such toolkits is still very rare, with the exception of Mac OS X, where the whole graphics subsystem is built on top of OpenGL on the desktop computer and OpenGL ES on the iPhone.

With the exception of Qt applications or games, it is generally necessary to re-implement the UI for your ported application. Even with Qt, if you’re porting from the desktop, major re-design for the mobile form factor may be required. If you’re porting from another framework, the choices are either to port to Qt or to implement the UI for the native Symbian C++ framework; which you do depends on the devices you are targeting. The Symbian platform unifies the S60, UIQ and MOAP native UI frameworks into a single framework, which is backward compatible with S60 only. However, it is still necessary to consider individual target devices, particularly their screen sizes and input methods. It might be possible to produce a single UI that scales across all target devices but it may also be desirable to produce optimized versions for certain groups of device. Fortunately, there is already a good range of examples of ported applications with native UIs available from Forum Nokia and the Symbian Developer Network.

For example, on the P.I.P.S. wiki page⁷ there is a port of the TightVNC client to both S60 and UIQ, complete with a paper describing how the port was accomplished. Forum Nokia has examples of an IRC client, an SMS encryption application and an FTP client;⁸ of course, these examples are only for S60. Between them, this set of examples covers a range of different UI implementation techniques. In some cases, it is possible to port a large portion of the existing UI functionality into a Symbian UI framework, particularly where there is a lot of custom drawing code. A simple example of this type of UI port is provided in Section 5.3.

5.1.2 Multimedia APIs

The situation for multimedia is very similar to that for UI functionality. Microsoft uses the DirectX APIs, Apple has QuickTime on Mac OS X

⁶ Clutter is hosted at clutter-project.org.

⁷ developer.symbian.org/wiki/index.php/P.I.P.S._is_POSIX_on_Symbian.

⁸ All examples are available for download from www.forum.nokia.com/main/resources/technologies/openc_cpp/documentation.

(which is also available on Microsoft platforms) and the open source community has built a number of cross-platform frameworks that aren't very widely used outside of Linux-based platforms. Mobile platforms all have their own solutions with very little portability between them.

At the time of writing, none of the cross-platform multimedia frameworks are available on the Symbian platform. However, Phonon is the multimedia component of Qt and it should be ported to the Symbian platform by the time you read this. Other potential targets for porting multimedia functionality are GStreamer⁹ (a plug-in-based open source framework that is very popular in both desktop and mobile Linux projects) and OpenMAX, a cross-platform standard defined by the Khronos Group (see Section 6.3.4). Nokia have previously revealed plans to port GStreamer to the Symbian platform but confirmation and exact timing were not available at the time of writing. Symbian have committed to the adoption of OpenMAX and several APIs are already available or in development. These options are promising for the future but to target existing devices the only options are:

- use the native Symbian C++ APIs
- use the RGA libraries which wrap the native APIs (see Section 6.1)
- port other multimedia libraries and codecs as part of the project.

It's important to be aware of licensing issues if porting codecs directly, since many are not free to use and often don't ship in standard distributions of the cross-platform frameworks. The open source community is trying to push the Ogg Vorbis and Theora standards as royalty-free alternatives to resolve this but there has been very limited uptake. So much content is now available in proprietary formats that this issue is very unlikely to go away.

Fortunately, if you use native (or RGA) APIs on the Symbian platform then most popular formats and codecs are already supported and licensed by the device manufacturers. Since RGA is currently a Nokia-only solution supported in a subset of S60 devices, in most cases it is necessary to use the native APIs.

The native multimedia subsystem has been designed from the ground up for mobile devices.¹⁰ It makes heavy use of active objects (see Chapter 3) behind the client interface to provide non-preemptive multitasking for the application programmer, while using multiple threads and hardware acceleration (where available) in the background to ensure the necessary performance. This makes it relatively simple to use when

⁹ See www.gstreamer.net.

¹⁰ For a full description of the Symbian multimedia architecture, refer to Rome, A., Wilcox, M. et al. (2008) *Multimedia on Symbian OS: Inside the Convergence Device*, John Wiley & Sons.

implementing a new application but rather inflexible from a porting perspective. None of the multimedia APIs can be used without an active scheduler running and this tends to force some re-architecting of any code that has to use them. The good news is that this doesn't necessarily mean a lot of change to the code, possibly just altering the context in which it is running and providing a simple wrapper to convert the input or output to match the APIs. An example of this is provided in Section 5.3.

5.1.3 Mobile-Oriented APIs

For the other API categories mentioned above (Bluetooth, PIM, telephony, messaging, device status and other hardware access), different mobile device platforms have their own solutions. In general you have to re-write these sections of a project to use native Symbian APIs. The necessary classes are mentioned briefly below. For further information, look up the classes in the documentation provided with your SDK.

Bluetooth

There are two options for accessing Bluetooth sockets on the Symbian platform. You can use the native `RSocket` and `CBluetoothSocket` interfaces or the non-standard extensions to the POSIX sockets API in P.I.P.S. that we covered in Chapter 4. There are also native APIs for service discovery and a standard way of interacting with the user to select another device to connect to. S60 also adds a simple 'send' UI (`CSendAppUi`) which hides the details of the bearer from the developer. It allows a user to select how they want to send a message or object giving options for Bluetooth, SMS, MMS, email and infra-red if they are available. If Bluetooth is selected, the Bluetooth Object Push Profile (which uses the OBEX protocol) is used.

PIM

Third-party applications are free to store their own contacts and calendar data in any way they like. However, in order to interact with the databases used by the built-in contacts and calendar applications, it is necessary to use the native APIs. `CContactDatabase` is the central class for contacts management and `CContactItem` represents an individual contact. Calendar data is managed by the calendar server, to which an application connects via `CCalSession` and creates a view of the data with `CCalEntryView` or `CCalInstanceView`. Individual calendar entries are represented by `CCalEntry`. There are papers on the Symbian Developer Network describing how to use each of these APIs; for contacts functionality, refer to

developer.symbian.com/main/downloads/papers/Using_Symbian_Contacts_Model.pdf and for calendar functionality, see ***developer.symbian.com/main/downloads/papers/Using%20the%20new%20v9%20Calendar%20API.pdf***.

Telephony and Device Status

Sometimes, when working with mobile device software, it's easy to forget that the primary function of the device is as a telephone. The `CTelephony` class allows you to dial, answer and control voice calls. It also provides support for retrieving phone settings, line information, network information and supplementary service settings. The functionality covered here includes most of the common device status information that developers require: battery level, charger connection, signal strength, flight mode status and cell ID. `CTelephony` provides asynchronous methods to access the data from underlying components. To achieve this it makes heavy use of the active object framework and requires an active scheduler to be running. There is a series of useful pages on the Forum Nokia wiki explaining how to use this class and they are all linked from ***wiki.forum.nokia.com/index.php/Usage_of_CTelephony***.

Messaging

For many mobile device users, the messaging functionality is even more important than telephony. The Symbian platform provides a powerful and complex messaging architecture, which can be extended to provide different types of message support with plug-ins known as 'message type modules' (MTMs). Projects that require a very high degree of control over messaging or want to retrieve stored messages can access individual messages and MTMs via the Message Server (using the `CMsvSession` class), or implement a new MTM. However, for most use cases, the functionality provided by the `RSendAs` and `RSendAsMessage` classes is perfectly adequate. In some cases, the `CSendAppUi` class mentioned in the Bluetooth section is sufficient and provides the simplest implementation option. For projects with more exotic use cases, be warned that the messaging architecture is quite complex, although it has been around for long enough to generate a lot of examples and documentation, both official and unofficial. Your favorite search engine is probably the best way to find the example you need.

Other Hardware Access

There are lots of interesting bits of hardware that are present on some mobile devices. Sometimes they can be accessed and sometimes not. It isn't usually possible to access DSPs or hardware accelerators

directly with public APIs. It is possible to query or modify backlights (for display and keys) on some devices; along with a range of other device hardware details, this can be found in the Symbian `HAL` class, although the reliability and availability of the implementation varies from device to device. Other models may have one or more cameras that can be controlled and used to capture images and video; these can be accessed via the `CCamera` class, which is another class that requires an active scheduler. More recently, devices have been adding sensors, such as accelerometers, magnetometers, tilt sensors and tap sensors. Nokia provides a common API for retrieving data from different types of sensor. Please read the warning on the wiki page for this API, wiki.forum.nokia.com/index.php/S60_Sensor_API, before using it.

For all of the APIs discussed in this section, there may be other options for porting as part of Qt, OpenKODE or the RGA libraries. These are covered in Chapter 6. For the rest of this chapter, we focus purely on hybrid code that mixes ported code with native Symbian C++.

5.2 How to Create a Hybrid Port

Having described the most commonly-used functionality that needs to be supplied by a platform-native alternative and provided a quick tour of the available alternatives, it's time to look at exactly how to mix portable code with native code and explain where you need to be careful.

All code in a `.cpp` file is compiled with C++ linkage unless you tell the compiler otherwise with the `extern "C"` declaration, as described in Section 2.6. If your project also includes `.c` files then make sure that you have the necessary C linkage declarations where you are calling C code from C++ or global C++ functions from C code. If you don't do this you'll get a mangled name for one and not the other and the linker won't match them, resulting in an 'undefined reference' error. An alternative approach is to supply an option to the compiler to tell it to compile all code as C++. Since C is not quite a proper subset of C++, this can often lead to a number of additional errors and warnings to fix, although these are usually trivial to resolve. The other place this is important is where you are looking up symbols in a DLL by name (for the `STDDL` target type). If you don't declare them with C linkage then the mangled name appears in the export table and you have to use it to look them up.

Given that you can mix standard C and C++ correctly, the next issue is mixing with Symbian C++. There are two major issues to consider; one is the context in which the code runs and the other is error handling. There are also some more general limitations relating to the way the standard APIs are implemented on the Symbian platform.

5.2.1 Active Scheduler vs Main Loop

The vast majority of native Symbian C++ code is executed, indirectly, from the `RunL()` method of some active object. As explained in Chapter 3, these active objects are non-preemptively scheduled by an active scheduler; that is, the `RunL()` functions are always allowed to run to completion before the scheduler selects the highest priority waiting active object to run next. This framework allows developers to implement multi-tasking systems and access resources asynchronously without having to deal with the complexities of multi-threading directly. Most of the native APIs make use of this, even if the active objects are hidden within a client library. This means that, to use the native APIs, an active scheduler has to be installed and started in the thread that is calling the function. In a native UI application, the application framework creates, installs and starts an active scheduler; it also automatically listens for events relevant to the UI such as keypad input and change of focus (e.g. an incoming call may force the application into the background).

In a non-UI ported application or server process, code usually runs from a main loop, often using `select()` or `poll()` to respond to external events. If your code needs to use native APIs, it is usually necessary to create and install your own active scheduler. The code to do this is very simple:

```
CActiveScheduler* scheduler = new(ELeave) CActiveScheduler;
CleanupStack::PushL(scheduler);
CActiveScheduler::Install(scheduler);
CMyActive* active = CMyActive::NewL();
active->StartSomeRequest();
CActiveScheduler::Start();
```

The `CActiveScheduler::Start()` function does not return until the active scheduler is stopped by an explicit call to its `Stop()` method. Note that an example active object, `CMyActive`, is created and a request is made before the active scheduler is started; this is because the scheduler waits for any requests on its active objects to complete before running any code; if there are no requests, it doesn't run anything. This means there are basically three options:¹¹

- 1 Port all of the code to run in the context of various active objects.

¹¹ There is a fourth option, although it's a little more complex. You can create an idle priority active object which immediately completes its own request and stops the active scheduler in its `RunL()` method. Then you can make a request to this active object and start the active scheduler to temporarily yield from your main loop to allow any waiting active objects to run.

- 2 Only start and run an active scheduler for as long as it takes to complete a single request or operation.
- 3 Run an active scheduler in a separate thread.

Option 1 usually requires quite a lot of work and makes the code much less portable. Option 2 has the disadvantage of blocking your main thread until the request or operation is completed, which could be a problem if there is a requirement to respond to other events which do not use the active object framework. Option 3 leaves you with the responsibility for implementing the thread synchronization and communication mechanism in order to use the functionality from your main thread.

At the time of writing, there is another option in the development version of P.I.P.S. (it is not yet available for use but we hope it will come soon). The asynchronous requests that are wrapped by active objects in the client libraries can often be accessed directly. For example, there is a `CTimer` class (and `CPeriodic` and `CHeartbeat` classes that derive from it) which accesses kernel timer resources via an `RTimer` class. It's possible to use the `RTimer` class directly without an active scheduler but then it is necessary to somehow wait on, or poll, the `TRequestStatus` variable passed to any request to find out about completion of the event. In order to improve the support for hybrid coding, the P.I.P.S. team have added an `eselect()` function which can wait on a number of file descriptors and `TRequestStatus` variables simultaneously. This enables use cases such as waiting for whichever comes first, a network event via a POSIX socket or a key press using a native API.

5.2.2 Error Handling

The code snippet in the previous section contains three functions that can 'leave'. The majority of native Symbian APIs include such functions and so almost all native code requires the use of a cleanup stack (refer to Chapter 3). The application framework creates the necessary cleanup stack and top-level `TRAP` handler for the main thread.

They are also created automatically if you use the 'glue code' or `STDEXE` target type described in Chapter 4. Similarly, if you create a POSIX thread with `pthread_create()`, then it has the same cleanup mechanism in place. However, if you create an `EXE` target without the UI framework or 'glue code' or create a thread with `RThread::Create()`, then you need to create the cleanup mechanism yourself with code such as this:

```
CTrapCleanup* cleanup=CTrapCleanup::New();
TRAPD(error, threadMainFunctionL());
```



```
delete cleanup; // destroy cleanup stack
return error;   // and return
```

This is the absolute minimum level of error handling that prevents your application or thread from causing a ‘panic’ when a function leaves. In practice, you will usually want to add additional TRAP or TRAPD macros around code that can leave and deal with the errors locally, rather than having the thread exit.

Internally, a leave is implemented as a standard C++ exception. However, it is not advisable to mix leaves with standard exceptions as this is likely to cause problems unless you know exactly what you’re doing and how the underlying leave mechanism works.¹² In most cases, you can simply use an error adaptor to convert from one to another. An error adaptor is a function that wraps a leaving function in a TRAP macro; if there is an error, it throws an appropriate exception instead. For example:

```
void CreateBuffer(RBuf8& aBuffer, TInt maxLength)
    throw (std::bad_alloc)
{
    TRAPD(err, aBuffer.CreateL(maxLength));
    if (err == KErrNoMemory)
    {
        throw std::bad_alloc();
    }
}
```

Alternatively, an error adaptor can wrap a TRAP macro in a leaving function. For example:

```
void AppendEntryL(node_array& myArray, node& element)
{
    try
    {
        myArray.append(element);
    }
    catch (std::bad_alloc)
    {
        User::Leave(KErrNoMemory);
    }
}
```

There are two things worth noting about this kind of construction. First, a fairly significant performance penalty is incurred when trapping one exception and throwing another. Second, you need to be particularly

¹² Further information is available at developer.symbian.org/wiki/index.php/A_Comparison_of_Leaves_and_Exceptions.

careful to avoid a nested exception occurring, as this is not supported on target builds for the Symbian platform (although it does work in the emulator, so beware). Simplifying as much as possible, when you are purely using the cleanup stack, the destructors for heap-based objects are called outside the exception handling mechanism and it is therefore safe to `TRAP` a potentially leaving function in the destructor of such an object. If you start mixing with standard C++ exceptions and the use of `auto_ptr`, then this assumption is no longer valid. It's necessary to check any Symbian C++ classes you use to ensure that they don't do this. For standard C++ on the Symbian platform in general, not only should destructors never throw any kind of exception, they also shouldn't call functions that can throw exceptions within an internal `try...catch` block.

5.2.3 Interleaving Standard C/C++ and Symbian C++

There is a restriction on mixing various P.I.P.S. and Open C/C++ function calls with similar native Symbian C++ calls due to the way that P.I.P.S. has been implemented (see Chapter 13 for more details of the P.I.P.S. architecture). A lot of the I/O operations in P.I.P.S. are buffered and it is not recommended to attempt to interleave standard API calls with native ones operating on the same resource. For instance, using `fopen()`, `fseek()` and `fwrite()` in conjunction with `RFile::Open()`, `RFile::Seek()` and `RFile::Read()` on the same file is unlikely to produce the desired result. It is best to stick to one set of operations for a specific resource. Otherwise, always ensure you do a `Flush()` operation or equivalent using the current interface before switching between APIs.

Another area for possible errors of this kind is with threads. The `pthread` implementation wraps the native `RThreads`, so it is possible to use both types together. However, `pthread`s share a heap by default whilst `RThreads` don't. If you create a `pthread` from an `RThread` other than the main thread, the `pthread` shares a heap with the `RThread` which created it, rather than the main thread. Again, if possible, stick to one type of thread unless you're sure you know what you're doing.

5.2.4 Hybrid Application Architectures

In Section 2.3, I described the ideal application architecture, where the UI has a nice clean separation from the engine. Even in this scenario, there is still some need for hybrid coding. The UI must call the functions of the engine and, in some cases, the engine needs to send events to the UI asynchronously. When the engine only has a code-level separation or is implemented in a separate DLL, this may require the use of some of the error adaptation techniques described in Section 5.2.1. Alternatively,

if the engine is implemented as a separate server executable, running in its own process, then there is a need for inter-process communication using either POSIX or native Symbian IPC mechanisms. So, either the UI has some POSIX IPC code or the engine has some native Symbian IPC code.

For simplicity and maximum portability of the engine, I'd recommend using POSIX mechanisms throughout. One simple way of doing this is to launch the engine process using `popen3()` and make use of the standard input and output pipes connected to the UI. This technique is particularly useful for porting an existing command-line utility and giving it a custom console, where the UI can use the `CConsoleBase` or `CEikConsoleScreen` classes, or adding a graphical UI to a ported command-line utility.

As mentioned in Section 5.1.1, there are a number of porting examples and papers already available that take a portable engine and give it a new UI. However, that isn't always possible. In reality, there are often parts of the project that don't separate so nicely but you still want to re-use as much of the code as possible.

5.3 Example: Guitune

This is an example of a porting project where the traditional UI and engine split doesn't make much sense because the UI is entirely data driven. The UI is basically the whole application, there is no internal state to store or manipulate and the user doesn't need to control anything. The project is a simple tuner for a guitar (or other instrument). It needs to use native multimedia functionality to access incoming audio from the device's microphone. In Section 5.1, we saw that cross-platform UI and multimedia APIs were not available on the Symbian platform at the time of writing. Since the application takes audio input, processes it and displays it, you may wonder if there is much effort to be saved by porting an application rather than writing one from scratch. It turns out that there is ...

I'd been thinking about getting an electronic tuner for my (acoustic) guitar for a while because, frankly, I'm unable to tune it properly by ear. The result is that my guitar is usually out of tune and makes my playing sound even worse than it really is. However, I didn't want to pay for one when my mobile device should be able to do a perfectly good job. There are some commercially available Symbian applications for this purpose but no information on how they work, so I didn't know if they'd be any good. Creating a utility I actually want to use and building an example of a hybrid application at the same time was too good an opportunity to miss. Full source code for this example,

along with the original project for comparison, is available from developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform.

5.3.1 Project Selection

A quick search of the Internet reveals four open source, desktop-based tuner projects written in C or C++: Gstring, GuiTuner, KGuitune and G-tune. Looking at the project websites reveals that Gstring and GuiTuner both use the FFTW library, a fairly heavyweight, optimized, floating-point Fast Fourier Transform (FFT) library. KGuitune uses a simple, integer-based, Schmitt-triggering algorithm (it basically counts the biggest oscillations) and G-tune uses a much simpler FFT algorithm to determine the sound frequency, but still uses floating point. The absence of intensive floating-point calculations makes KGuitune the top candidate, enhanced by the fact that there are already three versions: one for KDE, one using Qt and the third for gtkmm (the C++ bindings for the GTK+ libraries). All of the projects are fairly small one- or two-person efforts (as would be expected for a simple utility) and none appear to be abandoned or actively maintained. Since Qt support is planned for Symbian but not yet available, it made sense to select gtkGuitune, the gtkmm version of KGuitune. That should mean that this example is useful for anyone porting from GTK+ code in the future; if I'd ported one of the versions that use Qt, it would obviously make much more sense to port to the Qt libraries on Symbian.

5.3.2 Code Analysis

Scanning through the source files shows that there's a main container which hosts the various UI widgets, audio initialization and the main application logic. There are also three custom UI widgets, an oscilloscope-style view of the data, a logarithmic scale showing the note played and an LCD-style box for displaying frequencies. The settings boxes shown in Figure 5.1 are standard GTK+ widgets. The other points worth noting are that the audio is obtained by reading from `/dev/dsp` directly and that there is code to handle both unsigned 8-bit and signed 16-bit pulse code modulation (PCM) data, although the 16-bit option is used by default. There is very little use of other external libraries, only some simple mathematics and string formatting. The major negatives were that:

- It appeared to be basically C code in class wrappers.
- Some fundamental calculations were mixed in with drawing functions.
- There were practically no comments.

This might prompt a re-visit of the project selection but in this case the other options seemed far less suitable.

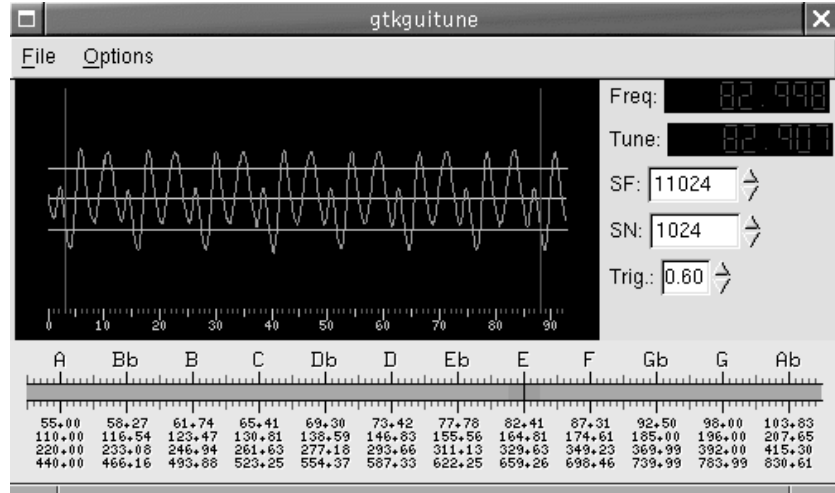


Figure 5.1 gtkGuitune

5.3.3 Architecture Changes

The main change required is to shift from polling the audio via a callback from the GTK+ main loop to driving the application via the callbacks from a Symbian C++ `CMdaAudioInputStream` object. The audio input stream allows you to read buffers from the audio device in a very similar way to reading the raw data from `/dev/dsp` on Linux. The custom widgets are all derived from `Gtk::DrawingArea`, which is equivalent to a custom control on the Symbian platform. Deriving these and the main container which holds them from `CCoeControl` on Symbian should make the bulk of the code portable between UI platforms, although I've chosen to target only S60 with this example.

The main widgets (controls) can be re-positioned to fit on the mobile screen, although the table of frequencies at the bottom of the logarithmic control do not fit and are not really necessary anyway. The settings boxes should be re-implemented as a native Symbian settings list in a separate view, if required.

5.3.4 Development Environment

For this project, I used Carbide.c++ v1.3 Developer Edition under Windows XP with the S60 3rd Edition (Maintenance Release) SDK and a Nokia N95 (although this example should work with minimal or no changes on any S60 3rd Edition device). At the time, it was possible to use the free Express Edition of Carbide.c++ for development (commercial and freeware) but I used the Developer Edition, available from Forum Nokia (www.forum.nokia.com/carbide), which you had to pay

for but it includes on-device debugging (which I do via Bluetooth) and a graphical UI designer tool. Both of these easily paid for themselves in terms of productivity gains and experience working with multimedia on the Symbian platform told me I'd probably want to use on-device debugging. Fortunately, Nokia have made all versions of Carbide.c++ free and contributed the source to the Symbian Foundation, so you no longer need to make this decision.

5.3.5 Creating the Project

Since I wanted a standard S60 application, I created one from a template using the new project wizard in Carbide.c++ (in fact, I used the UI designer to create a blank container application, in case I needed to add a settings list later, but the result is very similar). This generates a number of source, header and resource files. The MMP and PKG files are also generated automatically. With a few small exceptions that I'll discuss later (in Section 5.3.7), there was no need to edit any of these generated files apart from the application's main container. The container starts out as a blank control which fills the area between the status bar at the top of the screen and the soft keys at the bottom.

I created three copies of the container's header and source files, renaming them and doing a 'find and replace' on the class name internally, to produce the skeletons for the oscilloscope, logarithmic scale and LCD controls. Adding these files to the project in Carbide.c++ automatically adds them to the MMP file. Additional libraries, such as `mediaclient-audioinputstream.lib` (for `CMdaAudioInputStream`) have to be added to the MMP file manually, although there is a graphical editor with a drop-down selection box for this purpose.

My strategy was then to port the functionality across (mostly via cut, paste, find and replace) one control at a time, starting with the main container and the oscilloscope control which represent the core of the application. I debugged these before adding the other controls which are basically just additional (although more useful) views of the data. In this case, one control at a time equates to one source file at a time. It may be possible to use a similar incremental strategy with less UI-centric ports if the various classes or modules are not too closely coupled.

5.3.6 Getting It to Compile

The first major change was to re-write the audio handling. I based my implementation on an example in the Forum Nokia wiki.¹³ I did simplify the wiki example slightly as I used it, replacing an array of heap-allocated

¹³A search on 'recording audio' produced three article title matches. The second one, 'Recording audio with stream' was exactly what I wanted (wiki.forum.nokia.com/index.php/Recording_audio_with_stream).

TBuf8 objects with a single RBuf8 for the audio buffer¹⁴; although that wasn't essential, it made it easier to adapt to the ported code. Once you've got the data from the API you can access it as an array from ported code by using the `Ptr()` method to get a pointer to the start of the data.

At this point, I must also confess to using some detailed knowledge of the target device to select a buffer size (4096 bytes) to match that used internally by the API. A novice is likely to arrive at a working value by trial and error after discovering some quirks of the interface.¹⁵ Some slight modifications were needed to the function which processes the audio, as the original design read small blocks (256 bytes) until it had accumulated enough to do the frequency calculations. Using the new system, we just return from the function and wait for the next large buffer from the audio device if we can't find a suitable section of data to process. If you examine the `CGuituneContainer::proc_audio()` method in the example code, then you'll find that many more characters are required to describe the changes than to make them! In the gtkmm version, `proc_audio()` is the key function called from the GTK+ main loop. In the Symbian port, it is called whenever a new buffer of audio data is available, as indicated by a `MaiscBufferCopied()` callback from the audio input stream. This should be far more power-efficient than the original polling design.

The other major difference is in the way UI layout is done. The gtkmm code uses a number of nested horizontal and vertical boxes to arrange the controls. For the Symbian platform, you simply specify the rectangle that you want the control to occupy when you create it. There are two complicating factors to be aware of here. First, the container is a 'compound control', as it owns component controls. The class header contains an enumeration of the component controls and the methods `CountComponentControls()` and `ComponentControl(TInt aIndex)` are used by the UI framework to determine how many controls there are and to access each one individually in order to draw it. When you add a control, you must include an entry for it in the enumeration and return a pointer to it when that value is passed as `aIndex`. The second thing that's important to know is that the Symbian platform supports multiple screen sizes and dynamic scalable UIs where the display can switch orientation from portrait to landscape. This means that you need to re-do the layout of your component controls when `SizeChanged()` is called and the layout needs to work at different resolutions. You can test with the various screen sizes available on the emulator, even if you don't have all of the appropriate devices. The good news is that well written GTK+

¹⁴ I hope you recognize these as descriptor classes from Chapter 3. The Symbian platform code uses descriptors to handle string and binary data. A good place to get started with descriptors is developer.symbian.org/wiki/index.php/Descriptors_Cookbook.

¹⁵ See wiki.forum.nokia.com/index.php/TSS000318_-_Customizing_the_buffer_size_of_CMdaAudioInputStream.

widgets handle re-scaling internally since the windows they occupy on the desktop can be re-sized. The component controls share their parent's window (they could create their own windows but it's not very efficient) and all drawing is done relative to the origin of the parent window. This means you have to add offsets to the co-ordinates used for drawing whenever the layout changes. You can get the co-ordinates for the top left corner of a control, by calling its `Position()` method, and the size of the control's rectangle, with `Size()`.

The minor differences are changing `Gdk_Color` to `TRgb` and switching the graphics context from `Gdk_GC` to `CWindowGc`. In the original version, the widget classes each own a graphics context. On the Symbian platform, the standard model is to get the currently-active graphics context from the system when asked to draw. To minimize changes to the function internals, I modified all methods that draw in the original code to take a reference to a graphics context as an argument. There are a number of system events in the `gtkmm` version which cause a redraw: `expose_event_impl()`, `draw_default_impl()` and `draw_impl()`. With the Symbian platform, this is handled by the framework and anything the system does that requires a control to redraw itself causes a call to its `Draw()` method with the rectangle that needs to be redrawn supplied as an argument. As a result, these methods from the original version are simply not required. Other drawing-related differences are that functions for drawing lines and rectangles belong to the window class and take a graphics context as an argument in `gtkmm`, while they belong to the graphics context in the Symbian platform. Additionally, Symbian C++ has a more object-oriented style so it takes two pairs of co-ordinates, each encapsulated by a `TPoint` object, to draw a line or a point and a size to draw a rectangle, rather than a list of integer arguments. For example:

```
get_window().draw_line(gc, x1, y1, x2, y2);
```

becomes:

```
gc.DrawLine(TPoint(x1, y1), TPoint(x2, y2));
```

and:

```
get_window().draw_rectangle(gc, true, x1, y1, width, height);
```

becomes:

```
gc.DrawRect(TRect(TPoint(x1, y1), TSize(width, height)));
```


The Symbian platform uses a pen color and a brush color rather than foreground and background colors. The difference is subtle but a rectangle is drawn in the pen color and filled with the brush color. If all you want is a solid rectangle of one color, then it is easiest to set the brush color and use the `Clear()` method rather than `DrawRect()`:

```
gc.SetBrushColor(KRgbBlack);
gc.Clear(TRect(TPoint(x1, y1), TSize(width, height)));
```

Here's an example of one of the drawing functions before and after porting. Changes to the original function in the Symbian version are highlighted in **bold**.

Linux version:

```
void OsziView::paintSample(void)
{
    int i,x1,x2,y1,y2;
    unsigned char * samp_uc;
    short int *    samp_s16le;
    //erase();
    samp_s16le = (short int *) samp;
    samp_uc    = (unsigned char *) samp;
    if (!i_GC){
        Gdk_Window win = get_window();
        i_GC = Gdk_GC( win );
    }
    i_GC.set_foreground(i_col_bg);
    get_window().draw_rectangle( i_GC, true, xscr,yscr,
                                wscr+1,hscr+1 );

    i_GC.set_foreground(i_col_zero);
    get_window().draw_line( i_GC, xscr,yscr+hscr/2,
                            xscr+wscr-1,yscr+hscr/2);

    i_GC.set_foreground(i_col_data);
    for(i=1;i<sampnr;i++){
        x1=xscr+(i-1)*wscr/sampnr;
        x2=xscr+i*wscr/sampnr;
        if(i_sampfmt == U8){
            y1=yscr+(samp_uc[i-1]-128+i_divisor/2)*hscr/i_divisor;
            y2=yscr+(samp_uc[i]-128+i_divisor/2)*hscr/i_divisor;
        } else if(i_sampfmt == S16_LE){
            y1=yscr+((int)samp_s16le[i-1]+i_divisor/2)*hscr/i_divisor;
            y2=yscr+((int)samp_s16le[i] +i_divisor/2)*hscr/i_divisor;
        }
        get_window().draw_line( i_GC, x1, y1, x2, y2 );
    }
}
```

Symbian version:

```
void COsziControl::paintSample(CWindowGc& aGc)
{
    int i,x1,x2,y1,y2;
    unsigned char * samp_uc;
    short int *      samp_s16le;
    //erase();
    samp_s16le = (short int *) samp;
    samp_uc    = (unsigned char *) samp;
    // if (!i_GC){
    //     GdK_Window win = get_window();
    //     i_Gc = GdK_GC( win );
    // }
    aGc.SetBrushColor(i_col_bg);
    aGc.Clear(TRect(xscr,yscr,wscr+1,hscr+1));
    aGc.SetPenColor(i_col_zero);
    aGc.DrawLine(TPoint(xscr,yscr+hscr/2),
                 TPoint(xscr+wscr-1,yscr+hscr/2));
    aGc.SetPenColor(i_col_data);
    for(i=1;i<sampnr;i++){
        x1=xscr+(i-1)*wscr/sampnr;
        x2=xscr+i*wscr/sampnr;
        if(i_sampfmt == U8){
            y1=yscr+(samp_uc[i-1]-128+i_divisor/2)*hscr/i_divisor;
            y2=yscr+(samp_uc[i]-128+i_divisor/2)*hscr/i_divisor;
        } else if(i_sampfmt == S16_LE){
            y1=yscr+((int)samp_s16le[i-1]+i_divisor/2)*hscr/i_divisor;
            y2=yscr+((int)samp_s16le[i] +i_divisor/2)*hscr/i_divisor;
        }
        aGc.DrawLine(TPoint(x1, y1), TPoint(x2, y2));
    }
}
```

As you can see, although the changes are fairly extensive, they are also fairly mechanical, in that they require no deep understanding of the code. Also note that although there would have been fewer changes required if the graphics context continued to be called `i_GC`, rather than `aGc`, I have used the Symbian variable naming conventions to avoid confusing the argument with a member variable.

One further drawing-related difference highlighted by this port is for application-initiated drawing. In GTK+, you can draw wherever and whenever you like. On the Symbian platform, you must invalidate the rectangle you want to redraw and activate the graphics context before drawing. When the system initiates a redraw, it takes care of this for you. For application-initiated drawing you need to use the following code:

```
Window().Invalidate(invalidRect);
ActivateGc();
```

```
Window().BeginRedraw(invalidRect);
//call drawing functions here
Window().EndRedraw();
DeactivateGc();
```

The example code uses this pattern to update the oscilloscope and logarithmic scale controls for every new audio sample received. This is required because their existing drawing code only updates parts of the control in each case. The LCD controls need a complete update so they simply call `DrawNow()`, which requests an immediate redraw from the framework.

Text handling requires very similar changes to drawing¹⁶ but is slightly more complex due to the requirement to use 16-bit wide (Unicode) descriptors. It's simple enough to initialize an 8-bit wide descriptor with a zero-terminated C-style string or to use a `TPtr` descriptor type to point to the contents of one, but you then have to widen it to 16 bits. Fortunately, there is a `Copy()` method in the 16-bit descriptor base class (`TDes16`) which takes an 8-bit descriptor as an argument and does the right thing for ASCII text. If your string is UTF-8 encoded, then you'll have to use a text conversion utility (`CnvUtfConverter` should do the trick).

A couple of final points relate to mathematical and string manipulation functions. The application uses `log()` and `fabs()` from `math.h (libm)` as well as `sprintf()` from `stdio.h (libc)`. These functions are available as part of P.I.P.S. (see Chapter 4) but since they were only used a couple of times and there were simple alternatives, I've replaced them with native Symbian C++ operations (and a simple macro, in the case of `fabs()`) to avoid an unnecessary dependency. From Symbian^2 onwards, P.I.P.S. support is a mandatory part of the device firmware but many older devices require the user to install it as a separate SIS file. In this case, the P.I.P.S. install would be much larger than the application it is supporting. This wouldn't be an issue with a larger project; for one that makes extensive use of P.I.P.S. or Open C/C++ functionality, it wouldn't be at all advisable to switch to native calls.

5.3.7 Getting It to Work

As there is no test harness with the original project, I decided that some very simple manual tests would suffice. I used a signal generator program to produce sine waves across a range of frequencies and ensured that

¹⁶ Another quirk discovered during this port relates to fonts. `CWsScreenDevice::GetNearestFontToMaxHeightInTwips()` returns a font that is missing most of the ASCII characters on S60 3rd Edition but is fine on S60 3rd Edition FP1. `GetNearestFontToDesignHeightInTwips()` appears to work on both SDKs.

the application showed a sine wave on the oscilloscope and identified the correct note and frequency. After that, the final and most crucial test: could I use it to tune my guitar?

The first serious bug I came across was a blank container after starting the application, when I should have been seeing my custom controls! I spent a frustrated couple of hours checking that the drawing code was running correctly and trying to find out what was drawing over my controls or whether I'd made a mistake with the co-ordinates. The error was one that you could easily repeat if you're in a hurry so I'll share my embarrassment here. I'd copied RGB color values from the GTK+ code into ARGB values without realizing. The result is that the alpha value was set to zero for all the colors I was using. All of the drawing code was working, I was just drawing in invisible ink!

The next issue was that the oscilloscope control was being drawn too large. This was simply because, while copying the skeleton code, I'd forgotten to remove the following line from the `HandleResourceChange()` method of the control (which is called when the control is created and whenever it changes size or position). The control was setting its own rectangle to fill the entire window.

```
SetRect(static_cast< CEikAppUi* >(iCoeEnv->AppUi()) ->ClientRect());
```

With that solved, the application was basically working in the emulator, so it was time to move over to a real device (although note that at this point I'd only ported the oscilloscope control). In order to use the `CMdaAudioInputStream` API to capture audio, the application needs to have the `UserEnvironment` capability. This just needs to be added to the MMP file. If you miss a capability then your project will work in the emulator, where platform security enforcement is turned off by default, but not on a real device.

Unfortunately, as soon as the application started in the device, it exited with a 'Feature not supported' dialog. Experience (and common sense) tells me that the main difference between the device and the emulator for this application is access to the audio hardware. So, I fired up on-device debugging from Carbide.c++ and set a breakpoint in the audio initialization code. Sure enough, one of the set-up functions for `CMdaAudioInputStream` was leaving:

```
iAudioInput->SetAudioPropertiesL(iAudioSettings.iSampleRate,  
                                iAudioSettings.iChannels);
```

Initially, I thought that this was because of an unsupported sample rate but found that it doesn't work with any sensible value, so it seems that

the method is not supported on the Nokia N95 (although it was on earlier models). Differences such as this between devices are not that uncommon and it's good practice to TRAP the leave and deal with the failure at run time. However, in this case, the call is redundant because the audio properties are supplied in the call to `Open()` the audio input stream, so I simply commented it out. In theory, the `MaioOpenComplete()` callback should give you an error code other than `KErrNone` if the settings are not supported. In practice, I've found that this doesn't occur and you get a sample rate of 8000 Hz, no matter what value you request and no error. I hard-coded the sample rate accordingly, so that value doesn't require a setting item. Also, the number of samples displayed is somewhat constrained by the width of the screen so I decided that didn't require a setting either. Eventually, I found that the default value for the trigger level (0.6) worked very well for identifying frequencies and so I decided not to have any settings at all, making the application a simple single view.

Having decided that no user options were required, I changed the resource file that defines the soft keys (this is the application's main resource file, in this case `guitune.rss`) so that it used `R_AVKON_SOFTKEYS_EXIT` rather than `R_AVKON_SOFTKEYS_OPTIONS_EXIT`.

This had the desired effect of producing a single 'Exit' soft key but now I couldn't exit the application with it, on the emulator or the target. `HandleCommandL()` in the Application UI class, in this case derived from `CAknAppUi`, is where the keypad input goes by default, so I set a breakpoint in the debugger to look for the appropriate command. In this case, it seems that `EAnSoftkeyBack` is generated rather than `EAnSoftkeyExit`, so I added that value to the ones that result in the application exiting. These were the only changes I had to make to the generated application skeleton. If I were polishing the application for release I might want to change the application icon too.

I now had a working guitar tuner that passed all of my tests (see Figure 5.2). One final change was required to get a satisfactory result on the device. After a short period of use, the backlight on the display would go out, making the screen almost impossible to see. This is a standard power-saving feature during periods of user inactivity on Symbian devices and in most cases is what you want to happen. However, in this case we are using the device but it has no way of knowing that because we aren't pressing any keys. Guitar tuning is a two-handed operation, so stopping to press a key to get the backlight back on isn't at all user friendly. The solution here is to call `User::ResetInactivityTime()` at regular intervals, in this case in the callback from the audio input stream. With that in place we have a really usable little application.

To be a good mobile device citizen, we should also implement our own power-saving timeout of a suitable length and stop the audio input stream (and hence the inactivity resets). You'd probably then want to add

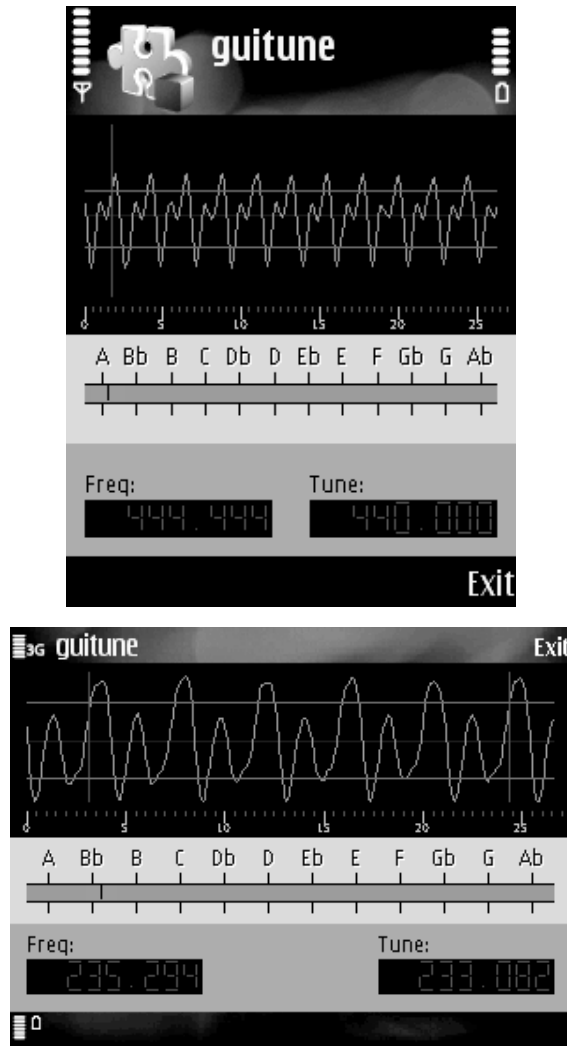


Figure 5.2 The tuner application running on a Nokia N95

a soft key to re-start it at a later time. The same stop–start mechanism could also be used if the application was moved to the background (currently it just keeps running). However, this is a porting example and not a finished product; these features are left as an exercise for the reader.

Note that the code changes from the original here are too significant to make any re-integration feasible. The ported application would have to be maintained separately from the original but, due to the GPLv2 licensing, full source code must be released along with a copy of the license if the project is distributed.

5.4 Summary

In this chapter, we've had a brief tour of the APIs available on popular desktop and mobile platforms which you won't find on the Symbian platform and their native equivalents. When your ported project uses these APIs, it's necessary to write some hybrid code that mixes portable code with native Symbian C++. The most common issues faced when writing hybrid applications were covered in some detail and an example was given. In the example, I discussed the issues involved in porting a simple application that uses both UI and multimedia functionality from Linux to the Symbian platform. Using the guitar tuner example, I hope I've shown that even where standard APIs from Linux are not available, porting to the Symbian platform is not too daunting a task. The example also highlights some common issues that need to be considered when porting to mobile platforms in general.

Obviously, it's impossible to cover all of the native APIs and issues that you'll face while writing hybrid code as part of a port but this chapter should provide a good starting point. For more specific details of the other mobile device platforms and how to port code from them to the Symbian platform, please refer to the guides in Chapters 7–9. Before that, Chapter 6 focuses on other APIs that are, or will be, available on Symbian devices and which could help to make code more portable. They provide alternatives for porting existing code that should also allow it to be ported to other platforms more easily.

6

Other Port Enablers

Most problems go away if you just wait long enough. It might look like I'm standing motionless but I'm actively waiting for our problems to go away. I don't know why this works but it does.

Dilbert

In this chapter, we look at some other APIs that have been recently released or are coming soon to devices based on the Symbian platform, which should all make porting code to the Symbian platform easier in appropriate circumstances. By the end of the chapter, you should have a good overview of the following technologies, including where and how they can be used:

- real-time graphics and audio (RGA) libraries
- Simple DirectMedia Layer (SDL)
- OpenKODE
- Qt.

These technologies can be used as target APIs for porting from other libraries and, with the exception of the RGA libraries, they provide cross-platform APIs which allow you to port code written for them to the Symbian platform with minimal changes. They should make a port easier and, in many cases, they also improve the portability of the code to other platforms.

6.1 Real-time Graphics and Audio Libraries

Nokia's real-time graphics and audio (RGA) libraries provide C++ developers with access to S60 functionality, without the need to use the

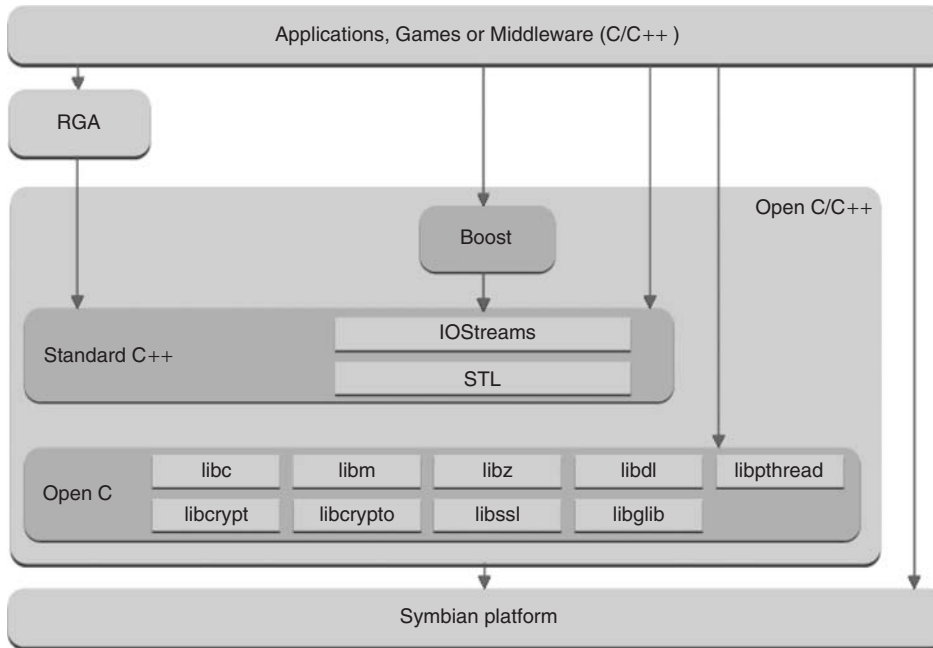


Figure 6.1 The relationship between RGA, Open C/C++ and Symbian C++

Symbian C++ idioms that the native APIs require. For example, you can use the RGA libraries to access multimedia and graphics services on the phone without having to use Symbian C++ at all. These libraries allow you to create more complex applications than those possible using Open C and Open C++ alone. Figure 6.1 illustrates the relationship between the RGA libraries and Open C/C++, on which they have a dependency.

The RGA libraries are of particular interest and utility to game developers. The libraries are shipped with the N-Gage SDK (*insider.n-gage.com*) to allow professional game developers to create high-quality, feature-rich games, either by writing them from scratch or by porting them from other platforms such as Sony PlayStation Portable, Nintendo DS or even a console or PC platform.

At the time of writing, the N-Gage SDK is restricted to developers working directly in a professional context with Nokia. However, the RGA libraries are available to all S60 C++ developers as a plug-in to the S60 3rd Edition SDK. This makes them available to game developers working outside of the N-Gage platform and to developers creating applications other than games.

The RGA libraries are shipped in the Open C/C++ plug-in for S60 3rd Edition SDKs, which can be downloaded from Forum Nokia at www.forum.nokia.com/Technology_Topics/Development_Platforms/

Open_C_and_C++. When you install it, the plug-in creates a subdirectory in your main 3rd Edition SDK installation directory (called `\nokia_plugin`, by default) and you'll find the documentation and example projects in the `\rga` subdirectory it contains. Also included in this subdirectory are the 'run-time plug-ins' which you must ship with your application binaries (they take the form of a signed SIS file that you should embed in your own SIS file) for installation onto the user's device.

The plug-in installation also places the header files and libraries required for development in the appropriate location within your S60 3rd Edition SDK (for example, the header files are placed under `epoc32\include\rga`).

6.1.1 Limitations of the RGA Libraries

The RGA libraries are not distributed on any S60 3rd Edition devices by default so, as I mentioned previously, you must embed within your application's SIS file a SIS file that deploys the RGA libraries to the device. In addition, the RGA libraries have a dependency on the Open C and Open C++ libraries, which you must also deploy on devices that don't embed those libraries. To run on S60 3rd Edition and 3rd Edition FP1 devices, you need to deploy both the Open C and Open C++ libraries; devices based on S60 3rd Edition FP2 have the Open C APIs delivered in their firmware, so you only need to deploy Open C++.

Any use of the RGA libraries requires that your application has the `SwEvent` capability and this in turn requires that your application be Symbian Signed (see the discussion of platform security and Symbian Signed in Chapter 14). Since the application requires restricted capabilities, the most elegant way to deploy the additional libraries with your application is to use the SW Installer Launcher API,¹ available from Forum Nokia, to install them as part of your application's installation process. Use of this API is beyond the scope of this book; please see the documentation at Forum Nokia.

Not all Nokia S60 devices (and no devices from other S60 licensees, such as Samsung) support the RGA libraries. No S60 5th Edition devices are supported. The following devices are supported at the time of writing:²

- Nokia 6290
- Nokia E70
- Nokia N73

¹ This API requires the `TrustedUI` capability, which also requires Symbian Signing. Documentation for the API can be found at wiki.forum.nokia.com/index.php/SW_Installer_Launcher_API.

² The list of supported devices comes from Forum Nokia (www.forum.nokia.com/Resources_and_Information/Tools/Runtimes/C++/Open_C_and_C++_Plug-ins_for_S60_3rd_Edition/Features.xhtml) and is updated regularly.

- Nokia N76
- Nokia N81
- Nokia N82
- Nokia N93
- Nokia N93i
- Nokia N95
- Nokia N95 8 GB.

In practice, this means that, if your application uses the RGA libraries, it will only run on the devices listed. You should never allow a user to be frustrated by installing an application onto a phone that does not support it. The best approach is to allow your application to be installed only to devices that support the RGA libraries. Unfortunately, the only reliable way to do this at present is to include a list of all the compatible devices in your application's package file.

By default, applications created using only the RGA libraries for the user interface do not appear in the task list. Correcting this is non-trivial and beyond the scope of this book.

6.1.2 Uses of the RGA Libraries

The Open C/C++ plug-in for S60 3rd Edition SDKs contains a number of examples of how to use the individual APIs provided by the RGA libraries, and the documentation includes instructions on how to run the examples on the Windows emulator and how to build and deploy the code to a device. However, an isolated example of how to use an API isn't always sufficient to illustrate its use in the real world, where you need to combine its use with other APIs and idioms. To help you get a better idea of how to use the RGA libraries, Forum Nokia provides a pair of game examples, Tetronimo and Biowaste, which can be found at www.forum.nokia.com/Resources_and_Information/Documentation/Open_C_and_C++.xhtml.

6.1.3 Features of the RGA Libraries

The RGA libraries can be subdivided into the following major categories: graphics, multimedia and utility APIs.

Graphics APIs

These APIs provide features for drawing to the screen, bitmap manipulation, rendering text and adjusting some of the display settings, such as the brightness and contrast.

Back Buffer

The Back Buffer API, `backbuffer.h` and `graphics.h`, allows drawing to the screen without tearing and flickering effects. It also allows changes to the orientation of the screen content. The API provides a device-independent, double-buffered graphics scheme consisting of a screen buffer and a secondary buffer (called the back buffer), which can be accessed and drawn into. When drawing is completed, the contents of the back buffer are moved to the screen buffer and become visible on the screen. Single and chained back buffers are supported.

Bitmaps

The Bitmaps API, `ngibitmap.h`, is an interface for creating and manipulating bitmaps. It includes functionality to:

- perform masked blits from one graphics device to another, such as a bitmap or the back buffer
- perform alpha-blended blits from one graphics device to another
- use self-defined color palettes
- scale and rotate the content of graphics devices
- access and modify pixels.

Display

The Display API, `display.h` and `displaymanager.h`, provides system-independent window handles for all of the phone's displays. These window handles are required to create a back buffer on a specific display. The API provides notification of changes to the orientation of the display, can be used to return the display settings and capabilities and allows adjustment of some of those settings, if the underlying system supports it. Display settings include brightness, contrast and gamma correction values. The capabilities returned by the Display API include the native size and resolution, orientation, and color formats.

Fonts

The Fonts API, `fonts_v2.h`, provides a system-independent interface for drawing text to a graphics device (back buffer or bitmap surface) using system fonts with a specific font face, size, and style. It also provides an interface for creating application-specific bitmap fonts and drawing text to a graphics device at a particular point.

Common features of system fonts and bitmap fonts include:

- drawing text to a graphics device (back buffer or bitmap)
- retrieving the size occupied by the text drawn

- applying text effects, such as underline and strikethrough
- drawing text in different directions
- drawing text with the user-specified color
- drawing text to the graphics device from a given target point.

The following system font features are available:

- installing and uninstalling system fonts
- getting the installed system font count
- getting information about the currently installed fonts
- converting unit size in twips to pixels.

The supported system font formats are True Type fonts (TTF) and Symbian bitmap (GDR) format. Bitmap font support includes:

- setting the kerning information between the characters
- setting the transparent color for the text in RGB format
- selecting whether transparency should be used or not
- setting default spacing between the characters.

Images

The Images API, `imageconverter.h`, allows you to load and save images from files of different color formats, and load images from memory buffers. Synchronous and asynchronous support is provided.

Multimedia APIs

These APIs can be used for audio clip and stream playback, recording and mixing of formats (such as MIDI) and playback of compressed audio data (such as MP3). APIs are also available for video playback, to access the device's cameras, and to manipulate the device's screen backlight and keypad lights.

Audio (High-Level and Compressed)

The High-Level Audio API, `highlevelaudio.h`, provides a set of functions for playing, recording, and mixing audio. Typical use cases supported include the following:

- retrieving the capabilities of the audio mixer
- initializing and configuring the mixer

- playing back sound on the mixer and stopping the mixer
- adjusting the mixer playback settings
- adjusting individual playback settings
- pausing or stopping playback of a sound
- managing sounds in logical units (tracks)
- recording audio data.

An additional Compressed Audio API, `audio.h`, is available for configurable low-level audio playback and recording. This includes support for playback of MIDI and compressed audio files, such as MP3 files, audio stream playback (buffer-based), and audio clip playback (file-based). Mixing functionality is also available, where it is supported by hardware, to allow playback of multiple streams simultaneously.

Camera and Video Playback

The Camera API, `camera.h`, supplies interfaces to access the phone's camera devices including functionality to:

- enumerate the available cameras
- query the capabilities of each camera to return information about the orientation of the camera, the possible optical and digital zoom values, as well as the supported image capture sizes
- get and set the configuration of a camera, including the zoom factor, contrast, brightness, flash mode, exposure mode, and white balance adjustment
- use the viewfinder to switch between what the camera sees and what the user would see if the device was a mirror; it is also possible to specify a clipping rectangle
- capture images and specify the color format and image size of the captured image; it is also possible to specify a clipping rectangle.

A Video Playback API, `videoplayback.h`, provides support for playing or pausing a video clip, and for configuring the playback image (the playback area size, position and orientation, the current and end position, and the playback volume).

Lights

The Lights API, `lights.h`, provides control of the screen backlight for the primary and secondary displays, and keypad lights. For example, the light sources can be set on for a specified time period with particular

intensity or can be set off for a specified time period. The lights may also have fade-in or fade-out effects applied. Lights can also be set to blink for a period of time at a specified intensity. The API can also be used to provide dynamic information about changes to the state of the lights.

Utility APIs

These APIs enable applications to perform a variety of tasks, including:

- determining the device's input layout
- receiving keyboard input
- determining the device's features and status
- setting and modifying alerts
- changing the active-theme and idle-screen wallpaper
- using programmable timers
- accessing the device's vibra motor
- loading code dynamically at run time.

Alerts

The Alerts API, `ngialerts.h`, provides interfaces to allow an application to create calendar alerts and to-do reminders that result in user notifications.

Device Capabilities and Device Status

The Device Capabilities API, `devicecapabilities.h`, provides access to information about the capabilities of an S60 device, such as whether it provides floating-point support, its CPU architecture and clock speed, and the amount of RAM available.

The Device Status API, `devicestatus.h`, provides interfaces to retrieve the current status of various hardware features, including specific information about the battery, network signal, connected accessories (such as headsets), telephony, alarms, and current profile. It can also be used to receive notification of changes to the device status, through a set of callback interfaces for the following:

- changes to the battery level
- connection or disconnection of the main charger
- changes to the network signal
- changes to the GSM cell ID or location area code of the phone

- connection or disconnection of an accessory
- changes to the phone's profile (such as the settings for keypad tones, warning tones, flight mode, ring tones, volume and display language)
- notification of an incoming phone call and its termination
- notification of an alarm.

Input

The Input API, `input.h`, provides information on the states and state changes of input devices, such as the embedded keypad and four-way controller, and external input devices connected to the phone via USB or Bluetooth wireless technology.

The Text Input API, `textinput.h`, handles text input on keypads in multi-tapping, predictive text input and numeric modes. It supports Western typing and the stroke, Shuyin, and Pinyin Chinese text input systems.

The Keypad Capabilities API, `input.h` and `runtime.h`, provides an interface to retrieve general information about the default keypad device connected to the phone, such as a list of the physical keys it supplies and the default action keys in its supported orientations.

Runtime and Application State

The Runtime and Application State APIs, `idle.h`, `runtime.h` and `object.h`, are provided to create a framework for code using the RGA libraries to load the necessary APIs and to handle asynchronous background tasks and notifications.

The Runtime API is used to instantiate objects of an API, loading each library as it is required. It also initializes the system and yields to the operating system to handle background events.

The Application State API notifies the application in which it runs of focus loss and gained events, and of events occurring when the operating system requests the application using the RGA libraries to shut down.

Themes

The Themes API, `ngithemes.h`, can be used to enumerate the themes currently installed on the device. It also provides the means to set and unset an installed theme, and to set the idle-state wallpaper.

Timers

The Timers API, `timing.h`, provides standard and high-resolution timers:

- Periodic timer: After the time has expired, the timer restarts automatically. The timer continues to run until it is stopped or released.
- One-shot timer: The timer has to be restarted manually after it expires.

Vibra

The Vibra API, `vibra.h`, provides control of the motor for phone vibration, allowing code to set the vibra on or off and receive notification of events indicating vibra state transitions and profile changes.

Virtual Code

The Virtual Code API, `ngivirtualcode.h`, provides a way to allocate, and later expand, a region of memory where code may be copied and executed, modified, and deallocated. The API provides the mechanism to synchronize and cache memory so that the loaded code can execute.

6.2 Simple DirectMedia Layer

The Simple DirectMedia Layer (SDL) is a cross-platform multimedia library designed to provide low-level access to audio, input methods, 3D graphics (via OpenGL) and a 2D video frame buffer. It has significant overlap with the RGA libraries; while not quite as comprehensive in functionality, it has the advantages of being cross-platform and supported across the entire range of open Symbian devices. It is used by video playback software, emulators and many popular games – particularly Linux versions.³

The official hosting for the SDL is at **www.libsdl.org**. This version supports Linux, Windows (desktop and embedded versions), BeOS, Mac OS X, various BSD versions, Solaris, IRIX and QNX. In addition, AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, Symbian and OS/2 are unofficially supported.

The most complete and up-to-date port for S60 devices is maintained by Markus Mertama at **koti.mbnet.fi/mertama/sdl.html**. This version has fairly comprehensive documentation to help you get started and several extensions to assist with adapting SDL applications to the S60 platform. A couple of noteworthy extensions are graphics scaling and orientation support. Graphics scaling resizes the canvas to fit the screen for each frame drawn. This feature can enable SDL-based projects to be ported to S60 with no code changes at all: simply creating the necessary build files should be enough to get the project up and running.⁴ The orientation support allows developers to manage rotation of the user interface and even has a mode that automatically selects the most appropriate orientation based on the dimensions of the video surface.

³ See **en.wikipedia.org/wiki/List_of_games_using_SDL** for the most popular titles or get a more complete list at **www.libsdl.org/games.php**.

⁴ However, using this feature can impose a significant performance penalty so most applications, and particularly games, eventually need to adapt to the screen size manually.

Markus Mertama's port of the SDL library has been used to create the popular S60 port of the classic first-person shooter, Doom, known as C2Doom. There is also an open source port of the game Rick Dangerous to the SDL, called XRick, which has been ported to S60; the source code is available from www.atopo.net/tool_rick.php.

Another port of the SDL to the Symbian platform is maintained by Lars Persson at anotherguest.se. Lars has ported a number of open source games to S60 and UIQ using his version of SDL, and those ports are also available from the site.

The SDL libraries are implemented in C and can be called from C or C++ code. They are provided under the terms of the GNU LGPL, which allows them to be used in both open and closed source projects. There is an abundance of SDL examples and documentation available on the main SDL website, so we won't cover it further here.

6.3 OpenKODE

OpenKODE is a royalty-free, open standard, defined by the Khronos Group, that aims to increase source portability for rich media and graphics applications. The Khronos Group is a member-funded industry consortium and the OpenKODE standard has been specifically designed for resource-constrained devices. From the perspective of a developer porting code to the Symbian platform, it's important to know that OpenKODE consists of several parts, some of which are already available in Symbian devices and some of which are currently under development.

The Khronos Group defines a number of standards⁵ for media authoring and acceleration. OpenKODE specifies a subset of the media acceleration standards along with some core operating-system abstraction APIs and an interface, called EGL, to enable the seamless interoperability of the other APIs and integration with the platform's native windowing system. Figure 6.2 shows the high-level structure of OpenKODE with its component APIs.

The OpenKODE standard is evolving. The first version, 1.0, included OpenGL ES and OpenVG. Version 1.1 also includes OpenMAX and OpenSL ES. Each of these standards is also evolving, along with the hardware that it addresses.

In future, it is likely that OpenKODE will include updated versions of the existing standards and possibly further standards, such as the embedded profile for the new OpenCL standard (which is a generic parallel computing API, allowing developers to make use of all the CPUs, GPUs and DSPs on a device for more general-purpose computing when maximum performance is required). However, sticking to those APIs that

⁵ For full details and access to the standards, see www.khronos.org.

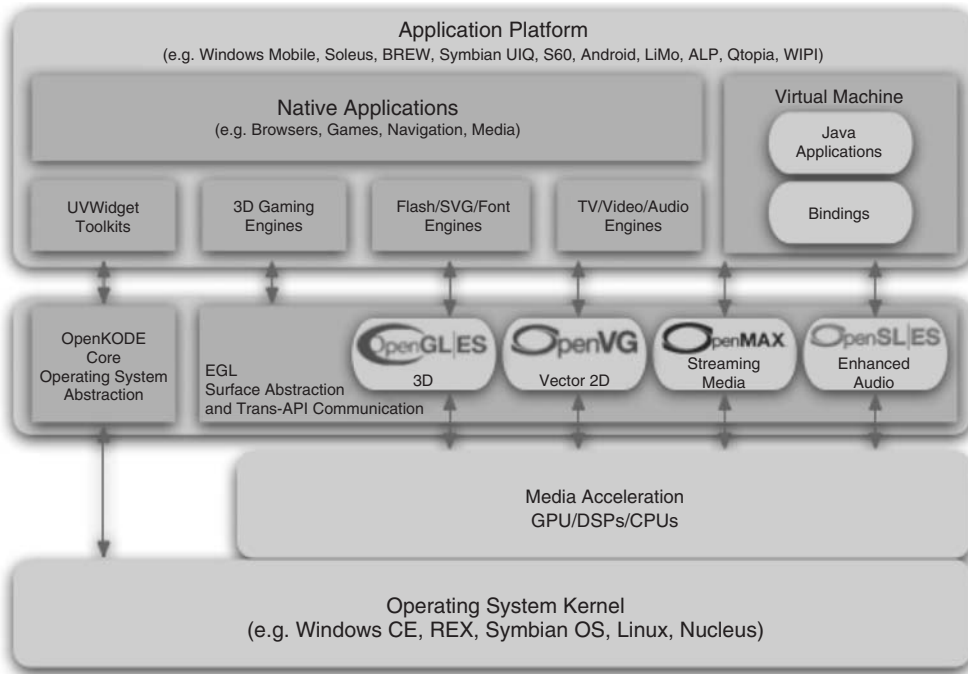


Figure 6.2 OpenKODE APIs

are available now, or coming very soon, the next sections cover the OpenKODE Core, OpenGL ES, OpenVG, OpenMAX and OpenSL ES.

6.3.1 OpenKODE Core

As you can see from Figure 6.2, the OpenKODE Core provides the operating system abstraction which includes:

- core OS libraries based on the POSIX standards
- thread creation and synchronization
- an event system
- mathematical functions – mainly standard C library analogs
- utility functions – assertions, logging, string conversion, memory allocation, random number generators, and so on
- timing functions and event-raising timers
- a virtual file system based on POSIX file I/O
- networking based on POSIX sockets

- cryptographic support
- user input and output abstractions
- windowing support.

There is a lot of overlap between these functions and the ones provided by P.I.P.S./Open C and the RGA and SDL libraries. This gives a choice of porting targets for code that's not already using these APIs. As a general principle, it makes sense to use the OpenKODE Core if the project also uses some of the other OpenKODE APIs, since it should increase portability to other platforms. That said, the functions in OpenKODE Core are generally C⁶ and POSIX equivalents with the names changed in predictable ways (a `k`d prefix and a capital letter at the beginning of the original name, so `memcpy` becomes `kdMemcpy`, for example) and the same arguments; this should make porting between OpenKODE and P.I.P.S./Open C APIs largely a matter of find-and-replace (or the OpenKODE implementation may simply `#define` the function names to refer to the relevant P.I.P.S./Open C functions). OpenKODE Core has the following advantages compared to a typical POSIX implementation:

- It is a subset of POSIX's standard C library that is chosen to be easier to implement on a range of platforms that lack support for standard C – that is, it is specifically designed to be more portable.
- It provides other portability features, such as an abstract file system, meaning that you can find your files irrespective of the layout of the host file system.
- It increases the precision of definition of some functions compared to their POSIX equivalents – there are situations where POSIX allows 'undefined behavior' while OpenKODE specifies an exact mode of failure.

The Symbian platform doesn't yet provide an implementation of OpenKODE Core but third parties, such as Ideaworks 3D,⁷ do.

6.3.2 OpenGL ES

Perhaps the most well known of the Khronos Group standards is the desktop 2D and 3D graphics API called OpenGL. The cut-down version of this API for embedded systems is OpenGL ES. Two profiles are defined for the standard: common and safety critical (known as OpenGL ES-SC).

⁶ All of the Khronos Group APIs are specified as C code APIs as this language is supported across the widest range of platforms and is also the easiest language for adding bindings to other languages.

⁷ See www.ideaworks3d.com.

Version 1.0 of the OpenGL ES standard (common profile) has been supported in Symbian OS since v8.0a. Although the API is designed to be hardware accelerated, a software implementation is provided on devices where hardware acceleration is not present. It should be noted that the software implementation of OpenGL ES is likely to be intolerably slow for all but the most simple graphics work.

Support for Open GL ES 1.1 is included in S60 3rd Edition FP1 devices by default and there is an upgrade plug-in for original S60 3rd Edition (and UIQ 3) devices. In order to use OpenGL ES, simply include the relevant headers and library in your source or header files:

```
#include <GLES/gl.h>
```

or

```
#include <GLES/egl.h> // this includes gl.h internally
```

Then in the MMP file add:

```
LIBRARY libgles_cm.lib
```

There are a large number of examples provided within the S60 SDKs.

Projects using OpenGL ES should port from one platform to another very easily and projects using only the basic features of OpenGL should also port to OpenGL ES with very little change. However, there is one large caveat and that is floating-point arithmetic. At the time of writing, the majority of Symbian devices don't have 3D graphics acceleration hardware or floating-point units. Attempts to use floating-point operations on these devices will result in very slow and power-intensive software emulation. Even where 3D hardware and a floating-point unit are present, the device may still default to emulating standard floating-point calculations, used in game physics, for example. Many of the 3D hardware accelerators in current phones also use only fixed-point arithmetic. This leads to two sensible options for porting projects to OpenGL ES on mobile devices:

- Target only devices in which both the 3D hardware and the CPU supports floating-point arithmetic, and optimize for that hardware. This gives a very restricted but growing target market and gets good performance from high-end hardware with limited porting effort.
- Convert all floating-point values in arithmetic and OpenGL function calls to fixed-point values. You'll still only be able to target devices with 3D hardware acceleration for high-performance graphics, but the CPU need not have a floating-point geometry processor.

If you target all devices with 3D hardware (not just those that are floating-point-enabled), you'll want to convert everything to fixed point, including floating-point values used outside the graphics code (e.g. for a physics engine). If you'd like your application to run on devices without 3D hardware then you need to simplify the graphics as well and consider using a more limited software renderer than the OpenGL ES implementation.

The solution for processors without floating-point support is to do decimal operations with integers by fixing the location of the decimal point and using some of the bits in the integer value to represent the fractional part of a number. In OpenGL ES this is always done with 32-bit values (this is the word size on the ARM processors used in mobile devices) split in the middle, using 16 bits for the integral part and 16 for the fractional part. In order to be able to represent negative numbers, two's complement is used for the integral part, meaning that the most significant bit is actually a sign bit. This is known as an S15.16 fixed-point representation (where the 'S' indicates the sign bit).

So, how does this work? In fixed-point arithmetic, all the digits to the right of the 'decimal' point are multiplied by negative exponents of the base. If that sounds a little too mathematical and you can feel a headache coming on at the thought of it, then consider this simple example. Here we represent the same number in base 10 (decimal) and base 2 (binary):

$$\begin{aligned}
 (10.8125)_{10} &= 1 * 10^1 + 0 * 10^0 + 8 * 10^{-1} + 1 * 10^{-2} + 2 * 10^{-3} + 5 * 10^{-4} \\
 &= 1 \text{ ten} + 0 \text{ ones} + 8 \text{ tenths} + 1 \text{ hundredth} + 2 \text{ thousandths} + 5 \text{ ten-thousandths} \\
 &= (1010.1101)_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} \\
 &= 1 \text{ eight} + 0 \text{ fours} + 1 \text{ two} + 0 \text{ ones} + 1 \text{ half} + 1 \text{ quarter} + 0 \text{ eighths} + 1 \text{ sixteenth}
 \end{aligned}$$

Since this is just a binary number, we can do bit operations on it, such as shifting. In this example, if we bit shift left by four we get rid of the fractional part and have just 10101101, which is equal to 173. As we have a fixed number of bits after the decimal point, we know that if we shift left by 16 bits then we will always get an integer. It turns out that we can fairly easily do all the standard arithmetic operations on fixed-point numbers by combinations of shift operations and standard integer arithmetic. The details get a little more complex and there is always a trade-off between precision and speed. For example, here are two ways you could do a multiplication on two S15.16 fixed point numbers stored in two 32-bit integers, *x* and *y*, (i.e. effectively pre-shifted left 16 bits each) and store the result as a similar fixed-point value in a 32-bit integer, *z*:⁸

⁸ Note that `long long` is not necessarily portable; an appropriate typedef, such as `int64` or `TInt64`, would be preferable.

```
z = (int) (((long long) x * (long long) y) >> 16);
```

or

```
z = x>>8 * y>>8;
```

The first option uses 64-bit arithmetic on the processor. In the case of current devices, this must also be emulated using multiple 32-bit operations (although it is much faster than doing floating-point calculations). However, the result retains the full precision of the original values. The second option removes 8 bits of precision from the values before they are multiplied but is significantly faster as it requires only standard 32-bit integer operations. The danger in both cases is that it's easy to have an overflow, where the result is too large to be expressed in the available bits. This kind of overflow happens silently so your math routines need debugging code to check for these errors. This is beyond the scope of this book.⁹

The optimization of fixed-point arithmetic can be critical to the performance of 3D games and it's worth taking some time to understand the issues involved, particularly concerning the operations available in the ARM instruction set (as opposed to the smaller Thumb instruction set that is used by default on the Symbian platform).¹⁰ Adequate performance can usually be achieved with inline functions and a good optimizing compiler, but on occasion it may be desirable to hand code complex calculations in assembler.

Looking to the future, OpenGL ES 2.0 supports custom shaders via an embedded version of the OpenGL Shading Language known as OpenGL ES SL. This should allow easier porting of code that takes advantage of the equivalent features in OpenGL 2.0. At the time of writing, this is supported in only a small fraction of mobile devices at the high end of the market but it has been the subject of some very impressive demonstrations at recent industry events.

6.3.3 OpenVG

The OpenVG API is designed to meet the growing need for high-quality vector graphics on devices with small screens and low power budgets. Like OpenGL ES, the rendering can be implemented in software but can also take advantage of hardware acceleration where it is available. An efficient hardware accelerator can reduce power consumption by up to 90% compared to a software rendering engine. The scalable nature of

⁹ Debugging code and the optimization of such operations is covered very well in Appendix A of Pulli, K. *et al.* (2008) *Mobile 3D Graphics with OpenGL ES and M3G*, Morgan Kaufmann.

¹⁰ You can force the build system to use the ARM instruction set by using the ALWAYS_BUILD_AS_ARM keyword in your MMP file.

vector graphics is perfectly suited for the mobile domain where it can enable a single application and set of graphics to target a range of different screen sizes. From a porting perspective, this is the most important aspect of OpenVG: for many types of application, porting graphical content to different platforms can be a large part of the cost. Since the OpenVG standard is not yet widely used it is unlikely that you will have OpenVG code to port but the API should be an appealing target for porting the following types of project:

- viewers – Flash, SVG, PDF and Postscript
- mapping applications – using vector maps
- e-book readers – for fast, low-power text rendering
- games – for defining sprites, backgrounds and textures (possibly in conjunction with OpenGL ES)
- scalable user interfaces – required for most mobile applications
- graphical toolkits – creating a higher level of abstraction on top of which to build scalable applications.

The OpenVG syntax has been designed to be immediately familiar for OpenGL ES programmers in order to ease the learning curve and make it simple to combine the two APIs. Both APIs use EGL to acquire graphics contexts and surfaces for rendering.

The current version of the OpenVG specification is 1.0.1 and is supported from Symbian³ onwards, although this is not used in any device at the time of writing.

6.3.4 OpenMAX

OpenMAX is designed as a complete solution for accelerated media playback and streaming media functionality. The standard consists of three layers. The lowest layer is the ‘development layer’ (OpenMAX DL), which defines a common set of functions to enable porting of codecs across hardware platforms. Generally, a silicon vendor would implement and optimize these functions for its specific hardware. So, from a porting perspective, this an ideal target API for codecs.

The next layer up is the ‘integration layer’ (OpenMAX IL), which serves as a low-level interface to the codecs to enable porting of media libraries across operating systems. This layer would be expected as part of the operating system and indeed Symbian has added support for OpenMAX IL audio from Symbian³ onwards. On future releases of the Symbian platform, OpenMAX IL may be used not only for audio but also for other multimedia codecs (e.g. video) and other processing units, such as sources, sinks, audio processing effects, mixers, and so on.

However, just because support for the standard is provided by the platform, it doesn't mean that it will be adopted immediately by device manufacturers.

The highest layer of abstraction is the 'application layer' (OpenMAX AL). This defines a set of APIs providing a standardized interface between an application and multimedia middleware. This specification was expected to be finalized by the end of 2008, although it still was not final at the time of writing. Symbian plans to support this API once the specification is complete.

6.3.5 OpenSL ES

OpenSL ES is the sister standard of OpenGL ES in the audio arena. It has been designed to minimize fragmentation of audio APIs between proprietary implementations and to provide a standard way to access audio hardware acceleration for application developers. OpenMAX AL and OpenSL ES overlap in their support for audio playback, recording and MIDI functionality. A device can choose to support OpenMAX AL and OpenSL ES together or just one of them. The standard supports a large set of features, but these features have been grouped into three 'profiles': phone, music and game. Any device can choose to support one or more of these profiles depending on its target market. OpenSL ES also supports vendor-specific extensions to add more functionality to the standard feature set of a profile.

The OpenSL ES API is identical to OpenMAX AL, but adds support for more objects (for example, listener, player and 3D Groups) and interfaces (for example, 3DLocation, 3DPlayer, 3DDoppler and 3DMacroscopic) for 3D audio support. As with OpenMAX AL, the OpenSL ES standard was expected to be finalized by the end of 2008 and Symbian have announced their intention to implement it when it is available.

6.3.6 The Future of OpenKODE

The Khronos Group is continuing to provide industry-wide standards to help solve the problems of source code portability, not just for mobile devices but also other embedded systems such as games consoles (the Sony PlayStation 3 already includes an OpenMAX implementation), set-top boxes, DVD players, portable media players and navigation devices. Although the standard is targeted at the embedded market there is an open source implementation in progress that also supports desktop platforms called FreeKODE (hosted at sourceforge.net/projects/freekode). However, apart from OpenGL ES, the standards are still more of a future solution to portability issues from an application developer's perspective. So next we look at something you can start using now to develop cross-platform applications.

6.4 Qt

Qt, pronounced ‘cute’, is a cross-platform application framework that was created by the Norwegian company Qt Development Frameworks, formerly known as Trolltech, now a wholly owned subsidiary of Nokia. The purchase of Trolltech by Nokia and subsequent porting of Qt to S60 is an important and exciting development for cross-platform mobile software and porting. As I hinted in the introduction, Qt is potentially the first truly cross-platform application framework for native mobile device development. There are already versions of Qt for mobile Linux platforms, Windows Mobile and Symbian/S60. As Nokia’s publicly stated strategy is to compete in the Internet services space, particularly on mobile platforms, it seems extremely likely that any mobile device platform that allows native application installation and gains a significant market share will have Qt ported to it in the future. This is just my personal opinion, with no evidence to back it but the opportunity to ride Nokia’s coattails to get application portability seems like a real one.

In addition to the support of the world’s largest mobile device manufacturer, Qt has other important benefits. First, and perhaps most important, is that the framework is designed to make it generally quick and easy to perform common operations with minimal new code. However, the design still makes it possible to modify the behavior of almost every object in fine detail if required.

Most C++ (or Java) developers learning about the various Qt classes for the first time will have several ‘why doesn’t everyone do it that way?’ moments. A few minutes with your favorite search engine will turn up hundreds of favorable comparisons to other frameworks made by independent developers. The framework is also well documented;¹¹ this is likely to be an enormous relief to existing S60 developers.

Another benefit of Qt is that the portability extends across all of the major desktop platforms. Qt is the main application framework for one of the most popular Linux desktops, KDE, and it is also supported on distributions using the rival GNOME desktop. Microsoft’s Windows platforms and Mac OS are also well supported. Qt is well proven in this cross-platform environment, being the basis for successful applications such as the Opera web browser, Google Earth, Skype and Adobe Photoshop Elements. My final point about Qt, before we get into the details, is that it has grown from a simple UI toolkit into a very comprehensive framework and developer tools. As Figure 6.3 shows, the framework is highly modular.

In the following sections, we briefly describe each of the modules in turn. For more complete documentation, please explore the resources

¹¹ Find the documentation at doc.qt.nokia.com.

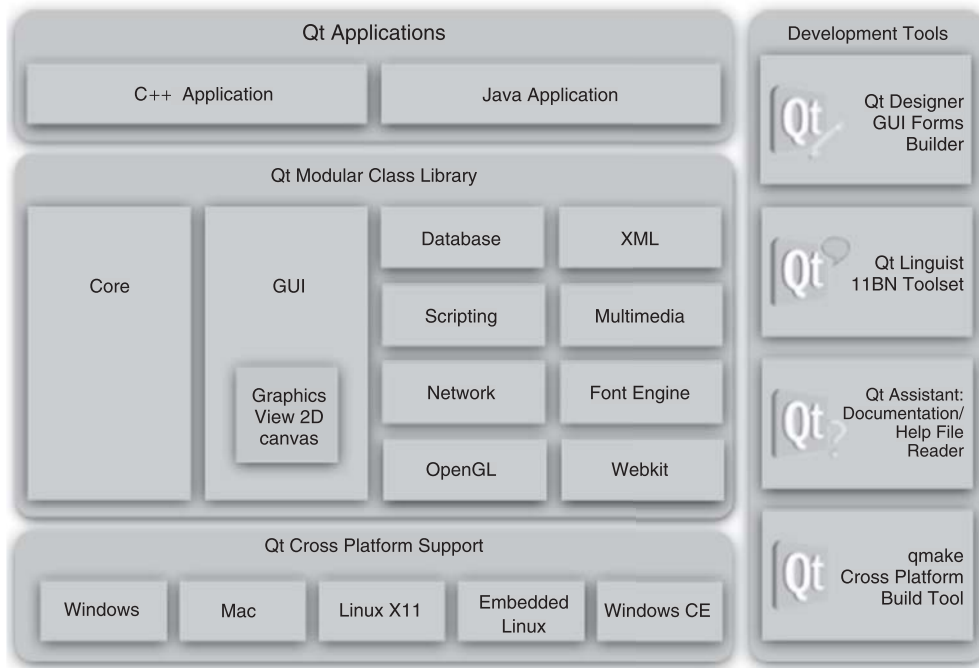


Figure 6.3 Qt architecture and tools

provided at www.forum.nokia.com/Resources_and_Information/Tools/Runtimes/Qt_for_S60. There is also an official book, although it doesn't discuss mobile Qt development for Symbian.¹²

6.4.1 QtCore

The QtCore module contains the 'core' non-GUI functionality and is the base on which all of the other Qt modules depend. It provides the main operating system abstraction layer and other useful general-purpose classes, including:

- a base object class with built-in support for event-driven programming and a seamless object communication mechanism via signals and slots
- strings, arrays, characters, linked lists, stacks and queues
- atomic operations on integers and pointers
- guarded pointers (automatically zeroed when the referenced object is destroyed)

¹² Blanchette, J. and Summerfield, M. (2008) *C++ GUI Programming with Qt 4*, 2nd Edition, Prentice Hall.

- date and time access and timers
- files, streams and file system management
- containers – pairs, vectors, maps, hashes and sets
- threads, processes, mutexes, semaphores and shared memory
- exceptions that can be transferred between threads
- text handling and internationalization support
- platform-independent application settings
- pattern matching using regular expressions
- convenience functions for handling URLs and XML
- support for loading libraries, plug-ins and resources
- basic geometry via points, lines, sizes and rectangles

There are many useful features of these classes but there are a couple that are worth pointing out to C/C++ programmers that may not be familiar with them. There is a ‘signals and slots’ mechanism available to all objects (assuming they are derived from `QObject` or one of its subclasses). As part of a class definition, it’s possible to declare signals that the class emits and slots that are the targets for signals. For example:

```
#include <QObject>
class Cell : public QObject
{
    Q_OBJECT // This macro is required for all classes
             // that implement signals and slots
public:
    Cell() { iValue = 0; }
    int value() const { return iValue; }
public slots:
    void setValue(int value);
signals:
    void valueChanged(int newValue);
private:
    int iValue;
};
```

These can be used in a similar way to pointers to callback functions in C or observer interface classes in C++ but they are superior in several ways. Any number of signals can be connected to a slot and a single signal can be sent to an arbitrary number of slots. This helps to decouple classes from one another; slots don’t know what signals they are connected to and signals don’t care if they are connected to any slots. A signal can also be connected to another signal so that the emission of the first automatically causes the second to be emitted. Connecting a signal, `send(int)`, on

an object of class `Sender` to a slot, `receive(int)`, on an object of class `Receiver` is as simple as:

```
Sender sender;
Receiver receiver;
QObject::connect(&sender, SIGNAL(send(int)),
                &receiver, SLOT(receive(int)));
```

Unlike callback functions, the signals and slots mechanism is also type safe. This mechanism is implemented as an extension to the standard C++ syntax through the use of the preprocessor and the Meta-Object Compiler,¹³ which change or remove `signals`, `slots` and `emit` keywords in the original code and generate a new source file that implements the necessary communication code. The slots must be implemented by the application programmer; they are just normal methods, but the signal functions are generated by the Meta-Object Compiler. Emitting a signal is almost as simple as calling a method; here is a possible implementation of the `Cell::setValue()` slot defined in the class above:

```
void Cell::setValue(int value)
{
    if (value != iValue)
    {
        iValue = value;
        emit valueChanged(value);
    }
}
```

The additional benefit of this approach is that it allows for very clear and readable code to implement object communication and the only downside is a slight performance penalty compared to other popular approaches. However, thanks to extensive optimization of the generated code, the performance penalty is relatively low when compared with the typical functionality activated by a signal. Note that the mechanism is extremely flexible; `QObject` instances can be used in multiple threads and emit signals that invoke slots in other threads.

The second feature you're likely to find unfamiliar, particularly if you're coming from a Symbian C++ background, is 'implicit sharing'. Many of the classes in `QtCore` and some in the other modules implicitly share any data they own to maximize resource usage and minimize copying. The string class, `QString`, is the most notable example of this. What this means in practice is that these objects can safely be declared as automatic types (i.e. instantiated on the stack) and they allocate the

¹³ The Meta-Object Compiler also facilitates Qt's run-time type information and dynamic property systems. See doc.qt.nokia.com/4.4/moc.html for more details.

required memory from the heap internally. When a copy of an object is taken, the class simply performs a shallow copy and increments a reference count on the shared data. The data itself is only duplicated when one of the shared copies needs to modify it.¹⁴ This allows safe and efficient passing of these classes by value, returning them from functions and simple assignment. Such classes can even be shared safely between threads as Qt uses atomic reference counting. Again, the trade-off here is a small performance penalty compared to using constant references where you know that the data won't be modified¹⁵ against a gain in simplicity of expression and decoupling of components.

6.4.2 QtGui

The QtGui module extends QtCore with graphical user interface functionality. At the heart of the graphics system is the `QPainter` class which provides the interfaces you'd expect to find on a graphics context in most other UI toolkits (although a lot of the associated functionality is actually implemented in `QPaintEngine` subclasses, these are hidden from the application programmer). The usual array of pens, fonts, images and brushes are available for drawing, as well as `QPainterPath`, which enables the construction and reuse of custom shapes. A `QPainter` object can be used to draw on any `QPaintDevice` subclass; this includes the base widget class, `QWidget`, and hence all widgets. Widgets in Qt (and many other UI toolkits) are equivalent to controls in the Symbian platform and should not be confused with the web-centric versions from Yahoo!, Google, Apple, and so on.

Qt provides an extensive set of widgets that can be easily derived from or themed with style sheets (similar to CSS) if customization is required. The following list of widgets is far from exhaustive:

- buttons, sliders and combo-boxes
- windows, dialogs, forms and frames
- scroll areas with scroll bars
- menus, tabs, toolbars and status bars
- splash screens and progress bars
- help search query and result
- single and multi-line text editors
- list, table and tree views
- calendar.

¹⁴ Full details are available at doc.qt.nokia.com/4.4/shared.html.

¹⁵ Of course there's nothing to stop you using constant references, rather than passing, returning and assigning by value, if you're sure that's what you want to do.

All of the widgets can be combined in flexible layouts (vertical, horizontal and grid) that can in turn be used in arbitrary combinations along with spacers to optimize positioning. The QtGui module also adds a range of UI-specific events to support actions, such as drag and drop or key input. Finally, the entire framework can use a range of styles, giving a different look and feel on different platforms. The usual Qt approach is to emulate the native style on a platform by default and indeed this is the case for S60, where Qt applications should be almost indistinguishable from native applications.

6.4.3 QtNetwork

The QtNetwork module is designed to make network programming both portable and simple. It covers sockets for low-level data transport protocols, application-level protocols and even a complete network access API, providing an abstraction layer over the specific operations and protocols used. There are classes for:

- UDP, TCP, SSL and local sockets
- key and certificate handling for secure communications
- HTTP and FTP protocol implementations
- host address lookup by name
- a managed request-and-response mechanism
- proxies, authentication and cookie management
- implementing local and remotely accessible (TCP) servers

Network operations may be performed synchronously or asynchronously at the application level.

6.4.4 QtOpenGL

The QtOpenGL module allows developers to combine a Qt user interface with 3D graphics rendered using OpenGL. On mobile device platforms, this module is expected to support the embedded version of the standard, OpenGL ES (see Section 6.3.2). This is already the case with Qt for Embedded Linux but at the time of writing the QtOpenGL module has not yet been ported to S60, so the exact details of available functionality cannot be confirmed.

6.4.5 QtScript

The QtScript module enables developers to make their applications scriptable. Qt Script itself is based on the ECMAScript language, which

in turn is the standard on which Netscape's JavaScript and Microsoft's JScript are based.¹⁶

The central class in the QtScript module is `QScriptEngine`, which can be used to evaluate scripts that are typed by the user, loaded from disk or downloaded from the network. The extra features provided by Qt's Meta-Object Compiler enable Qt Script to interact with `QObject` instances. Any subclass of `QObject` can be made available to script code by passing it to the `QScriptEngine`. It is then possible to query and modify dynamic object properties from the script and also to mix signals and slots between C++ and script. That is, C++ application code can connect a signal to a script function, a script can connect the signals and slots of C++ objects dynamically, and a script can also define its own signal handlers (effectively slots) and set up the connections that utilize those handlers. The only restriction is that scripts are currently not able to define new signals, although they can emit a signal that is defined by a C++ class.

6.4.6 QSql

The `QSql` module enables seamless database integration with applications. The architecture has three layers:

- The driver layer abstracts the details of the various database types that are supported, providing a bridge to the SQL API layer. On the Symbian platform, the extremely popular open source SQLite¹⁷ database is supported as the back end.
- The SQL API layer provides access to databases, allowing developers to make connections and query databases using the structured query language.
- The UI layer links data from the database to data-aware widgets. This is achieved via model classes which can be combined with view classes from the `QtGui` module as part of Qt's model-view architecture.

6.4.7 QtSvg

The `QtSvg` module is provided for displaying scalable vector graphics (SVG). The SVG specifications define a way of describing 2D static and animated graphics using XML. There are mobile SVG profiles, SVG Basic and SVG Tiny, aimed at resource-limited devices. The profile widely supported on high-end mobile devices is SVG Tiny and the `QtSvg` module currently supports only the static features in version 1.2 of this standard. As one of the primary targets of the OpenVG standard

¹⁶ See www.ecmascript.org for details. In fact, Netscape's JavaScript preceded ECMAScript and the language was presented to Ecma International for standardization.

¹⁷ See www.sqlite.org.

(discussed in Section 6.3.3) is the accelerated rendering of SVG graphics and animations for mobile devices, it seems likely that this support will be expanded in future versions of the Qt framework. In the current version, you can very simply display the contents of an SVG file by creating a `QSvgWidget` and loading the file with it.

6.4.8 QtWebKit

One of the features added to Qt 4.4 was the QtWebKit module. This module integrates the WebKit browser engine¹⁸ with Qt. WebKit is almost standard on mobile devices and S60 had a WebKit-based browser even before Nokia's acquisition of Trolltech. At the time of writing, the QtWebKit module was not available in Qt for S60 and the extent of any functionality to be provided in the first official release was not yet known.

The main class in the QtWebKit module is the `QWebView` widget. It can be embedded into forms and other views, providing complete functionality for downloading and rendering web sites. In fact, using `QWebView`, a website can be downloaded and displayed in just three lines of code:

```
QWebView *view = new QWebView(parent);
view->load(QUrl("http://www.qt.nokia.com/"));
view->show();
```

This class has a very simple interface which allows developers to integrate web-browsing functionality into their applications with relative ease. However, the source of the really impressive demonstrations that have been shown at various developer conferences is integration going the other way. QtWebKit provides a bridge between the JavaScript engine in WebKit and `QObject` instances, very much like the functionality provided by the QtScript module (Section 6.4.5). Qt widgets can be rendered in web pages and animated with JavaScript. This offers the possibility of creating very rich clients for Internet-based applications (or even more of a hybrid between current web widgets and native applications).

6.4.9 QtXml and QtXmlPatterns

The QtXml module provides utility classes for reading and writing XML. There are very simple stream reader and writer classes for XML documents as well as C++ implementations of parsers based on Simple API for XML (SAX)¹⁹ and Document Object Model (DOM).²⁰

¹⁸ The WebKit project is hosted at webkit.org.

¹⁹ See the official website at www.saxproject.org.

²⁰ See the specification at www.w3.org/DOM.

The QtXmlPatterns module adds support for the XQuery/XPath language, which enables the traversing of XML documents to select and aggregate items of interest and to transform them for output.

6.4.10 Phonon

Another new feature in Qt 4.4 was the cross-platform multimedia framework called Phonon. This framework started as part of the KDE project and was subsequently adopted by Trolltech to become part of Qt (presumably the reason for it not having a Qt-prefixed module name).

The term ‘multimedia framework’ is slightly misleading here as Phonon is really a common interface to various separate back ends which provide the multimedia functionality on each supported platform. For example, there is a GStreamer back end on Linux, a DirectX back end on Windows and a QuickTime back end on Mac OS. A back end for Symbian using the Helix client from RealNetworks²¹ is in development but has not been released at the time of writing. Longer term, a fairly obvious target for a back end on the Symbian platform would be OpenMAX (see Section 6.3.4) as this should also be usable on several other embedded platforms. The early versions of Phonon support only simple playback and streaming of audio and video formats, plus some audio effects. There are plans for future versions to support media capture, audio mixing and more complex processors for audio and video effects.

6.4.11 Platform-Specific Extensions

However good Qt is, there will probably always be things that some application developers want to do which the framework doesn’t provide yet, or are only applicable on certain platforms. In this situation, platform-specific extensions are necessary. Nokia is developing a range of extensions to access other functionality via a Qt-style interface. At the time of writing, available extensions include:

- sensors (e.g. accelerometer)
- location
- Internet access point management
- messaging
- contacts
- telephony
- camera
- system information

²¹ This is also an open source project; see symbian.helixcommunity.org for details.

- profiles
- calendar.

There are also plans to provide many more so that most use cases can be covered without the need to write any Symbian C++. Clearly, some of these APIs will be relevant for other platforms and will, I hope, evolve into fully cross-platform Qt APIs at some point in the future.²² It seems quite likely that S60 will become something of a test-bed for new Qt APIs, particularly for mobile devices.

6.4.12 Using Qt

Qt has historically been dual-licensed, giving developers a choice of either the GPL or a commercial license. This policy made the libraries free for free software but developers had to pay if they wanted to keep their source closed. There is also a GPL exception²³ that allows a commercial licensing alternative and code developed under most common open source licenses to link to the open source version of the Qt libraries, including licenses that are incompatible with the GPL.

On 14th January 2009, Nokia announced that from Qt version 4.5 onwards, the framework would be available for all platforms under the GNU LGPL. This effectively makes Qt free for all open and closed source development, as long as developers are able to comply with the terms of the LGPL v2.1. The license is fairly permissive so this should not be a problem, unless developers want to modify the Qt source code and not release the source code for their changes or they need to link to the libraries statically. Certainly on S60, with the Qt libraries expected to ship in device firmware, everyone is likely to link dynamically to unmodified libraries. Along with the licensing change, the announcement also indicated that the development model for Qt would be opening up, making it easier for third parties to contribute and, we hope, accelerating the development of the framework.²⁴

Since Qt is a cross-platform framework, it has its own platform-independent project files and build tools that are used to generate the necessary platform-specific build system inputs. For Symbian/S60 the `blt.inf` and MMP files are automatically generated along with an extension makefile that invokes the other Qt build tools, such as the Meta-Object Compiler. In this case, the standard Symbian build files should not be edited and any changes should be made via the platform-independent project files. Further details of this can be found in the example Qt application ports in Chapters 10 and 12. Some more information about

²² See the Qt Mobility project at labs.qtsoftware.com/page/Projects/QtMobility for the first steps in this direction.

²³ See doc.qtsoftware.com/4.4/license-gpl-exceptions.html for details.

²⁴ See qt.gitorious.org.

the port of Qt to S60, as an example of how to port middleware, is available in Chapter 11.

6.5 Summary

We have looked at APIs that make it easier to port code to the Symbian platform:

- real-time graphics and audio (RGA) libraries
- Simple DirectMedia Layer (SDL)
- OpenKODE
- Qt.

In Chapters 7–9, we take a closer look at porting to the Symbian platform from specific other mobile device platforms.

7

Porting from Mobile Linux

How should I know if it works? That's what beta testers are for. I only coded it.

Linus Torvalds

In recent years, we have seen a massive expansion in Linux-related efforts – its presence in the desktop, laptop, netbook and mobile space has steadily and progressively grown. This is not a big surprise. Linux is a free, open source, highly customizable and mature platform. The healthy pool of developers with experience, knowledge and enthusiasm to develop for the platform is equally important.

Linux is typically offered to end users in what are called ‘distributions’. Typically, distributions offer a similar set of core packages and distinguish themselves in providing installation support, customizations or branding. Some distributions focus on specific submarkets, such as servers, media centers, or headless, diskless or embedded systems. There are hundreds of distributions available, many more than necessary, however this segmentation has provided a crucial set of concepts and tools, allowing Linux to expand in many directions, including the mobile space.

One of the challenges in the Linux space is that there are usually several – often many – competing solutions for a specific problem. This is especially interesting from the point of view of porting to the Symbian platform; the amount of required effort is tightly related to which underlying libraries and frameworks are used by the application being ported. If all the libraries are available, porting may be fairly straightforward. In many cases, however, this is not the case.

Porting from Linux to Symbian may sound like a daunting task, especially if you are new to the Symbian platform. However the Symbian platform has taken major steps towards compatibility in recent years with more to come. One extraordinary porting success story proves that there is nothing to fear: Personal Apache–MySQL–PHP (known as Personal

AMP or PAMP)¹ has been successfully ported to Symbian and is now available on SourceForge.

The general approach is to make use of available libraries and, when libraries are not available, either to port them or rewrite the code to use native Symbian APIs. As discussed in previous chapters, S60 already offers basic POSIX compatibility through Open C/C++, and there are several library ports available, most notably Qt,² OpenKODE³ and the SDL⁴ multimedia framework.

In the remainder of this chapter, we first discuss some well-known Linux distributions that have a specific focus on the mobile phone market. Then we cover several core libraries used in mobile Linux distributions and their portability and availability for porting applications to the Symbian platform. This is split into three sections: basic support, covering core libraries, threading, IPC and similar topics; user interface support; and feature support, where we discuss multimedia, connectivity and similar topics.

7.1 Major Players in the Mobile Linux Space

Mobile Linux is a somewhat divergent effort. There are many companies, open source groups and distributions entering this space. The end result is a fragmented market for developers. In addition, this is a fairly new market and new entries are regular and expected for the foreseeable future. Further still, existing distributions are in constant flux with many variations.

In order to get at least the basics right, the Linux Foundation has set up the Mobile Linux Workgroup. The Workgroup has been around for several years now, but has a somewhat limited influence in the market. It has produced several platform guidelines addressing security, file systems, tools, performance and virtualization. The guideline documents are useful for device creators and define the functionality that should be present in devices. However, the Workgroup has no ambition to set up or prescribe standard frameworks and APIs and therefore has little impact on our porting discussion.

A common theme in mobile Linux distributions is reuse of the kernel and some middleware while providing a custom set of APIs for mobile-specific tasks. Different distributions draw the separation lines in different places. Use of different middleware libraries and the presence of distribution-specific APIs often force some design decisions that facilitate application portability. Clearly, if an application takes this approach, it can

¹ See wiki.opensource.nokia.com/projects/PAMP.

² See www.qt.nokia.com/developer/technical-preview-qt-for-s60.

³ See www.khronos.org/openkode.

⁴ See koti.mbnet.fi/mertama/sdl.html.

be an asset when porting to the Symbian platform – well designed applications are easier to port because platform incompatibilities can be dealt with in isolation and the ‘meat’ of the application can escape refactoring.

Let us now examine the current Linux distributions for mobile devices. As discussed, this market is moving fast; however the real players are already established. Table 7.1 summarizes the features of the main Linux platforms.

Table 7.1 Linux platforms

	Maemo	Moblin	Openmoko
Base distribution	Debian	Fedora	Various
User interface	Hildon, Gnome, GTK, Matchbox	Clutter, Qt, GTK+	Qtopia, GTK+
C services	Glib, GObject	Glib, GObject	–
IPC/threading	DBus	DBus	DBus
Media	GStreamer, ALSA, Video4 Linux	GStreamer, Helix, PulseAudio	GStreamer, ALSA
PIM	Evolution	Evolution	–
Telephony	–	Telepathy	–
Connectivity	Telepathy BlueZ	BlueZ	BlueZ
Messaging	Email client	–	–
Location	Gpsd	Gipsy, GeoClue	–
Networking	ConlC, curl Samba, wland	ConnMan	ConnMan
Database	SQLite	–	–
Security	OpenSSL	–	–

7.1.1 Maemo

The Maemo platform is an open source, Linux-based, mobile information device (MID) platform. Its architecture is shown in Figure 7.1. There are several Maemo devices currently in the market, notably Nokia’s Internet tablets – N770, N800 and N810 with several further devices in the pipeline at the time of writing. None of the devices released so far are phones, however the platform is maturing with extensive UI improvements in Maemo 5 and it seems to be only a matter of time before Maemo reaches phones. On the other hand, current Maemo devices are quite powerful – the tiny N800 tablet is capable of running Android as a VMware virtual machine.

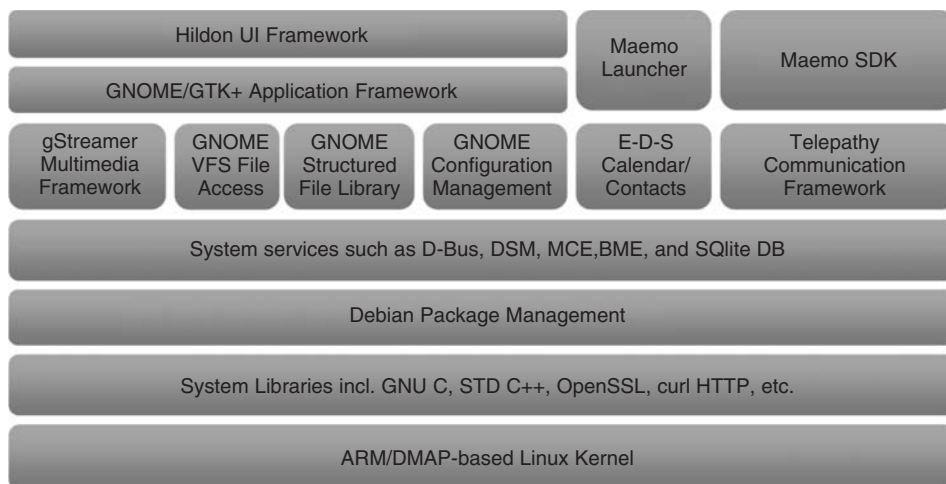


Figure 7.1 Maemo architecture

Maemo 5 includes significant advances in the UI and graphics components, but also adds a Location API, screen orientation control and a host of other features. Maemo 5 is currently available in alpha and a full release is expected during 2009.

7.1.2 Openmoko

Openmoko⁵ is an open source, Linux-based platform (see Figure 7.2 for an architecture diagram) specifically focused on selling the ‘hackable’ Neo range of phones. The idea behind the Neo is to develop open source hardware, as well as software. As of March 2009, there are at least 10 different distributions (including Android, Debian, Gentoo, etc.) that can run on Neo phones, as well as a variety of Openmoko software distributions.

Openmoko has succeeded in producing a good base of packages covering the main functionality we have come to expect from mobile phones and to develop, produce and market a phone based on their software stack. The Openmoko community appears to be very active and development seems to progress rapidly. The challenge is the extremely small production volumes of Neo phones – success is unlikely without entering the mass market.

7.1.3 Moblin

Moblin⁶ is an open source (GPL) project targeting mobile information devices (MIDs). Moblin has strong support from Intel and recently

⁵ See www.openmoko.org.

⁶ See moblin.org.

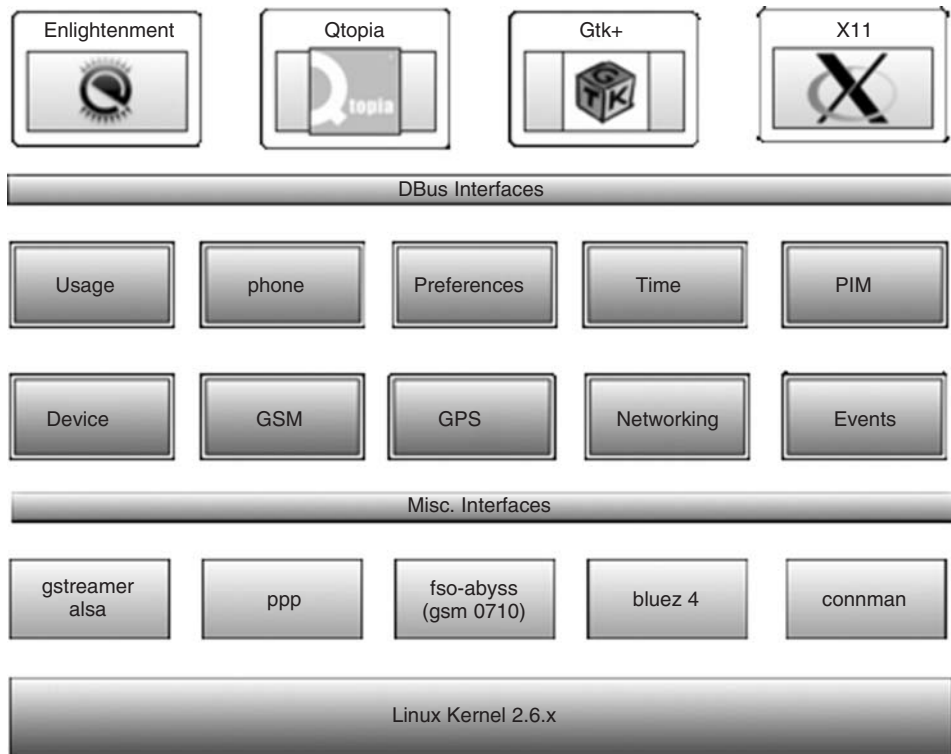


Figure 7.2 Openmoko architecture diagram

from the Linux Foundation. The architecture of Moblin is shown in Figure 7.3.

As of April 2009, a single Moblin device has been announced – GigaByte M528. The M528 utilizes powerful hardware: the Intel Atom processor (x86), an 800x480 screen and 512 Mb of RAM which puts it in its own MID category, possibly competing with netbooks rather than phones. This could change quickly however.

7.1.4 LiMo

The LiMo Foundation⁷ is an industry consortium for delivering hardware-independent mobile Linux. It is quite different from Openmoko and Moblin in that it has backing – including financing – from major worldwide industry players. LiMo aims to produce a new, clean set of APIs, specifically for creating Linux phones, in a way that mimics the Symbian platform in many architectural and operational aspects. The architecture of the LiMo platform is shown in Figure 7.4.

⁷ See www.limofoundation.org.

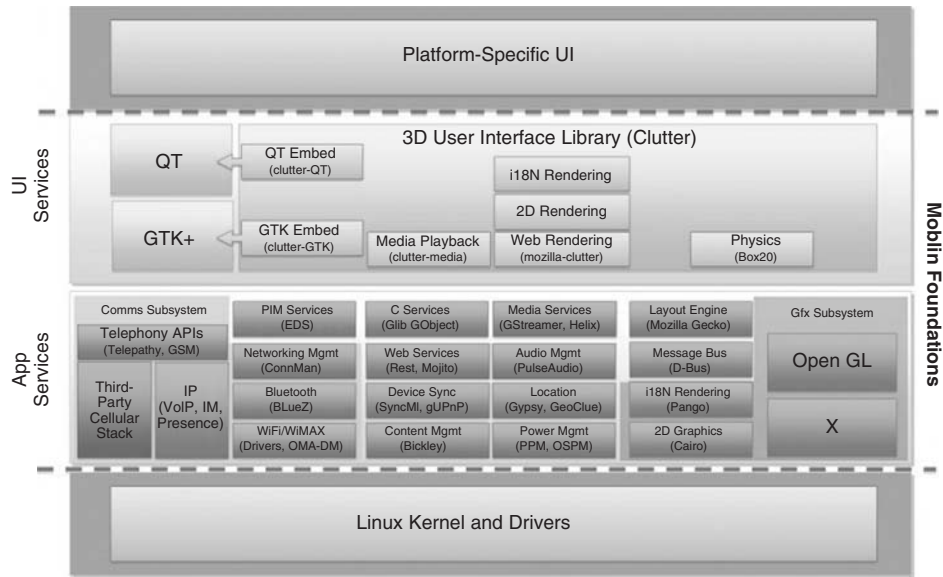


Figure 7.3 Moblin architecture

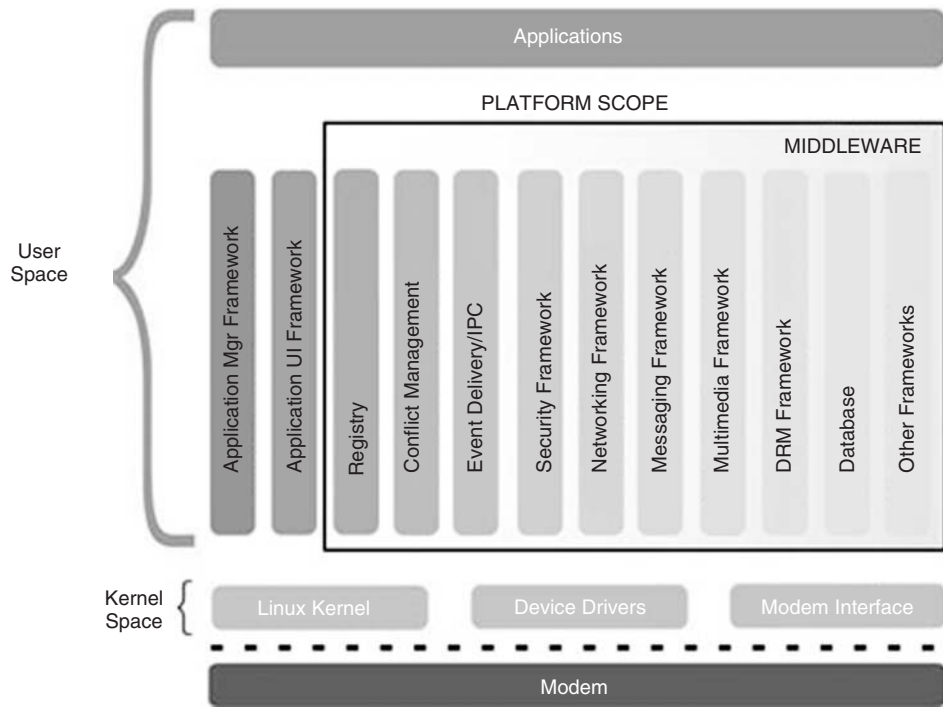


Figure 7.4 LiMo architecture

Because LiMo defines its own APIs, porting from LiMo can be challenging for any target platform, including other Linux platforms. LiMo also includes DRM, which can add a layer of complexity to the platform.

The main strengths of LiMo are tied-down, non-divergent APIs and a single distribution model. In the device production world, this significantly reduces costs. However, the way this appears to be implemented could also be its main weakness – lack of compatibility with anything else on the market.

Even if the brand doesn't have the high visibility its market share should warrant, LiMo is a clear leader in delivering Linux phones to the market. A plethora of models has been produced by Motorola, NEC/NTT DoCoMo, Panasonic, Samsung and LG.

7.2 Porting from Linux to Symbian

The main porting goal is to simplify the task by using ported components such as Open C/C++ and Qt for S60, or other existing components that are easier to manage in a Linux-based code base, for example, UNIX-like APIs such as Khronos' OpenKODE.

Even though the libraries are available, not all may be present on devices currently in the market. While Open C/C++ is present in all modern S60 phones (see Chapter 4 for more details), Qt is currently not and requires either a separate SIS installation or bundling the Qt SIS file with your application. This situation is temporary as Qt is proposed to become the base UI in Symbian 4 and should therefore be present in all subsequent Symbian devices.

Finally, it may sometimes be necessary to combine Open C with Symbian C++ APIs, producing 'hybrid' code (see Chapter 5). This may be more challenging than it looks at the first glance, primarily due to the way asynchronous calls are often implemented in Symbian C++ using Active Objects. General advice is to endeavor to keep the two programming styles separate by using the Façade pattern (see Section 15.2.5) to abstract away Symbian native APIs.

In the remainder of this chapter, we examine porting code that uses various features, starting with basic compatibility libraries and then covering UI, multimedia, networking, security, database and other APIs.

7.2.1 Basic Support

Open C/C++ Libraries

Although we have discussed Open C/C++ throughout the book, we venture into it once more. This is because Open C and POSIX compatibilities have special relevance for porting from Linux. Most importantly, Open C provides core compatibility libraries, as shown in Table 7.2.

Table 7.2 Open C libraries⁸

Library	Description	Source	Function Coverage
libc	Standard C libraries providing standard input–output routines, database routines, bit operators, string operators, character tests and operators, DES encryption routines, storage allocation and time functions	FreeBSD	47%
libpthread ⁹	APIs for thread creation and destruction, an interface to establish thread-scheduling parameters, and mutex and condition variables to provide mechanisms for synchronizing access to shared process resources	FreeBSD	60%
libm	Arithmetical and mathematical functions that operate according to the Standard C library	FreeBSD	42%
libdl	APIs that load DLLs	FreeBSD	100%
libz	In-memory compression and decompression functions, including integrity checks of the uncompressed data	Zlib	100%
libcrypt	Functions for encrypting blocks of data, messages, and password hashing	OpenSSL	100%
libcrypto	Services used by the OpenSSL implementations of SSL, TLS, and S/MIME; also used to implement SSH, OpenPGP and other cryptographic standards	OpenSSL	77%
libglib	Data types, macros, type conversions, string utilities, file utilities, and a main-loop abstraction.	Gnome	77%
libssl	SSL 2.0/3.0 and TLS 1.0 protocols	OpenSSL	86%

While function coverage may seem low in places, supported functions are strategically selected to produce a minimum set designed specifically for portability. The next section covers missing functionality in more detail.

The `libglib` library is essentially GTK Glib, which also includes GObject support. As we have seen, these libraries are used in all Linux MID platforms and are a major asset in porting. Open C++ expands on this by including support for IOStreams, STL and Boost APIs (see Table 7.3).

Open C Incompatibilities

There are some very important incompatibilities in Symbian/S60 versions of the Open C libraries. Unfortunately, many of these are unlikely to go

⁸ Source: Nokia Technology Datasheet.

⁹ IEEE Std 1003.1c (POSIX) is the standard interface for implementing multiple threads of execution within a traditional user process.

away. The S60 Developer Library¹⁰ maintains a full list of incompatible and missing APIs.

Table 7.3 Open C++ libraries

Library	Description	Source	Function Coverage
IOStreams	The C++ counterpart of the standard C <code>stdio.h</code> library, using streams to provide input–output functionality on sequences of characters	STLPort	100%
STL	Well-structured generic C++ components that work together seamlessly: <ul style="list-style-type: none"> • <code>Algorithm</code> defines computational procedures. • <code>Container</code> manages a set of memory locations. • <code>Iterator</code> provides a way to traverse through a container. • <code>Function</code> encapsulates a function for use by other components. • <code>Adaptor</code> adapts a component to provide a different interface. 	STLPort	100%
Boost	Portable, efficient libraries that work well with the standard C++ library: <ul style="list-style-type: none"> • Smart pointers provide objects that store pointers to dynamically allocated (heap) objects: <code>scoped_ptr</code>, <code>scoped_array</code>, <code>shared_ptr</code>, <code>shared_array</code>, <code>weak_ptr</code>, and <code>intrusive_ptr</code>. • Containers are generic data structures capable of holding many different types of data: <code>array</code>, <code>dynamic_bitset</code>, <code>graph</code>, <code>multi_array</code>, <code>multi_index</code>, <code>pointer container</code>, <code>property map</code>, and <code>variant</code>. • <code>Math</code> provides special math-template functions. 	Boost	100%

The following APIs and features are missing or not supported:

- `fork` and `exec`
- `long double`
- time stamps for file entries
- `wchar` works only with UTF8
- no concept of user or group on Symbian

¹⁰ See library.forum.nokia.com.

- floating-point exceptions
- complex number APIs.

The following APIs and features are partially supported or have non-standard behavior:

- `popen()` is partially supported
- `wait()` and `waitpid()` behave differently
- `dup2`
- `pthread_attr_setscope()` supports only `PTHREAD_SCOPE_SYSTEM`
- `pthread_attr_setschedpolicy()` supports only `SCHED_RR`
- `pthread_join()` enables only one thread to join a specific target thread
- `pthread_setschedparam()` supports only `SCHED_RR` policy
- limited support for cryptographic algorithms due to intellectual property rights issues.

Threading and Processes

The Symbian platform has full support for multi-threaded applications. As one would expect from a modern operating system, Symbian offers the standard suite of synchronization and inter-process communication tools such as semaphores, mutexes, message queues and shared memory with semantics similar to Linux. In addition, Open C offers nearly full support for `libpthread`, including thread pools, making porting basic threading code very easy.

IPC and Porting from DBus

Most mobile Linux distributions use DBus for inter-process communication (IPC) or as a generic message queue system. DBus is usually not used directly in applications; rather, the library is used through bindings, such as GLib bindings. This is potentially very useful because GLib defines a main loop, a type system, and an out-of-memory handling policy so that DBus operation keeps behaviors similar to the remainder of the application.

The challenge, however, is the way DBus is often used on Linux platforms. It appears that Linux distributions – including some mobile ones – have taken their cue from Symbian’s client–server mechanism to benefit from added resource efficiency and security. The concept

is, in essence, that the operating system and middleware functionality is exposed through IPC servers as opposed to directly through libraries. From the application developer's point of view, there is little difference; apart from the need to maintain sessions, the effects on the API are minimal.

There is no DBus port available for Symbian or servers to support DBus-based APIs, so if DBus support were available, it would be of little use without feature servers. If your application uses DBus-based services, it is likely that service functionality must be ported to corresponding Symbian APIs.

Let us now take a closer look at how we would port a DBus-based service or an application that uses DBus directly. Symbian's alternative to DBus is 'message queues'. There is no one-to-one API equivalence but the main message transfer functions are very similar.

CreateGlobal()

In Symbian, the code is as follows:

```
_LIT(KGlobalName, "GlobalMessageQueue");
const TInt KNumberOfSlots = 5;
const TInt KMessageLength = 16;
RMsgQueueBase queue;
TInt ret = queue.CreateGlobal(KGlobalName, KNumberOfSlots,
                             KMessageLength, EownerProcess);
```

There is no equivalent in DBus.

CreateLocal()

In Symbian, the code is as follows:

```
const TInt KNumberOfSlots = 2;
RMsgQueueBase queue;
TInt ret = queue.CreateLocal(KNumberOfSlots,
                             RmsgQueueBase::KMaxLength);
```

There is no equivalent in DBus.

OpenGlobal()

In Symbian, the code is as follows:

```
_LIT(KGlobalName, "GlobalMessageQueue");
RMsgQueueBase queue;
TInt messageSize = 0;
```

```
TInt ret = queue.OpenGlobal(KGlobalName1);
if (ret == KErrNone)
{
    messageSize = queue.MessageSize();
}
```

In DBus, the code is as follows:¹¹

```
DBusError err;
DBusConnection* conn;
int ret;
// initialize the errors
dbus_error_init(&err);
// connect to the bus
conn = dbus_bus_get(DBUS_BUS_SESSION, &err);
if (dbus_error_is_set(&err)) {
    fprintf(stderr, " Error (%s)\n", err.message);
    dbus_error_free(&err);
}
if (NULL == conn) {
    exit(1);
}
```

Send()

In Symbian, the code is as follows:

```
_LIT(KGlobalName, "GlobalMessageQueue");
RMsgQueueBase mqueue;

TInt ret = mqueue.CreateGlobal(KGlobalName1, 1, sizeof (TInt));
if (ret == KErrNone)
{
    TInt src = 45;
    ret = mqueue.Send(&src, sizeof (TInt));
    TBool full = (ret == KErrOverflow);
    //blocking send

    src = 32;
    mqueue.SendBlocking(&src, sizeof(TInt));
    mqueue.Close();
}
```

In DBus, the code is as follows:

```
// unique number to associate replies with requests
dbus_uint32_t serial = 0;
DBusMessage* msg;
DBusMessageIter args;

// create a signal and check for errors
```

¹¹ The DBus code is from *freedesktop.org*.

```

msg = dbus_message_new_signal(
    "/test/signal/Object", // object name
    "test.signal.Type",    // interface name
    "Test");               // name of the signal
if (NULL == msg)
{
    fprintf(stderr, "Msg Null\n");
    exit(1);
}
// append arguments onto signal
dbus_message_iter_init_append(msg, &args);
if (!dbus_message_iter_append_basic(&args,
    DBUS_TYPE_STRING, &sigvalue)) {
    fprintf(stderr, "OOM!\n");
    exit(1);
}
// send the message and flush
if (!dbus_connection_send(conn, msg, &serial)) {
    fprintf(stderr, "OOM!\n");
    exit(1);
}
dbus_connection_flush(conn);

// free the message
dbus_message_unref(msg);

```

Receive()

In Symbian, the code is as follows:

```

_LIT(KGlobalName, "GlobalMessageQueue");
RMsgQueueBase mqueue;

TInt ret = mqueue.OpenGlobal(KGlobalName1);
if (ret == KErrNone)
{
    TInt data;
    ret = mqueue.Receive(&data, sizeof (TInt));
    TBool empty = (ret == KErrUnderflow);
    //blocking receive
    mqueue.ReceiveBlocking(&data, sizeof(TInt));
    mqueue.Close();
}

```

In DBus, the code is as follows:

```

// add a rule for which messages we
// want to see
dbus_bus_add_match(conn,
    "type='signal',interface='test.signal.Type'",
    &err); // see signals from the given interface

```

```

dbus_connection_flush(conn);
if (dbus_error_is_set(&err)) {
    fprintf(stderr, "Match Error (%s)\n", err.message);
    exit(1);
}

```

Wait for Messages

In Symbian, the code is as follows:

```

TRequestStatus stat;
mqueue.NotifyDataAvailable(stat);
mqueue.CancelDataAvailable();
User::WaitForRequest(stat);

```

In DBus, the code is as follows:

```

// loop listening for signals being emitted
while (true) {
    // non blocking read of the next available message
    dbus_connection_read_write(conn, 0);
    msg = dbus_connection_pop_message(conn);
    // loop again if we haven't read a message
    if (NULL == msg) {
        sleep(1);
        continue;
    }
    // check if the message is a signal from the correct interface
    // and with the correct name
    if (dbus_message_is_signal(msg, "test.signal.Type", "Test")) {
        // read the parameters
        if (!dbus_message_iter_init(msg, &args))
            fprintf(stderr, "Message has no arguments!\n");
        else if (DBUS_TYPE_STRING != dbus_message_iter_get_arg_type(&args))
            fprintf(stderr, "Argument is not string!\n");
        else {
            dbus_message_iter_get_basic(&args, &sigvalue);
            printf("Got Signal with value %s\n", sigvalue);
        }
    }
    // free the message
    dbus_message_unref(msg);
}

```

7.2.2 User Interface

Remarkably, there are significant developments in porting Linux user interfaces to Symbian. The official port of Qt to Symbian and Nokia's commitment to maintain the port is a compelling reason to use Qt UIs.

However, the main advantage of using Qt is portability to a variety of platforms, including desktop Windows, Linux, Mac OS X and mobile Windows CE, Linux and Symbian/S60.

Applications with a UI based on Qt still require some porting work. Mainly this is down to differences in screen size, orientation and input methods (e.g. touchscreen vs keypad). As is usually the case, ‘write once, test everywhere’.

At the time of writing, Qt for S60 is still in pre-release and several modules are missing. That being the case, Qt is still not present in phone ROMs and must be installed by the end user before installing Qt-dependent applications. Luckily, the installation process can be streamlined. For example, a Qt-dependent application can specify the dependency in the Symbian PKG file and the end user is notified during installation if the Qt libraries are missing. Further still, the Qt SIS file can be included in your application’s SIS file and installed as part of application installation. The following is just an example – please note that actual field values must be updated for the correct Qt version (here 4.5.0), UID (here 0x23456789) and component name (here simply ‘Qt’):

```
; Embedded sis
@"qt-4.5.0.sis", (0x23456789)
; Dependency and required version
(0x23456789), 4, 5, 0, {"Qt"}
```

For users of other UI toolkits such as Hildon or GTK, the options are porting to Qt or moving to Symbian native UI. At the time of writing, there are some efforts to port GTK to Symbian and S60, however these are still in very early stages. Qt looks like a more future-proof option – first, because of the portability to other platforms and also because it is likely to reach all Symbian devices without significant delay.

For application development which requires fast 2D and 3D graphics with UNIX-like APIs, standard OpenGL ES and Open SVG APIs are also available on Symbian in direct screen access mode. See the `CDirectScreenAccess` class in the Symbian Developer Library for details on direct screen access.

Additional APIs are available via Khronos OpenKODE. This set of portable APIs also promises hardware acceleration and API quality on a par with Microsoft’s DirectX and is definitely worth evaluating.

Future developments in the Symbian UI space include a declarative user interface based upon an up-and-coming Qt widget set called ‘Orbit’. Orbit and a declarative UI will offer XML-based UI definition in a similar way to XUL.¹²

¹² See www.mozilla.org/projects/xul.

7.2.3 Other Features

System Functions

For many system-level functions (such as getting the battery or signal level, or the phone's IMEI), there is usually no common ground between Linux distributions or any of the available mobile platforms for that matter. However, this typically represents a comparatively small amount of code.

A variety of examples of how to perform these operations can be found on the Forum Nokia and Symbian Developer websites. The most commonly used 'recipes' have also been collected and published in Aubert, M. (2008) *Quick Recipes on Symbian OS*, Symbian Press, and others are available at developer.symbian.org/wiki/index.php/Category:Recipe.

Multimedia

There are several ways to program multimedia on Symbian, including Symbian C++ native APIs, the port of SDL/CSDL libraries and Qt's Phonon. Symbian native APIs are by far the most complete in terms of functionality.

Most Linux MID platforms use GStreamer, however. Interestingly, a GStreamer port to S60 appears to be underway in the community, however at the time of writing, this effort is still in its very early stages.

Networking

Porting networking code from Linux should be fairly straightforward with Open C sockets. The QtNetwork module is also supported in the current pre-release of Qt – porting an application that uses QtNetwork should not require more than a recompilation.

Security

The Symbian platform features a unique and well-developed security model not commonly seen in other operating systems. The system, dubbed 'platform security', defines a set of capabilities that are assigned to executables (including DLLs). A capability roughly corresponds to a permission to use specific system resources. Applications request and are assigned capabilities during installation, while system services and file system locations come with pre-defined capabilities. At run time, capabilities are enforced by the system.

Understanding platform security is essential for developing for the Symbian platform. It is a complex topic which is covered in more detail in Chapter 14. The reader is encouraged to find out more about

Symbian's platform security architecture from Heath (2006),¹³ the Symbian Developer Library, Forum Nokia¹⁴ and the Symbian Developer wiki.¹⁵

In terms of security APIs, Open C offers the good and compatible libraries `libcrypto` and `libssl`. The main limitation is that, due to export control issues, not all security algorithms are available at the time of writing. Specifically, the following algorithms are missing: `Rc5`, `IDEA`, `Blowfish`, `cast`, `ripemd`, `md4`, `mdc2`, `ecc`, `ecdh` and `ecdsa`. The transition of the Symbian platform source code to open source should solve this limitation in the near future.

Database

Database access for mobile devices has essentially two meanings: on-device databases and connectivity to off-device databases.

Two on-device database engines currently ship in all S60 phones: Symbian DBMS and SQLite. Symbian DBMS is a legacy engine with very limited SQL support but with a complete API and good performance. Support for SQLite¹⁶ on-device databases is offered via the Symbian SQL API and the usual SQLite C API. The latter is likely to be very attractive for porting applications from Linux because SQLite appears to be ubiquitously adopted as a database of choice in Linux MIDs.

Finally, we have already mentioned the PAMP stack, which offers a MySQL port to Symbian, among other things. However, MySQL requires several times more memory than SQLite. Also, the MySQL port has not benefited from the rigorous performance tests, tuning and optimizations that SQLite has been put through for the Symbian platform.

For accessing off-device databases, developers must use APIs provided by database vendors. Oracle 10 g Mobile Client¹⁷ has been available for Symbian OS since 2006 and Sybase iAnywhere¹⁸ Symbian client is available and actively updated.

Other APIs

There is a wealth of other APIs that have not been specifically covered so far in this chapter. Code using connectivity (e.g. Bluetooth), location

¹³ Heath, C. (2006) *Symbian OS Platform Security*. John Wiley & Sons.

¹⁴ See community.forum.nokia.com.

¹⁵ See developer.symbian.org.

¹⁶ See www.sqlite.org.

¹⁷ See www.oracle.com/technology/products/lite/symbian.html.

¹⁸ See www.sybase.com/ianywhere.

(GPS/AGPS) and PIM (contacts, calendar, etc.), are likely to require re-writing to use native Symbian APIs or as Qt applications using the mobile extensions discussed in Chapter 6.

A good approach to dealing with hybrid code would be to introduce a Façade that separates UNIX-like code from Symbian C++ native code. There are several main benefits to this – minimizing the amount of hybrid code that needs to be written, keeping the code that implements core functionality clean and separating out Symbian C++ asynchronous code from the rest of the application code.

7.3 Summary

Targeting the Symbian platform for application development and porting has become much easier in recent years. A variety of readily available standard and UNIX-like APIs for Symbian provide a good basis for porting Linux applications. The set of features not available in these APIs is continually reduced and further libraries are continually ported. Using phone features, however, still lacks POSIX-compliant APIs and requires hybrid code, mixing Symbian C++ native APIs and UNIX-like code.

8

Porting from Microsoft Windows

Success is a lousy teacher. It seduces smart people into thinking they can't lose.

Bill Gates

Microsoft Windows is one of the world's most widely used operating systems and is available in desktop, server and embedded variants. Due to its ease of development and widespread adoption there are a large number of developers, development tools, applications and resources worldwide.

Developers porting code from Windows to Symbian often find difficulties when attempting to map the ideas and concepts in Windows onto Symbian. To assist you in doing this, there are four major ways that porting code can be accomplished:

- create a library of base components in both platforms
- refactor the code bases to bring the high-level designs closer together
- use the Open C/C++ libraries
- do a total rewrite of the code base.

Ideally the application would have been written using a modular design and developed using ANSI standard C or C++, so the actual porting is largely down to reworking the user interface rather than any significant rewrite. Unfortunately, in the real world, this is seldom practical or even possible, especially if the program has been through a number of revisions. Most projects use one or some of the options above to move their product set onto Symbian devices.

Since Open C/C++ has already been discussed in depth in this book, this chapter's focus is on migrating the code and the design decisions that need to be made when porting from Windows to Symbian. Furthermore,

unless otherwise indicated, this applies to all of the Windows family. Due to the commonality of the Windows APIs, the variants are, to all intents, the same, though Windows Mobile does not have the security architecture of the desktop and server products and Windows desktop and server variants do not have the code-signing requirements of Windows Mobile.

With C# being a very popular development language on Windows, the chapter closes with a section on porting .NET Compact Framework code from Windows Mobile to Symbian.

8.1 Architecture Comparison

In many respects, Windows and Symbian are very similar in that they are designed around the traditional models of an operating system. They are both fully preemptive, robust, multi-tasking and extensible operating systems. The traditional kernel–user split is used by both to abstract access to the hardware and applications. Both provide a rich API set that encourages third-party development of applications and extensions to the core functionality for both hardware and software. Processes, threads, dynamic libraries, windows, IPC and ITC are all supported natively by the underlying operating system.

Since both the operating systems at their core use C and C++, the APIs provided to access the system are also in C and C++. Whilst the Windows API is primarily based on C, it has become increasingly prevalent to use third-party libraries to provide object-oriented features over the C functions in the move towards a more modern framework. The most natural mapping of Windows onto Symbian and vice versa is, however, the Microsoft Foundation Classes (MFC) that provide high-level C++ class wrappers over the existing Win32 APIs and is part of Microsoft Visual Studio. For example, where a window would be created using the Win32 API `CreateWindowEx()` call, you create it in MFC by constructing a new object of type `CWnd`.

As well as providing a wrapper over the Win32 libraries, MFC provides a full asynchronous messaging and document view architecture very similar in nature to the Active Object and EIKON/AVKON frameworks on Symbian. Developers extend the framework for their own needs rather than building the functionality from the ground up.

Since most Symbian devices are not touch enabled, it often needs a significant amount of thought as to how navigation and interaction would be easily accomplished with just the five-way navigation button and two soft keys. Often it involves redesigning the client to cater for this and sometimes even back porting these user interface changes to Windows Mobile. It must be said, however, that this is somewhat easier if the application has already been updated to cater for Windows Mobile Standard which does not support a touch screen. A significant

number of applications are designed to work with a mouse and, with this functionality not being emulated on S60 devices, a redesign of the user interface is sometimes the only way to handle the lack of pointer services in the user interface.

In Windows, the API functionality is typically exposed as a DLL loaded into the process and then the process uses the functions the DLL provides. Inter-process communication (IPC) is rarely used and it is more common to use threading to perform asynchronous functionality. Symbian however is designed so that services are normally provided by processes and typically there is a very lightweight wrapper DLL that encapsulates the call to the server. For example, the bitmap server provides services to do with bitmap creation and information. The API layer is stored in `FBSCli.dll` and accesses the single running `FBSErv.exe` instance which provides the real service for all clients on the device so, for clients using the bitmap services, the complexity is hidden behind the API.

But the most contentious issue is the handling of errors. Windows tends to be very forgiving in a number of areas such as handle and pointer usage. Typically Windows functions return an error code if the handle used is invalid. Likewise accessing memory outside of the allocated block tends to hide errors until such a time as the program reports an access violation and pointer mismanagement happens all too easily. Symbian however treats all programming errors, such as using an invalid handle, as critical errors and stops the program running.

When looking at architectural issues, we also have to consider the wider aspects of the platform. These platforms are optimized to reduce power consumption rather than processing speed so they run slower. In addition, it is typical for them to have much less memory than their desktop counterparts. This may require redesigning some algorithms for performance or space. In particular, many devices do not have native support for floating-point arithmetic so floating-point code is either emulated or used only within the Digital Signal Processor (DSP) and so a subsequent degradation in performance can be expected. The devices also do not have large amounts of memory and no swap files, so memory can be highly constrained. Furthermore, since they are designed to be mobile devices, they are not generally extensible with new hardware. Adding new hardware to the device is intended to be done via Bluetooth or the USB connector rather than the more traditional approach of adding a new board, as would be done with a server or desktop.

8.2 Application Compatibility

In Windows, the developer needs to carefully target the minimum operating system configuration on which their application will run and then select the APIs available for use on that platform. For example, the NT

Security APIs are not supported or present on Windows 95, so any form of access control would mean that the application would have to be built using the Windows NT SDK.

Symbian projects may be built on a number of SDKs and it is crucial that you understand the numbering system used for SDKs so that the appropriate SDK is chosen for the set of devices to be targeted.

Nokia, as the UI platform provider, and Symbian, as the core operating system supplier, guarantee that all APIs in the public SDK will remain backwardly compatible for that major version, thus for an application to run on the widest possible range of devices, the lowest possible feature pack should be chosen. For example, if an application is to run on all S60 3rd Edition devices, including the latest Feature Pack 2 devices, such as the N78, then the S60 3rd Edition MR SDK should be chosen, as applications built using this SDK work on all base release, FP1 and FP2 devices. Take care to avoid building an application using the Feature Pack 2 SDK and expect it to work on base edition devices. There is special code in later releases of ROM to allow the device to understand file and resource formats from earlier versions. Newer devices may also contain new hardware that was not available when the base edition devices shipped, such as a new codec or hardware acceleration.

More on compatibility and binary breaks can be found on the book's wiki page.¹

8.3 Development Languages and SDKs

Both Windows and the Symbian platform support a wide range of programming languages, from assembly language all the way up to the newest desktop languages such as .NET and Ruby. Traditionally C and C++ have been the language of choice for embedded systems due to the small size and high performance which is of great benefit for resource-constrained devices. The APIs for Windows and Symbian are the most mature in C/C++ and have the widest range of functionality and device-level support. Due to Windows' age, its API is based on handles and coding can be done in plain C. More recently, the focus has moved to using MFC and C++. Symbian has always used C++ and there is no C API to access native modules.²

8.3.1 Development Environments

Development in Windows is typically accomplished with an IDE, though applications can be built from the command line using tools such as

¹The wiki page for this book contains links to useful resources that are referenced in this chapter. It can be found at developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform.

²The Open C/C++ libraries are a wrapper over the native C++ libraries.

make. The primary development environment on Windows is Microsoft Visual Studio. This IDE has an integrated editor, debugger and code assistance tools to reduce the edit–compile–link–debug cycle. This also offers mobile device development tools and on-device debugging in the professional versions.

The equivalent for Symbian is the use of the free Carbide.c++ IDE. This is built on the Eclipse platform and offers a similar set of tools to Microsoft Visual Studio. I strongly encourage you to use Carbide.c++ as this offers the most up-to-date environment for developing Symbian applications. Whilst Carbide.vs (which targets Symbian development in Microsoft Visual Studio) is available, it has a much smaller feature set and tools such as code analysis, on-device debugging and up-to-date help files are notably absent.

The Symbian SDKs also come with a number of examples and tutorial projects which should be studied to get an understanding of the Symbian operating system and how the supplied APIs work. This gives you a deeper understanding of the operating system architecture and how applications are built and run.

Carbide.c++ and the relevant SDKs can be downloaded from ***developer.symbian.org/main/tools_and_kits/index.php***.

8.3.2 Building Applications

Microsoft Visual Studio is the *de facto* standard when it comes to development of Windows applications. If any automation of builds is to be accomplished, developers typically achieve this through the command-line scripting interface tools for Visual Studio or via MSBuild.

Symbian however uses its own Perl-based toolchain that does a multi-stage build from various component files. The root file for a Symbian project is the `bld.inf` file. This file defines which compilers are supported, which projects are to be included and in what order they are to be built. The `bld.inf` file is the equivalent of the Visual Studio solution (`.sln`) file.

Each project file in `bld.inf` has the extension `.mmp` and contains information about how the project is to be built, including the type of project, its name and the location of files and libraries required to construct the final executable. In addition to MMP files, `bld.inf` may also contain extension makefiles to build additional components not handled by the toolchain. The MMP file is the equivalent of a Visual Studio project (`.vcproj`) file.

Building a project in Symbian from the command line involves using the `bldmake` command. This command creates the files required by the toolchain to create the binaries. This is similar in many respects to `MsBuild`.

The final step is to create the installation file. The components of the installation file are stored in a `.inf` file in Windows Mobile and in a `.pkg` file on Symbian. In Windows Mobile, all the executables are signed using SignTool and then cabwiz is used to read the `.inf` file and create the final `.cab` file. SignTool is used again to sign the `.cab` file with the appropriate signature. In Symbian, the equivalent is to use makesis, which reads the `.pkg` file and creates the SIS file. The SIS file is then signed using the signsis program and the resultant file can be deployed to the device. It is important to note that, unlike in Windows Mobile, the binaries are not signed but the resultant SIS file must be signed and must obey the rules described in the security section.

More information on the toolchain and building applications can be found on the Symbian developer wiki at [**developer.symbian.org/wiki**](http://developer.symbian.org/wiki).

For the most part, Visual Studio and Carbide.c++ take away a lot of these issues as they provide wizards to generate new projects with all the correct files and settings.

8.3.3 Restricted APIs

Symbian is much more restrictive about API usage than Windows. Access to some APIs on Symbian devices is limited and requires additional approval, licenses and partnerships for use in products. This can require significant time to arrange as it normally requires legal agreements and contracts.

Nokia have opened some of the most common APIs to third-party usage with a simple click-through agreement. More information on the partnering SDK and examples can be found at [**forum.nokia.com**](http://forum.nokia.com).

8.3.4 Coding Style

There are various coding styles that are used to make code easier to understand. In Symbian, it is very important to follow the standards otherwise it can be very difficult to diagnose issues quickly.

The Windows APIs are built around a convention called ‘Hungarian notation’ that encodes the type of variable in the variable name. For example in the Windows SDK, a string parameter passed to a function is typically `LPCTSTR lpszString`, indicating a long pointer to a zero-terminated string. Examples of this type of encoding of parameters is liberally spread throughout the Windows header files.

Symbian uses a comprehensive and sophisticated variable and function naming system to illuminate side effects of function calls and variables. Table 8.1 illustrates common mappings.

Table 8.1 Mappings between Windows and Symbian types

Windows	Symbian	Notes
BOOL	TBool	Boolean type
BYTE	TUint8	Single byte
DWORD	TUint32	Four-byte unsigned number
WORD	TUint16	Two-byte unsigned number
LONG	TInt32	32-bit signed number
UINT	TUint	32-bit unsigned number
BSTR	HBufC*	Pointer to a string
LPSTR	TText8*	Pointer to a narrow string
LPTSTR	TText*	Pointer to a string
LPWSTR	TText16*	Pointer to a wide string
LPOINTER	TAny*	Pointer to a void pointer
RECT	TRect	A rectangle
POINT	TPoint	A two-dimensional point
HANDLE	RHandle	A handle

The significant differences between Windows and Symbian types are as follows:

- Handles are effectively `DWORD` objects in Win32. In Symbian, `RHandle` is a C++ class that manages the lifetime of the handle. There is no single class in MFC that manages handles as each type of object has their own internal handle that it manages.
- In Symbian classes, instance variables (or member variables to use the MFC terminology) have a prefix of `i`, whereas in MFC the prefix is `m_`.
- In Symbian, function parameters are prefixed with `a`; Windows encodes the type of variable in Hungarian notation and there is nothing to indicate that it is an argument in a function call.

I recommend that developers moving from Windows to Symbian C++ spend a significant amount of time learning these idioms to ensure reliable and standard code is produced. Unless these idioms are followed, they make the code challenging for people who are familiar with Symbian C++. The CodeScanner and LeaveScan tools provided in Carbide.c++ do extensive checking of these idioms to isolate and highlight possible coding issues and so it becomes more difficult to isolate issues when these rules are not followed. Further information on Symbian idioms can be found in Chapter 3.

8.3.5 Debugging

The Windows Mobile SDK provides a simulator for debugging code. The simulator runs an ARM emulation of the processor and is a very

high-fidelity emulation of the device ROM. Symbian provides a Windows emulation of the device; it does not execute the same code that would be placed on the device and so low-level issues, such as segment alignment faults and hardware emulation, are not reliably handled. High-level issues such as look-and-feel and input are accurately reflected on the emulator but there is still the potential for unexpected behavior when moving over to a real device. For this reason, the application should be regularly tested on a real device (or, even better, on a few devices) to ensure the application's features remain usable. External testing services can also be used to test an application's functionality on a number of devices including preproduction devices.

With Carbide.c++, on-device debugging is supported through the ID. Implicit in the use of Carbide.c++ is the expectation that the hardware and application are targeting S60 3rd Edition or later devices.

Symbian is more rigorous with error handling and applications exit when a panic occurs. If you are trying to isolate issues, it is essential to remember to enable extended error codes so errors are displayed when they occur; by default, applications end without warning which can make it confusing to developers.

More information on enabling extended panic codes can be found at [*developer.symbian.org/wiki*](http://developer.symbian.org/wiki).

8.4 SDKs and APIs

Modern operating systems such Windows and Symbian have a rich architecture and API set to facilitate development of applications. There is full support for preemptive processes and threads, storage and communications to name but a small segment of the full API set.

When developing in C/C++ on Windows, there are two major development libraries. The C SDK is supported by all the Windows platforms and the Microsoft Foundation Classes (MFC) library is a C++ wrapper over the C libraries to provide a more object-oriented framework.

In addition to encapsulating the Windows API in a class framework, MFC provides a set of rich data structure classes, an asynchronous messaging framework, extensible document view classes and a persistence framework that helps reduce the overall burden of developing high-quality applications on Windows.

8.4.1 Comparing the Win32 API with Symbian APIs

The Windows API is largely based around handles and functions. There is no object-oriented native API to manage resources via standard C++. When a resource is allocated, a handle is returned to the caller and then the services this API provides can be accessed using the handle. You must

remember to release the resources back to the operating system and to handle errors when any of the functions fail.

The APIs for Symbian are all based in C++ and use Symbian C++ styles and idioms extensively. This can make it difficult for Windows developers to understand how to work with Symbian code. The Windows API and the lower level Symbian APIs work in a similar way. They are handle-based, except Symbian encapsulates the functionality in a class to control the acquisition and relinquishing of the resources. The major differences start to appear with more sophisticated services such as the Windowing APIs which, although handle-based, perform their services in a totally different manner and so there is no clear mapping in Symbian for the functionality.

Symbian also has a more rigorous memory management model and when working with ‘blobs’ of memory, raw pointers are eschewed in favor of tracking the pointer and the size of the memory allocation. This is called the descriptor API. Accessing system services almost always requires a descriptor to be used, making it difficult to avoid using them at some stage of development. More information on descriptors can be found in Chapter 3.

When working with Symbian, one common issue is how windows are implemented. In Microsoft Windows, it is normal for each window to be represented with a real Window handle and the Kernel manages the windows. Since much of Symbian’s internal workings revolve around server processes, the windowing framework is also implemented as a server process. A Symbian application typically has a single ‘real’ window, represented by a handle in the window server and then many container windows that are embedded within the real window.

Since the window resides in a different process on Symbian, there is no concept of the window message queue or message passing being used to implement asynchronous messages and events. Instead controls are all derived from a common base class, called `CCoeControl`, with many functions representing quite closely the available windows messages. `CCoeControl` manages the windows, both real and container, depending on whether `CreateWindow()` or `SetContainerWindowL()` is called. Drawing of the real window then calls the draw function for the container controls and also works out the visible regions and z-ordering of the window. This allows for significant optimization when buffering changes.

It is also possible to avoid using the EIKON environment. In this case, a session to the Window server needs to be created and then an instance of the `RWindow` object can be created and used. All drawing will then occur through that window. This is similar to the technique that the Qt team used in the early technology previews (see Section 11.2) but does mean that all drawing and controls would need to be implemented from scratch, a non-trivial task.

The typical windows loop, `GetMessage()`, `TranslateMessage()`, `DispatchMessage()` does not have any relationship to the way Symbian works since this is handled by the more abstract `CActiveScheduler` class and active objects which do not need a window to function. Probably the closest to the Windows event queue are the `RWsSession::EventReady()` and `RWsSession::GetEvent()` calls, though these are the lowest level APIs that a window can generate.

8.4.2 Comparing the MFC/ATL Libraries with Symbian APIs

A much improved mapping of Windows onto Symbian is done via the Microsoft Foundation Classes (MFC) libraries. These provide an object-oriented framework over the Win32 libraries as well as providing additional functionality over the standard SDK.

Table 8.2 shows a mapping of the common MFC classes and how they relate to the Symbian classes and frameworks. This is by no means exhaustive and some classes, such as `CBrush` and `CPen`, have no equivalent functionality in Symbian; they are managed by the `CGraphicsContext` object since they are intrinsically part of the graphics context and not objects in their own right.

Most of the mappings between Symbian and MFC are fairly self explanatory and many of the classes are similar to their corresponding implementations in Symbian. One point worthy of mention is that, in many respects, the `CString` class (and its Win32 counterpart `_bstr_t`) is similar to the descriptor API since both provide range-checked dynamic memory allocation.

The major difference is in the command-handling routing code. MFC implements its own document view architecture where commands are routed in message maps from the view to the frame and, if neither of those handle it, it is routed to the document. The EIKON framework has a specific mixin class called `MEikCommandObserver` that handles the commands and they get routed via the `CEikAppUi` to the current view.

The persistence framework in MFC is derived from `CArchive` and is used to serialize objects to a file. The equivalent in Symbian is to use the stream APIs. Like descriptors, the stream APIs consist of a number of classes inheriting from `RReadStream` and `RWriteStream` to store data in a number of ways.

8.5 Porting an Application

There are a number of recommended strategies for porting an application. Whilst each case is unique, there are some tried and trusted blueprints

Table 8.2 Mappings between MFC and Symbian APIs

MFC	Symbian	S60 (licensee extensions)
CObject	CBase	
CWinApp	CEikApplication	CAknApplication
CDocument	CEikDcoument	CAknDocument
CFrameWnd	CEikAppUi	CAknAppUi
CDialog	CEikDialog	CAknForm/CAknDialog
CView	MCoeView	CAknView
CWindow	RWindow/CCoeControl	
CButton	CEikButton ^a	CAknButton*
CCombobox		CAknChoiceList*, CAknPopupFieldText
CEdit	CEikEdwin	
CHtmlEditCtrl		CBrCtlInterface
CIPAddressCtrl		CAknIpFieldEditor
CListBox	CEikListbox	CAknSingleStyleListbox is one of a number of example classes
CMFCTabCtrl		CAknNavigationControl- Container
CToolBarCtrl		CAknToolbar
CFileDialog		AknCommonDialogs
CArray	CArray/RArray/ RPointerArray	
CList	TSglQue	
CMap	RHashMap	
CCriticalSection	RCriticalSection	
CMutex	RMutex	
CSemaphone	RSemaphore	
CDynLinkLibrary	RLibrary	
CDatabase	RDb/RDbNamedDatabase	
CRecordset	RDbView	
CSocket	RSocket	
CInternet- Connection	RConnection	
CInternetSession	RHttpSession/ RHttpTransaction	
CRect	TRect	
CPoint	TPoint	
CSize	TSize	
CString	HBufC/TBuf/RBuf	
CTime	TTime	
CTimeSpan	TTimeInterval	
CDC	CGraphicsContext	
CClientDC/ CWindowDC	CWindowGC	
CFont	CFont	AknFontUtils
CRgn	RRegion	
CBitmap	CWsBitmap/CFbsBitmap	CAknIcon

^a This is only available in S60 5th Edition touch devices.

that can be used to make the process easier and the resulting code base is more stable and maintainable.

A review of porting strategies and tips that is more independent of the operating system can be found in Chapter 15.

8.5.1 Layered Design

Good design practice recommends that an application be broken down into three tiers: the user interface, the logic and the storage layers. I strongly recommend that you follow this design practice when porting to Symbian (or any other operating system) as it makes the process of migration much easier since the logic and storage layers tend to be very similar on the different platforms, but the user interface usually involves a total rewrite to provide the best user experience on the platform or to provide for different or missing features, such as touch and color.

8.5.2 Modular Implementation and Unit Testing

When porting a complex application, it is often useful to have unit tests for modules that can identify that the same code behaves correctly across all operating systems for which the implementation is to be provided. Modular design and unit tests help capture errors and issues early in the development cycle where they can easily be fixed, as well as providing a base for regression tests. Both Symbian and Windows support automated unit test tools.

8.5.3 Refactoring an Application

Often a Windows application is written using the standard Windows SDK and so is not object-oriented. The port may be an opportunity to move the code base into a more object-oriented framework before the porting process begins, as it is easier to port a project based on C++ classes to Symbian.

8.5.4 Cross-platform Libraries

It is significantly easier to port applications across operating systems using an ANSI-compliant C/C++ library, ideally with POSIX support.³ This makes it easier to migrate code other than the user interface into a standard form that works across the systems. In addition, many open source solutions (where licensing permits) can be utilized to produce a common framework for a particular layer. For example, SQLite (www.sqlite.org)

³ POSIX support is not available by default on Windows Mobile but can be licensed in third-party libraries such as the one from www.mapusoft.com.

can be used to provide a common database interface and implementation across the supported platforms. Likewise, Qt (www.qt.nokia.com) can be used to build a common user interface framework.

8.5.5 Common Platform Code

As an alternative to using standard libraries, or if the code is to be shared across operating systems that do not support ANSI standard libraries, a common library may be built that provides a set of common interfaces across all platforms and the underlying implementation is specific to a platform. This approach offers a common code base for the application architecture and makes the platform-agnostic common code simpler to understand. It does however frequently make the code less resilient to change since developers make a change in one platform and break another without realizing it.

For example, a string class called `CExString` could be defined as follows in Windows:

```
// WinExString.h
class CExString
{
public:
    CExString();
    ~CExString();
public:
    ExInt Length() const;
    ExInt AssignString(const ExChar* data);
private:
    void* _extension;
}
```

It is implemented in a different source file for each operating system. The `_extension` variable can be internally typecast, normally via a `#define` to the class of which it is representative; for example, on Symbian, it would be an `HBufC*` and on Windows Mobile, it would be a `BSTR` or `TCHAR*` depending on the expected use of the string.

In Windows, the implementation would be similar to this code:

```
// WinExString.cpp
#include <windows.h>
#define BSTR_CAST (BSTR)_extension
CExString::CExString()
{
    BSTR_CAST = NULL;
}
CExString::~CExString()
{
    SysFreeString(BSTR_CAST);
}
```

```

ExInt CExString::Length()
{
    return (ExInt)SysStringLen(BSTR_CAST);
}

ExInt CExString::AssignString(const ExChar* data)
{
    BSTR tmp = SysAllocString(data);
    if (tmp == NULL)
        return ExErr_NoMemory;
    SysFreeString(HBUFC_CAST);
    HBUFC_CAST = tmp;
}

```

In Symbian, the implementation would be similar to this code:

```

// SymbianExString.cpp
#include <e32std.h>
#define HBUFC_CAST ((HBufC*)_extension)
CExString:: CExString()
{
    HBUFC_CAST = NULL;
}

CExString::~~CExString()
{
    delete HBUFC_CAST;
    HBUFC_CAST = NULL;
}

ExInt CExString::Length()
{
    if (HBUFC_CAST == NULL)
        return 0;
    else
        return HBUFC_CAST ->Length();
}

ExInt CExString::AssignString(const ExChar* data)
{
    HBufC* tmp = HBufC::New(User::StringLength(data);
    if (tmp == NULL)
        return ExErr_NoMemory;
    delete HBUFC_CAST;
    HBUFC_CAST = tmp;
}

```

8.6 Windows-specific Issues

When porting code from Windows and Windows Mobile to Symbian, you will encounter some or all of the following issues in all but the most trivial of programs. This is the hardest part of any porting project, as missing core operating system functionality in one platform results in new code to cater for these issues.

Because operating systems are designed differently, at different times they offer different functionality and so what is central to Windows may

be totally missing in the Symbian platform (as in the case of the registry) or may be different (in the case of database support) or may work differently due to operating system limitations (in the case of services).

8.6.1 Unsupported Features

There is no support in the Symbian platform for a number of features that are part of Windows Mobile or Windows desktop. This list is by no means exhaustive but includes the most common issues encountered:

- multiple document interface (MDI)
- distributed COM
- interface definition language (IDL)
- IDispatch and late binding of interfaces
- shell folder APIs (SHLWAPI)
- common controls (including listview and treeview)
- DocList API
- ODBC/ADO
- printing APIs
- common dialogs (limited support for Open and Save via the AKN-CommonDialogs API)
- input method editors or soft input panels – consider the front-end processor architecture as a replacement.

These modules require a total rewrite for the Symbian platform as there is no similar functionality in the platform as it currently stands.

8.6.2 Windows Registry

In Windows, application and driver settings are almost always stored in the registry as it offers a secure, transactional, robust and scalable solution to storing different types of data for an application. Symbian offers a number of solutions to this problem and which one should be used is dependent on a number of factors.

If the application is being designed so the settings do not need to be shared with other applications then the settings storage could be accessed via a custom API and the implementation can then store the data in whatever form it needs to depending on the chosen implementation. This could be as simple as a flat file or a more complex database. It is often advantageous to port this solution back to Windows and expose the same API so that common patterns are maintained across the ports.

If the project requires the data to be shared amongst applications then this approach does not easily work. In that case, it is more beneficial to create a Symbian server to expose an API similar to the registry APIs to load and save settings. Access to the settings can then be controlled via the settings server. The settings server maintains its own data management and provides a robust and managed solution for settings. It may again be advantageous to port the client API back to the Windows version and so maintain a similar pattern between the ports.

It may be preferable to use the central repository, which performs the same functions as the registry but has the overhead of complex installation when being initialized from non-ROM sources and requires licensing.

8.6.3 Databases

Typically on Windows, database solutions range from the device object stores all the way through to SQL Server CE. The Symbian RDBMS solution tends to sit in the middle of this. It offers a subset of the SQL APIs as well as offering a record-based (ISAM) API to access the data.

The major problem with databases is, surprisingly enough, not the API to access the data, but rather the data itself. As the data is normally stored in a format that is optimal for the platform, just copying a database file over from one operating system to the other does not work. Data should be stored in an intermediate format that is portable across platforms with care being taken to ensure that the data correctly handles multibyte characters and byte ordering issues as well as addressing any performance and size limitations.

For simple projects, the object store implementations found on Windows Mobile can be replaced with the Symbian database APIs in non-SQL mode. These offer the same performance and feature set as the object store on Windows Mobile. This is a highly efficient scheme and is a much lighter solution to accessing data, albeit covering a much smaller set of data access scenarios.

For more sophisticated database manipulation, the native SQL APIs can be used. These offer a subset of SQL specifically tailored for mobile devices and so do not offer some of the more heavyweight features of relational databases, such as joins. For later releases of Symbian, it is possible to use the SQLite engine which provides a cross-platform implementation that can be shared between Windows and Symbian.

Other vendors, such as Sybase and Oracle, offer paid database solutions that are portable across the operating systems and so migration of data between the two is less of a problem in these cases.

8.6.4 Inter-process and Inter-thread Communication

Symbian has a rich inter-process (IPC) and inter-thread (ITC) services API to support isolated and secure access to operating system and third-party services. Many system services in Symbian exist as a separate process and applications connect to the process, which decides on the basis of the security identifier, the vendor identifier and the caller's capabilities the functionality that is made available. For example, the graphics environment, the windows framework and the telephony services are all implemented as separate processes and applications connect to and exchange data with these processes to perform their functionality.

On Windows Mobile, these application services are typically performed in the process and it is rare to find applications using process boundaries as a means of exchanging data. However, when it happens there are a number of ways it can occur:

- **Message Queues:** Point-to-point message queues are available in Windows Mobile and, to a certain extent, via the MailSlot API on Windows desktop and server. Symbian offers a similar solution with its own message queuing implementation to be found in the `RMsgQueue` class. It offers similar features but it must be stressed that the Symbian implementation allows a maximum of 64 bytes per message so if more than that needs to be transferred an alternative solution should be used.
- **Memory-mapped files:** This API is specific to Windows and allows a file to be treated as if it were a memory block. There is no equivalent to this on Symbian as the virtual memory manager available in Windows is not present on Symbian. One solution to this is to use shared chunks via the `RChunk` class on Symbian as this makes a global memory block available to share between processes.
- **Sockets:** This is the most common mechanism for IPC on Windows. A TCP/IP server is created on the device and local applications connect to the server and exchange data via a well defined binary API. Whilst it is possible to do this in Symbian, it is not generally recommended as it negates much of the security offered by the Symbian IPC framework. Additionally, constructing a client and server socket solution on Symbian can require a significant amount of code.
- **Services API:** A limitation of the Windows Mobile architecture is that only 32 processes can run simultaneously. To alleviate this, an application server is built as a DLL and is hosted inside the

`services.exe` application. Access to the service is then done via IOCTL function calls. Of course, a wrapper DLL or library can be built over the IOCTL calls to make the access model transparent and safe. The downside to this approach is that the DLL being hosted inside `services.exe` requires a privileged certificate, and thus signing, in order to be able to be hosted.

If any of these solutions is to be used, a well-defined API should be created on the Windows Mobile client and applications should access the other process through this API. This abstraction of the access methods to the server make it easier to map onto Symbian's native support for IPC. More information on Symbian's server architecture can be found at [developer.symbian.org/wiki/index.php/Client-Server_Framework_\(Fundamentals_of_Symbian_C++\)](http://developer.symbian.org/wiki/index.php/Client-Server_Framework_(Fundamentals_of_Symbian_C++)).

On Windows, inter-thread communication is a much more common scenario where synchronization primitives (such as critical sections) are used to arbitrate access to shared data structures. There are some caveats when dealing with threads:

- It is not possible to access user interface services outside of the primary thread.
- If a thread uses Active Objects, then it requires a cleanup stack and an active scheduler to function. Active objects are linked to an active scheduler and so by definition are implicitly attached to a thread and cannot be used outside of that thread. It is also important to remember that client-side services can encapsulate active objects within themselves and not expose this to the caller, so implicitly require both a trap harness and an active scheduler to function.
- Handles to sessions (such as the socket server or file server) cannot be passed between threads without marking the handle as shared (where supported) and generally cannot be passed between processes.
- Recursive synchronization primitives are not supported so re-entrant calls to threads that have already acquired the primitive result in a panic.
- Spinlocks are not supported. For semaphores, `RFastLock` may be used in place of a semaphore and offers some of the advantages of a spinlock.
- There is limited support for atomically changing a 32-bit value. Only increment and decrement are supported; anything else needs to be wrapped around one of the synchronization primitives in order to prevent multiple threads corrupting the value.

Table 8.3 shows the mapping between the common synchronization objects available in the Windows API, MFC and Symbian.

Table 8.3 Mappings between synchronization objects

Win32	MFC	Symbian
CreateEvent	CEvent	RThread::Rendezvous()
CreateMutex	CMutex	RMutex
CreateSemaphore	CSemaphore	RSemaphore
InitializeCriticalSection	CCriticalSection	RCriticalSection
InterlockedIncrement		User::LockedInc()
InterlockedDecrement		User::LockedDec()

An example of threading in Win32:

```
HANDLE hThread;
HANDLE hEvent

DWORD WINAPI ThreadProc(LPVOID)
{
    int finish = 20;
    for(int i=0; i<finish; i++)
        ; // log a message
    SetEvent(hEvent);
    return 0;
}

hEvent = CreateEvent(NULL, FALSE, FALSE, "ExSignal");
hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0);
WaitForSingleObject(hEvent, INFINITE);
CloseHandle(hEvent);
CloseHandle(hThread);
```

To do the equivalent in Symbian is very similar:

```
const TInt KStackSize = 8192;
const TInt KMinHeap = 4096;
const TInt KMaxHeap = 1024*1024;

RSemaphore s;
RThread t;

LOCAL_D ThreadProc(TAny* aAny)
{
    TInt finish = 20;
    for (TInt i=0; I < finish; i++)
        ; // log a message
    s.Signal();
}

s.CreateLocal(0);
t.Create(_L(" "), ThreadProc, KStackSize, KMinHeap, KMaxHeap, NULL);

t.Resume();
s.Wait();
s.Close();
t.Close();
```

Further information on synchronization primitives can be found at [developer.symbian.org/wiki/index.php/Threads,_Processes,_and_IPC_\(Fundamentals_of_Symbian_C++\)](http://developer.symbian.org/wiki/index.php/Threads,_Processes,_and_IPC_(Fundamentals_of_Symbian_C++)).

8.6.5 Binary Executables and DLLs

Windows developers normally put binaries under \Windows or \Program Files. In addition, it is rare for applications to be code signed or for the process to check and enforce security for an application. When moving over to Symbian, these issues need to be addressed otherwise the application will not execute successfully on a device. There are some significant differences for binaries when porting from Windows Mobile to Symbian.

8.6.6 Compiler Differences

When building for the device, Symbian supports two toolchains. The first is the free GCCE toolchain. The second is the Realview toolchain (RVCT), which may be purchased from ARM. Version 2.2 must explicitly be ordered otherwise the compiler will not compile Symbian code. The RVCT toolchain offers significant size and performance advantages over the GCCE compiler.

It is also important to note that when the RVCT compiler and the Windows Mobile SDKs are installed, the RVCT compiler needs to appear in the path before the Visual Studio tools as Microsoft have licensed the ARM compiler from ARM but it is a different version from the one used by Symbian and is not compatible.

8.6.7 Functions Exported by Ordinal

Unlike Windows, the current versions of Symbian do not export function names, so all run-time linking of a compiled application when a process is loaded is done by ordinal number rather than the more familiar name linking performed by Windows. This means that care needs to be taken to maintain the order of exported function ordinals between releases so that old versions of applications using that DLL do not break if the DLL is upgraded to a newer release. In Symbian, this term is referred to as freezing and is part of the toolchain. When a DLL has functions added to it, the DLL must be rebuilt and refrozen with the `abld freeze` command to ensure the new functions are added to the end of the existing DEF file and not in any random order.

It is important to remember that ensuring binary compatibility is more difficult in Symbian than in Windows Mobile and there are a number of

suggestions that need to be enforced to ensure that seamless upgrades can occur, especially where the components being upgraded are to be shared amongst multiple applications.

Further information on binary compatibility can be found at [developer.symbian.org/wiki/index.php/Compatibility_\(Fundamentals_of_Symbian_C++\)](http://developer.symbian.org/wiki/index.php/Compatibility_(Fundamentals_of_Symbian_C++)).

8.6.8 Code Signing

Whilst Windows desktop and server do not mandate code signing for user-side code, the network operator may do so for Windows Mobile devices. When deploying applications for Windows Mobile, the binaries may need to be signed with a certificate so that the operating system can verify the integrity of the binary. As is typical with Windows, any executable code can be stored in any folder and executed from any folder on the device.

Symbian's model differs markedly from Windows. When porting from Windows Mobile, it is imperative that you remember the capability access rules that were summarized in Table 4.5:

- `\sys\bin` is not visible to applications and is highly restricted.
- Executable code is only run from `\sys\bin`.
- The executable code is only installed via the software installer and not copied to `\sys\bin` any other way.
- Applications must not attempt to read the binary or search for binaries in the `\sys\bin` folder as the operations will fail with an 'access denied' error.
- Additional security is enforced for removable media.
- Each process has its own private folder under `\private` and this folder is only accessible to the application; it is not accessible to other programs or visible to end users. This folder is where application-specific settings and sensitive files should be stored to prevent tampering.

8.6.9 Capabilities

On Windows desktop and server, many developers are familiar with local security policy. This is used to establish user rights, for example, 'Logon as a service', 'Log on locally' and 'Load and unload driver'. Windows mobile has a much simpler system in that there is a single API

to check whether the process hosting the DLL can execute privileged code or not.

Symbian has a similar model called the capability model. Each binary file has associated with it a set of capabilities that identify the operations the binary is allowed to perform by the operating system. There are five simple rules to remember to ensure the application can load and run on the device:

- APIs define the capabilities they require.
- The binary name and UID are unique on the device.
- Capabilities are set when the application is compiled and enforced whenever the application is run.
- A DLL can only be loaded into a process if the DLL has at least the capabilities of the parent process.
- A DLL can only load another DLL if the DLL being loaded has the same or more capabilities than the requester.

8.6.10 UIDs

The Symbian loader has special rules to enforce security which the Windows loader does not. Whilst Windows Mobile uses the signature on the binary to decide what level of trust to assign to the application and DLL, Symbian has a much stronger security model designed to work with the IPC framework in Symbian. UIDs in the ranges 0xAxxx and 0x2xxx are guaranteed to be unique and are allocated by Symbian.

UIDs in the range 0xExxxxxxx are development UIDs and are for code that is not expected to be released or for development code. These UIDs can clash with existing applications or DLLs since they are not unique. All executable code is required to have a unique UID and this is enforced by the installer and is a common reason why applications fail to install or run.

8.6.11 Installer

Windows Mobile deploys code via a cab file or the developer can replace the executable on the device. Because each DLL is signed with a digital signature which is checked by the loader during run time, the Symbian software installer is very much a tool to manage an application and components. Symbian treats the software installer as the gateway to the device. Any executable code that is deployed to the device is validated for capabilities and certification and then the binaries are deployed to the operating system restricted folder `\sys\bin`. In addition, any resources

for the application are deployed to the read-only folder in `\resources` to prevent third-party applications changing them.

The Symbian software installer also enforces additional rules that the Windows installer does not. The developer must not change the filename or UIDs of files and the package name, package id and vendor name may not be changed either because this would cause the package not to be installed since the software installer would treat it as an entirely new package. Since binaries with the same name cannot be installed from multiple packages, care needs to be taken to ensure DLLs that may be shared are installed in their own SIS file⁴ so they can be installed and upgraded independently of the main application. The standard technique of deploying common DLLs (such as MFC) in multiple cab files under Windows Mobile does not work on Symbian.

There is a fuller set of installer error messages documented at [**developer.symbian.org/wiki/index.php/Troubleshoot_install_errors**](http://developer.symbian.org/wiki/index.php/Troubleshoot_install_errors).

8.6.12 Localization

It is normal in Windows to create a satellite DLL with just resources and, at run time, select the appropriate DLL by examining the language in which the operating system is running using the `GetUserDefaultUILanguage` API. When coding for localization, both the strings and the resource definitions are normally placed into a single resource file per language and then that file is localized into the required languages. The resource file is then compiled and linked into a DLL.

Symbian has a different process for localizing resources based on the way the toolchain and resource compilers function. The resource file contains the resource definitions and is language neutral. Language-specific strings are identified by a macro include from a separate file. There is then one file of strings per supported language and the MMP file contains a keyword indicating which languages should be built. The resource file compiler selects the appropriate language file for the language being compiled and generates a resource file for each language required. This file has an extension indicating the language. For example, English resources have the extension `.r01`. The Symbian run-time engine selects the appropriate language by using `BaflUtils::GetNearestLanguage()`.

The most used localization function in Windows is `LoadString()`. The equivalent in Symbian is called `StringLoader` and offers a number of additional formatting APIs over the Win32 equivalent.

It is up to the package creator how the languages are deployed. The installation script files need to be localized to support multiple languages, either by installing all the languages and allowing the operating system

⁴ Note that the SIS file can be embedded in the application's main SIS file, so this doesn't imply that the user has to install the DLLs separately.

to choose the correct one at run time or by having the installer prompt the end user to select a language when the application is installed.

8.6.13 Component Object Model

One of the key aspects of modern programming languages is the use of interface-based programming and the subsequent introduction of the use of shared modules. Windows particularly places a lot of emphasis on this and it is termed the Component Object Model (COM). Much of the extensibility of Windows comes directly from the fact that objects can implement interfaces and the system can use and discover new implementations of interfaces.

Symbian has adopted a similar model called the Epoc Component Object Model (ECOM) framework. ECOM works in a similar way to COM but there are clearly very different routes to implementation. ECOM is much simpler than COM in that there is no support for threading (apartments), no need for an interface language (IDL), no need to use VARIANT or BSTR data types and the reference counting framework is not needed. ECOM objects are treated the same as any C++ objects in Symbian.

A set of COM objects under Windows resides in a DLL. Each interface and implementation is associated with a UID and these are linked via the system registry. Because Symbian does not have the concept of a public settings registry, the link between the DLL, interface and implementation is stored in a resource file in the `resource\plugin` folder and the ECOM service application monitors this for changes and updates its internal databases accordingly.

In the Windows implementation, interfaces are expected to derive from `IUnknown` and manage their own lifetimes, destroying themselves when their reference count reaches zero. In Symbian, each object is derived from `CBase` and all the methods in the class should be pure virtual, aside from the static inline factory function that creates an instance of the class. The creation and destruction of the object are all done by the ECOM run-time system and it is up to the caller to manage the lifetimes as regular C++ objects. When implementing the class, a class should be derived from the interface class and implementations provided for the pure virtual functions. A static factory function needs to be provided so that the ECOM run-time system can find the class in the DLL. This is exposed through the single entry point in the DLL.

A typical Windows COM class and implementation would be:

```
class IMyInterface : public IUnknown
{
public:
    virtual void MyMethod() = 0;
}
```



```

class CMyImplementation : public IInterface
{
public:
void QueryInterface(REFIID iid, void **ppvObject);
{
if (riid == IID_IUnknown)
{
*ppvObj = static_cast(this) ;
AddRef() ;
return S_OK;
}
if (riid == IID_IMyInterface)
{
*ppvObj = static_cast(this) ;
AddRef();
return S_OK;
}
return E_NOINTERFACE;
}

ULONG AddRef()
{
InterlockedIncrement(&m_nRefCount)
};

TInt Release();
{
long nRefCount=0;
nRefCount=InterlockedDecrement(&m_nRefCount) ;
if (nRefCount == 0)
delete this;
return nRefCount;
}

private:
int m_nRefCount;
}

```

To use the object and interface is then a matter of instantiating an instance of `IMyInterface` and calling the method:

```

IMyInterface* pIFace = NULL;
CoCreateInstance(CLSID_MyImpl, NULL, CLSCTX_ALL,
                IID_IMyInterface, &pIFace);
pIFace->MyMethod(); // use the object
pIFace->Release(); // free the object

```

The corresponding ECOM implementation in Symbian is markedly simpler:

```

class CMyInterface : public CBase
{
public:
static inline CMyInterface* NewL(TUId aImplementationUid);

```

```

inline ~CMyInterface();

public:
virtual void MyMethod() = 0;
private:
iDtor_ID_Key
}

```

The corresponding implementation which would live in a DLL is defined thus:

```

class CMyImplementation : public CMyInterface
{
public:
static CMyImplementation* NewL(const TUid aImplementation);
{
TAny* interface = REComSession::CreateImplementationL(
                                aImplementation,
                                _FOFF(CMyImplementation, iDtor_ID_Key));
return (CMyImplementation*)interface;
}

public:
void MyMethod();
}

```

The object is instantiated as follows:

```

CMyInterface* p = CMyInterface::NewL(KMyImplementationUid);
p->MyMethod();
delete p;

```

In Windows, identifying and creating new instances of the objects that are implemented in the DLL is done through a single function that provides the object creation factory:

```

STDAPI
DllGetClassObject(const CLSID& clsid, const IID& iid, void** ppv)
{
(*ppv) = NULL;
if (clsid != CLSID_MyImplementation)
return CLASS_E_CLASSNOTAVAILABLE

if (iid != IID_IUnknown && iid != IID_MyInterface)
return E_NOINTERFACE

CMyImplementation* obj = new CMyImplementation
if (obj == NULL)
return E_OUTOFMEMORY

return obj->QueryInterface(iid, ppv)
}

```

The corresponding operation in ECOM is simpler. When the ECOM service loads, it builds a mapping of the implementation UIDs and the DLLs in which they reside. When a call is made to create an implementation, the DLL's factory function is queried to obtain the method that creates an instance and then that method is called and the instance is returned to the caller.

```
const TImplementationProxy implementationTable[] =
{
    IMPLEMENTATION_PROXY_ENTRY(KMyImplementation,
                                CMyImplementation::NewL)
}

const TImplementationProxy* ImplementationGroupProxy
(TInt& aTableCount)
{
    aTableCount = sizeof (implementationTable) /
                  sizeof (TImplementationProxy);
    return implementationTable;
}
```

8.7 Signing and Security

Both Windows Mobile and Symbian verify that the binaries have not been tampered with using cryptographic techniques. The way the two verification systems work however is fundamentally different. The Symbian Signing program and platform security are discussed in more detail in Chapter 14. This section is intended only to highlight the differences between the Symbian and Windows Mobile approaches.

Windows Mobile requires that each binary is checked before it is run and the code execution policy is checked to see that the executable and its dependent modules allows the execution of the code. The application will not run on the device if the policy check fails. In addition, a certain subset of functions and registry keys is classed as privileged and these cannot be called or modified unless the hosting process is signed with the privileged certificate. It up to the network operator as to the strictness of the policy enforcement on the device.

Symbian enforces the security when the application is installed and the strictness of the policy enforcement is determined by the Symbian licensee, though the network operator could amend the ROM image if required.

8.7.1 Types of Certificate

Both Windows Mobile and Symbian have different kinds of certificate and these restrict what classes of APIs can be called. Windows Mobile

has a privileged mode certificate and a regular developer certificate, in addition to the Microsoft signed certificate for released code.

Symbian has a similar set of certificates. If only user-granted capabilities are being used then a self-generated certificate may be used. If the application is being tested and is using enhanced set capabilities, then it may be locked to a specific phone by applying for a developer certificate (if a publisher ID has been obtained or manufacturer set capabilities are being used) or by submitting the application to Symbian Signed Online (if the correct UIDs are being used). Finally if the application has been through Symbian Signed then the application can be installed on any device as it is signed with the Symbian root certificate.

Carbide.c++ automates much of the process of building, deploying and testing as the toolchain integrates the certificates into the signing step of the build process.

8.7.2 Enforcing Security in Code

Windows Mobile applications tend to be built around DLLs that are loaded into the process and expose APIs to the host process whereas Symbian has a lightweight wrapper that provides the framework to talk to another process that then performs the functionality subject to the caller passing the security checks.

In Windows Mobile, the key function for enforcing security is `CeGetCurrentTrust()`. This API is used to determine whether or not the process is allowed to access APIs from the privileged set. Note, it returns the trust status of the current process; the operating system enforces the security.

There is a much richer security set on Symbian but, in general, `RProcess::SecureId()` and `RProcess::HasCapability()` are the two major functions used to identify the application and the process security policy and these are the most closely aligned APIs to the Windows Mobile ones.

8.7.3 Signing Programs

Windows Mobile applications do not have to pass any form of quality assurance. If the application is going to use the Microsoft brand then the application is required to pass the tests in the 'Designed for Windows Mobile' guidelines and must also be submitted to a third-party test house for further testing. This process is called the 'Mobile to Market' program.

Symbian have a similar, albeit mandatory, program called 'Symbian Signed'. Before an application can be run unrestricted on any phone, it needs to be passed to a test house that performs the Symbian Signed (and sometimes the licensee) testing criteria to ensure the application functionality reaches the minimum level of acceptance. Once an application

is signed, it is eligible to use the Symbian Signed logo and can be part of the licensee application catalog.

8.7.4 Symbian Signed Test Criteria

When application development and quality assurance is complete, the next step is to ensure that it passes the mandatory steps of the Symbian Signed test criteria. The test criteria are used by the test house to ensure that typical errors that affect the functionality of the phone are avoided in release code. In addition, the application may be required to pass additional testing if it is to be distributed by a device manufacture or network operator.

Unlike Windows Mobile, which does not have capabilities, the developer is required to provide a rationale for each of the capabilities being used by the application to ensure overly broad use is discouraged. This is normally done by identifying the operating system calls the application is making. Carbide.c++ provides a tool, called the capability scanner, that scans the project source files, identifies which functions are used and builds a list of the capabilities required for the application to run. Overly broad use of capabilities result in the application failing testing.

8.7.5 Signing Process

As www.symbiansigned.com describes in more detail, the development process is quite straightforward and involves the following steps:

1. A publisher ID is purchased from Trust Centre.
2. A developer certificate is obtained with the IMEI and capabilities required by the application.
3. Development and testing is performed until the application is stable enough to release.
4. The developer signs the SIS file with their publisher ID (note that this does not enable the application to install on the phone; it is solely to prove to the test house that the SIS file came from the submitter). Any relevant files such as a user manual or instructions should be included with the SIS file signed with the publisher ID.
5. The application is submitted via a web portal to the test house and the submitter justifies their use of the capabilities.
6. The test house performs the Symbian Signed tests and verifies that the application functions as specified. When the application passes, it is re-signed with a certificate that chains to the Symbian root certificate which is already present on the phone ROM.

7. The SIS file is returned and is now no longer locked to developer certificates.

8.8 Porting from C# and .NET

Whilst most developers are likely to be porting a C or C++ application, there are significant commercial and vertical market applications that have been developed using .NET Compact Framework for Windows Mobile devices and would benefit from porting the application over to Symbian. As an alternative to rewriting the application in C++, the existing .NET Compact Framework code can be ported over as .NET-compliant code to run on the device. Red Five Labs have a version of the .NET Common Language Runtime (CLR) that runs on all S60 3rd Edition and 5th Edition devices.

The Net60 Mobility Framework is a cross-platform development solution for .NET developers targeting both Windows Mobile and S60. The Mobility Framework publishes one API across both platforms so that device features and services may be accessed using a single uniform API, removing the need for platform conditional invocation.

The API richness of the Mobility Framework includes the following components:

- messaging (send and receive SMS and email)
- location (from GPS)
- PIM (write and read contacts and calendar items)
- call functions (dial, answer and terminate a call)
- OpenGL ES
- vibration.

The Mobility Framework, used in conjunction with Net60, makes low-level S60 device features easily accessible in a managed .NET environment.

8.8.1 .NET and the Common Language Runtime

The Common Language Runtime (CLR) is the byte code engine for .NET. It manages memory, threads, type and code safety. Net60 implements its own version of the run-time engine that is suitable for S60 3rd Edition devices and complies with all the requirements for Symbian Signed. A simplified version of Net60's place in the Symbian architecture is shown in Figure 8.1.

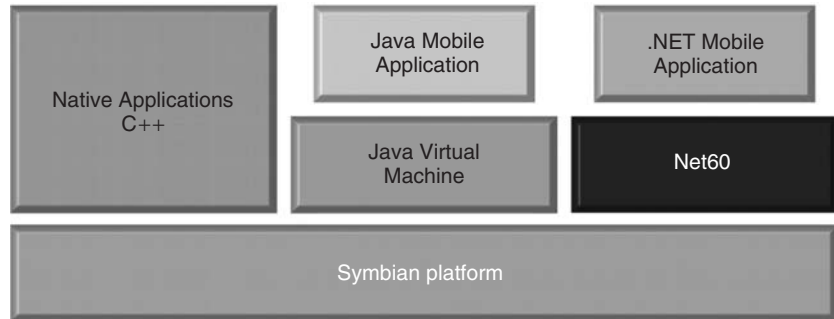


Figure 8.1 Net60 – enabling .NET applications on the Symbian platform

Net60 is compatible with Microsoft Compact Framework 2.0 and supports integrated development in Visual Studio 2005 and Visual Studio 2008. Applications written in C# or Visual Basic can very easily be ported over to Symbian by using Net60. This is particularly true of vertical applications which are built for in-house use and so tend to be developed with less specialized APIs and with maintenance in mind, making them ideal to port to new platforms quickly.

8.8.2 Libraries

To make porting easier, Net60 provides an implementation of the Base Class Library (BCL) and the Framework Class Library (FCL) on Symbian. Together they provide a rich library of reusable, reliable, extensible and tested code for you to use within your applications.

The BCL provides classes to manage common data structures, such as types, strings, dictionaries, vectors and IO operations. The FCL is a layer above the BCL and provides Microsoft-specific functionality, such as the Forms framework and controls.

In addition to providing an implementation of the BCL, Net60 provides managed FCL assemblies for Controls, WebServices, XML and an ADO.NET layer into SQLite. Additionally Red Five Labs also supplies .NET managed assemblies for Symbian, providing APIs into native GPS, Telephony, SMS, Audio and graphics from C# or Visual Basic. As a last resort, if the required functionality is missing, then the P/Invoke API can be used to access unmanaged APIs residing in native code DLLs.

8.8.3 Build Process

To build a C# application, the code is compiled by the C# compiler to an output file containing Intermediate Language bytecode (IL). The CLR

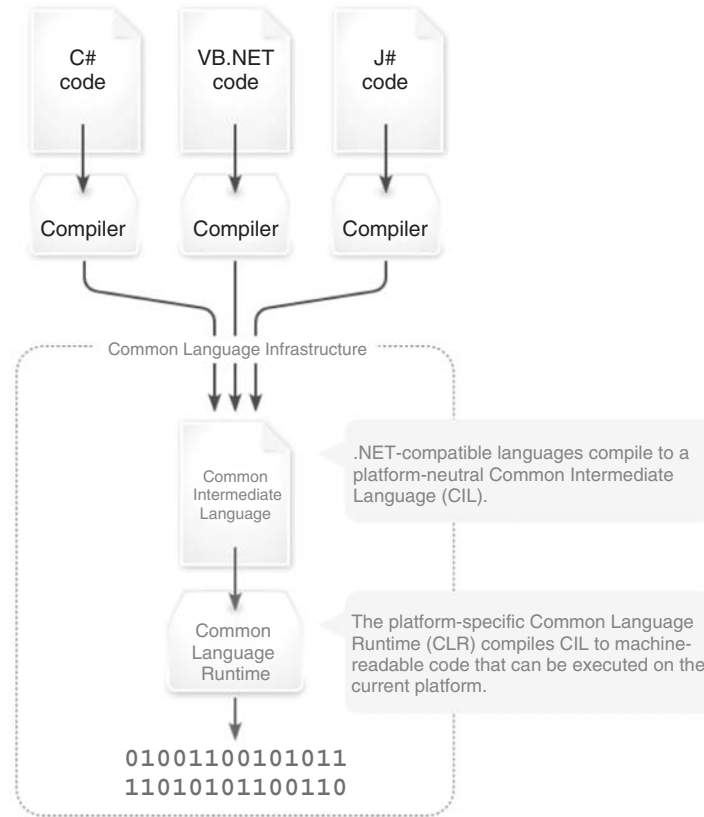


Figure 8.2 .NET compilation process

understands IL and executes it on the target platform. Figure 8.2 illustrates the process of compiling an application into a .NET executable suitable for execution on a target device.

Net60 takes the output of stage 3 and uses that with its own engine and package creation tool, called Genesis, to generate a SIS file that is suitable for deployment to any Series 60 3rd Edition device. All these steps are largely invisible to users as they are done within the Visual Studio IDE.

The creation and development of .NET applications, including accessing services via P/Invoke, is covered in more detail at developer.symbian.org/wiki/index.php/.NET_Quick_Start.

8.8.4 User Interface

One of the areas that is fraught with trouble is the user interface. This is always problematic as different platforms provide their own

implementations and designs for it, making it very difficult to leverage any kind of reuse. This is made more difficult when porting to Symbian as the user interface does not support touch or have a mouse-driven interface. It should be noted however that from 5th Edition there is now a touch user interface and support for touch controls.

Net60 supports a wide range of native Windows Mobile controls. These controls are binary compatible with their Windows Mobile counterparts and behave as if they were Windows controls. The lists below show the binary compatible controls supported by Net60 and the Microsoft Compact Framework 2.0.

Non-Touch Controls

Checkbox
 ComboBox
 Datagrid
 DateTimePicker
 HScrollBar
 ImageList
 Label
 LinkedLabel
 ListView
 MainMenu
 Panel
 PictureBox
 ProgressBar
 TextBox
 TreeView
 VScrollBar
 WebBrowser

Touch Controls

Button
 ContextMenu
 DocumentList
 InputPanel
 ListBox
 MonthCalendar
 Notification
 NumericUpDown
 OpenFileDialog
 RadioButton
 SaveFileDialog
 Splitter
 StatusBar
 TabControl
 ToolBar
 TrackControl

8.8.5 Porting a .NET Application to Symbian

Fortunately this is a much easier task than porting a C++ application. This is because the underlying code is processor agnostic and also because Red Five Labs have kept the same behavior and controls as Windows Mobile. For the most part, this is a seamless port and requires very little change if standard controls are used or controls have been subclassed from existing controls in the FCL.

The first version of the application starts with development targeting Windows Mobile Standard Edition (which does not have support for touch). The application is built in Visual Studio and rolled out to Windows Mobile devices.

When a need for a Symbian version is identified, the existing code is analyzed for compatibility with Net60. Normally this revolves around unmanaged (native) code, rather than C# code. Typical examples include

new platform features, such as location-based services or sensor APIs. Once these sections are identified, they need to be refactored into a common class that can be shared by both platforms. Due to the way .NET dynamically loads classes, the same binary can at run time decide which implementation to choose for the appropriate platform. For example, if GPS support is required then the application can choose to use the Red Five Labs assembly providing the GPS services, rather than the Microsoft solution on native devices.

You can implement this at run time using code similar to the following:

```
if (System.Environment.OSVersion.Platform != Platform.WindowsCE)
{
    // The device is a Symbian device running Net60
}
else
{
    // The device is a Windows Mobile device
}
```

Once the platform-specific code is identified and refactored, the final stage is to add an installer to create the installation file for the device. This involves creating a Net60 Genesis project. In a manner similar to the Setup Project in Visual Studio, Genesis is a project type and gathers the components required from the project dependencies as well as the bootstrap code and CLR binaries into a single package. This package is then built and signed and the resulting SIS file can be installed in the same way as any other Symbian application.

Net60 makes it very easy to take existing code from the Windows Mobile Compact Framework and migrate it onto Symbian devices with the minimum amount of rewriting or refactoring whilst retaining the commercial advantages of C# and the .NET Platform.

For more information on writing device-independent .NET applications or porting .NET applications from the desktop to mobile devices, see msdn.microsoft.com/en-us/magazine/cc163387.aspx or visit the Red Five Labs website at www.redfive labs.com.

8.9 Summary

Unfortunately there is no magic bullet for porting from Windows to Symbian for the majority of applications. Typically, good design practice and implementation in the form of building highly cohesive and loosely coupled applications with numerous module tests will make the most difference when it comes to porting to other platforms, as covered in Chapter 15.

There are some significant points to remember when converting from Windows to Symbian (and vice versa):

- Layered and modular design from the outset allows easier porting from Windows, as both platforms support DLLs to encapsulate functionality.
- C++ structured exception handling and Symbian exception handling should not be mixed since it does not reliably handle resource deallocation.
- When writing new Symbian code, coding conventions are critical. They must be followed to avoid problems in maintenance and to work correctly with the existing APIs and tools.
- Error handling must be stricter. Using invalid handles in Windows is normally benign and returns an error code. On Symbian, however, it results in the application terminating.
- Symbian has stricter security and signing requirements than Windows Mobile.
- Smaller heap memory means applications need to consider reducing memory requirements.
- .NET code can generally be ported much more easily than C++ since the run-time engine uses processor-agnostic byte code and has a rich set of APIs that are largely platform independent, particularly when moving to and from Windows Mobile.

9

Porting from Other Mobile Platforms

We are currently not planning on conquering the world.

Sergey Brin

Like many of the other chapters in this book, the topics covered here could be the subject of one or more separate books. To provide complete guides for porting between the Symbian platform and other mobile platforms such as those discussed here – Android, BREW and iPhone OS – would be a near impossible task. Instead we have focused on providing useful information about the most common issues that you may face when moving from one platform to another.

Additionally, we've attempted to minimize your porting effort by taking a view on the most similar run-time environment to target when porting 'native'¹ code from the originating platform to Symbian. In the case of Android, we suggest porting to native Symbian C++ because, at the time of writing, there is no other match for the rich API set provided by Android. In contrast, the API set for BREW is more limited and so most applications could more easily be ported for Symbian using Java ME. Porting from iPhone OS is somewhere in between and, because Cocoa and the UIKit have some close similarities with Qt in places, we recommend using Qt as the target framework for your Symbian port.

When reading this chapter, it's important to keep in mind that the suggestions presented here represent only a subset of the available options for moving between platforms. Since all of the platforms are still evolving, the simplest targets to port to are also likely to change. Finally, please note that due to significant differences between these platforms, including the primary programming languages for Android and iPhone OS, it is not really possible to 'port' code from one platform to another;

¹ I use the word 'native' in quotes here because for Android the only run-time environment available is Java-based, but it has a similar feature set to genuine native environments on other platforms. For more information see code.google.com/android.

practically every line will have to be re-written. However, it is often still worth considering how to ‘translate’ the original application rather than starting from scratch, so it is in that spirit that we try to provide a mapping between concepts and APIs.

9.1 Android

The Open Handset Alliance (OHA) led by Google was formed with the intent of fostering innovation in the mobile industry with a fresh approach to ‘change the way people access and share information in the future’.² Although there’s only a single device on the market at the time of writing, the T-Mobile G1 manufactured by HTC, OHA’s Android platform is the subject of much interest throughout the mobile developer community.

Android is based on Linux, yet the public APIs that developers can use require knowledge of the Java programming language, tools and environment. Although the format of the bytecode generated by the Dalvik virtual machine is not compatible with any existing Java edition, the target developer group is mainly Java ME and Java SE developers. This makes for an obvious difference in the development environments used by Android developers and Symbian C++ developers. However, the techniques and methods can be ported rather than the actual code, so it is worth having a look at some practical tips.

9.1.1 Development Environments Compared

While developers can write programs for Android on their favorite desktop computer running Windows, Linux or Mac OS X, Symbian C++ development can take place only on Windows. Even though Symbian modules can be built on Linux machines, the set of development tools is incomplete (on-device debugging is not possible, for example) and are unofficial, thus they come with limited support.

At the time of writing, the latest Android SDK is v1.1. This low version number alone clearly indicates that Android is a young platform with only one previous mature version, v1.0. As you’d expect from an early release, the APIs are not yet stable, with some source compatibility breaks between the first two releases.

The idea of a resource compiler always proves useful on a resource-constrained device. Both the Android and Symbian resource compilers transform data into a binary, fast-loading format that is efficient during execution. There are identifiers on both platforms that can be used as references to resources from the code. On Android, `R.java` is the name of the file that contains the identifiers of resources that belong to the

² www.openhandsetalliance.com/member_quotes.html.

application. The resources can be in different files but their IDs are compiled into the same file. This mechanism is a bit different in Symbian, where a developer has to work with three kinds of file connected to resources: you define resources in one or more text RSS files; the resource compiler compiles these files to RSC files so that each RSS file has a corresponding compiled file; RSG files are automatically generated during compilation: they contain identifiers that can be used to refer to resources from code. Please note that there are as many RSC and RSG files as there are RSS files.³

On Android, the most important external library that every program imports is called `android.jar`. This approach is completely different in Symbian: a Symbian application doesn't import Symbian functionality unless it really needs it, in order to keep the size of binaries as small as possible.

Although command-line support is available on Symbian, it's more productive to use an integrated development environment. The official IDE for Symbian C++ development is Carbide.c++ which is based on Eclipse. This is good news, since the preferred IDE for Android development is also Eclipse (NetBeans is also supported by Android), thus most Android developers will find Carbide.c++ comfortable to use. Even though there are differences between these two tools as to what features are supported (mostly due to the differences between the C++ and Java programming languages and build environments), the user experience and core (and a wide range of complementary) features are the same.

When it comes to testing, Android and forthcoming Symbian SDKs provide a PC-based emulation of a typical device which can run the target binaries.⁴ However, developing for any emulator is typically not the ultimate goal: the software must be tested on a real device to see if it works as expected. At the time of writing, the only Android-powered handset on the market is T-Mobile G1 (also known as HTC Dream). However, Google also offers a SIM- and hardware-unlocked device, the Android Dev Phone, that can be purchased from anywhere around the world and used without any limitations.⁵ There are over 250 handsets based on Symbian; in fact, there are so many that the variations between models must be considered carefully (for example, different devices have different screen sizes and input methods, such as touch only, keypad only, or hybrid).

A developer doesn't have to have a real device at hand to test the software. There are web-based services that offer the possibility of uploading and testing applications on remote devices. For example,

³ See [developer.symbian.org/wiki/index.php/Tool_Chain_\(Fundamentals_of_Symbian_C++\)](http://developer.symbian.org/wiki/index.php/Tool_Chain_(Fundamentals_of_Symbian_C++)).

⁴ Note that earlier Symbian SDKs provided only a higher level emulation environment, for which you had to build separate x86 binaries.

⁵ It is not yet possible to purchase copy-protected applications from Android Market with Android Dev Phones.

DeviceAnywhere.com is such a service where developers can test Symbian and Android applications alike. It's worth noting, though, that it's not free. For Symbian developers, the Nokia Remote Device Access (RDA) service is available for testing applications on Nokia devices free of charge. Note that there are things that cannot be tested remotely (such as the accelerometer, GPS, etc.) but the vast majority of features are available.

9.1.2 Android and Symbian Compared

In this section, we compare the basics of the Android and Symbian platforms.

Symbian has a multi-tasking, multi-threading operating system allowing more than one application to run concurrently, and this behavior applies to third-party software, too. Each application runs in a separate process, making code execution more robust and secure. This is the same on Android. However, a Symbian program has a main entry point, the `E32Main()` function, which is called by the system upon application start-up. This is different to an Android application, where there's no method that has a special meaning to the system, but there can be one `Activity` (and only one!) that is interested in getting notification about a special `Intent` with the name of `android.intent.action.MAIN` and category of `android.intent.category.LAUNCHER`.

A typical Symbian GUI application follows the Model–View–Controller (MVC) design pattern.⁶ The model role is usually implemented by a `Document` (or an engine) class; the view role is usually implemented by a `View` (or one or more `CCoeControl`-based classes); the controller role is usually implemented by an `AppUi` class. Please note that a native Symbian GUI application must have at least a `Document` and an `AppUi` class;⁷ the application framework is designed to force the structure of an application to comply with this pattern. Figure 9.1 depicts the typical architecture of a Symbian GUI application.

The architecture of a similarly simple GUI application on Android is completely different: the two keywords that are relevant in this context are `Activity` (a screen component) and `Intent` (used for message exchange between activities, but also used for application launch). An `Activity` usually represents a screen in an Android application. It is one of the fundamental building blocks in the platform. It's very common for users to traverse between several screens during the lifetime of an application and applications usually contain more than one activity. It is possible to switch between activities, send messages back and forth, start activities from within the same application or an external application,

⁶ See en.wikipedia.org/wiki/Model-view-controller.

⁷ It also contains an `Application` class, but that's not relevant from the MVC pattern's point of view.

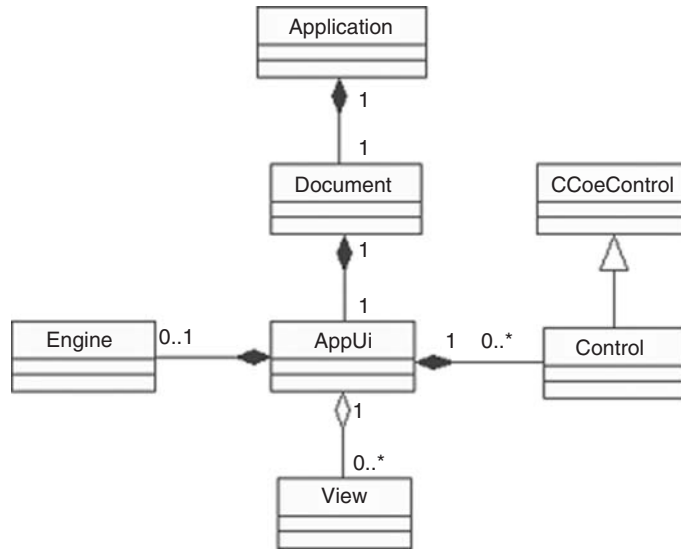


Figure 9.1 Architecture of a typical Symbian GUI application

and so on. One of the activities is typically the main entry point for an application, namely its `onCreate()` method.

Switching between activities is achieved by the use of an `Intent` object that encapsulates the action to perform and the data to act upon. An explicit intent specifies the component that handles the action; in contrast, an implicit intent contains enough information for the system to find and start the right activity for the purpose. Although their use is not mandatory, activities often use intent filters (instances of the `IntentFilter` class) to describe the actions in which they're interested.

Multiple screens are available on Symbian, too, and a similar mechanism is in place to allow developers to implement (and switch between) multiple screens in their applications. The most commonly used solution is to write a view-based application, where views are containers of controls – graphical components which are responsible for drawing operations. This may sound rather confusing so let's put it simply: Android activities can be thought of as views in Symbian and Android views (`ListView`, `ImageView`, etc.) as Symbian controls.⁸ A view-based application architecture is not the only way to write an application in Symbian: there are also dialog-based applications, where focus jumps from one dialog to another, as well as those that implement graphics in a custom way (no dialogs or views, e.g. in a game). This is somewhat different from Android, where activities play a vital role in the system.

⁸ For more information, consult the Symbian Developer Library documentation for the following Symbian classes: `CAknView` (`aknview.h`), `CAknViewAppUi` (`aknviewappui.h`) and `CCoeControl` (`coecntnl.h`).

On Symbian, views can exchange data with one another during a view switch. It's typical that most heap-allocated objects a view owns are freed up right before the view is de-activated; they are then re-allocated and re-initialized only when the corresponding view is activated again. In contrast, a Symbian view is never the entry point of a Symbian application (the entry point is instead `E32Main()`). Permission-checking never takes place on view activation – at least not at view-level – and platform security capability enforcement is done at process-level, when one process exchanges data with another.

There is no mechanism in Symbian similar to Android intents. For view activation, a view identifier can be passed along with some extra data, which can be treated as an intent without encapsulation. However, the intents mechanism is more widely used on Android than in Symbian. Intent filters are not used by Symbian either.

Event Broadcasting

On Android, applications can execute in reaction to external events. An example of such an event is when the system notifies listeners that the time zone has changed. It's worth noting that an application doesn't have to run in order to be able to receive events, but it is automatically started by the system if necessary. Finally, applications themselves can broadcast events.

The publish and subscribe (P&S) framework provides a similar mechanism for Symbian programs: information providers can publish their data and others can subscribe to be notified when that data changes. In addition, P&S is also used to broadcast system events, such as network availability, charger status, current call status, and so on. There is no restriction on whether a P&S client has a UI or not – anyone can listen and publish. Like Android, security enforcement can be enabled for a given event – if the listener doesn't hold the right capability or permission (in Symbian and on Android, respectively) then it simply won't be able to handle it. For the Symbian platform, further information can be found in the Symbian Developer Library documentation for class `RProperty` (`e32property.h`) and in a comprehensive article about data sharing at developer.symbian.org/wiki/index.php/Data_Sharing_and_Persistence_with_Symbian_C++.

Content Providers

Sometimes Android applications want to share the data they operate on with other applications. Since there is no common storage area for applications that all packages could share and applications cannot access each other's private storage area, a new mechanism has been introduced on the Android platform. Content providers follow common conventions

that ensure that querying for data, returning results, and so on is always done in a similar fashion. They do it without revealing anything about how their data is stored internally. A few examples of widely used content providers are the call log, contacts, media store (audio, image, video), and browser bookmarks.

On Symbian, it is possible to provide content for public use, although there's no common framework for doing so. It is possible to hide sensitive application data from curious eyes (private data caging ensures that), but nothing forces developers to expose this hidden data in a consistent way. In other words, each API is different, as described in the article about data sharing found at developer.symbian.org/wiki/index.php/Data_Sharing_and_Persistence_with_Symbian_C++.

Services

It's a common requirement for a program to run in the background for an arbitrary period of time and serve the requests of other processes. In effect, a program acts as a server to satisfy the requests of one or more clients. On Android, the server is called a service. Services don't have a visual user interface. Permissions may be defined to prevent the unauthorized launch or use of a service.

On Symbian, servers don't have a user interface, either. A server may have multiple clients connected to it and each client may potentially have more than one outstanding request to the server at any one time. These requests may be either synchronous or asynchronous, meaning that the client-side program execution is either blocked until the server replies or the client continues to run until it is notified about the result via some callback mechanism at a later point. IPC is done with the assistance of the kernel, which applies to both message exchange and data passing (no process is allowed to read from or write to the memory of another process, except the kernel).

The client interface of a server is not defined by interface definition languages (as opposed to Android's AIDL), but the server's author must provide a client DLL which can be used to interact with the server. In a very low memory situation, the server may be terminated by the system, similarly to Android, but the traditional way for a server to stop running is that it terminates itself when no clients are connected to it.

On Android, the user either *starts* a service and forgets about it or binds to it and makes use of the exposed services. In Symbian, this differentiation simply does not exist and it's always the same API that is exposed to the client. More information about the Symbian client-server framework is available from the Symbian Developer Library documentation and at [developer.symbian.org/wiki/index.php/Tool_Chain_\(Fundamentals_of_Symbian_C++\)](http://developer.symbian.org/wiki/index.php/Tool_Chain_(Fundamentals_of_Symbian_C++)).

9.1.3 Deployment Compared

On the Android platform, every application must be signed. An application can be signed with a self-signed certificate (in fact, this is the typical way) without any restrictions other than that the expiry date be in the future. Signing by a certification authority is not mandatory. The developer can use standard Java signing tools, `Keytool` and `Jarsigner`. The entry fee for selling applications on Android Market, the main distribution channel for Android devices, is currently \$25/year. Upgrades are free, although they must be signed with the same certificate as the original application.

For comparison, in Symbian, signing of C++ applications is mandatory. However, in contrast with Android, the certificate is used for signing and is also used to control what an application can do (self-signing can be used for a subset of all available security capabilities as Chapter 14 describes). Symbian's own tools must be used for signing (`signsis`) and generating keys (`makekeys`). There is no central distribution channel: device manufacturers and network operators use their own solutions for application distribution.⁹

As to deployment, there's another aspect that might be new to Android developers who haven't got used to the number of Symbian devices. Since there are lots of devices, their characteristics often vary, requiring some customization of a deployment package. The Symbian package file is a human-readable file that contains instructions on how the installation file should be built. It influences installation, specifying localization options and the minimum required platform version, for example. In addition, customization can be used to:

- embed additional installation files alongside which third-party code must be installed
- provide device-specific or platform-specific files to handle variations among different devices or device families
- provide condition checking to differentiate installation based on model, device family, memory size, display dimensions, manufacturer, and so on.

Since the security model affects application deployment, it's worth reading Chapter 14, where the differences between the security model of Android and Symbian are discussed.

9.1.4 API Mappings

The following tables contain the most widely used APIs sorted by functionality. Android class names are listed along with their package names and then the corresponding Symbian class names with their appropriate

⁹ Nokia's Ovi Store can be found at <https://store.ovi.com>.

header files and library files in parentheses. These classes and APIs may not fully overlap each other's functionality, however, they're the closest to each other in terms of supported features.

Table 9.1 File I/O APIs: Opening, reading, writing, deleting files, directories

Android Package	Class	Symbian Class	Header (library)
java.io	File, FileReader, FileWriter, BufferedReader, BufferedWriter	RFile, RFs, CFileMan, CDir	f32file (efsrv)

Table 9.2 PIM – Contacts: Adding, querying, changing, removing contacts

Android Package	Class	Symbian Class	Header (library)
android.content.*	ContentResolver, ContentUris	CPbkContact-Engine, CPbkContactItem, TpbkContactItem-Field	cpbkcontact-engine, cpbkcontactitem, tpbkcontact-itemfield (pbkeng)
android.provider.Contacts.*	People	CContactDatabase, CContactChange-Notifier	cntdb (cntmodel)

Table 9.3 PIM – Calendar: Adding, querying, changing, removing calendar items

Android Package	Class	Symbian Class	Header (library)
N/A	N/A	CCalSession, CCalEntryView, CCalEntry, CCalAlarm, CCalDataExchange	calsession, calentryview, calentry, calalarm, caldataexchange (calinterimapi)

Table 9.4 Telephony: Observing incoming/outgoing calls, answering incoming calls, initiating outgoing calls

Android Package	Class	Symbian Class	Header (library)
android.telephony	Telephony-Manager, Phone-StateListener (both for status detection – Android is very limited in its call controlling capabilities)	CTelephony	etel3rdparty (etel3rdparty)

Table 9.5 Networking: Sockets communication

Android Package	Class	Symbian Class	Header (library)
java.net	Socket, Inet- SocketAddress, ServerSocket	RSocket, RSocketServ	es_sock (esock)
		TInetAddr	in_sock (insock)

Table 9.6 Networking: HTTP communication

Android Package	Class	Symbian Class	Header (library)
java.net	HttpURL- Connection, URL	RHTTPSession, RHTTPTransaction, RHTTPRequest, RHTTPResponse, RHTTPMessage, MHTTPTransaction- Callback	rhttpsession, rhttptransac- tion, rhttprequest, rhttpresponse, rhttpmessage, mhttptransac- tioncallback (http)
org.apache. http.*	HttpClient, HttpGet, HttpPost, HttpServletResponse	RStringPool	stringpool (bafl)

Table 9.7 Messaging: Reading messages from Inbox, sending SMS

Android Package	Class	Symbian Class	Header (library)
Android .telephony .gsm	SmsManager, SmsMessage	MMsvSession- Observer, CMsvSession, CClientMtmReg- istry, CMsvOperation, CMsvEntry- Selection	msvapi, mtcclreg, mvstd (msgs)
Android .content	Broadcast- Receiver	CSmsClientMtm CMtmUiRegistry	smsclnt (smcm_gsm) mtudreg (mtur)

Table 9.8 Database: Database management

Android		Symbian	
Package	Class	Class	Header (library)
Android .database .sqlite.*	SQLiteDatabase, SQLiteOpenHelper, SQLiteCursor	RDbNamedDatabase, RDbStoreDatabase, TDbCol, RDbTable, RDbView, RDb, RDbNotifier	d32dbms (edbms)

Table 9.9 Camera: Using on-board camera, taking photos, making videos

Android		Symbian	
Package	Class	Class	Header (library)
Android .hardware	Camera (and its nested classes)	CCamera, MCameraObserver, MCameraObserver2	ecam (ecam)

Table 9.10 Media: Playing and recording audio/video

Android		Symbian	
Package	Class	Class	Header (library)
Android .media	MediaPlayer, MediaRecorder	CMdaAudioPlayer- Utility, CMdaAudio- RecorderUtility	MdaAudioSample- Player, MdaAu- dioSampleEditor (mediaclientau- dio)
		CVideoPlayer- Utility, CVideo- RecorderUtility	videoplayer, videorecorder (mediaclient- video)

9.2 BREW

Binary Runtime Environment for Wireless (BREW) is an application development platform developed by Qualcomm¹⁰ for mobile phones. It is

¹⁰ For more details, visit brew.qualcomm.com.

most widely used on CDMA handsets. Developers typically write BREW applications in C/C++ but the APIs available are relatively limited in comparison with those provided by the Symbian platform. Further information is provided in an article on the Symbian Developer wiki.¹¹

From a portability perspective, BREW provides an implementation of the standard C library, known as AEESTdlib. The naming conventions used in this library mean that you can't compile code written for it directly in another standard C environment but it does translate very easily. For example, consider the following code, which allocates an AECHAR array and uses the AEESTdlib, MALLOC and FREEIF macros:

```
pzFileNameBuf = (AECHAR*) MALLOC( MAX_FILE_NAME * sizeof( AECHAR ) );
if( !pzFileNameBuf )
{
    return ENOMEMORY;
}
...
FREEIF( pzFileNameBuf );
```

In fact, in contrast to Android where it makes sense to port Java code to C++ in order to achieve similar functionality, the features of the BREW environment are similar to those provided by Java ME on Symbian. It makes more sense to port BREW C/C++ to Java ME on Symbian. Since Java ME is supported by a wide range of devices running on mobile platforms beyond Symbian, the port can be deployed widely; it would only make sense to port to native Symbian C++ from BREW in a few rare cases.

Since this book is about C/C++ development on the Symbian platform, rather than Java ME, porting from BREW to Symbian is not discussed further, but information about Java ME on Symbian can be found in a separate title from Symbian Press¹² and on ***developer.symbian.org/wiki***.

9.3 iPhone OS

Since an SDK became available for download by third parties, allowing them to create native applications for the iPhone and iPod Touch devices,¹³ an ever growing number of developers and professional companies have developed add-on software for iPhone OS. The success can be quantified by looking at the number of applications available

¹¹ See ***developer.symbian.org/wiki/index.php/Symbian_C++_Primer_for_BREW_Developers***.

¹² Hayun, R. et al. (2009) *Java ME on Symbian OS: Inside the Smartphone Model*, John Wiley & Sons.

¹³ The SDK was officially announced on March 6, 2008.

on Apple's App Store¹⁴ (the single point of sale for additional applications), which listed over 65,000 applications and had passed 1.5 billion downloads by 14 July 2009.¹⁵

Presumably you are a direct or indirect contributor to this number and would like to know how to make your applications run on Symbian devices as well. In this chapter, after a short comparison of the two platforms, we discuss the portability of iPhone OS applications in general and examine the choices that could minimize the effort required to make the port. As a programming aid, you may find it useful to see how to map fundamental iPhone OS programming techniques and APIs to their Symbian platform counterparts.

9.3.1 iPhone OS and Symbian Compared

Both iPhone OS and the Symbian platform are multi-tasking, multi-threading and preemptive, and allow third-party software to be installed, but they provide a different level of openness. On iPhone OS, not all subsystems are opened up for installable applications and there are rules for what you can do even where the APIs are available (see the iPhone SDK Agreement accompanying the SDK). In Symbian, almost all user-side subsystem APIs (with the exception of the installer) are ready for use by developers, although a number are protected by a requirement for a platform security capability, and others are marked as non-public indicating that they may change, affecting compatibility.

Cocoa Touch¹⁶ is the environment to use for developing iPhone UI applications. It is a set of high-level Objective-C APIs comprising the UIKit framework and the Foundation framework. The Foundation framework, as its name suggests, is a collection of commonly used non-UI classes such as string and container classes, and others that wrap resources managed by the operating system, such as threads, locks and more. With the use of the Foundation framework, it is possible to write an iPhone OS application fully in Objective-C, using UIKit for the UI, basing lower level code on Cocoa. iPhone OS, being a subset of Mac OS X, brings standard C/C++ and support for a number of POSIX APIs. Thus, where Cocoa Touch is not enough, the gaps can be filled with standard C++ or POSIX C code.

9.3.2 Porting Scenarios

The simplest porting scenario is when your application's engine was written in standard C or C++. This may be a consequence of the fact that

¹⁴ www.apple.com/iphone/appstore.

¹⁵ See www.apple.com/pr/library/2009/07/14apps.html.

¹⁶ When we discuss issues that also hold for Apple's desktop environment, called Cocoa, the 'Touch' is omitted for brevity.

the application itself is already a port. If that is the case, you may want to consider going back to the original version instead of using the iPhone OS code. Chapters 2 and 4 can give you guidance in porting this part of the application.

If you are porting a game, it may need special treatment. Although OpenGL ES is available on the Symbian platform, most devices don't have hardware acceleration, and you will have to check if your game will run at the required frame rate (see Chapter 6 for more details).

If you are porting code written in Objective-C (using Cocoa Touch), you will have to re-write it from scratch for Symbian devices. The good news is that Qt¹⁷ will shortly be supported by Symbian devices (probably many of those based on the Symbian^3 release in mid-2010 and all devices in the Symbian^4 release in late 2010).

The similarities between Qt and Cocoa include the following:

- the Foundation framework and `QtCore/QtNetwork` module classes
- reference counting
- Qt signals and slots, which are analogous to the Cocoa Target–Action solution
- inter-thread communication
- introspection, with the property feature of Qt.

In addition, porting to Qt has a clear advantage if you plan to port your application to other platforms, since Qt is a cross-platform framework which will shorten any subsequent port efforts. However, porting to Qt still requires redesign and may require brushing up your C++ skills!

Qt is based on C++ but it adds the meta object compiler (MOC), which extends objects with information that is available at run time. It's possible to make method invocations by name (represented as a string); you are unlikely to use this often, but it is a trait of Qt that resembles Objective-C selectors. The signals and slots mechanism (see Section 6.4) makes it possible to connect composed form elements to developer code in the same way that Cocoa uses the Target–Action paradigm.

9.3.3 Cocoa Foundation Framework and Qt Compared

The general-purpose data structures and access to some basic I/O are encompassed by the Foundation framework in Cocoa Touch. In Qt the equivalents are mostly, but not exclusively, found in the `QtCore` library. Below, we take a quick look at these classes to see what you can map your Cocoa Foundation class into. It is not intended to be an exhaustive description of the classes. For details, please consult the Qt reference.¹⁸

¹⁷ See Section 6.4 and Chapters 10, 11 and 12 for more information.

¹⁸ See doc.qt.nokia.com/4.5.

Base Class

The equivalent of `NSObject` for Cocoa is `QObject` for Qt. This class is the base class of all classes in Qt. It carries the extra metadata that Qt uses for introspection and signal–slot messaging. The base class also remembers the thread in which it was created (and messages are delivered to it in that thread).

`QObject`s form a hierarchy, where the parent `QObject` instance automatically cleans up its children. A `QObject` encapsulates a periodic timer that calls back the `QObject` that has started it via the thread's thread message queue.

The metadata generated for base class instances makes it possible to use Qt's lightweight type identification. As C++ is a statically typed language, it is not possible to send messages to objects whose static types would not be able to handle it (i.e. they do not implement the appropriate method). Often the semantics of the code allows you to assume a certain dynamic type for an object. Qt offers the `qobject_cast<Type*>(object)` cast operator to safely do the casting in these cases. This returns `NULL` if the object does not have the appropriate dynamic type or returns the object cast to the specified `Type`.

String Handling

In place of the `NSString` and `NSMutableString` that you would use in your iPhone OS code, you should use Qt's `QString`, which is based on `QChar`, a 16-bit character type used to store a Unicode character. Differences between `QString` and `NSString` and `NSMutableString` include:

- Wherever `QString` methods accept `const char*`, they are treated by default as C-style ASCII character strings (this behavior can be changed by calling the `QTextCodec::setCodecForString()` static method), whereas in Cocoa you are expected to specify the encoding.
- `QString` uses implicit sharing with copy on write (see Chapter 6); with `NSMutableString`, a developer can decide based on the situation whether to spawn a copy or just increase the reference count.
- A `QString` is best created on the stack, like a fundamental type. Returning `QString` by value is fine too as the copy constructor just increases the reference count on the shared data to which the `QString` object holds a pointer.
- `QString` makes use of the C++ language features operator overloading and implicit conversion. As a result, you end up with simpler and more readable code than you typically get with Cocoa.

Initialization of a `QString` object is as easy as it can be. The Objective-C line:

```
NSString* myString=@"Porting is easy";
```

turns into the following Qt code:

```
QString myString="Porting is easy";
```

This uses the constructor that accepts a `const char*`. It assumes that the passed string is ASCII. Also `QChar` arrays may be used to construct the string. For safety reasons, the array is copied (use the `QString::fromRawData()` method to avoid initial copying, as long as the `QString` does not need changing).

A literal can always be passed to functions where `QString` is expected because the constructor makes a `QString` of a `const char*`. In the other direction, care has to be taken and it can be accomplished with the `qPrintable` macro, which uses the codec most suitable for the current locale.

For string comparison, `QString` overrides the `==` and `!=` operators (multiple times). For example, you can do something as natural as:

```
if (myString == "Porting is hard"){  
    //...  
}
```

A Cocoa version would look like this:

```
if ( [myString isEqualToString:@"Porting is hard"] ){  
    //...  
}
```

The `>=` and `<=` operators can be used too but using `QString::compare()` provides more information in one function call.

The string manipulation functions of `QString` are explanatory in most cases. The functions you are used to in Cocoa are provided, such as those for inserting, appending (also in the form of the `+=` operator), removing and replacing characters. Each method returns a reference to the changed string object for convenience, so that operations can be chained.

Some additional `QString` methods (always `const` methods) return a new `QString` object while leaving the original string unchanged.

Some representatives of this category are the `QString::left()`, `QString::right()` and `QString::mid()` methods that make a copy of a substring of the original string.

`NSString` and `NSMutableString` cooperate nicely with `NSData`, which is the container of binary data on Cocoa. The same is true for the Qt analog of `NSData` and `NSMutableData`, called `QByteArray`.

It is possible to iterate through a string and work on the individual characters making it up, using the `QString::at()` method or the `[]` index operator as well as STL-style iterators. This requires some extra care as some Unicode characters can only be described by two 16-bit `QChar` data. It should be noted that the version of the index operator that returns a `QCharRef` allows the caller to modify the character in the string as if it was a `QChar&`.

There are some classes that are strongly connected to strings and are fairly commonly used. For example, `QStringList` is a list of strings and is inherited from `List<QString>` (see the Containers section on page 260). It is more or less equivalent to using `NSArray` for storing `NSString` objects. A call to `QString::split()` would give you an object like this. It has a feature that may come in handy when developing a UI with an incremental search field that allows filtering. It can create a new list of the existing items if a match criterion is given. Filtering is done with the two overrides of the `filter()` methods.

There is no strict equivalent of `NSScanner` in Qt. However there is a fairly elegant way to process a string iteratively. You can make a `QTextStream` based on a string and then use the standard C++ stream operators:

```
float f;
int i;
QString s;
QTextStream input("0.32 23 apple");
input>>f>>i>>s;
```

In contrast, the Cocoa version with `NSScanner` takes the following shape:

```
float f;
NSInteger i;
NSString* str;
NSScanner* scanner = [NSScanner scannerWithString:@"0.32 23 apple"];
[scanner scanFloat:&f];
[scanner scanInt:&i];
[scanner scanCharactersFromSet:[NSCharacterSet
                               alphanumericCharacterSet] intoString:&str];
```

Containers

Qt provides a wealth of container classes. Table 9.11 helps you to decide which one to use in specific cases.

Table 9.11 Qt containers

Cocoa Container	Qt Class	Short Description
NSArray NSMutableArray	QList<T>	A list of items that uses contiguous memory, with fast access by index and quick append and prepend operations
NSArray NSMutableArray	QLinkedList<T>	Similar to QList, with elements accessible through iterators and faster insertion and removal
NSArray NSMutableArray	QVector<T>	A dynamic array implementation for which prepending incurs a performance penalty compared to QList, which is usually a better choice
NSCountedSet		There is no corresponding class – you will probably have to write it yourself
NSDictionary NSMutableDictionary	QMap<Key, T>	A dictionary of key–value pairs, internally sorted by the keys; one key can be associated with multiple values, however QMultiMap has a more convenient interface
NSDictionary NSMutableDictionary	QHash<Key, T>	A faster version of QMap In which the key order is not maintained
NSSet NSMutableSet	QSet<T>	A set implementation in which no multiplicity of elements is maintained

Further information about Qt containers can be found in the Qt reference.¹⁹

An example of enumerating items in Cocoa containers is as follows:

```
for (NSMutableString* str in list){
    //do something with str
}
```

In comparison, Qt takes the same approach as standard C++ and provides iterator classes: STL-style `const` and non-`const` iterators. Java-style iterators are also available however they are not as efficient as

¹⁹See doc.qt.nokia.com/4.5/qt4-tulip.html.

the STL ones. Here is a simple example of how you could multiply all elements by two in a list of integers using an STL-style iterator:

```
QList<int> list;
...
QList<int>::iterator i;
for (i = list.begin(); i != list.end(); ++i)
    // do something with *i
```

Alternatively, to enumerate the items in the container, Qt provides `foreach`:

```
QLinkedList<QString> list;
...
foreach (QString str, list){
    //do something with str
}
```

In Qt, algorithms such as quick sort and binary search are also based on STL-style iterators. The comparison of elements relies on the overriding of C++ comparison operators rather than on helper classes such as `NSSortDescriptor` in Cocoa.

Basic I/O

In Cocoa, the Foundation framework provides a small set of classes that deal with basic I/O. These classes are `NSFileHandle`, `NSStream`, `NSInputStream`, `NSOutputStream`, `NSFileManager`, `NSData` and `NSString`. The latter two can simply be initialized from the content of a file. In this case, the content of the file is fully read into these objects and further manipulation can be done by the objects (e.g. splitting the string into lines and feeding them into the cells of a table view). `NSStream` is mainly used for socket-based communication as there is no advantage in using it for file I/O over `NSFileHandle` other than serving as an abstract layer every now and then when the underlying channel can be either socket- or file-based. `NSStream` works only with three devices: sockets, files and byte buffers; if further devices are needed, they must be implemented in user-derived subclasses.

Qt is stream-focused; its container classes can be manipulated with stream operators. For example, the following code appends 3, 5 and 7 to the end of an integer list:

```
list<<3<<5<<7;
```

In Qt, behind each stream there is always a `QIODevice` object that represents devices that support reading and writing blocks of data. `QFile`, `QBuffer`, `QTcpSocket` and `QNetworkReply` classes are notable examples of these devices. In Cocoa, there is no such base class for devices; instead the ways of use are built into the `NSInputStream` or `NSOutputStream`, which can be opened on a file or on a buffer (`NSData`).

Events on the stream are delivered as messages to the registered delegate in Cocoa. In Qt this is implemented by the `QIODevice` emitting a signal to which your code can connect a slot. For instance, you can get notification if there is available data on a TCP stream by connecting a slot to `QIODevice::readyRead()` signal.

A `QDataStream` can be opened on those devices providing a single interface for all I/O operations in a device-independent way. `QDataStream` implements serialization of basic data types as well as other Qt classes with `>>` and `<<` operators. Serialization has seen quite a lot of changes as Qt has evolved; to keep consistency between externalized and internalized data, it is important to indicate the Qt version when using more complex types using the `QDataStream::setVersion()` method. Here is a short example:

```
int score;
int level;
...
QFile file("gamestate.bin");
QDateTime dateTime = QDateTime::currentDateTime();
file.open(QIODevice::WriteOnly);
QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_0);
out << dateTime.toString();
out << level;
out << score;
...
```

Reading from the file is done similarly, simply by reversing the stream operator.

Networking

Qt networking classes can be found in the `QtNetwork` module providing low-level and high-level solutions for exchanging data over a network.

Cocoa and Qt networking classes are similar. In Qt, common application-level network access such as HTTP, HTTPS and FTP is encapsulated in the `QNetworkAccessManager` class, which can be considered the Qt equivalent of `NSURLConnection`. Whereas in `NSURLConnection` the event handler is the specified delegate, `QNetworkAccessManager`'s events can be reacted upon in the registered slots.

`QNetworkAccessManager` operates on a request object of `QNetworkRequest` (compare with `NSURLRequest`) and the reply is wrapped in a `QNetworkReply` object (compare with `NSURLReply`). `QNetworkRequest` is an implicitly shared class that holds a URL, request headers and relevant attributes.

Events are delivered through the network access manager object's signals:

- `authenticationRequired()`
- `proxyAuthenticationRequired()`
- `sslErrors()`
- `finished()`.

Authentication-related signals are expected when using HTTP and the server requires basic or digest authentication. The signal provides a pointer to a `QAuthenticator` object that needs to be filled in with user credentials. The credentials are then cached internally to the network access manager. In Cocoa, a similar thing happens except the credentials are managed (when you allow the framework to store it) by a shared instance of the `NSURLCredentialStorage`, which uses the underlying user key chain to hold the permanent credentials.

The `sslErrors()` and `finished()` signals are also available in `QNetworkReply` and that is a convenient way to handle them if you expect only one reply. Lower-level errors are also delivered by the `QNetworkReply` object's error signal. `QNetworkReply` collects the data received from the network and emits signals that indicate the progress for downloads and uploads.

Since `QNetworkReply` is a subclass of `QIODevice`, it is possible to open an input stream on a `QNetworkReply` object and process the incoming reply in chunks, when needed. As with other devices, data arrival is signaled by `readyRead()`, for example when a web server sends a response with chunked transfer encoding.

`QNetworkAccessManager` comes with some useful satellite classes that can help developers create fully fledged HTTP clients. `QNetworkCookieJar` is a concrete base class that implements in-memory cookie storage. Persistent cookie storage can be implemented by subclassing this class. `QNetworkAccessManager` manages cookies internally with the help of its associated cookie jar. `QNetworkCookieJar` is somewhat similar to `NSHTTPCookieStorage`.

In Cocoa, `NSURLCache` is a complete persistent cache implementation. A different approach is used by Qt. `QAbstractNetworkCache` is a cache interface that is used by the network access manager to save cacheable content and build up replies based on cached data as much as possible. This class is intended for derivation and is not

implemented in Qt 4.4; however, an implementation is provided from Qt 4.5 onwards.

In Qt, just as in many other software environments, sockets are a means of transport-level communication. `QAbstractSocket` provides an asynchronous object-oriented communication endpoint. `QTcpSocket`, `QSSLSocket` and `QUdpSocket` are well-known subclasses.

Threading at a Glance

Both Cocoa Touch and Qt provide some level of threading control, building a messaging system on top of low-level thread support. This is an area where a short comparison may prove useful. Using your own threads should be the last resort: if possible, use the asynchronous APIs provided by Qt (e.g. `QNetworkAccessManager`).

In Cocoa, threads are managed via `NSThread` objects. Threads can be started in one of two ways:

- Fire and forget using the `detachNewThreadSelector:toTarget:withObject` class-level message, which creates a new thread and sends the specified message to the object passed as the `toTarget` parameter. The last argument is simply passed to the message as a parameter. A simpler version is `NObject's performSelectorInBackground:withObject` which does the same but the message is sent to the current object.
- Sending the `start` message on a thread that is initialized with either `initWithTarget:selector:object` or `init`. In the latter case, the `main()` method is invoked.

The second option is more generic and the closest to the way it is done in Qt. In Qt, threads can be accessed by `QThread` instances using the usual thread-related primitives to start the thread, query its state, modify its priority and send it to sleep. In Qt, the entry point of the thread is the `QThread::run()` pure virtual method, which you must override. The thread itself is started by calling `QThread::start()`.

Both frameworks support the concept of a run loop or, in Qt terminology, an event loop. These are loops that wait without consuming any CPU time (practically sleeping) until an event happens and then call the right handler for that event.

Setting up a run loop in Cocoa requires more effort unless the thread is the application's main thread. Usually a while loop is used in which individual iterations are called. One such iteration waits for signals and then reads the relevant event sources (queues) and processes the data associated with one event found on them. Timer sources provide event delivery in a synchronous manner (no queuing), whereas selector sources contain serialized selectors to call within the thread thus enabling the

sending of messages from one thread to another, without the first thread needing to wait for the message to be posted. A port source is a base for connection-oriented communication between two threads. A run loop iteration has a mode associated with it that defines what sources are checked, so it can practically act as a filter on the events that occur. In addition to setting up your run loop, for secondary threads Cocoa demands you to manually set up an automatic release pool to avoid leaking memory. An exhaustive guide to this topic is available on Apple's developer site.²⁰

In Qt, the loop is there for you and can be started by calling `exec()` from your `run()` method implementation, when you need a loop. Qt's API is easier to use for this purpose, although it does not give the developer as much control. However, it is fit for purpose in most common scenarios (and is very close to the native Symbian platform code). As stated previously, a thread's event loop can be started with the `QThread::exec()` call. Before starting the event loop, we must guarantee that some event will happen and there is a handler for it in the code, otherwise the loop will just wait forever. In the most common scenarios, Qt's events come as signals and the handlers are the slots (only true if the `QObject::connect()` does not use `Qt::DirectConnection`, which would simply call the slot synchronously from the thread in which the signal happened). Qt also serializes slot functions to call them when the target thread's event loop can handle them. Using this approach ensures that the emitting thread is not blocked during the execution of the handler unless this is explicitly requested during `QObject::connect()` by specifying `Qt::BlockingQueuedConnection` as the connection type. Why is it relevant? Because it works in exactly the same way as `performSelector:onThread:withObject:waitUntilDone` messages in Cocoa.

The most common synchronization utilities are also available in Qt. `NSLock`'s equivalent is `QMutex`. `QSemaphore` has no Cocoa equivalent, the closest is `NSCondition`, but this can be more easily translated into `QWaitCondition`. `NSRecursiveLock` has no Qt counterpart. Only a read-write lock is present in Qt (`QReadWriteLock`).

Both systems support the concept of thread-local storage: in Cocoa, the `NSThread` class gives access to a dictionary for storing user data and Qt encapsulates it in the `QThreadStorage` template class.

`QThreadPool` is a utility class that maintains a limited number of threads and keeps them alive for recycling after they have completed their run. It works on instances of classes implementing the `QRunnable` interface.

Qt has no equivalent of `NSOperationQueue`, which runs operations in an order that is calculated from the developer defined dependency

²⁰ The Threading Programming Guide is at developer.apple.com/documentation/Cocoa/Conceptual/Multithreading.

relation between them in a specified number of threads. Internally, it recycles threads as `QThreadPool` does, but it is more versatile.

9.3.4 Porting the User Interface

UI programming is a rather wide category and we don't cover it in detail here. The purpose of this section is to indicate some of the issues you need to consider.

Supporting only touch-based Symbian devices may seem to be a natural choice when porting from iPhone or iPod Touch devices. However, it heavily restricts the number of compatible devices you can target, since the Symbian platform is also available on a large number of handsets that do not have touch screens. If you decide to support some or all of these you must consider how the input capabilities of a device affects the design of your UI state transitions. For example, putting the focus on an active item (e.g. a line editor) using the navigation keys is not user friendly when compared to a simple selection on a touch screen device.

When porting to Symbian, it is worth using the relevant UI style guides as not only does your application need to be usable on its own but it also should resemble other 'well behaved' applications on the device. For Nokia devices, the style guides can be found on the Forum Nokia website (www.forum.nokia.com).

Tools

Both platforms offer great tools for application development. When you create a new Qt project, the Carbide.c++ IDE prompts you to select the boilerplate code your project will be based on. QtGUI Main Window templated projects correspond to the window-based application selection in XCode. They both relieve the developer from creating the code that initializes the GUI application, including starting its event loop and creating the empty main window.

Qt Designer is a similar tool to Apple's Interface Builder, which is used for populating your UI with available building blocks and connecting up your own code with them. In Qt, the composed UI is stored in one or more XML files with `.ui` extensions. One notable difference between UIKit and Qt is that, in Qt, this file is processed at build time creating a source file in which the `setupUi()` method reflects the connections defined in the Qt Designer; in UIKit, the generated NIB files are loaded into memory and processed at run time.

Initializing the Application

In both frameworks, application initialization is done for you when you create a new project. You hardly ever have to change the way this is done. In UIKit, the following code does the initialization:

```
int main(int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

In Qt the main function looks like this:

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MyMainWindow w;
    w.show();
    return a.exec();
}
```

Note that Cocoa creates the main window for you from the `Main-Window.xib` file, whereas with Qt the main window is created explicitly in the main function. It is not actually an object of the `QMainWindow` class that is instantiated but a subclass of it (when it is created by Carbine.c++, it is automatically named after your project). The slots of this subclass serve as the first event handlers coming from the underlying widget hierarchy.

UI Building Blocks

UIKit and Qt come with common building blocks, called views and widgets respectively. These representation layer classes can draw themselves and change their states on reception of user events. They also notify the handlers of relevant higher-level, widget-specific events (e.g. in the case of a button, such an event is that the button was clicked) and they have a well-defined semantics as to how they expect data to be displayed.

Some typical examples include buttons, lists, containers, line editors, color pickers and scrollbars. The full set of available Qt widgets can be examined in the Qt Widget Box tab of the Qt Designer. Qt widgets derive from the `QWidget` class.

The top-level widget (view) in both environments is the main window that contains further widgets (views). In UIKit, the main window is a `UIWindow` instance, it has the distinguished role of being the entry point for UI events in the view hierarchy, which it dispatches further along the responder chain it maintains. In Qt, the `QMainWindow` is derived from `QWidget` and thus serves as a top-level container widget.²¹ In neither

²¹ Note that `QMainWindow` is not the only possible choice; the top-level container can be any Qt widget.

UIKit nor Qt do you have to set up the main window manually if you use the appropriate templates as described in the ‘Tools’ section above.

During the UI design phase, widgets can be picked from Qt’s widget box tab and dragged to the designer editor. With the application of layout elements and spacers (these are not widgets even if they appear in the Widget Box tab of the Qt Designer), you can define how each element changes its position when its container is resized. In UIKit, the same would be partly accomplished by using the Interface Builder’s Size Inspector tab and changing the auto-resizing options.

Individual widgets can be fully customized through the Qt property editor, where you can set such things as icons, fonts, shortcuts (for buttons) and a whole lot of other widget-specific attributes. The properties, along with the defined widget hierarchy, are saved in the `.ui` file. There is one big difference between the way these descriptor files are handled by Cocoa and Qt: the `.ui` file is compiled into a C++ class file at build time. This class has a special `Ui` name space and is instantiated by your main window class. The individual widgets can be accessed in your main window class through its `ui` data member. In Cocoa Touch, you connect the views of the hierarchy you want to access to the appropriate outlets of your source (this can be done from within the Interface Builder using Control–Click on the views), which can usually be found in your controller classes.

Writing your own Qt widgets is also possible. When you implement your own `QWidget`-derived class, it needs to:

- set its preferred size by re-implementing `sizeHint()`
- set its size policy using the `sizePolicy()`
- draw itself properly when requested by overriding `paintEvent()`
- handle events in which it is interested by overriding the relevant event handling methods, for example, `mousePressEvent()`.

Routing Widget-specific Events to Handlers

In Cocoa, you use the Target–Action pattern to send high-level events from the UI to handlers in the form of message passing. In Qt, the usual way is that you define the slots of the main window class instance that capture the signals sent from within the hierarchy of the widgets stored in the `.ui` file. You could connect widget signals with slots in other objects, but if you decide to do so, you must do the connections manually whereas with the proposed way you can use the Signal Slot editor of the IDE. An immediate consequence is that in Qt, controllers are generally implemented in custom subclasses of `QWidget`.

Event delivery is another example where message passing is replaced with the signal and slot mechanism of Qt (we have seen the same happening in case of inter-thread message passing). To make things easier, Qt automatically connects a slot to the signal if you use the following naming convention on the slot in the main window class:

```
private slots:
    void on_<sender name>_<signal name>();
```

where <sender name> is the name of the widget (you can set it in the property editor) that emits the signal. The <signal name> is the name of the signal function as defined in the header of the widget.

Variants of the Model–View–Controller Design Pattern

A good design separates the code where data is stored (the model) and the logic that manipulates it from the components that present that data to the user (the view). A model exposes its API which client code can use to query and manipulate data and request notifications when data is changed. For example, in a phone book application, the model can return the current contact count, makes it possible to iterate over the contacts and query the details of a given contact as well as edit or delete an existing contact and add a new one. The benefit of a clearly separated model lies in its reusability.

The third fundamental component in the Model–View–Controller (MVC) pattern is the controller. There are well-defined paths of interaction between the three components: in the original MVC pattern, the controller is the object that takes information from the view as a response to a given UI event (e.g. a button was clicked) and changes the model based on it. The view directly queries the model for the data it needs and also receives notification from the model when its state changes.

In Qt, the controller and the view are merged; in Cocoa, the controller becomes a mediator between the view and the model (i.e. the view and the model are fully decoupled). In Qt, `QListView` and `QTreeView` require the use of the MVC variant. In Cocoa, `UITableView` demands that you use the decoupled variant when you choose to subclass the ready-made `UITableView` controller class.

A `QListView` is a list widget that obtains its data for display from its `QAbstractListModel` object. The view requests the model to map a ‘view position’ (its row and column) into a `QModelIndex` object, which acts as some kind of item ID for the view to pull further information from the model about the given item.

9.3.5 Mobile-Specific APIs

Mobile-specific APIs provide access to features and data provided by the platform. This typically includes several items that you wouldn't get as standard in a desktop environment. On Symbian devices this includes:

- user data, such as call logs, contacts, media file, profile and settings
- data (processed or raw) from built-in devices, such as cameras, GPS sensors and microphones
- device state information, such as signal strength or battery level
- telephony services.

Developers of iPhone applications have access to similar functionality via APIs such as `AVRecorder`, `ABAddressBookRef`, `CLLocationManager`, `UIImagePicker` and `UIAccelerometer`.

For Symbian devices, you can access almost any feature supported by a device using native Symbian C++ and this can be mixed with Qt code (carefully). However, to avoid the need to learn Symbian C++, Qt wrappers for the most popular APIs are being provided. These are first prototyped as part of the Mobile Extensions project described at the end of Chapter 6 and should eventually move to fully cross-platform APIs as part of the Qt Mobility project.²²

9.4 Summary

In this chapter, we have covered the options for porting to the Symbian platform that provide the best trade-off between porting effort and functionality from Android, BREW and iPhone. These trade-offs may change over time: for example, as the Qt environment is better supported and extended to cover more mobile-specific functionality, it may make more sense to port an Android application to Qt rather than to native Symbian C++. However, it's also possible that an Android-like environment could be made available on the Symbian platform in future since there is at least one effort to port the Dalvik VM from Android at the time of writing. Similarly, there is already a Qt port to Cocoa on Mac OS, which could be fairly easily adapted for the iPhone OS, but Apple currently doesn't allow the distribution of other run-time environments for the platform, so no such port is planned.

This is a very fast-changing area and although we have checked details at the time of writing, please check the latest status of your selected platforms before deciding on a porting strategy.

²² See labs.qt.nokia.com/page/Projects/QtMobility.

10

Porting a Simple Application

Engineers like to solve problems. If there are no problems handily available, they will create their own problems.

Dilbert

This chapter provides an example of the porting process described in Chapter 2 applied to a complete working application with a user interface. Full source code for this example project can be found on the book's wiki page at developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform.

Since the most portable UI toolkit available for smartphones is Qt, I restrict the project selection to projects which make use of it. I also introduce the workflow, tools and build files involved in a typical Qt-based project on the Symbian platform. At the end of the chapter, I describe how to extend the port to use mobile-specific controls via native Symbian C++ APIs.

By the time you've read this chapter, you should understand:

- how to use Qt on the Symbian platform
- how Qt integrates with the Symbian build process
- how to manage platform-specific extensions in a cross-platform project
- how portable a well-written Qt project can be.

10.1 Selecting a Project

The more portable the code you are working with, the less the effort required to get something working on a new platform. This makes it

possible to use a much more complex and functional example than the typical ‘Hello World’ application that you find in most introductory examples.

In the rather artificial situation of choosing a project for a simple porting example, I have the primary requirement of choosing something that will be easy to port. After that there is the somewhat conflicting requirement that it be sufficiently complex to be interesting. It should also be reasonably well written in the first place so that it makes good example code. Additionally, in this case, I wanted a project to which it would be fairly natural and easy to add a platform-specific extension, in order to demonstrate hybrid coding techniques in Qt. Finally, it ought to be something fun because it’s more motivating for the learning experience – and it sells more books! A simple game sounds like it would be ideal.

Anyone looking for a practice porting project using Qt need go no further than www.qt-apps.org where there are details of hundreds of free applications developed using Qt. Sadly, free games are not quite as common as other free applications and at the time I checked there were only 21 projects in the games section. With a thought to using the accelerometer, I checked the Arcade subsection. Just three games remained, all of which looked promising. I was immediately drawn (due to a misspent youth playing the original) to The Battle for Birera, an Elite remake. Unfortunately it was only available for Qtopia Phone Edition,¹ in many ways a good thing due to the similar target hardware but not 100% compatible with the version of Qt ported to the Symbian platform. There was also some doubt about the licensing for the project and its legality as it is based on Elite-TNK,² which has been withdrawn from distribution.

Another tempting prospect was Flyer 2D, a game inspired by Sopwith³ (which has great retro appeal from my very early PC days). This project was discarded due to being in an early alpha state and using an open source physics engine called Box2D⁴ that has not yet been ported to the Symbian platform (although it should be platform independent) and makes heavy use of floating-point mathematics. As discussed in Chapters 2 and 6, hardware floating-point support is not available in most current smartphones and is not enabled by default in the Symbian toolchain. This forces the use of software emulation for floating-point operations which is very slow and power-intensive.

¹ Qtopia was re-branded Qt Extended in October 2008 and was a more complete UI and application platform for embedded Linux devices. It was discontinued as a standalone product in March 09. See www.qt.nokia.com/about/news/qt-software-discontinues-qt-extended for more information.

² See www.christianpinder.com/games for details.

³ See [en.wikipedia.org/wiki/Sopwith_\(computer_game\)](http://en.wikipedia.org/wiki/Sopwith_(computer_game)).

⁴ See www.box2d.org.

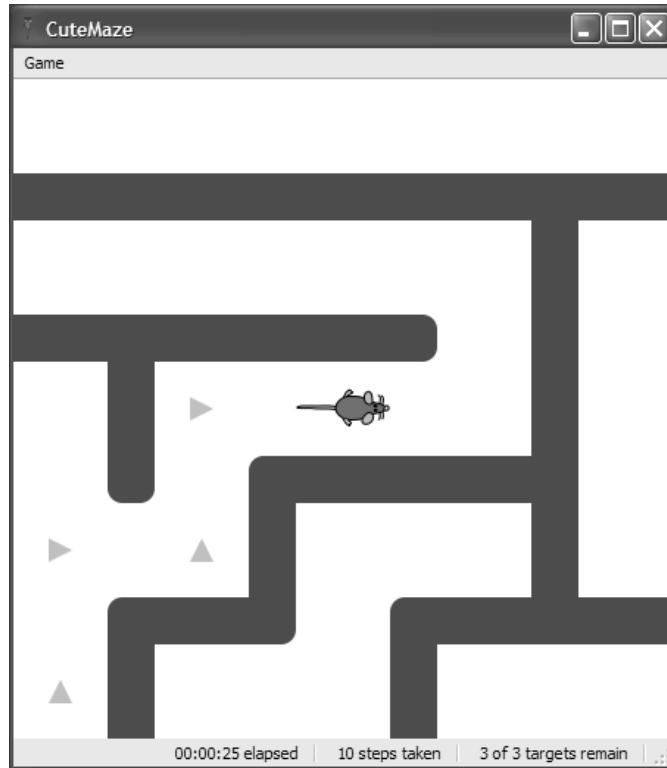


Figure 10.1 CuteMaze running in a desktop environment

The remaining project was called CuteMaze, a top-down maze game that can generate random mazes (see Figure 10.1). Players move a character through the maze hunting for targets. The entire game can be re-skinned or themed with a simple set of SVG files, ideal for re-scaling the content to a mobile screen. The only dependency is Qt 4.3 or later and there are downloads for Linux, Mac and Windows, proving the portability.

10.2 Analyzing the Code

As I mentioned in Chapter 6, Qt projects have a platform-independent project file (with a `.pro` extension) that is used to automatically generate the platform-specific build files; no build files need to be created manually to port the project to the Symbian platform. Here is `cutemaze.pro`:

```
TEMPLATE = app
QT += svg
```

```

CONFIG += warn_on release
macx {
    # Uncomment the following line to compile on PowerPC Macs
    # QMAKE_MAC_SDK = /Developer/SDKs/MacOSX10.4u.sdk
    CONFIG += x86 ppc
}
qws {
    qtopia_project(qtopia app)
}
MOC_DIR = build
OBJECTS_DIR = build
RCC_DIR = build
unix: !macx {
    TARGET = cutemaze
} else {
    TARGET = CuteMaze
}
HEADERS = src/board.h \
    src/cell.h \
    src/maze.h \
    src/scores.h \
    src/settings.h \
    src/theme.h \
    src/window.h
SOURCES = src/board.cpp \
    src/cell.cpp \
    src/main.cpp \
    src/maze.cpp \
    src/scores.cpp \
    src/settings.cpp \
    src/theme.cpp \
    src/window.cpp
RESOURCES = themes/theme.qrc preview/preview.qrc
macx {
    ICON = icons/cutemaze.icns
}
win32 {
    RC_FILE = icons/icon.rc
}
unix: !macx {
    isEmpty(PREFIX) {
        PREFIX = /usr/local
    }
    icon.files = icons/cutemaze.png
    desktop.files = icons/cutemaze.desktop
    qws {
        target.path = /bin/
        icon.path = /pics/cutemaze
        icon.hint = pics
        desktop.path = /apps/Games
        desktop.hint = desktop
    } else {
        target.path = $$PREFIX/bin/
        icon.path = $$PREFIX/share/icons/hicolor/48x48/apps
        desktop.path = $$PREFIX/share/applications/
    }
    INSTALLS += target icon desktop
}

```

As you can see, the project file holds a lot of information necessary for the creation of build files and deployment on different platforms. Most of this need not concern us at this point, it's just good to have an idea what sort of thing is specified in the project file. However, there are a few items worth pointing out related to analyzing the project from a porting perspective. The following line specifies that the QtSvg module is required:

```
QT += svg
```

The previous line declaring that the target is an 'app' automatically includes the QtCore and QtGui modules. There are no LIBS lines in the project file so these really are the only library dependencies. The references to Qtopia and qws show that this project has also been ported to an embedded Linux device, so it's possible that the code has already been tried on an ARM processor with limited memory and run on a smaller display – for additional reasons, as you'll see later, an unexpected bonus. Finally, the following line shows that the project uses the Qt Resource System⁵ to store the application graphics in the main executable:

```
RESOURCES = themes/theme.qrc preview/preview.qrc
```

This makes deployment to real devices easier. The .qrc files have a simple XML-based format to specify which files to include in the binary. Further information on qmake and the syntax of the .pro file can be found at doc.qt.nokia.com/4.4/qmake-advanced-usage.html.

A quick examination of the source files reveals that the embedded Linux port is for Qtopia Phone Edition which should ensure that the code performs adequately on device hardware. There is minimal use of floating-point arithmetic (to calculate a scale factor for the graphics, which is then converted to the nearest integer for use) plus the code has a nice modular design and is well commented. The project is licensed under the GNU GPL, which is fine for use with the open source version of Qt. We are all clear to go ahead and start porting.

10.3 Setting Up the Development Environment

You need a standard Symbian development environment (see Section 2.4) plus an appropriately configured Qt installation. I recommend that you

⁵ See doc.qt.nokia.com/4.4/resources.html.

use Carbide.c++ v2.0 or later as this comes with built-in support for Qt projects, including the graphical UI designer (Qt Designer) which can be used to create and edit a UI project (although it was not used for CuteMaze). Carbide.c++ v1.3 requires an unofficial update to the Windows compiler in order to be able to build Qt projects.

The Qt framework depends on Open C and so the exact setup you require depends on which SDKs you are using for development. SDKs earlier than S60 3rd Edition FP2 require you to separately install the Open C plug-in and, when you deploy your code to a phone, the corresponding Open C SIS file must also be installed to pre-FP2 devices (see Chapter 4 for more details).

At the time of writing, this port of Qt is still in a technology preview state and it is necessary to install special technology preview versions of Qt and compile them for the Windows emulator environment yourself. However, it is expected that when the port is finalized (scheduled for the third quarter of 2009), plug-ins for the SDKs will be made available. Eventually, the port will be part of the standard Qt releases (you'll simply have to configure for cross-compilation on Symbian devices) and also part of at least Nokia's Symbian device firmware.

In order to set up Qt for your development environment, please refer to the latest instructions from developer.symbian.org or qt.nokia.com. You can find a link to this information on the wiki page for this book.

10.4 Integrating with the Symbian Build System

Since Qt project files can be used to create the necessary Symbian build files, this is a very simple step. In Carbide.c++ v2.0, simply import the .pro file (select File, Import, Qt Project); the bld.inf and MMP files are created automatically. If you're building from the command line, you need to ensure that you have the bin folder for your Qt installation set in your PATH environment variable, then you can simply go to the directory which contains the .pro file and type `qmake`.

In order to make changes to the MMP file that are specific to Symbian, it is necessary to change the .pro file to add a Symbian-specific section. For example, to add specific UIDs and allow the application to use the `NetworkServices` platform security capability, you would need to add:

```
Symbian: {  
    TARGET.UID2 = 0x100039CE  
    TARGET.UID3 = 0xA0000EFF  
    TARGET.CAPABILITY = NetworkServices  
}
```

If no UID3 value is specified then a random value is selected from the testing range. If no capabilities are specified then the application is not

granted any. However, in the case of CuteMaze, the defaults are fine so we don't need to do anything at this stage except import the project and try to compile it.

10.5 Getting It to Compile

On SDKs with the Open C/C++ plug-in installed, this project compiled without changes – the magic of Qt at work! However, the S60 3rd Edition FP2 SDK ships with Open C but not Open C++ supported by default. Compiling against a standard SDK without Open C++ produced some errors at compile time related to the inclusion of `<ctime>` in order to access the `time()` function to initialize a random seed for the maze generation. Includes of this form (with no `.h` extension on the file name) are used in standard C++ development and so perhaps it's not too surprising that this header file is found within the STLport include directory. Fortunately, in this case (as is very common for these includes), the header file is basically just a wrapper that includes the standard C library's `time.h`. By making a simple include file change, we can remove an unnecessary dependency on Open C++. We could simply replace `<ctime>` with `<time.h>` but without access to all of the environments that this project is compiled in to check we aren't breaking anything, it is safer to make this a platform-specific change as follows (see `board.cpp` in the example project):

```
#if defined(Q_OS_SYMBIAN)
#include <time.h>
#else
#include <ctime>
#endif
```

`Q_OS_SYMBIAN` is defined for all Symbian targets but not other Qt platforms. It's worth noting that `Q_OS_UNIX` is also defined for Symbian builds due to the use of Open C for the Qt port making it most similar to other Unix/Linux environments. As well as macros specific to the operating system, Qt also has defines to determine the window system being used, such as `Q_WS_X11` and `Q_WS_MACX`. These are often more relevant for platform-specific UI customizations.

10.6 Getting It to Work

The first run of the application in the emulator reveals two extremely common problems for desktop applications ported to mobile devices:



Figure 10.2 CuteMaze running on Symbian, unmodified

the application's main window doesn't fit on the screen and many of the application controls are designed for use with a mouse. Figure 10.2 shows CuteMaze running in the emulator after the initial compilation attempt.

Fortunately this project has already been adapted for a small screen in order to run on Qtopia Phone Edition. A quick examination of the code shows that the application's main window resizes itself and saves a default size to the application settings in `window.cpp`:

```
#if !defined(QTOPIA_PHONE)
    resize(QSettings().value("Size", QSize(448, 448)).toSize());
#endif
```

As you can see this line is not compiled in for Qtopia Phone Edition, so we can simply do the same for Symbian, as follows:

```
#if !defined(QTOPIA_PHONE) && !defined(Q_OS_SYMBIAN)
    resize(QSettings().value("Size", QSize(448, 448)).toSize());
#endif
```

Searching the source reveals a similar piece of functionality setting a minimum size for the Board widget in its constructor (in `board.cpp`), which we adapt accordingly:


```

#if !defined(QTOPIA_PHONE) && !defined(Q_OS_SYMBIAN)
    setMinimumSize(448, 448);
#endif

```

In fact, searching the entire project for ‘QTOPIA_PHONE’ reveals all of the places where we need to make changes in order to adapt to a mobile device screen. In nearly all cases, we can simply modify the `#if` statement to use the same solution for the Symbian platform as has been provided for Qtopia Phone Edition. This involves the removal of some features and buttons where they are not applicable or the device softkeys are used by default instead. The exceptions are where the code uses features that are specific to Qtopia Phone Edition. One example is an ‘auto-save on close’ feature implemented simply as:

```

#if defined(QTOPIA_PHONE)
    connect(qApp, SIGNAL(aboutToQuit()), m_board, SLOT(saveGame()));
#endif

```

This involves a signal provided by the Qtopia application manager that isn’t available on the Symbian platform at the time of writing, so we simply leave the code as it is and exclude the feature. Another example covers the handling of menu items and the conversion to softkeys.

The main window sets up the menus in its `initActions()` method as follows:

```

void Window::initActions()
{
    #if defined(QTOPIA_PHONE)
        QMenu* game_menu = QSoftMenuBar::menuFor(this);
        game_menu->addAction(tr("Quit Game"), qApp, SLOT(quit()));
        game_menu->addAction(tr("Settings"), m_settings, SLOT(show()));
        game_menu->addAction(tr("High Scores"), m_scores, SLOT(show()));
        m_pause_action = game_menu->addAction(tr("Pause Game"));
        game_menu->addAction(tr("New Game"), m_board, SLOT(newGame()));
    #elif defined(Q_OS_UNIX) && !defined(Q_OS_MAC)
        // Fetch icons for toolbar
        int actual_size = 0;
        QList<QIcon> icons = findNativeIcons(actual_size);
        // Create toolbar
        QAction* action;
        QToolBar* toolbar = new QToolBar(this);
        toolbar->setIconSize(QSize(actual_size, actual_size));
        toolbar->setFloatable(false);
        toolbar->setMovable(false);
        toolbar->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);
        action = toolbar->addAction(icons.at(0), tr("New"), m_board,
                                   SLOT(newGame()));
        action->setShortcut(tr("Ctrl+N"));
    #endif
}

```

```

m_pause_action = toolbar->addAction(icons.at(1), tr("Pause"));
toolbar->addAction(icons.at(2), tr("Scores"), m_scores,
                  SLOT(show()));
toolbar->addAction(icons.at(3), tr("Settings"), m_settings,
                  SLOT(show()));
action = toolbar->addAction(icons.at(4), tr("Quit"), this,
                           SLOT(close()));

action->setShortcut(tr("Ctrl+Q"));
addToolBar(toolbar);
setContextMenuPolicy(Qt::NoContextMenu);
#else
QMenu* game_menu = menuBar()->addMenu(tr("Game"));
game_menu->addAction(tr("New"), m_board, SLOT(newGame()),
                    tr("Ctrl+N"));

m_pause_action = game_menu->addAction(tr("Pause"));
game_menu->addAction(tr("Scores"), m_scores, SLOT(show()));
game_menu->addAction(tr("Settings"), m_settings, SLOT(show()));
game_menu->addAction(tr("Quit"), this, SLOT(close()), tr("Ctrl+Q"));
#endif
}

```

The Qtopia code uses the `QSoftMenuBar` class. Something similar may be added to Qt on Symbian in the future but the current solution is to mirror the Windows Mobile port of Qt and use the standard desktop `QMenuBar` (as in the `#else` clause towards the bottom of the method), mapping the actions into an options menu available from one softkey and allowing the developer to select the action to perform with the opposite softkey via a `setDefault()` method. Note that by default in this case we get the `QToolBar` instead, since `Q_OS_UNIX` is defined and `Q_OS_MAC` is not. This is OK for a device with a touchscreen⁶ but, in order to support the widest possible range of Symbian devices, we simply add the following code to the `#elif` condition:

```

&& !defined(Q_OS_SYMBIAN)

```

Opinion is divided on the use of preprocessor directives (`#if`, `#ifdef` etc.) in source files to manage platform-specific code. A popular alternative approach is to derive from the platform-independent class and keep all platform-specific code in separate files. The preprocessor directives make it easier to see where there are differences and require fewer files to be maintained. On the other hand, extensive use of preprocessor directives can make a source file very difficult to maintain. I'd suggest using a subclass if there are a lot of platform-specific changes, particularly if there are also multiple platforms. Otherwise it's probably easier to use

⁶ Conveniently, the Nokia S60 SDK emulators support mouse clicks on the screen as if a touchscreen were present even before S60 touch support was introduced in 5th Edition. This can be very useful in the early stages of porting a Qt desktop application.

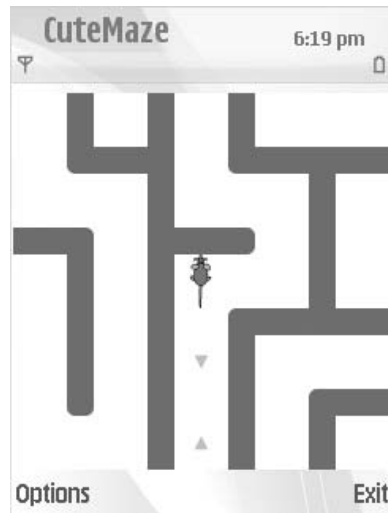


Figure 10.3 CuteMaze adapted to S60

the preprocessor but be ready to refactor to a subclass approach if new platforms are added or the level of platform-specific code increases. In the case of CuteMaze, the number of platform-specific differences is very small and doesn't warrant separation into multiple files.

You can examine the example code for full details of the changes but basically we can have the application completely working using Qt on S60 without making any real code changes, just a few modifications to preprocessor statements and changing one included header. Figure 10.3 shows a screenshot of CuteMaze adapted to S60.

With all of this easily achieved in a couple of hours, you can spend your development time on adding more functionality. The next section contains slightly more advanced material and can be safely skipped if you're just trying to get started with Qt on S60.

10.7 Extensions Specific to Mobile Devices

A fairly obvious and popular extension for games on smartphone platforms recently has been to use the accelerometer hardware present in many devices as a method of user input. Instead of pressing keys to move a character around on the screen, you simply tilt the device in the appropriate direction. This can create a very natural feeling and addictive gameplay. It would be possible to create a basic physics model and a theme for the game which replaces the character with a marble that is rolled around the maze, much like a once-popular children's toy. However, for this example, I simply use the input from the accelerometer to simulate key presses.



Figure 10.4 Samsung emulator running CuteMaze with GSensor API

The challenge here is that there are several different accelerometer APIs for different Symbian devices: Nokia has a Sensor Plug-in API and a Sensor Framework API (for pre- and post-3rd Edition FP2 devices, respectively, although there are some minor exceptions⁷) and Samsung has its own GSensor API. At the time of writing, the Nokia APIs only provide support (and, in some cases, import libraries) for target devices, not for the Windows emulator. In contrast, the Samsung SDK plug-in for S60 3rd Edition FP2, available from their Mobile Innovator site,⁸ provides the GSensor API and additions to the emulator to enable basic simulation of accelerometer input (see Figure 10.4).

⁷ See wiki.forum.nokia.com/index.php/Nokia_Sensor_APIs for details.

⁸ See innovator.samsungmobile.com.

I decided to implement the feature using the Samsung API initially, in order to be able to develop on the emulator, but to provide support for all three APIs, abstracting the details away from the rest of the application. The implementation is based on an early version of the Qt mobile extensions from Forum Nokia and it should be noted that the `QObjectPrivate` class that it uses is not part of the public Qt API and could be changed in future releases. However, as this area of hardware support is still under development and the sensor APIs are still evolving this does not really add to the existing burden of creating multiple builds for different devices. The implementation may need to be revisited in the future when Qt binaries are included in firmware and there is a need to consider binary compatibility. However, it is quite likely that such extensions will eventually be made a part of the firmware as well or at least released as libraries, so by following the ‘official’ implementation now there is a greater chance of retaining compatibility in the future. This kind of portability issue is, in any case, quite common if you work with the latest features on smartphones. The goal is to have the minimum impact on the core project code and allow easy configuration of the features for different devices.

The interface for the accelerometer sensor is abstracted by the following class:

```
#include <QObject>
class S60QAccelerationSensorPrivate;
QT_BEGIN_HEADER
QT_BEGIN_NAMESPACE
class S60QAccelerationSensor: public QObject
{
    Q_OBJECT
public:
    S60QAccelerationSensor(QObject *parent = 0);
    virtual ~S60QAccelerationSensor();
    int version() const;
    bool open();
    void close();
    bool startReceiving();
    void stopReceiving();
Q_SIGNALS:
    void xChanged(int value);
    void yChanged(int value);
    void zChanged(int value);
private:
    Q_DECLARE_PRIVATE(S60QAccelerationSensor)
};
```

Note that this class emits signals when the accelerometer sensor values change for each axis. The `Q_DECLARE_PRIVATE` macro allows access to a private internal implementation in a class called `S60QAccelerationSensorPrivate`. The method implementations simply call matching methods in the private class, for example:

```
bool S60QAccelerationSensor::startReceiving()
{
    Q_D(S60QAccelerationSensor);
    return d->startReceiving();
}
```

The corresponding private implementation uses the preprocessor to select the appropriate code for the device (again, we could derive separate classes for each type of sensor, but the implementation is probably too small to make it worth the overhead):

```
bool S60QAccelerationSensorPrivate::startReceiving()
{
    if(sensor)
    {
#ifdef SAMSUNG_GSENSOR
        sensor->StartObservingAccelerationInfo();
        return true; // No failure possible at this stage
#elif !defined(__WINSWC__)
#ifdef SENSOR_FRAMEWORK
        bool error = true;
        TRAP_IGNORE(
            sensor->StartDataListeningL(this, 1, 1, 0);
            error = false;
        )
        return !error;
#else // SENSOR_PLUGIN
        sensor->AddDataListener(this);
        return true; // Again, can't fail here
#endif
#endif
    }
    return false;
}
```

This is messy but covers all of the targeted sensor APIs and the Nokia cases where we want to build for the emulator without sensor support but the target device has support (using the `__WINSWC__` define that is always present when building for the emulator). Also note the use of the `TRAP_IGNORE` macro and a Boolean error flag to convert any leave from the Sensor Framework API to a simple indication of failure.

Details of which API is used are not required in the core of the application, just a simple flag to decide whether or not to include sensor support. A new member variable, `m_sensor`, is added to the `Board` class which currently handles the key processing and the sensor is initialized in the constructor as follows:

```
#ifdef S60_SENSOR_SUPPORT
    m_sensor = new S60QAccelerationSensor();
```

```

connect(m_sensor, SIGNAL(xChanged(int)),
        this, SLOT(sensorXChanged(int)));
connect(m_sensor, SIGNAL(yChanged(int)),
        this, SLOT(sensorYChanged(int)));
m_sensor->open();
m_sensor->startReceiving();
#endif

```

Aside from destruction of the sensor object and stopping and starting receiving events when we pause and continue the game, the only other impact on the application code is two new slots to handle changes to the sensor readings (we ignore changes in acceleration perpendicular to the plane of the screen):

```

#ifdef S60_SENSOR_SUPPORT
void Board::sensorXChanged(int x)
{
    if (x>>4 > 0)
    {
        QKeyEvent right(QEvent::KeyPress, m_controls_right, Qt::NoModifier);
        keyPressEvent(&right);
    }
    else if (x>>4 < 0)
    {
        QKeyEvent left(QEvent::KeyPress, m_controls_left, Qt::NoModifier);
        keyPressEvent(&left);
    }
}
void Board::sensorYChanged(int y)
{
    if (y>>4 > 0)
    {
        QKeyEvent up(QEvent::KeyPress, m_controls_up, Qt::NoModifier);
        keyPressEvent(&up);
    }
    else if (y>>4 < 0)
    {
        QKeyEvent down(QEvent::KeyPress, m_controls_down, Qt::NoModifier);
        keyPressEvent(&down);
    }
}
#endif

```

The slots simply ignore the bottom four bits of the sensor values, which cuts out sensor noise and small movements, then determine the direction of tilt and fake an appropriate key press.

In order to control the various preprocessor flags and add the necessary libraries, we simply add a new section to the `.pro` file:

```

symbian {
    # Uncomment sensor definitions depending on device

```

```

HEADERS += src/s60/s60qaccsensor.h \
           src/s60/s60qaccsensor_p.h
SOURCES += src/s60/s60qaccsensor.cpp \
           src/s60/s60qaccsensor_p.cpp
#DEFINES += S60_SENSOR_SUPPORT
# Samsung
#DEFINES += SAMSUNG_GSENSOR
#LIBS += -lHWRMAccelerationSensorClient
# Nokia (note hardware only - use custom MMP rule)
#sensorPluginMmpRule = \
#"$${LITERAL_HASH}ifdef WINSCW" \
#"LIBRARY RRSensorApi.lib" \
#"$${LITERAL_HASH}endif"
#MMP_RULES += sensorPluginMmpRule
#DEFINES += SENSOR_FRAMEWORK
#sensorFrameworkMmpRule = \
#"$${LITERAL_HASH}ifdef WINSCW" \
#"LIBRARY SensrvClient.lib sensrvutil.lib" \
#"$${LITERAL_HASH}endif"
#MMP_RULES += sensorFrameworkMmpRule
}
}

```

Here we demonstrate ways of customizing the MMP file from the .pro file. DEFINES in the .pro file are converted to MACRO statements in the MMP file, which end up as preprocessor defines. To give complete flexibility, it is also possible to add completely custom content to an MMP file via a new MMP_RULES keyword in the .pro file.

10.8 Deploying and Testing on Target Hardware

In order to assist in the deployment of projects to target hardware, qmake also creates package files automatically including the target binaries and adding a dependency on the Qt libraries (which first need to be installed separately). Users of Carbide.c++ can have SIS files automatically generated and signed (see Chapter 14 for more details of Symbian platform security and signing) as part of the build process. For command-line builds, the port of Qt to the Symbian platform provides a convenience script called `createpackage` to do this for you. If there is any requirement to add resource files to the package, this can be achieved with the DEPLOYMENT keyword in the .pro file. In the case of CuteMaze, the use of the Qt Resource System (see Section 10.2) means that no further files need to be installed beyond those included by default. If using the accelerometer extension from Section 10.7, it may be necessary to make some modifications to calibrate for the sensor output. Different device models may produce different results. I tested on a Nokia N95, a Nokia N96 and a Samsung i7110; in my tests, the application worked without any further changes.

10.9 Re-integrating

The original project is developed and maintained by a single person, as is common with small open source projects. The person is Graeme Gott and he hosts his projects at ***gottcode.org***. I contacted Graeme after my first few hours of porting effort to let him know about my plans and he was happy to integrate patches for a new platform.

A patch file is a very common way of communicating differences to code in the open source and free software world. A patch file can be created in a number of ways and some source code versioning systems can create them automatically. The most common way of creating a patch file manually is to use the DiffUtils tools, part of the GNU project. There is a free version for Windows available from: ***gnuwin32.sourceforge.net/packages/diffutils.htm***.

To create a patch file, execute the following command:

```
diff -u original_directory new_directory > changes.diff
```

On the command line, you specify the root directory of your project and a clean version of the original you started with (see the DiffUtils documentation for more advanced usage). You may need to clean the project up manually before the process as manual use of DiffUtils can't differentiate between original files and those that are generated as part of the build process. A patch that has been created in this way can be applied with the Patch program, with a Windows version also available from ***gnuwin32.sourceforge.net/packages/patch.htm***. A patch can be applied by running the following command in the appropriate directory on a copy of the original project:

```
patch < changes.diff
```

This can work even when there has been some change to the project since the version against which the patch was created, since the patch files contain some contextual information to assist in applying the changes in the correct locations.

10.10 Summary

In this chapter, I have described how a simple application developed using Qt was ported to the Symbian platform with very little effort. In Section 10.7, I also showed how such a project can be extended using

native Symbian C++ APIs while keeping the platform-specific additions separate from the rest of the code, preserving portability. While discussing the port, I've covered a number of the common issues you'll come across while porting Qt projects as well as solutions for the simplest cases. The coding for this port and extension was completed in less than a day. This is a lot less effort than would be typical for a porting project but it keeps the example small enough for thorough study. In the next two chapters, we look at some more complex open source ports that are in progress at the time of writing. Chapter 11 examines some examples of porting middleware and Chapter 12 describes an application that uses the middleware covered in Chapter 11.

11

Porting Middleware

I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone.

Bjarne Stroustrup

This chapter is about porting middleware to the Symbian platform. We examine the subject via two case studies for middleware that is in the process of being ported. The first of these is an excellent example of platform-independent middleware, the open source Geospatial Data Abstraction Library (GDAL).¹ As it deals with converting and manipulating various data formats, it has very few dependencies on system services. The second example, the Qt application framework, shows how highly platform-dependent middleware can also be portable. In this case, the libraries necessarily interact with a wide range of system services in order to provide a platform-independent interface for applications.

So, what exactly is middleware? It's a term used to refer to components that sit 'in the middle' between applications and lower-level APIs, such as those provided by the operating system, or third-party libraries. Originally used in distributed computing environments to abstract away network interface details, it now has much wider use, including application programming frameworks, game engines and multimedia libraries. A defining characteristic of middleware is that it doesn't do anything on its own, instead it provides services to an application or server, as well as a higher level of abstraction for the APIs on which it is built, making them easier to use.

Both of the ports discussed in this chapter are works in progress at the time of writing and the aim of the chapter is to provide an insight into the strategies used and issues involved in the porting of large, widely used middleware projects, rather than to present complete porting examples.

¹ See www.gdal.org.

However, since both projects are open source and the ports are expected to be at least close to completion at the time of publication, interested readers are invited to study the latest releases to learn how they have progressed. You can find out more at the website for each project, or by following links on the wiki page for this book.

11.1 GDAL

GDAL is a translator library for raster geospatial data formats that is released under an X/MIT style open source license by the Open Source Geospatial Foundation.² As a library, it presents a single abstract data model to the calling application for all supported formats. It also comes with a variety of useful command-line utilities for data translation and processing. The related OGR³ library lives within the same source tree and provides similar functionality for simple features vector data. The translators for both raster and vector formats compile into a single GDAL/OGR library; usually referred to simply as GDAL. The GDAL library is a fundamental building block in many geospatial software projects. In the words of Howard Butler, a director of the Open Source Geospatial Foundation, ‘I see GDAL as the glibc/glibc++ of the geospatial software world. It’s open, it provides core functionality, I can’t understand how anybody gets anything done without it.’

11.1.1 Selecting a Project

In this case, the project selection was forced because the library is used by another project being ported, Bluemapia (see Chapter 12). However, GDAL has several characteristics that are desirable for a porting project: it’s maintained by a large, active community and has already been ported to multiple platforms, with bindings to several other languages and a massive user community. As mentioned above, since its core functionality is the translation of data formats, it should be inherently platform-independent.

11.1.2 Analyzing the Code

I downloaded the latest release of GDAL, version 1.6.0, and examined the source. The top-level directory contains build files for various platforms and there are sub-directories for various parts of the project. One sub-directory, named `port`, looked particularly promising for further investigation. It turns out that GDAL uses a porting layer to maintain cross-platform compatibility. Any functions that have to be implemented

² See www.osgeo.org.

³ The name OGR has only historical meaning; it no longer stands for anything.

differently on one platform from another are called via the porting layer, which hides the differences from the rest of the library. A good example of this can be seen in the `cpl_multiproc.cpp` file that provides facilities for threading, mutexes, file locking and waiting (sleep). There are three completely separate implementations of the interface within the file, conditionally compiled based on the available threading model: Win32, POSIX threads or stub. The stub model doesn't provide threads at all and sleep is just a busy wait: it's a minimal implementation for platforms that don't support the other models. Fortunately, thanks to P.I.P.S. (see Chapter 4), we can use the POSIX threading model on the Symbian platform.

Support for different formats is provided through 'drivers', which can be optionally included in the build, depending on the availability of other libraries to parse the formats in some cases. However, there are internal versions of libraries maintained for some of the most popular formats.

After a quick look through the code, two things are clear:

- GDAL is big, in terms of source files at least. Including documentation, the project contains over 2000 files in nearly 200 directories.
- Despite its size, porting the project should essentially be a case of configuring the porting layer correctly and possibly re-implementing any functions where existing options aren't suitable.

Since the project selection is forced in this case, with no alternatives, and the port looks feasible, no further analysis of the code was necessary at this stage. Even though it's still hard to estimate how much effort is likely to be required at this point, starting the port and assessing the level of issues with compilation is going to provide a more accurate basis for estimation more rapidly than further code investigation.

11.1.3 Setting Up the Development Environment

For a large library project like this one, it can be more efficient to use the command-line tools than to use an IDE, at least in the early stages before the code is actually running. The main reasons for this are that there is little to be gained from project wizards or templates for a library (static or dynamic – on other platforms, GDAL can be built in either form) and Carbide.c++ tends to rebuild more than is strictly necessary to ensure that everything is correctly updated. For small projects, a full rebuild is so fast that it is certainly worthwhile avoiding the occasional nightmare debugging session that turns out to be due to an out-of-date object file or similar. In the case of larger projects, it can be too much overhead. To be fair to Carbide.c++, it does provide the facility to compile, or pre-process, individual C/C++ files that are part of a larger project to help address this issue. Even this feature seems to be slower than using the command-line tools to recompile a target.

When you've created `bld.inf` and MMP files, you can attempt a first full build for the emulator with:

```
bldmake bldfiles  
abld build winscw udeb
```

or for target hardware with:

```
bldmake bldfiles  
abld build gcce udeb
```

If you don't make any changes to the MMP file, only to source or header files, then you can rebuild after this simply by using:

```
abld target winscw udeb
```

or:

```
abld target gcce udeb
```

This can save significant amounts of time when the MMP file is large or you have multiple MMP files referenced within the same `bld.inf`.

11.1.4 Integrating with the Symbian Build System

On Linux, GDAL uses a configuration script generated by Autoconf that allows various command-line options to add and remove features. On Windows, makefiles with manually editable options are provided. Neither option provides a simple basis for creating the MMP files for the Symbian platform. As there are hundreds of source files and nearly all of them should be included in the build, I decided that the fastest approach would be to start by trying to build everything and then remove parts that wouldn't compile due to missing dependencies or other non-trivial issues.

Manually creating an MMP file for the entire project was not a very appealing prospect. This is one of those repetitive tasks where a software engineer should always consider if a computer can do it faster and more accurately. There's a clear case here for a script that will at least examine the files and add `.c` and `.cpp` files to an MMP in the correct format. Fortunately something very close to this functionality (and more) is provided by the `qmake` utility provided with Qt (ported to the Symbian platform). In the root directory of the project, the following command creates a `.pro` file that combines all the source as a Qt application:

```
qmake -project
```

If you then run `qmake` (without arguments) in the same directory, it creates `bld.inf` and MMP files, as well as an extension makefile, application registration and package files. You can delete all the generated files (including the `.pro` file) except `bld.inf` and the MMP files. Then it's just necessary to remove the reference to the extension makefile from the `bld.inf` file, edit the MMP to remove Qt-related content (macros, options, libraries and include paths) and change the target type if required. In this case, it was desirable to build GDAL as a static library, although in the longer term a dynamic library option is also likely to be required. Since the C++ pre-processor is run on MMP files, it's possible to provide both options via conditional compilation.

11.1.5 Getting It to Compile

The first logical task to get the whole library to compile is correctly configuring the porting layer. I used the `qmake` trick mentioned in Section 11.1.4 to generate build files for creating just the porting layer as a static library. At the first compilation attempt, the most obvious error was a missing header file, `cpl_config.h`, which is generated by the configure script on Linux. This makes sense in a desktop environment where you can re-compile the library to match the available external components on your individual system.

When cross-compiling for an embedded device, it would seem more reasonable to have a fixed configuration that can be manually edited if features are added in future releases. In fact, there is a version of the header for Windows CE in the port folder called, `cpl_config.h.wince`. I created a version of this for the Symbian platform, which can be copied to `cpl_config.h` when required. I took a minimal Linux configuration as a starting point, gradually adding a few more features that it should be possible to support on Symbian. There were also missing standard C++ headers which I corrected by adding the STLport include paths to the MMP file.

The next major source of errors for a Windows emulator build is that many Windows-specific functions were being included. This is because the file `cpl_port.h` uses some standard defines that all Windows compilers include, determining that it is building for a Windows platform, as follows:

```
/* ===== */
/*      We will use WIN32 as a standard windows define.      */
/* ===== */
#ifdef _WIN32 && !defined(WIN32) && !defined(_WIN32_WCE)
#  define WIN32
#endif
```

```
#if defined(_WINDOWS) && !defined(WIN32) && !defined(_WIN32_WCE)
#   define WIN32
#endif
```

This is understandable and since Symbian's Windows emulator environment is unique, it needs to be treated as a special case. Although the `__SYMBIAN32__` define has been stable and consistent, I follow the convention used in GDAL and isolate the library from any potential future changes to this define:

```
/* ===== */
/*      We will use SYMBIAN as a standard Symbian define.      */
/* ===== */
#ifdef __SYMBIAN32__
#   define SYMBIAN
#endif
```

I then changed the lines determining whether we are building on Windows to exclude the Symbian emulator case:

```
/* ===== */
/*      We will use WIN32 as a standard windows define.      */
/* ===== */
#if defined(_WIN32) && !defined(WIN32) && !defined(_WIN32_WCE) &&
    !defined(SYMBIAN)
#   define WIN32
#endif

#if defined(_WINDOWS) && !defined(WIN32) && !defined(_WIN32_WCE) &&
    !defined(SYMBIAN)
#   define WIN32
#endif
```

This isn't quite the entire story for these platform-specific defines though. When attempting to build `cpl_minizip_unzip.cpp`, there is an undefined reference to `strnicmp()`, which is a Windows-specific function. In this case, there is a clash with the `zlib` implementation in Open C. The `zlib` implementation internally defines `WIN32` in a similar way to GDAL for emulator builds and the Symbian port currently doesn't do anything to correct this. As a result, including `zlib.h` before `cpl_port.h` was causing the wrong set of defines to get picked up here:


```

#ifndef EQUAL
#ifdef WIN32 || defined(WIN32CE)
# define EQUALN(a,b,n)      (strnicmp(a,b,n)==0)
# define EQUAL(a,b)         (strcmp(a,b)==0)
#else
# define EQUALN(a,b,n)      (strncasecmp(a,b,n)==0)
# define EQUAL(a,b)         (strcasecmp(a,b)==0)
#endif
#endif

```

In order to correct this, ideally I'd like `zlib` changed for the Symbian platform so that it doesn't export this `WIN32` define to anything that uses it. Currently this kind of change is very difficult to achieve but hopefully proposing such changes will be much easier when the Symbian Foundation is in full operation. In the mean time, using `#undef` to remove the `WIN32` define for all Symbian builds of GDAL is a simple solution.

```

/* ===== */
/*      We will use SYMBIAN as a standard Symbian define.      */
/* ===== */
#ifdef __SYMBIAN32__
# define SYMBIAN
// Remove any WIN32 define made by zlib.
# undef WIN32
#endif

```

This kind of issue is extremely common where a project uses multiple third-party libraries, since every library tries to decide for itself which platform it is being built for.

The only remaining changes required to get the porting layer to compile were the exclusion of some files from the build. In two cases, this was obvious as the files had `_win32` as part of their names and were platform-specific extensions for Windows. Since there is no support for ODBC database access on the Symbian platform, the relevant source file was excluded and the header modified to be effectively empty on Symbian (the conditional compilation flags were already in place for Windows CE). Finally, another file also had an issue relating to `zlib`. There was a missing symbol at link time, `inflateCopy()`, used in `cpl_vsif_gzip.cpp`. This function is missing from the Symbian

zlib port, despite the Open C documentation claiming 100% function coverage! Having contacted the core GDAL developers about this issue, it was determined that the functionality affected is new and potentially optional and could thus be safely excluded without further issues. Again, a future update to Open C may allow this to be re-instated.

Having completed the porting layer, I moved onto the core GDAL functionality and API in the `gcore` sub-directory. Again, the STLport system include paths were required, as were user include paths for the `port` and `ogr` sub-directories, it appears that GDAL and OGR are currently inseparable! The only compilation error remaining was a use of Linux-style locale support in `gdal_misc.cpp`, as part of a command-line argument parsing function for use in various utilities supplied with GDAL. Searching the project revealed that this use of the platform-specific functionality had just been missed; other uses were flagged with appropriate conditional compilation flags, so I simply added the same in this case:

```
#if defined(HAVE_LOCALE_H) && defined(HAVE_SETLOCALE)
/* ----- */
/*      --locale      */
/* ----- */
    else if( EQUAL(papszArgv[iArg], "--locale") && iArg < nArgc-1 )
    {
        setlocale( LC_ALL, papszArgv[++iArg] );
    }
#endif /* defined(HAVE_LOCALE_H) && defined(HAVE_SETLOCALE) */
```

There was an almost identical occurrence in `ogrutils.cpp` in the `ogr` sub-directory. Apart from this, the core OGR functionality and API also built for the Symbian platform without problems once the appropriate system and user include paths were added.

With the core parts of the project building, the only essential additions were any required raster and vector formats. A large number of the formats supported by GDAL require external libraries, some of which are proprietary. Even though very few of the formats are actually required for the Bluemapia port, I decided it was worth trying to compile all of the available formats and simply removing the ones with missing dependencies to start with. Making more of the functionality available on the Symbian platform should increase use of the library on the platform and hopefully encourage further testing by others. A surprisingly large number of the formats built with no changes at all. There were also a number of files which gave compiler errors for the emulator (although not the target) due to the implicit casting of a pointer to an unsigned type to a pointer to a signed type of the same size. This is a common issue when compiling a code base with a new compiler. Because the code has already been compiled on Linux using GCC, it's no surprise

that GCC-E doesn't have any problems with it; however the Nokia x86 compiler finds new issues. There are three possible solutions to this type of issue:

1. Fix the code so that compilers for all platforms are happy.
2. Make casts explicit.
3. Turn off the relevant compiler warning or error for the project.

Option 1 seems like the 'right' solution, although if you aren't in a position to test the change against all supported platforms then you may introduce a defect or compilation failure for someone else. Option 2 can sometimes be equal to Option 1, with no danger of breaking compilation on other platforms, although it is not always the correct solution even when it is possible. It's essential to check that there isn't a real defect in the code which the compiler has highlighted before making a cast explicit – if there is a real defect, making a cast explicit removes the warnings, makes the cast look intentional, and makes the defect much harder to find and fix. Option 3 is something of a last resort, if you have many tens or hundreds of similar errors and the effort required to check them all and fix them is too great. You can turn off many errors and warnings by checking the compiler documentation and adding an appropriate `OPTION` statement in the MMP file. In each of the GDAL cases, I determined that an explicit cast was safe: data was being read in from or written out to a file as unsigned characters (binary data) but cast to or from a signed character or numeric value for interpretation based on its position within the file header.

Following this, some of the other drivers expected internal library headers to be in system include locations and were including the relevant headers with angle brackets rather than quotes. Many build systems search both locations but this doesn't happen on the Symbian platform – if you want headers included in this way to be found, then the paths to them must be added as system includes; a few changes to the MMP file of this nature were required. Another problem was a file (`rawblockedimage.cpp`) that was using the `_WIN32` define rather than the internal GDAL equivalent, and was therefore trying to include a header that isn't available on Symbian in emulator builds – easily fixed. Supporting the TIFF and GeoTIFF formats required the internal version of `libtiff` supplied with GDAL to be used. This needed another configuration option in the porting layer that I had missed previously:

```
/* Set the native CPU bit order (FILLORDER_LSB2MSB or
                                FILLORDER_MSB2LSB) */
#define HOST_FILLORDER FILLORDER_LSB2MSB
```

With this set correctly in `cpl_config.h`, the TIFF library and related drivers compiled correctly but would not link.⁴ A missing function, `lfind()`, was simply extern'd in the case where no `search.h` header file was available on the platform. There was no matching implementation in the project but fortunately the full version of the `libtiff` library⁵ had a simple implementation of the function that I was able to re-use in GDAL by adding another source file to the porting layer. Another issue was easily solved at the link stage by the removal from the build of some basic test harnesses for some of the drivers – there were multiple conflicting `main()` functions in each of these files.

At this point, an unusual and difficult-to-resolve issue was discovered. The port had been too successful and so many formats were able to be compiled that it wasn't possible to link them all into a single library with the GCC-E toolchain (although there was no problem on the emulator). The actual error that occurred was 'make (e=87)' and I was able to discover from the Forum Nokia discussion boards that this had been found before and investigated in a very similar situation. The command to the linker is constructed and passed through an internal buffer in make that is 32 KB in size. With several hundred files being linked into a single library, it is possible to exceed this limit. As far as I'm aware, the only flexibility the Symbian build system has to get around this at the time of writing is to split the project into smaller static libraries built with subsets of the code and link those into a larger dynamic library or executable. I was unable to find a way to create a single static library for the whole project using the standard build system (although it should be possible by using some of the individual build tools directly). However, restricting the drivers built to just those required by Bluemapia avoids this problem and I decided to leave it for future investigation.

11.1.6 Getting It to Work

Testing middleware can be a very complex and difficult task. Often, libraries provide a lot of different functionality that can be used in many combinations of ways. At the same time, since they don't do anything on their own, some kind of test application is required to exercise the functionality. Good libraries should always be accompanied by test suites and this is the case with GDAL. However, the primary testing for GDAL is performed via its Python bindings, using a large set of test scripts and associated data. The documentation for the test scripts suggests that Python version 2.4 or later is required. There is a version of Python for the Symbian platform but, at the time I got the port to compile, the

⁴ In fact, this was not discovered until attempting to link the static library into a test executable, as discussed in Section 11.1.6. However as this is a build-time error, rather than a run-time error, I include it here.

⁵ www.libtiff.org

stable release was Python version 2.2, and there had just been a new alpha release of Python version 2.5.⁶ Not wanting to complicate the development further by being the first to try to port a new Python module to the latest interpreter, I decided to wait for a later release and explore other testing alternatives. There is a much smaller set of tests for GDAL written in C++, which was created for testing the Windows CE port. Since Bluemapia has extremely limited use of GDAL, basically just for reading a few map formats, there was also the option of a simple viewer application that had been written using Qt.⁷

With an open source project, it's generally a good idea to release early and often to get feedback and allow other developers to contribute. In this case, other developers had already expressed an interest in contributing, so I decided to follow an incremental testing strategy. I used the viewer application first to ensure that the port was usable to view the necessary map formats. I could then make an initial release of the library so that others could start working on the Bluemapia port and further testing of the GDAL library. The plan is to port and run the C++ test suite to increase confidence in the port and make initial submission of changes back to the core GDAL maintainers. After this, I hope it will be possible to get the Python-based test suite running before making the Symbian platform officially supported for the project. Please check the GDAL website for the latest status.

When using the viewer application to test GDAL, only one error was identified and fixed before maps were displayed correctly. On the first run, GDAL was unable to read files of any format. Although the drivers for the formats were being compiled, they weren't being registered for use by the API. Each driver implements its own registration function but they are all called from a central location in `gdalallregister.cpp`. The file contains a single function `GDALAllRegister()` which must be called by clients before using any other features of the library. It looks like this:

```
void CPL_STDCALL GDALAllRegister()
{
    GetGDALDriverManager()->AutoLoadDrivers();
#ifdef FRMT_vrt
    GDALRegister_VRT();
#endif

#ifdef FRMT_gdb
    GDALRegister_GDB();
#endif
}
```

⁶ Porting of this version of the Python interpreter to the Symbian platform was yet another project made possible by Open C.

⁷ The viewer was originally written for the Win32 API and was ported to Qt in a matter of hours by another contributor to the project.

```
// Many, many formats follow ...
...

#ifdef FRMT_georaster
    GDALRegister_GEOR();
#endif

/* ----- */
/* Deregister any drivers explicitly marked as suppressed by */
/* the GDAL_SKIP environment variable. */
/* ----- */
    GetGDALDriverManager()->AutoSkipDrivers();
}
```

The conditional compilation flags refer to defines that are passed to the compiler from the makefile, rather than included in a header. These are generated by the configure script on Linux, based on command-line options that are passed to it. For the Symbian port, the necessary defines can be added as `MACRO` statements in the MMP file. To make configuration of included drivers easier on the Symbian platform, a separate formats file was created in which defines for the required drivers can be included. The formats file is included in the MMP file so that when the pre-processor runs through it, it conditionally includes the relevant source files and `MACRO` statement.

11.1.7 Re-integrating

Like most open source projects, write access to the GDAL source code repository is only granted to proven contributors. Suggested changes to allow GDAL to be built for the Symbian platform, along with any new build files required, had to be submitted to the project's maintainers. The method for doing this is to create new 'tickets' in the project's Trac instance.^{8,9}

Despite the large size of GDAL, it was extremely portable. The time taken from downloading the source to getting a map image open in a test viewer in the emulator was just two days. Partly this is due to the inherent portability of the project, purely by its nature, partly it is due to the project maintainers' very disciplined use of a porting layer to isolate platform-specific changes, but most importantly it is due to Open C/C++ providing an environment that is very close to the Linux/Unix platforms on which GDAL was already being used.

⁸Trac is an integrated source code and project management tool that is quite popular with open source projects. The Trac project is hosted at trac.edgewall.org and the instance used for GDAL is hosted by OSGeo at trac.osgeo.org/gdal.

⁹Another popular method for submitting patches, bugs and feature requests to open source projects is via an instance of Bugzilla (www.bugzilla.org), the tool that has been taken into use by the Symbian Foundation.

11.2 Qt

In contrast to GDAL, the Qt application framework needs to have quite deep integration with the target platform; it is not all inherently portable code. The features of Qt and its port to the Symbian platform were discussed in Chapter 6, so here we focus on it purely as an example of how to go about porting a large middleware project. Trolltech had a history of developing new ports and features in secret until there was something developers could actually try for themselves and then releasing a technology preview. Several months after the purchase of Trolltech by Nokia, it was no great surprise to anyone when the first technology preview of a port of Qt to S60 (as the Symbian platform was then known) was announced.

11.2.1 Technology Previews

Following the open source ‘release early, release often’ strategy, three technology previews were scheduled before the official beta release of the port:

- Pyramid – Support for the QtCore, QtGui, QtNetwork and QTest modules only
- Temple – Fixes and build system improvements along with the addition of QtScript, QtSvg and QtXml modules
- Garden – Integration with the S60 UI and an S60 UI style; further modules added.

The first release contained all the core modules that would allow early adopters from the Symbian developer community to start experimenting with Qt and pioneers from the Qt developer community to start learning some Symbian concepts and porting their applications. It also allowed the porting team at Qt Software, now known as Qt Development Frameworks, to get rapid feedback on the major issues that developers had installing and using the libraries, in order to prioritize work for the next release.

The second release added some of the least platform-dependent modules, which were presumably fairly easy to port. Some missing functionality from the original modules was also included. The major improvements were fixes to the build tools and installation process, making it much easier for novice developers to use. Around the same time as this release, Nokia also released Carbide.c++ v2.0 including new Qt integration as a free development tool. The Temple release didn’t have an enormous amount of new content, but enough to keep developers interested while the porting team was working on the third release in parallel.

As well as adding most of the remaining modules, the third release was a very big shift visually and architecturally. The S60 UI integration made

Qt applications look and behave like native applications. To achieve this, the porting team had to re-write a lot of the initial work on the QtGui module. This was an intentional part of an incremental porting strategy.

11.2.2 Incremental Porting

In any large porting project with multiple software engineers, there is often a lot of dependencies between tasks. Even when the team members are physically all in the same location, it can require careful planning to ensure that no-one is blocked waiting for the work of others; in an open source project with contributors spread across the world, it becomes even more difficult. In the case of Qt, all of the other modules have some dependency on QtCore and most of them also on QtGui. However, these modules are among those requiring the greatest effort to port to a new platform. To avoid the problems associated with too many developers trying to work on the same module, one solution is to do the minimum amount possible to get a usable module, then go back and do it properly while others work on the rest of the port in parallel. When porting your own project to an unfamiliar platform, this strategy also has the advantage of providing a prototyping stage and valuable learning experience for the developers.

There are excellent examples of this that could be used by someone porting another GUI toolkit or middleware that does low-level drawing and input handling, such as a game engine. The Pyramid and Temple releases of Qt for S60 interfaced directly to the Symbian window server, both for drawing and notification of input events. Effectively, Qt controlled the entire screen, completely bypassing the S60 Avkon UI framework. This couldn't be a final solution if Qt applications are supposed to behave like native applications. The native application framework handles the status pane and notification icons, as well as dynamically responding to changes of orientation on some devices. From the Garden release onwards, Qt for Symbian processes the key events passed to it by the application framework and draws in the client rectangle it provides, unless the application is explicitly set to full-screen mode, in which case it lets the application framework know this too. Similarly, the early releases had only crude keypad input with a fixed mapping; there was no support for multi-tap or predictive text entry. The later releases made Qt applications FEP-aware,¹⁰ providing support for the full range of input methods.

If you want to interface directly to the window server for your own project, you simply need to create a session with it using the `RWsSession::Connect()` method. You can then use this session to create a `CWsScreenDevice`, to get information about the screen size and

¹⁰The front end processor (FEP) architecture on the Symbian platform is a unified framework for character input methods.

display mode, and a window group (RWindowGroup) and register to receive key events and redraw events from the window server. The event notifications come via the standard asynchronous request mechanism on Symbian, so you can either wrap them with active objects or use `eselect()` from Open C/C++ to wait on them – see the `EventReady()`, `RedrawReady()` and `PriorityKeyReady()` methods of `RWsSession`. For more details on implementing a deeper integration with the Symbian application framework, please examine the latest source code. You should find the file `qapplication_s60.cpp`, in the `src\gui\kernel` directory, a good starting point for your investigations.

Another way in which Qt has been ported incrementally is in terms of library versions. Qt, like most active software projects, is constantly evolving. If you plan to start a port of a code base that's going to take a reasonably long time then you need some kind of strategy for managing updates to the original code. Trying to keep up-to-date with the latest development branch is likely to result in a lot of wasted time chasing new defects that aren't relevant to the porting task. However, staying on a stable branch throughout the porting project can result in a merging nightmare at the end, potentially introducing large numbers of additional defects. This is common to most large software development tasks but is even more relevant for porting because there is often a broad set of changes involved and the likelihood of parallel development on the same files is significantly increased. Ideally some middle ground is required where the porting branch regularly merges in changes from the main development branch but keeps away from the leading edge of development. Some projects have a 'stable' branch which is ideal for this purpose, while others have a more cyclical development process. The port of Qt to the Symbian platform was started with a similar timeframe to the development of Qt version 4.5. The Pyramid release used the (relatively old and stable by that time) version 4.4.2 of the libraries, which was upgraded to version 4.4.4 for the Temple release. The move to Qt version 4.5 was not made until shortly before the Garden release when version 4.5 itself was close to final release.

11.2.3 Architecture of Qt on the Symbian Platform

The Qt port was another of those made feasible by Open C. Although it was necessary to interface directly to the native Symbian platform code in a number of places, a large part of the port uses the same code as various Unix-compatible systems via the Open C APIs. However, as application code still runs natively and in the context of an active scheduler (provided by the native application framework), it is very easy to mix Qt code with native Symbian C++ as well as using Open C/C++ directly. These architectural dependencies are shown in Figure 11.1.

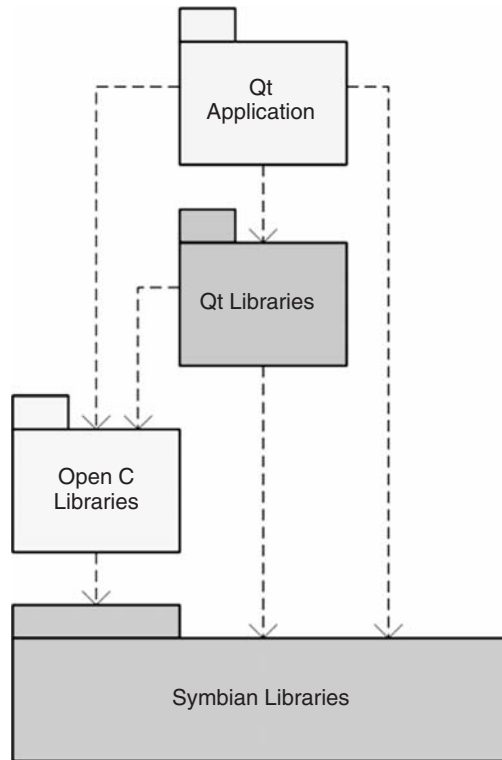


Figure 11.1 Application environment for Qt on the Symbian platform

Since all of the code for the Qt libraries is open source, you can examine it for yourself to see the platform-specific differences. Searching the code for files with 'symbian' or 's60' in the names highlights the areas with the most significant changes. Similarly, searching for the defines `Q_OS_SYMBIAN` and `Q_WS_S60` within the source files should find the areas where smaller changes were required.

Perhaps understandably, there is quite a high level of changes in the `QtGui` module, whereas networking, threads and file access have very little change from the Unix code. In fact, for networking and threads, the only changes required are to work around limitations of the current Open C implementation and allow some inter-working with native threads. For the file system, there are some special Symbian features that are catered for. One of these is that Symbian provides a built-in facility to monitor the file system (via the `RFs::NotifyChange()` methods), which is used in preference to Qt's own polling implementation. Another difference is that Symbian uses separate drives with drive letters in what is often referred to as a 'DOS' style, so any use of absolute paths has to be modified accordingly.

Another porting issue that has been dealt with in a way that could be replicated in other projects is that of searching for and querying plug-ins.¹¹ On Symbian, the platform security system prevents applications from accessing the `\sys\bin` directory where all binaries are stored. If you can't access the directory, how do you discover which plug-ins are present? One solution would be a separate system server that has read access to the directory in question, however this is not ideal as it adds both complexity and latency to the process of loading plug-ins. The alternative favored in the Qt port to Symbian is simply to add another file, a plug-in stub, to the file system in a known accessible location for each plug-in installed. With these extra files in place, querying the plug-ins is transparently replaced by querying the plug-in stubs. The actual plug-in location is only used when it is necessary to load it.

In contrast to the parts of the port that have significant platform dependencies, many of the Qt modules build entirely on other core modules and required very little change. At the time of writing, the latest code for the QtScript module had only one change for the Symbian port, which was to limit the maximum call depth for a script slightly, to account for the limited stack size on the Symbian platform. The QtXml module had no changes specific to Symbian.

11.2.4 Integration with the Symbian Build System: qmake

As they have to be built on all of the supported platforms, it is not surprising that the Qt libraries use the same platform-independent project files as all Qt applications. With the build files common across platforms, the task of integrating with the Symbian build system was reduced to porting the build tools and adding support for platform-specific extensions. The first step was to ensure that all of the existing tools could be compiled with the Nokia x86 compiler since it is the one most likely to be available for the host platform in a Symbian development environment.¹² For many of the tools, such as `moc`, `uic` and `rcc`, this was all that was required, since they have inherently platform-independent functions. However, the `configure` and `qmake` utilities needed platform-specific modifications in order to be usable. The `configure` utility is a special case because it controls the building of the other tools. New functionality had to be added to include the host platform (denoted `win32-mwc`) and the target platform for which the tools are cross-compiling (denoted `symbian-abld`).

The vast bulk of the work involved in getting Qt projects to compile on the Symbian platform was the addition of new functionality to `qmake`. Since `qmake` generates makefiles and other build files for all of

¹¹ You can read about Qt's plug-in system at doc.qt.nokia.com/4.5/plugins-howto.html. However, the details of the plug-in system are not important for this discussion.

¹² Assuming use of the emulator – strictly speaking, a host compiler is not required to do Symbian development.

the platforms that support Qt, it is internally structured to use a range of independent makefile generator classes. By far the largest change required was the creation of a new generator for Symbian build files, which can be found in the `qmake\generators\symbian` sub-directory of a Qt installation. An interesting thing to note is that the generator is written using C++; in fact, it uses Qt classes (particularly `QString`, `QStringList`, `QTextStream` and `QMap`) extensively to do the necessary text manipulation. This is a good thing from two perspectives: Qt developers use their own toolkit¹³ and there is no dependency on an external scripting language. The other file generated by `qmake` specifically for Symbian projects is the package file to enable deployment to target devices. For a Qt plug-in, the necessary plug-in stub (as mentioned in Section 11.2.3) is also created.

11.2.5 The Route to a Working Application Framework

The core Qt porting team was part of Trolltech before it was acquired by Nokia and the porting began before the acquisition was completed. Only one of the original team had any experience with Symbian before the project was started.¹⁴ However, they did have some training and consultancy from a third-party service provider with extensive Symbian platform experience. The team was in the common position of having to learn a new environment and port to it at the same time. Several members of the porting team were generous enough with their time to talk to me or answer my emails during the writing of this book, including the team leader, Espen Riskedal.¹⁵ He said that the initial training was extremely valuable because there are some idioms on Symbian which are very different from other platforms (you can read about them in Chapter 3). Beyond that, the team had found that starting their testing early, working with running code, had helped them to get to know the platform better and that the debugger in `Carbide.c++` was a really useful tool for this purpose, particularly with its built-in support for many Symbian platform types and classes.

As the creators of an open source application framework, Qt Development Frameworks is in an excellent position when it comes to testing their ports. In addition to their own module tests, they produce a large range of examples and demonstrations for developers which can also be used for testing. Having previously produced embedded Linux and Windows

¹³ I'm always suspicious of platform providers that don't use their own tools. Mobile device manufacturers have pushed Java ME as the portable solution for developers for a long time but almost never leveraged the alleged portability to create common applications for their own device portfolios!

¹⁴ In fact, that developer had started his own port of Qt to Symbian as a 'creative Friday' project. Creative Fridays were a long-standing tradition at Trolltech, where developers were allowed to work on their own projects for one day a week.

¹⁵ See labs.qt.nokia.com/blogs/author/espenr.

Mobile ports, they already had some good tests and demonstrations for mobile platforms that they could re-use on the Symbian platform. These points should apply to any toolkit or framework. With a popular open source framework, there are the added benefits of a large community of developers to test the port with their own applications and the ability to get more detailed feedback. As the alpha and beta tester community has all the source code, they can even fix the bugs they find occasionally. Then by porting the framework to a new platform, an even larger community of potential testers and contributors can be accessed. Even with this community assistance, the task of testing all the libraries thoroughly in order to find all the tiny issues and unexpected interaction with other software on the platform is by far the largest task. This is made worse by the requirement to support millions of devices that are already in use – any major quirks in the firmware of existing devices need to be worked around.

At the time of writing, the intention was for Qt 4.5 to be released without support for the Symbian platform, with the Symbian port remaining separate until it is more mature and then hopefully re-integrated with the other platforms in the main branch in the release of Qt 4.6.

11.3 Summary

In this chapter, we've looked at the issues involved in porting large middleware projects. As can be seen from the two examples, the technical details and effort required for such projects can vary enormously. The main thing all middleware projects have in common is that the customers are developers rather than end users. This is both a challenge and an opportunity, since the interface to your customers is much more complex than for an application but those customers are also potentially very useful contributors to the project. In the Web 2.0 world of development, driven by perpetual beta and user feedback, there's a new level of expectation for software products and services. Both open and closed source middleware projects need to find ways to involve their developer communities as early and as widely as possible in ports to new platforms.

The other important similarity highlighted by the two projects presented in this chapter is that the testing, validation and maturing of the port is likely to be significantly more effort than the initial development work by many, many times. This suggests that if there is some choice at the project selection stage then availability and quality of test code should be given added consideration and importance for middleware projects.

In the next chapter, we look at another work in progress. This time, it's an application that makes use of platform-independent middleware but also has some highly platform-specific components.

12

Porting a Complex Application

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

The project discussed in this chapter is the open source mobile client for a web-based community called Bluemapia.¹ Like the projects discussed in Chapter 11, it is a work in progress with only the early stages of porting completed at the time of writing. However, as you read this, the project should be much nearer completion and you are encouraged to visit the project website² in order to study the latest code; you could even get involved and help finish or improve it!

While the majority of the examples in this book are already cross-platform, this project is currently only available for Microsoft Windows, both desktop and mobile platforms. Although the primary goal of the porting project is to get a working client on the Symbian platform, there is also a longer term goal to support embedded Linux variants on Internet tablets, netbooks and other similar devices. In addition to porting the application, a major goal for the project is to make the code more portable. For this reason, the decision was taken very early to port or re-write the user interface for the Qt application framework, rather than the native S60 (Avkon) framework.

Readers who prefer to learn from examples first and get the theory later should find this chapter a useful source of practical details on separating platform-specific code from portable modules in order to make future porting easier. The relevant theory and more general advice on this subject can be found in Chapter 15; those who prefer to get the overview first and examples later might want to read Chapter 15 first.

¹ See www.bluemapia.com.

² See launchpad.net/bluemapia-qt.

12.1 Selecting a Project

Since all of the example projects in this book are open source, there is an inherent bias towards porting code that is primarily developed on Linux. However, there are relatively few native applications written for mobile Linux platforms at the time of writing, so developers porting mobile applications often need to do so from a platform that doesn't have such wide support of standard APIs. To address this, it was desirable to find a native mobile application for another smartphone platform to port to the Symbian platform as an example. The obvious platform choices, from the perspective of market share and developer interest, were Windows Mobile and iPhone OS. Open source applications for both platforms are extremely rare; indeed Apple's NDA, which used to be required for the iPhone SDK, basically prohibited collaborative open source development.³ Most of the open source iPhone software consists of trivial example applications or adds functionality that's part of the Symbian platform by default. This left me with Windows Mobile. One of the main deterrents to open source software on that platform is the need to purchase the full version of Microsoft Visual Studio in order to be able to develop anything.⁴ This means that there is also a fairly limited selection of open source projects to choose from here.

At this point in my search, fortune stepped in with a happy coincidence. The author of one of the newest and most interesting (at least from a porting perspective) open source applications for Windows Mobile was actively looking for help to port to the Symbian platform. His application, Bluemapia Mobile (which I'll refer to simply as Bluemapia for the rest of the chapter), is a client for a web and mobile social mapping service for boaters; allowing the sharing of details, locations and pictures of points of interest for a specific niche market. Having a very Web 2.0 concept, as well as using the GPS, camera and network access features of the device, made it an ideal source of example material. On further investigation there were some significant negative points from a porting perspective:

- It was written by a single developer for the bulk of the project.
- It had not previously been ported to any non-Windows platforms.
- Several external library dependencies would also need porting.

However, there were also some significant positive points:

- It is a relatively complete, stable and robust application with an active user community.

³ Apple removed the relevant NDA restriction on public discussion of iPhone development in October 2008.

⁴ Microsoft does offer a free 90-day trial, which could be very useful for a one-off porting project.

- It had recently acquired some public funding to continue development on Windows Mobile and help encourage open source contributions for ports.
- External library dependencies were already ported to multiple platforms.

Taken together, this probably represents a fairly typical porting project, rather than an idealized example, which is exactly what I was hoping for. There was also a desire on the part of the original author and another developer working on related software to build a community and produce a re-usable core of a mapping application. The opportunity to help build a genuinely useful example made this a very easy selection decision.

12.2 Analyzing the Code

Bluemapia was originally developed with the Win32 API, although some significant effort has been made to use POSIX-compliant code and separate out platform-specific parts where possible. Desktop and mobile versions of the application are both built from the same code base, which separates out the mobile-specific features and those that are too resource hungry for a mobile environment from a common core. The user interface is also split off into a `win32` sub-directory although the separation is not entirely clean – many of the files that are intended to be platform-independent include Windows-specific headers and some functions take platform-dependent arguments, such as handles to windows and device contexts. The code is fairly sparsely commented but is logically structured with functions, parameters and variables generally having meaningful names. Unfortunately there is relatively little test code, particularly at a module test level.

As mentioned previously, the project makes use of several external libraries and indeed files from other open source projects are also included directly. The libraries included in the source tree are:

- Exiv2 for handling Exif metadata (www.exiv2.org)
- Expat, a popular XML parser (expat.sourceforge.net)
- GDAL, the Geospatial Data Abstraction Library (see Chapter 11)
- JSON-C, for reading and writing JavaScript Object Notation (oss.metaparadigm.com/json-c)
- MiniWeb, a mini HTTP server (miniweb.sourceforge.net)
- libbsb, for reading and writing BSB format images (libbsb.sourceforge.net)

- b64, Base-64 Encoding Library (www.synesis.com.au/software/b64.html)
- nmeap, the NMEA parser (sourceforge.net/projects/nmeap)
- wcelibcex, standard C and POSIX extensions for Windows Mobile (wcelibcex.sourceforge.net).

The last of these, wcelibcex, is not required: it is specific to Windows Mobile, and Open C provides a much more complete alternative for the Symbian platform. The MiniWeb library is only used for the desktop version of the application so we needn't consider it further. The other libraries have no dependencies or just standard C and POSIX requirements.

The libraries are provided under a variety of licenses and it's important to understand the restrictions these place on an application using them. Most of the libraries use MIT or BSD licenses; this is some of the most permissive open source licensing available and basically allows you to do anything with the code as long as you retain and display the original copyright notices.⁵ Other libraries use the GNU LGPL, which has slightly more stringent requirements, although is still usable by most open and closed source projects. The Exiv2 library uses the GNU GPL v2 or later or can be licensed commercially. For Bluemapia, this is acceptable because the project is also licensed under the GNU GPL. For other projects this might force unwanted disclosure of source code or, in some cases, excessively restrictive licensing. There may also be licensing clashes, such as the use of the Eclipse Public License (as used by the Symbian Foundation), which can't legally be combined with code using the GNU GPL in the same project. Fortunately, all of the licenses used by these libraries are compatible with the GNU GPL. When porting a project that makes use of multiple open source components under a variety of licenses, you should always check for any potential conflicts or non-compliance with the license terms. It isn't safe to assume that the original authors have done so, since anyone that redistributes the code can be sued for copyright infringement.

12.3 Re-architecting

Having examined the code, it was clear that the contents of the win32 sub-directory would need to be re-written to port the user interface to Qt. The bulk of the external library code should be trivial to port with few changes and the rest of the application would need some refactoring to separate out the platform-specific parts. Beyond this there are some mobile-specific features that could be dealt with in a variety of ways.

⁵ Please read the licenses carefully to ensure you comply with the terms, even in these cases.

In the Windows Mobile version, the camera is controlled directly by the application. For the port, it should be possible to use one of the Qt mobile extensions provided by Forum Nokia (see Chapter 6) or provide a richer camera UI by using the native New File Service Client API⁶ to launch the device's built-in camera application and retrieve the image when it is captured. A third possibility would be to use the native Symbian CCamera API directly. These options all have their own strengths and weaknesses:

- The Qt mobile extension is simple to use and provides a cleanly separated interface for porting to other platforms but is not yet mature and stable.
- The New File Service Client API provides lots of functionality for very little effort but is not part of the public SDK⁷ and is not portable.
- The CCamera API provides maximum control but also maximum effort to implement and lots of source and binary breaks in the interface to deal with across devices and platform versions; it is not portable.

I decided that the best course of action was to use the New File Service Client API for the initial implementation and, if compatibility became an issue, then move to using the Qt mobile extension.

Another feature with multiple options is the access to GPS data. The Windows Mobile implementation for this feature is not optimal since it requires that the user set the correct COM port for an internal or Bluetooth-based GPS device and it uses the nmeap library to parse the data. The same implementation could be ported fairly directly to the Symbian platform, since internal GPS devices are typically attached to COM4 and the location of a connected Bluetooth RFCOMM serial port could be queried. However, the COM4 connection is just a convention and not an official standard; it's even possible that an application processor with a built-in GPS module might not use a COM port connection at all in some future devices. Symbian has a location framework that allows direct querying of the device location via any supported positioning method on the device, not just GPS data. It is also possible to get NMEA data directly from the location framework for GPS devices. For most applications, it is best to use the high-level API from the location framework to get latitude and longitude information, via a Qt mobile extension when Qt is being used.

In the case of Bluemapia, there are further complications. Bluemapia has some built-in GPS track simulation functionality that can inject NMEA data or read it from a file, which could be very useful for testing.

⁶ See wiki.forum.nokia.com/index.php/New_File_Service_Client_API.

⁷ It is included in the SDK API Plug-in (see wiki.forum.nokia.com/index.php/SDK_API_Plug-in) and comes with no binary compatibility guarantee.

There is a Symbian-specific alternative,⁸ which can inject simulated (or recorded) GPS tracks into the location framework. The fact that moving to the platform-specific solution breaks portability for the test tool as well as the code should be taken into account. An additional complication is that NMEA data is not just for GPS. The National Marine Electronics Association (NMEA)⁹ provides a common standard for communication between many marine electronic devices including echo sounders, sonar, anemometers (wind speed and direction indicators), gyrocompass, autopilot and GPS devices. Since Bluemapia is intended for use by sailors, communication with these other devices may be a future requirement.

The requirement for future portability tips the balance in favor of sticking with the current solution as closely as possible. However, future enhancements should be taken into consideration, such as the possibility of keeping the existing NMEA-parsing implementation for other marine electronics devices and moving the GPS functionality to the location framework to get access to additional positioning methods as they become more widely available. If that option is not taken, then an alternative enhancement would be to implement an auto-detection feature to detect internal and external GPS devices.

The other major architectural change will occur as a natural consequence of the move to Qt. Win32 applications typically run a message loop from `WinMain()`; here is the Bluemapia Mobile version:

```
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Dispatched messages are ultimately processed by the current window. The `WndProc()` function that processes messages for the main window in Bluemapia is 374 lines long, most of which are part of one giant `switch` statement. This is a very different style of programming to that used with Qt. The move to an object-oriented framework means that most of the functionality in this message-processing routine will migrate to the relevant widgets and a lot of it is automatically provided by the framework.

⁸ See wiki.forum.nokia.com/index.php/Simulation_PSY.

⁹ The standard is called NMEA 0183 but everyone uses NMEA for short (see www.nmea.org).

12.4 Setting Up the Development Environment

As with the GDAL port in Chapter 11, I started with the command-line environment to create some projects and get some of the external libraries to compile before moving to the Carbide.c++ IDE for debugging. I also used Microsoft Visual Studio and the Windows Mobile SDK while refactoring some of the code on the original platform. There were three reasons for going to the effort of setting this up:

- Opening the ‘solution’ files in Visual Studio helped to clarify exactly what is built for the mobile version of the application.
- Being able to run the application in the Windows Mobile emulator is good way of verifying what the functionality should be when implementing parts of the new UI.
- It allowed me to build and test refactoring work on the core while others worked on the first version of the new UI.

In general, when porting, it’s a very good thing to get a working build environment for the original platform if it’s possible and relatively easy (as long as you don’t need to go and buy new hardware or an IDE).

12.5 Integrating with the Symbian Build System

I decided to start with the external library dependencies,¹⁰ initially building them all as static libraries to link into the main application executable rather than deploying them as separate DLLs. The idea here was to do the minimum necessary to get something up and working, deferring any work on sharing these libraries to a later date. However as these are popular libraries it may well make sense to distribute them separately as DLLs in the future. GDAL was the largest and I decided to tackle that first (see Chapter 11).

12.5.1 Expat

Next I ported Expat; I had the advantage here that this library has already been ported to the Symbian platform several times. There is even a tutorial for porting an older version of the library to earlier versions of Symbian OS on the popular Symbian developer website called NewLC.¹¹ I was

¹⁰ I call them external here but Bluemania actually delivers versions of them in its source tree – they are only external in the sense that the source code has come from various third parties.

¹¹ See www.newlc.com/en/Building-an-XML-Parser-for-Symbian,555.html.

able to borrow and slightly modify the MMP file and configuration header provided there, create a `bld.inf` file and get the library compiling in less than half an hour.

12.5.2 JSON-C

Moving on to the JSON-C library, I made another brief search for previous port attempts but was unable to find any. A quick look at the library code suggested that it would be a trivial port. I manually created the `bld.inf` and MMP files via cut and paste from the Expat library with some simple edits. Examining the project file for the working version on Windows Mobile showed that all of the source files needed to be included except two test files (`test1.c` and `test2.c`) and a utility for reading and writing JSON objects to and from files (`json_util.c`). The library also has a configuration header file which is used to determine the presence or absence of certain system functions and headers. I made a first pass edit of the configuration, correcting some of the obvious issues before the first compilation attempt.

12.5.3 Exiv2

Exiv2 is another library that doesn't appear to have been ported to the Symbian platform previously. It is a lot larger than Expat and JSON-C in terms of the number of files, although much smaller than GDAL. I used the `qmake` utility with the `'-project'` option (as discussed in Chapter 11) to create a `.pro` file. I then edited the `.pro` file to remove source files for the many test executables and utilities that ship with Exiv2, before running `qmake` to generate the `bld.inf` and MMP files. The build files were then edited to remove all the Qt specifics and add standard C++ include paths. Following the convention used in the project, I created these files in a new `symbian` sub-directory in the root of the source tree. It was also necessary to hand-craft a configuration header for Symbian based on the `config.h.in` file used to generate configuration headers for Linux/Unix systems. The configuration header is called `exv_conf.h` and I decided to create the Symbian version of it within the `symbian` sub-directory, with the build files for the platform.

The process for creating configuration headers like this is very simple and since the format is very common I'll explain it here (perhaps someone will port Autotools to the Symbian platform one day and this won't be necessary, but until then it isn't a major task anyway). The `config.h.in` file has a number of entries like this:

```
/* Define to 1 if you have the 'alarm' function. */
#undef HAVE_ALARM
```

```

/* Define to 1 if you have the declaration of 'strerror_r',
   and to 0 if you don't. */
#undef HAVE_DECL_STRERROR_R

/* Define to 1 if you have the 'gmtime_r' function. */
#undef HAVE_GMTIME_R

/* Define to 1 if you have the <inttypes.h> header file. */
#undef HAVE_INTTYPES_H

/* Define to 1 if you have the <libintl.h> header file. */
#undef HAVE_LIBINTL_H

```

In most cases, you simply search in `\epoc32\include\stdapis` to see if the relevant functions or header files are available for the Symbian platform. If they are, you simply change the line in question, for example to:

```
#define HAVE_GMTIME_R 1
```

In the case of Exiv2 (and a good practice to follow for your own libraries), the defines also have a prefix added as part of the generation process, so the correct line is actually:

```
#define EXV_HAVE_GMTIME_R 1
```

You can do this manually for the whole configuration file, making careful use of find and replace; for particularly large configuration files, it may make sense to run the configure script in a Linux environment and then copy and modify the generated header. The only place where I didn't just mechanically define values where the relevant functions or headers are available was:

```

/* Define to 1 if you have the 'mmap' function. */
#undef HAVE_MMMap

/* Define to 1 if you have the 'munmap' function. */
#undef HAVE_MUNMAP

```

Both of the functions `mmap()` and `munmap()` (they're used as a pair) are provided in Open C/C++, but there are some limitations on the implementation imposed by the operating system (see Section 4.7.1). For the initial porting effort, I decided not to enable the use of these functions until the exact usage could be investigated.

An interesting observation from Exiv2 is that despite there being about 50 configuration options almost all of them are supported on the Symbian platform, demonstrating that the standard C and C++ environment is fairly complete.

12.5.4 The Main Executable

The remaining libraries used by Bluemapia are included directly in the project that builds the main executable, rather than linked separately. To create the necessary build files for the main executable, I used the 'qmake -project' trick again with some extra options to avoid including Windows-specific files:

```
qmake -project -norecursive -nopwd SRC SRC\Formats\BSB INC
```

Using these options allows you to specify that you don't want to include files from the current directory and qmake shouldn't automatically include all sub-directories, only adding files from the ones you explicitly specify. Note here that BSB is the only format in the `Formats` sub-directory because all the other formats are handled by GDAL.¹²

This was used for the first stage of the port when there was no user interface code for Qt. Another sub-directory, called `Qt`, parallel to the `Win32` directory was created for the UI files and these were then added manually to the `.pro` file as they were implemented and integrated.

12.6 Getting It to Compile

In order to get the application to compile without a user interface, three main tasks were necessary:

1. Resolve all compilation errors in each of the external libraries.
2. Refactor the core of the application to remove Windows-specific dependencies.
3. Comment out or stub all user interface references.

These are discussed in the sections below.

12.6.1 Compiling Expat

For the Expat library, there were some code changes required in the older port (see Section 12.5.1) relating to writeable static data; the data was

¹² In fact, GDAL can also handle BSB files but apparently libbsb does it better!

actually intended to be constant but had either not been declared as such or, in some cases, had not correctly been declared as such. Interestingly, these changes have been incorporated into the more recent releases but the Symbian build files are not part of the official distribution. I think it's worth giving an example here to illustrate the point addressed by the code changes, as you may come across this issue in other ports. Older versions of Expat used:

```
static const XML_LChar *message[] = {
    /* array initialization here */ };
```

Newer versions correctly use:

```
static const XML_LChar* const message[] = {
    /* array initialization here */ };
```

Note that the extra use of `const` makes both the data and the pointer to the data constant; omitting either of these forces the compiler to use some writeable static data.

Although Expat compiled without any issues at all, it should be noted that this port is now using the Open C implementation of the standard C library rather than the older `estlib.lib` that was used for the previous port. That is, it now uses a new library and the configuration includes additional features that are yet to be tested on the Symbian platform.

12.6.2 Compiling JSON-C

There was only a single error on my initial compilation attempt for JSON-C. The source of the error was this section of the file `printbuf.c`:

```
#if !HAVE_VSNPRINTF && defined(WIN32)
# define vsnprintf _vsnprintf
#elif !HAVE_VSNPRINTF /* !HAVE_VSNPRINTF */
# error Need vsnprintf!
#endif /* !HAVE_VSNPRINTF && defined(WIN32) */
```

As you can see, there is a configuration option which is not really an option! The only slightly puzzling issue was that this option, `HAVE_VSNPRINTF`, was not actually included in the configuration header in the project at all (defined or otherwise). However, this worked for Windows Mobile since `WIN32` is defined in that case. Fortunately, `vsnprintf()` is available in Open C so it was easy to define this value in the configuration header, at which point the library compiled without further issues. If this

define had been for a function that isn't supported on the Symbian platform then the only real options at that point would have been to find out what the function does and implement my own version or to port an implementation from another platform.

12.6.3 Compiling Exiv2

My first attempt to compile Exiv2 produced many tens of errors for the emulator build relating to the type `wchar_t`. The errors weren't produced by the GCC-E toolchain; indeed, the project compiled for GCC-E without any problems.¹³ The errors for the emulator build were simply because I'd forgotten to add the statement to the MMP file which enables the type:

```
OPTION    CW - wchar_t on
```

This is detailed in Chapter 4 as a standard requirement for using the STL, as Exiv2 does. Having resolved the `wchar_t` issue, I was left with relatively few errors in the emulator build. Unfortunately, they were all related to some very complex-looking use of templates. Here's an example of a template function that the compiler didn't like:

```
template <int N, const TagDetailsBitmask (&array)[N]>
std::ostream& printTagBitmask(std::ostream& os, const Value& value)
{
    const uint32_t val = static_cast<uint32_t>(value.toLong());
    bool sep = false;
    for (int i = 0; i < N; i++) {
        // *& acrobatics is a workaround for a MSVC 7.1 bug
        const TagDetailsBitmask* td = *(&array) + i;

        if (val & td->mask_) {
            if (sep) {
                os << ", " << exvGettext(td->label_);
            }
            else {
                os << exvGettext(td->label_);
                sep = true;
            }
        }
    }
    return os;
}
```

Specifically, there were problems with this line:

```
const TagDetailsBitmask* td = *(&array) + i;
```

¹³With a patch to the STLport to fix signed/unsigned comparison errors in `_sstream.cpp`, which should be in the official releases by the time you read this.

The comment above the line suggesting a bug in the Microsoft compiler when attempting to use this template construct is a big warning flag.

If you have to look twice at a clever use of templates to understand what it is supposed to be doing and why it's there then you can bet that anyone trying to implement a compiler will have had to think very hard about it. In many cases they won't have come to the same conclusion and compiler bugs are quite likely, even after many years of the C++ standards being in place. Even when you are relying on the GNU compiler for all platforms, it's not always safe to assume this kind of template magic is portable. I've seen a case where a use of templates worked fine on ARM targets but not for MIPS processors. Worse yet, the bug only became apparent at run time! The moral of this story is: 'Don't do clever things with templates if you want your code to be portable'.

Returning to this particular problem, I tried removing the workaround and just indexing into the array as follows:

```
const TagDetailsBitmask* td = &array[i];
```

Unfortunately this didn't work and the same was true for several other permutations on the syntax, including explicit casts where the compiler was complaining about implicit ones. At this point, I decided to try to contact the author of the code to find out if there was a simpler solution that avoided the template usage that my compiler didn't like. I used the forums on the project hosting site for Exiv2 and the author, Andreas Huggel, responded the next day. This is fairly typical of the excellent support provided in the open source community, assuming you do your homework before asking questions. It turns out that the templates are required to create a separate function for each of a fairly large number of arrays of constant data. However, after refreshing my memory on non-type template arguments, I rewrote the function shown above as:

```
template <int N, const TagDetailsBitmask* array>
std::ostream& printTagBitmask(std::ostream& os, const Value& value)
{
    const uint32_t val = static_cast<uint32_t>(value.toLong());
    bool sep = false;
    for (int i = 0; i < N; i++) {
        const TagDetailsBitmask* td = array + i;
        if (val & td->mask_) {
            if (sep) {
                os << ", " << exvGettext(td->label_);
            }
            else {
                os << exvGettext(td->label_);
                sep = true;
            }
        }
    }
    return os;
}
```

Replacing the reference to a fixed length array with a pointer like this seems to work for all of the compilers for the Symbian platform and also builds without errors in Microsoft Visual Studio. I made a similar change in another place where the same template construct was being used and the library then built without errors. I then reported my findings on the Exiv2 forum, suggesting that the changes could be integrated back to the main version to enable official support for the Symbian platform.

12.6.4 Refactoring the Application Core

When I first attempted to compile the core of the application, it became obvious that there were places where portability had not been considered carefully. Most of the files included the `stdafx.h` file which in turn provides most of the standard includes on Windows. It seems that this was originally included for a very user-visible form of debugging, where a standard Windows message box is used to display error messages to the user. With hindsight, a better option would have been to create a wrapper for the message box call and move the actual Windows-specific code into the UI layer. Although this might seem like an unnecessary piece of extra work initially, once a platform-specific header is included in platform-independent files, the compiler no longer catches the use of other platform-specific functions and types. Since a lot of code is created via the intelligent application of cut-and-paste with some editing, this can lead to ever declining portability. Although the Bluemapia code makes use of standard C types and some portable typedefs, such as `uint8_t`, these had become mixed with `TCHAR`, `BOOL`, `UINT`, `WORD` and `DWORD` throughout the code. An early task was to remove all references to `stdafx.h` and then go through and replace the non-portable types. Adding a wrapper for the debug messages also allows the debugging mechanism to be replaced at a later date. For example, you could have message boxes only produced in debug builds via some conditional compilation flags; release builds could then either do nothing with the messages or log them to a file or serial port.

Another area where platform-specific code was being used in the portable core was thread synchronization. Global mutex handles were used with the Windows functions `CreateMutex()`, `WaitForSingleObject()` and `ReleaseMutex()`. The main alternatives to this were to write a platform-independent layer that wraps the mutex handles and operations, replace with POSIX alternatives and then add appropriate implementations for Windows to `wcelibcex`, or use the Qt equivalents for both platforms. Further platform-specific code was used to access the Internet to load maps from URLs and upload data about points of interest.

There isn't a POSIX equivalent for HTTP access, only socket connectivity, so the options of a porting layer or use of Qt are preferred, since it's better to be consistent across the project for the purposes of future maintenance.

Now, if we were to write a porting layer, what would we wrap on the Symbian platform? The choices are basically native Symbian C++ or Qt code. However, Qt is already effectively a porting layer wrapping a combination of underlying POSIX and native Symbian C++ APIs; it doesn't make a lot of sense to wrap it further or to replicate the work Qt has already done wrapping the native APIs. Since we've already decided to use Qt for the project it makes sense to use it to replace platform-specific code wherever possible, although it may still be desirable to separate its use from POSIX/standard C code at a file level. The value of that separation is something to keep under review as the port progresses.

Beyond fixing the accidental use of non-portable types and wrapping or replacing the necessary use of platform-specific code where there were no portable alternatives, there is another type of refactoring we can make at this stage. This is where platform-specific functionality is unnecessarily mixed with portable code. There were several examples of this in the project but I present just one of them here, since the underlying point is the same in all cases. Here is (a slightly cleaned up version of) a function from the file `WebMap.cpp` in the original code:

```
// Returns the middle of the client area in the current image's
// coordinates.
bool GetWebMapOrigin(HWND hWnd, long *centerX, long *centerY)
{
    long w,h;

    GetOriginScreen(centerX, centerY);

    // Webmap originScreenX e originScreenY cannot be negative
    if(*centerX<0)
        *centerX=0;
    if(*centerY<0)
        *centerY=0;

    if((*centerX==0) && (*centerY==0)) // Not valid, return false,
        return false;
    else
    {
        // Middle of the screen
        GetScreenCenter(hWnd, &w, &h);
        *centerX=*centerX+w;
        *centerY=*centerY+h;
        return true;
    }
}
```

The purpose of the function is one of pure geometry: it is intended to provide the current center of the on-screen map in terms of pixels in a displayed map image (to understand this, realize that we may be displaying only a fraction of a full map image on the screen or parts of multiple tiles joined together). However, in order to determine the center of the image displayed in screen pixels, it needs to know about the size of the area in which a map is currently being displayed. This part of the functionality is encapsulated by the call to `GetScreenCenter()`, which understandably takes a handle to the window which is displaying the map. This forces the window handle to be passed into `GetWebMapOrigin()` as a parameter. To increase the portability of this code, the call to `GetScreenCenter()` can be pulled out of the function and the coordinates of the center of the screen can be passed in instead.

Of course, this change requires you to change any code that uses the function. In this case, a search of the project reveals it is only used by one other function, which is in the same file – `ManageWebMapCoordinatesAfterZoom()`. This function takes the window handle as its only parameter and passes it in to `GetWebMapOrigin()`, `SetWebMapOrigin()` and `GetScreenCenter()`. Examining `SetWebMapOrigin()` reveals that it uses the window handle to pass to `LatLonToImage()` and `GetScreenCenter()` yet again! Further enhancing the case for refactoring here, it turns out that `LatLonToImage()` doesn't actually use the window handle internally at all. So, the only purpose of the window handle and consequent platform dependency of this group of functions is to get the coordinates of the center of the screen, multiple times! Refactoring this code so that the necessary coordinates are passed into each of the functions rather than passing in the window handle makes the entire section of code platform-independent and also smaller and more efficient, since there is no longer any need for multiple calls to get the screen center. Searching the project again, I discovered that `ManageWebMapCoordinatesAfterZoom()` is only called from one place within the user interface code, so the impact of the change on the rest of the project was minimal and it was easy to test on the original platform.

12.6.5 Separating Out the User Interface

Fortunately the user interface for *Bluemapia* was already quite well separated from the rest of the application. There were only a couple of major exceptions:

- Portable code mixed with user interface code in `ChartCoord.cpp`
- Callbacks from the application core into the user interface to update position and status values.

The solution for `ChartCoord.cpp` was to split it into two files and move the portable parts to the application core. Callbacks are a little more difficult to deal with. The form they take in Bluemapia is direct function calls into the user-interface code. This violates the layering of the architecture, making the core of the application depend on the user-interface code. A popular solution to this for projects written primarily in C, like Bluemapia, is to use callback function pointers. Typically the user interface initiates an action and provides the layer underneath with a pointer to a function to call when there is any update. This has the disadvantages of not being type-safe and being rather more difficult to read, understand and debug (and hence also to maintain). In C++, the more popular solution is to define an observer class which the user interface layer then implements. Qt provides the signal and slot mechanism, which is more flexible than any of the others, but obviously requires the use of the Qt libraries for both parts of the application.

In Bluemapia, some of the callbacks were artifacts of the Win32 programming style. Moving the user interface to Qt simply removes the need for them. Other callbacks could be replaced with signals and slots where Qt classes are used to replace platform-specific functionality, to indicate network loading progress, for example. The remaining cases depend on the decision about the extent of Qt usage in the application core. It is likely that existing functionality will be wrapped inside Qt objects and the signal and slot mechanism will be used. If that option isn't taken, then the C++ class wrappers will be used and an observer class defined instead. Until the decision is taken and the user interface is fully implemented, the callbacks are simply commented out, or stubbed in a test harness where the result in the callback is required for testing.

12.7 Re-writing the User Interface

Writing a new user interface for an application is beyond the scope of this book. Porting from Win32 to Qt is generally much closer to re-writing than simply porting. As a general guide to the level of change required, I'll illustrate the process used with the coordinates dialog as an example. Within Bluemapia, this dialog allows the user to input coordinates in a range of formats, convert them to decimal degrees and, if desired, jump to the coordinate location on the map. Figure 12.1 shows the dialog displayed on a Windows desktop.

The only fundamental difference from the dialog on Windows Mobile is that the buttons at the bottom are hidden and a menu accessed from the softkey is used instead (obviously you don't get the semi-transparent Vista-style frame around the outside either). I use the desktop version here purely because it's easier to see all the available functions of the dialog in a single figure. The way that dialogs are constructed in Qt is

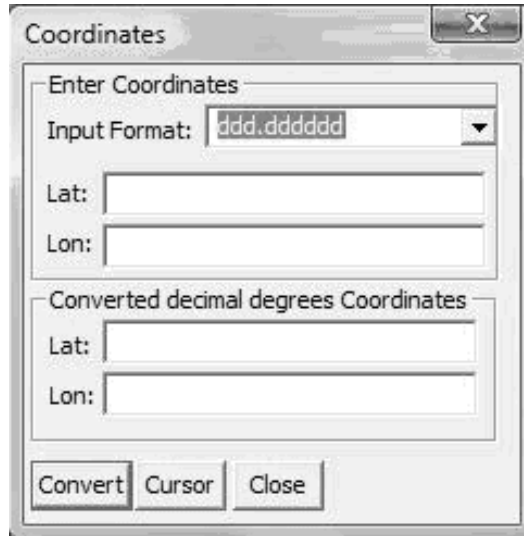


Figure 12.1 Coordinates dialog

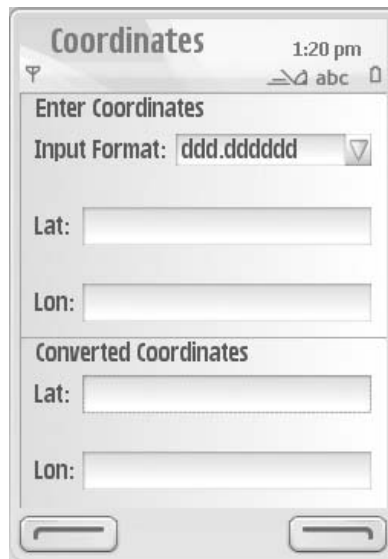


Figure 12.2 A simple clone of the dialog elements in the Windows emulator

completely different from the Win32 style, so rather than look at the code at this point I decided to use the Qt Designer to quickly build a dialog with the same elements as an initial basis for the new component. The

result is shown in Figure 12.2. (Note that there are no softkeys because this was developed with an early release of Qt for the Symbian platform, which is described in Chapter 11. This is not a final look and feel for the dialog.)

This dialog has very little functionality. The Convert button in the Windows dialog causes the function `ReadConvertCoordinates()` to be called:

```
static void ReadConvertCoordinates(HWND hwnd, double *lat, double *lon)
{
    TCHAR szValue[MAX_DLG_FIELD_LEN];
    TCHAR szValue2[MAX_DLG_FIELD_LEN];
    char strCoordLat[MAX_DLG_FIELD_LEN], strCoordLon[MAX_DLG_FIELD_LEN];
    int coordFormat = 0;

    coordFormat = (BYTE)SendMessage(GetDlgItem(hwnd,
                                                IDC_COORDLG_FORMAT),
                                    COORDLG_GETCURSEL,
                                    (WPARAM)0, (LPARAM)0);

    GetDlgItemText(hwnd, IDC_ENTER_COORDINATES,
                   szValue, MAX_DLG_FIELD_LEN);
    wtoc( strCoordLat, szValue );

    GetDlgItemText(hwnd, IDC_ENTER_COORDINATES2,
                   szValue, MAX_DLG_FIELD_LEN);
    wtoc(strCoordLon, szValue);

    if((*lat = ConvertCoordToDecimalDegrees(coordFormat, strCoordLat))
        != -1000.0)
    {
        sprintf(strCoordLat, "%.6lf", *lat);
        ctow(szValue, strCoordLat);
        SetDlgItemText(hwnd, IDC_CONVERTED_COORD, szValue);
    }
    else
    {
        AME_NotifyError(INVALID_LAT_COORD, 0);
    }

    if((*lon = ConvertCoordToDecimalDegrees(coordFormat, strCoordLon))
        != -1000.0)
    {
        sprintf(strCoordLon, "%.6lf", *lon);
        ctow(szValue2, strCoordLon);
        SetDlgItemText(hwnd, IDC_CONVERTED_COORD2, szValue2);
    }
    else
    {
        AME_NotifyError(INVALID_LON_COORD, 0);
    }
}
```

This function retrieves the current values from the entry text fields, converts and validates the input and then either notifies the user that the input is invalid or updates the converted coordinate values. In Qt, `QLineEdit` objects used for text entry can be configured to automatically validate their input. Where custom validation is required, a new subclass of `QValidator` can be created. In this case, the validator object can be updated dynamically when the coordinate format is changed. A signal is emitted when editing is finished and a valid value has been entered, so this can be used to trigger conversion and an update of the converted coordinates. This makes the Convert button and the function it calls entirely redundant in the Qt version of the UI.

The other function of this dialog, slightly mysteriously named `Cursor` in the original, is a 'show on map' feature. That is, it causes the application's main view to jump to the coordinates entered in the dialog. Here's the code that executes when this button (or softkey option) is selected:

```
ReadConvertCoordinates(hwnd, &lat, &lon);
if (!IsInsideBlueMapiaImage(lat,lon))
{
    AME_NotifyError( COORD_OUT_OF_MAP, 0);
    return FALSE;
}
hdcDlg = GetDC(hwnd);
InitLoadbar(hwnd,hdcDlg);
StartLoadbarWithText(10,"Moving Map to coordinates...");
IncrementLoadbar(1);
LatLonToCurrentImage(hwnd, lat, lon, &x, &y);
hdc = GetDC(hMainWnd);
IncrementLoadbar(3);
CenterMapImage(hMainWnd, hdc, x, y);
IncrementLoadbar(6);
ReleaseDC(hwnd,hdcDlg);
ReleaseDC(hMainWnd, hdc);
EndDialog(hwnd, FALSE);
```

The first thing to note is that this also calls `ReadConvertCoordinates()` and we've already established that this function is no longer needed because its functions will be performed by the relevant widgets. Following this, it does some range checking on the coordinates to ensure they are within the bounds of the maps we are currently accessing. This is a platform-independent function that can be preserved in the new version. If the check fails, an error is shown to the user using a standard Windows message box. The equivalent can be achieved with Qt using a `QMessageBox` widget.

Assuming the check succeeds, the dialog then sets up a progress bar and locates the coordinates on the current map image before centering the view there and closing itself. This functionality logically belongs to the main view of the application, so for the Qt version of the user interface

we just have the dialog emit a signal to say that the coordinates have been updated (providing the latitude and longitude) and connect it to a slot on the main view which generically handles coordinate updates (for example, for a new GPS position).

As you can probably see from this example, the existing code is mostly used as a guide to the functionality that a component provides, rather than actually being ported to the new platform.

12.8 Testing and Debugging

At the time of writing, only a little progress has been made towards testing and debugging this application. However, the strategy employed so far has been to build and test components independently and then integrate them incrementally. Fortunately, the third-party libraries are all supplied with their own test code in the form of command-line test executables which were fairly easy to build and run for the emulator environment. I also ran a subset of them in the target environment. Since the libraries are all essentially doing platform-independent data processing, the main reason for testing on the target is to ensure there are no data access alignment issues running on an ARM processor, otherwise the emulator should provide a reasonably accurate simulation for this type of code. The other potential issue is performance, but since the libraries are already in use on Windows Mobile, both issues should be fairly well covered.

For the user interface components, module testing has been left to the discretion of the contributors developing them, for the first phase of the port at least. In the case of the coordinates dialog described above, I used the QTest module to create simple unit tests that simulate key input and check the contents of the `QLineEdit` objects that display the converted output. A good tutorial for following this method can be found at doc.qt.nokia.com/4.5/qtestlib-tutorial.html.

As the components are integrated to create a more complex application, there is going to be a need for more comprehensive integration testing. It is possible to use the QTest module to develop this kind of test, but as the application grows, the complexity of the test cases starts to increase significantly and this is not desirable as it makes the tests hard to write and maintain. Other alternatives are manual testing, which is still very popular in mobile devices, or using some kind of automated GUI testing tool. A particularly impressive option for Qt-based applications is Squish for Qt from froglogic at www.froglogic.com/squish. It allows a very high level of flexibility in defining test cases, including simply performing the actions you want automated in the application while it records them. At the other end of the scale, you can also access an application's internal APIs from your test script to simulate unusual failure

cases. The tool supports all of the platforms on which Qt runs and this should also include the Symbian platform in the not too distant future. The only major downside is that this is, perhaps unsurprisingly, not a free tool.

12.9 Re-integrating

At present, there is no intention to re-integrate this port to Qt with the original Bluemania code and releases. In fact, a more likely solution in the medium to long term is that the Qt version of the application is ported back to Windows Mobile (and multiple desktop platforms) so that there is a single common code base for all versions. Essentially, the Qt version could be seen as an official fork of the project that will eventually replace the original.

12.10 Summary

In this chapter, I have discussed the issues involved in the selection, analysis, re-architecting, coding and testing of a port of an application based on Windows Mobile to the Qt application framework on Symbian. In the process, I've ported several external library dependencies, each of which was accomplished with relatively little effort thanks to Open C/C++. There are a few points I've covered along the way that are worth remembering:

- Porting to Qt rather than the native Avkon framework can significantly improve future portability.
- Don't use the most recent or exotic language features and syntax, such as complex use of templates, if you want your code to be portable.
- Refactor code to separate out portable parts at the beginning of the project rather than re-writing all functions that have platform-specific arguments or variables.
- User interface paradigms can be so different that it is often better to clone the visual appearance of an interface than attempt to translate the code.

This is the last of the example chapters. I hope you have found them useful, and don't forget that you can find additional URLs and example code on the wiki page for this book (developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform).

In the remaining chapters of the book, we examine the architecture of P.I.P.S., which forms the core of Symbian's standard C and C++ support and look at the security model on the Symbian platform and compare it to other mobile platforms. We finish the book with some advice on writing portable code and maintaining ports, so if you do have to write new platform-independent code, you should only have to do it once.

13

The P.I.P.S. Architecture

Fools ignore complexity; pragmatists suffer it; experts avoid it; geniuses remove it.

Alan Perlis

This chapter describes the architecture of P.I.P.S. An insight into the design and implementation of this framework will provide you with a better understanding of what the framework is capable of and what its limitations are. While you can generally rely on the architectural information presented in the chapter, implementation details of the various individual components may change as P.I.P.S. evolves.

The P.I.P.S. framework is composed of the following components: the backend, the core libraries, the addons, the `stdcpp` library, the emulator's writeable static data (WSD) library, the `locale/stdio` subsystems and the glue code. These components are shown in Figure 13.1.

The core libraries are those mandated by the POSIX standard on any compliant operating system: `libc`, `libm`, `libpthread` and `libdl`. The bulk of the code in `libc` and `libm` was ported from the FreeBSD libraries.¹ On the other hand, `libpthread` and `libdl` were written from scratch around native thread and DLL management interfaces.

The addons are utility libraries that were ported or written using the P.I.P.S. framework and made available on Symbian OS. These are included in the Symbian platform from the first release onward² but are also available as part of the Open C plug-in for earlier versions of Symbian OS with S60 3rd Edition. These include `libcrypto`, `libcrypt`,

¹ See www.freebsd.org.

² Technically, the first version of the platform released by the Symbian Foundation is Symbian^2, although in practice Symbian^1 (S60 5th Edition) devices should all include these libraries as well.

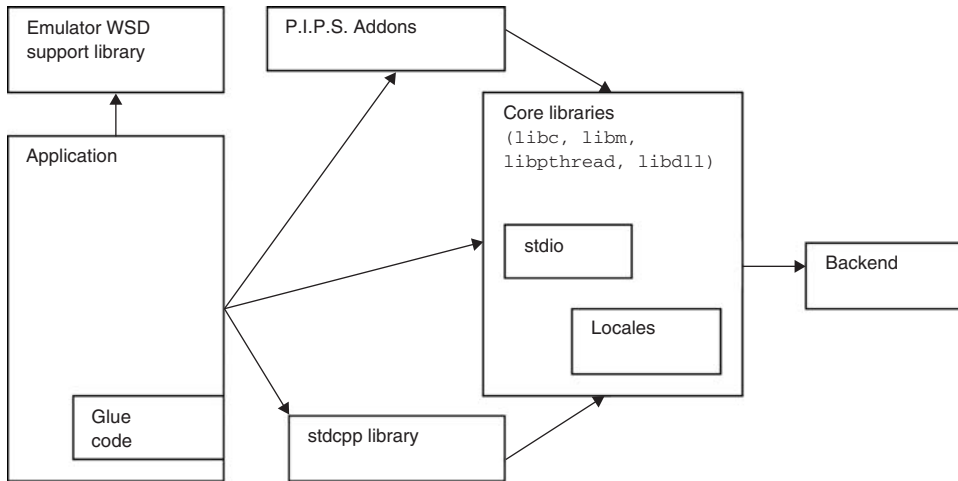


Figure 13.1 Components of the P.I.P.S. framework

`libssl`, `libz` and `glib`. Other libraries are regularly added to this portfolio.

13.1 The Glue Code

The entry point of an executable on the Symbian platform is the `E32Main()` function. POSIX-compliant programs, on the other hand, assume `main()` as their entry point. The glue code in P.I.P.S. is a static library that bridges the gap by providing a default `E32Main()` routine that performs some initialization (specific to P.I.P.S. applications) and then invokes the user-supplied `main()` function. The default `E32Main()` function transparently sets up a run-time environment for POSIX processes that enables them to execute as they are, without the need for extra function calls or unfamiliar programming idioms in their (ported) source code to initialize P.I.P.S. subsystems.

The glue code performs the following initialization steps before calling `main()`:

1. Create the backend (CLSI) singleton object (discussed in Section 13.3).
2. Check the origin of the process. If it was a result of a `popen/popen3/system/posix_spawn` call, it inherits a subset of open file descriptors and environment variables from the parent process.
3. Map the process's `stdin`, `stdout` or `stderr` to pipe ends, in the case of `popen/popen3`.

4. Set up certain environment variables (`HOME`, `PWD`, `TMPDIR`, `TZ`, etc.) for the process. These are accessed by the application via the `environ` global variable.
5. Initialize the Thread-Local Storage (TLS) and, if running on the emulator, the process-local variables (see Section 13.3). The TLS is the native thread-local store that allows Symbian platform DLLs to maintain thread-local information. It is implemented as a single word-sized field that the system maintains uniquely for each thread per DLL. In this field, we store a pointer to a dynamically allocated struct that contains data specific to this thread, such as the `errno`.
6. Set the current working directory of the process to the private directory of the application on the drive it was launched from.
7. Establish and configure a session with the `stdio` server (see Section 13.3.1).
8. Create a signal pipe and a watcher thread, in P.I.P.S. releases from v1.5.1.
9. Create a cleanup stack for the main thread.

After your program returns from `main()`, the glue code invokes `exit()` and performs cleanup operations mandated by the POSIX specification.

13.2 The Core Libraries

13.2.1 The Locale Subsystem (`libc`)

POSIX requires locale information to be maintained on a per-process basis. The Symbian platform does not support this natively – its `TLocale` class operates on the system locale. `libc` therefore maintains locale information in a separate `CLocale` class. An instance of the class is created per process and initialized with settings from the system `TLocale`. A change of locale effected by the application is a change to this `CLocale` object and, thus, remains local to the process.

13.2.2 The `stdio` Subsystem (`libc`)

The `stdio` subsystem in `libc` was, for the most part, ported unaltered from FreeBSD. We do, however, use methods from the native descriptor and `Math` classes for formatting and precision logic. P.I.P.S. v1.6 reverts to using a patched version of FreeBSD's formatting and precision routines – constraints, such as precision limits, in native methods have proved to be limiting for users of `stdio`.

13.2.3 The Math Library (`libm`)

Functions in `libm` do not, in general, use the native `Math` class. Algorithms to perform the various arithmetic and geometric operations were ported from FreeBSD.

13.2.4 The Dynamic Linking Interface Library (`libdl`)

STDEXE and STDDL are new target types introduced by P.I.P.S. Binaries with these target types have symbol information added to their headers. `libdl` uses this symbol information to support symbol lookup by name (`dlsym`). This symbol information is not present in the usual EXE and DLL target types.

13.3 The Backend

The backend is most important component of the framework, providing the bridge between the POSIX API layer and the underlying operating system. In that sense, it is the layer that performs system calls. It comprises a number of subcomponents which are described in detail in the following sections. Figure 13.2 shows the components of the subsystem.

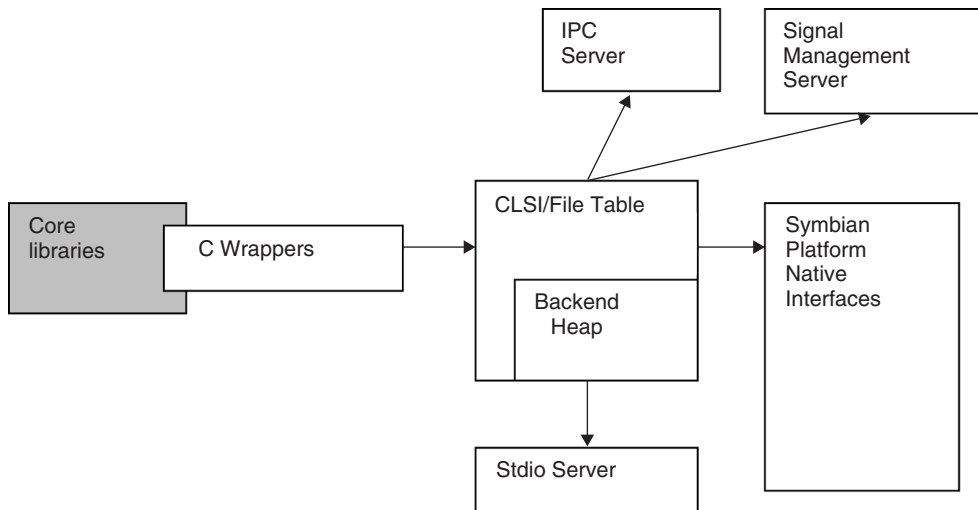


Figure 13.2 The backend components

13.3.1 The `stdio` Server

The `stdio` server services all `stdin` and `stdout` operations from P.I.P.S. processes. It is a transient, system-wide server³ created when the first P.I.P.S. process launches or when a hybrid application first invokes a `stdio` API. The user can set the media used for `stdin` and `stdout` operations in a configuration file `C:\System\Data\config.ini` (or `stdio.ini`), that is shared across applications in the system. This file is read every time a process connects to the server – any changes to the file affect processes that are launched after the change; processes that are running already are not affected.

The user can configure `stdio` media to be the console, a file, a serial port or a null device. Each medium has certain attributes that can also be set; for example, the path of a file or the width and height of a console. The default configuration file that ships with P.I.P.S. is reproduced below:

```
[STDIO]
STDIN = MEDIA1
STDOUT = MEDIA3

[MEDIA1]
type = file
path = C:\system\data\in.txt
max size = 100

[MEDIA2]
type = serial
baud = 214
port = COMM::10

[MEDIA3]
type = console
; The next two parameters set up the dimensions of the console window.
; -1 implies the default
width = -1
height = -1

[MEDIA4]
type = file
path = c:\system\data\out.txt
; The next parameter is currently ignored
max size = 1000
```

To change the default media for `stdout` to the file `out.txt`, change the `STDOUT` value in the `STDIO` section to `MEDIA4`. You can also add custom media options.

³ A transient server is one that is launched on demand, typically in the course of the first operation requiring the server. It terminates automatically after the last client drops its connection.

There is currently no provision for per-process configuration of `stdio` operations. Support for this is planned in a future revision of P.I.P.S.

Operations on `stderr` are performed locally in the process and do not involve the `stdio` server.

13.3.2 The IPC Server

The IPC server implements P.I.P.S. Sys V IPC interfaces – message queues, semaphores and shared memory. The semaphores discussed in this section are intended to be used for inter-process synchronization. Though nothing prevents them from being used between threads, there exist lightweight alternatives, such as `sem_t` and `pthread_mutex_t`, for this purpose.

Like the `stdio` server, the IPC server is a transient, system-wide server created when the first P.I.P.S. process attempts an IPC operation. Since these IPC resources are intended to be used between processes, they need to be visible across the system. One way to do that on the Symbian platform would be to assign them global names – this, however, could expose them to malicious access. Further, POSIX requires specification and application of access permissions when operating on IPC objects. Both of these considerations require the implementation of a server to oversee the lifetime and use of these IPC objects. While this adds a certain overhead to the process of creation, access (shared memory is an exception, see below) and deletion of these objects, it does ensure secure inter-process communication:

- Message queues are implemented as a linked list of buffers in the IPC server and do not use native `RMsgQueues`.
- Sys V Semaphores are implemented using a simple counter in the IPC Server. Since the IPC Server is a single-threaded process, updates to the semaphore happen synchronously.
- Shared memory objects are implemented using `RChunks`. While creation and deletion of these shared memory segments involve the IPC Server, reads and write happen directly. Thus, the speed associated with use of shared memory for IPC is preserved.

Permissions can be set during the creation of each IPC object or by using the `msgctl/semctl/shmctl` APIs. These decree whether a process can access a particular IPC object and (in the case of server-mediated operations, such as those on message queues and semaphores) whether a process can perform a particular operation. `RChunks` can be marked read-only or read-write on a global basis, not per handle. Since

access to shared memory is direct, it is possible to write to what is (as per `shmget`) a read-only shared memory segment.

Since the buffers, counter objects and RChunks that make up P.I.P.S. IPC objects are all created in the context of the IPC server, the system-wide limit on the number or size of these is bound by resource limits imposed by the Symbian platform on the server process. Although these have been raised to offer reasonable defaults, you still need to keep this constraint in mind when working with a large number of processes or IPC objects. The P.I.P.S. team is actively working towards making P.I.P.S. IPC more scalable. In a redesign planned shortly, the IPC server will only manage metadata pertaining to IPC resources in the system; the resources themselves will be hosted by the creator processes.

Pipes are the only IPC objects in P.I.P.S. that do not use or require the IPC Server. Both unnamed pipes and named FIFOs are implemented using native RPipes.⁴

Local sockets (those belonging to the `AF_UNIX/AF_LOCAL` address family) are implemented using native RSockets bound to `localhost` and a unique port. The mapping between the path (the address of the local socket) and the port number is stored in the IPC server. This is queried whenever the receiver needs the sender's address – implicitly in `recvfrom()` and `accept()` and explicitly in `getpeername()`.

13.3.3 C Wrappers

The C Wrappers are a layer between the core libraries and the backend. Their primary function is to provide C-callable front ends to backend APIs. They decouple the platform-specific code in the backend from the core libraries, allowing the ported code to remain largely untouched by platform semantics. This enables either or both of the backend and the core libraries to be upgraded separately. The C Wrappers are not so much a distinct component as they are a logical assembly of functions distributed between the core libraries and the backend:

- They convert arguments given in 'standard' datatypes to the native data types required as parameters to the real implementations.
- They perform some basic error checking (mostly on arguments) so as to be able to quickly return to the application in case of error.
- On the return path, they map native error codes to POSIX `errno`s, in cases where the real implementations haven't done so already.

⁴RPipes are a new Symbian IPC interface introduced for P.I.P.S. They provide kernel support for both named and unnamed pipes on the Symbian platform.

- In APIs that take string arguments, the wrappers perform conversions between ASCII/UTF-8 and Symbian's default UCS-2.

```
// NB: Illustrative only.
EXPORT_C int _stat_r (int* errno, const char *name, struct stat *st)
{
    // Some early error checking
    if(!st)
    {
        errno = EFAULT ;
        return -1 ;
    }
    // Convert from ASCII/UTF-8 to UCS-2
    wchar_t tmpbuf[MAXPATHLEN+1];
    if ((size_t)-1 != mbstowcs(tmpbuf, name, MAXPATHLEN))
    {
        TInt err = Backend()->stat(tmpbuf, st);
        return MapError(err, errno);
    }
    // Failure to convert
    errno = EILSEQ;
    return -1;
}
```

13.3.4 The CLSI

The Local System Interface class (CLSI) is the backbone of the backend component. A single instance of the class is created per process – either in the glue code or when the first P.I.P.S. API is invoked from a hybrid application. The CLSI object stores session handles to the various system servers (RFs, RSocketServ, etc.) and P.I.P.S. support servers (the `stdio` server, the IPC server, etc.). It also plays host to the file table.

Nearly all of the functionality of the backend is implemented in methods of the CLSI object or its subobjects.

13.3.5 The Backend Heap

On the Symbian platform, each thread can have its own heap. This implies that any memory allocated from one thread (say, buffers allocated during an `fopen()`) cannot be freed from another (during `fclose()`). Also, certain system resources are thread-local, by default. To enable them to be created and closed from any thread in the process, they need to be constructed on a shared heap. This allows any internal allocations they make to be freed without error, when the resource is closed from outside the creator thread. POSIX APIs assume a process-wide heap. This is implicit in the fact that file descriptors and IPC resources can be created, used and deleted from any thread in the process. To enable this semantic for native resources in P.I.P.S., we create a new P.I.P.S.-only heap called the backend heap. P.I.P.S. APIs transparently switch to this heap when

creating (or destroying) file descriptors, native resource wrappers and IPC objects.

P.I.P.S. also uses this heap to store some internal objects such as the CLSI and the File Table that need to exist through the process lifetime. These objects are only cleaned up after the process exits `main()`. If they were allocated on the user heap and the application used memory-leak detection routines (UHEAP macros) before exiting `main()`, incorrect leaks may be reported.

13.3.6 File Table

The File Table is, essentially, an array of open file descriptor objects. Methods are available to create, close or duplicate file descriptors. The File Table stores information on child processes created by the current process. It also provides methods that enable its file descriptors to be inherited during process creation. `RConnection` objects that are used in socket operations in P.I.P.S. are members of the File Table object.

Every file descriptor in P.I.P.S. is a wrapper around the equivalent native resource. Thus, regular and temporary file descriptors map to `RFiles`; socket descriptors map to `RSockets`; pipes and FIFOs map to `RPipes`; and serial and `IRCOMM` port descriptors map to `RComm` objects.

Each file descriptor type is represented by a class. These inherit from a common base class that describes a number of virtual functions for derived classes to re-implement according to their type.

The `CFileDesc` class, below, represents regular files:

```
class CFileDesc : public CFileDescBase
/*
Abstractions for a regular file
*/
{
public:
    CFileDesc();
    ~CFileDesc();
    void SetState(const TDes& params);
    TInt Open(RFs& aSession, const TDesC& aName, int mode, int perms);
    TInt LSeek(int& offset, int whence);
    void Read(TDes8& aDesc, TRequestStatus& aStatus);
    void Write(TDes8& aDesc, TRequestStatus& aStatus);
    TInt FStat(struct stat *st);
    void Sync(TRequestStatus& aStatus);
    TInt IoctlCompletion(int aCmd, void* aParam, TInt aStatus);
    TInt Fcntl(TUint anArg, TUint aCmd);
    static void MapStat(struct stat& st, const TTime& aModTime,
                       TUint& aAttr, const mode_t aMode = S_IFREG);
    TInt Truncate(TInt anOffset);
    TInt SetAtt(TUint aSetAttMask, TUint aClearAttMask);
    TInt ProcessLockParams(TInt& pos, TInt &lock_len,
```

```

        TInt& lock_type, struct flock* anArg);
inline int CreateLock()
{
    return iLock.CreateLocal();
}
inline RFile& FileHandle()
{
    return iFile;
}
inline const TInt Offset() const
{
    return iPos;
}
inline const TUint32 Size() const
{
    return iSize;
}
virtual TInt Type()
{
    return EFileDesc;
}
protected:
    virtual TInt FinalClose();
    TInt Alloc();
private:
    TInt FileRead(TUint8* aPtr, TInt aLength);
    TInt FileWrite(TUint8* aPtr, TInt aLength);
    TInt Flush();
    TInt DoSync();
    TInt DoRead(TDes8& aDesc);
    TInt DoWrite(TDes8& aDesc);
    TInt Pos();
    TInt Ext();
protected:
    RFile iFile;
    ...
};

```

13.3.7 Signal Management Server

The Symbian platform has no native support for signals. Signals are emulated in P.I.P.S. (version 1.5.1 and onward) using a signal pipe and a watcher thread for each process. A central signal management server mediates signal delivery across processes. When a process launches, it creates a pipe and registers this with the management server. It then creates a thread that monitors this pipe for data. This thread, called the `SignalHandlerThread`, runs with the highest priority in the process but is nearly always idle and does not impact execution of the other threads. When one process invokes `kill` on another, it sends a message to the server. The server inserts this message into the recipient's pipe, after verifying the sender's right to issue such a signal (`SIGKILL` and `SIGSTOP`

require the sender to have the `PowerManagement` capability; all other signals are just passed through). The watcher thread on the recipient awakes and checks the signal against the signal mask of the main thread in the process. If the signal isn't blocked, it marks the signal as pending on that thread. When the thread next enters or leaves a system call,⁵ it executes signal handlers against each signal in its pending queue.

Since this is only an emulation of signals, we cannot interrupt threads or hook into context switches to run signal handlers. This implies that if an application spends most of its time outside system calls (in some computation loop, for example), it does not receive a signal unless it calls the non-standard `issigpending()` libc API intermittently to check for outstanding signals.

Though this is not currently enabled owing to backward compatibility considerations, blocking system calls (such as a blocking `read/recv` or `sleep`) are also interruptible. To accomplish this, P.I.P.S. waits for a signal event alongside the actual event in such blocking APIs. If this arrives first, the thread interrupts the event wait, cleans up and returns to the user with `errno` set to `EINTR`. P.I.P.S. developers are currently evaluating options to make this feature available.

Signals may be directed to a specific thread using `pthread_kill()`.

13.4 Emulator Writeable Static Data Support

13.4.1 Using Writeable Static Data in DLLs on the Emulator

In the Symbian platform emulator, processes are emulated with Win32 threads. DLLs are Win32 shared DLLs. This means that all processes end up sharing the same copy of writeable static data (WSD) variables in the DLL, violating the per-process semantics we expect. To enable DLL writers to work around this and emulate per-process WSD, P.I.P.S. provides a library called `ewsd` with a single interface `Pls()`. To use this library, you need to:

1. Wrap all your WSD variables in a `struct`:

```
#ifdef __WINS32__
struct Globals
{
    int gref;
    int gerr;
};
#endif
```

⁵ System calls in P.I.P.S. are APIs that are implemented in the backend – in CLSI or one of its subobjects. For example `strcpy()` and `siblings` are not system calls.

2. Include `pls.h` and define a getter that uses `Pls()` to create and return a singleton reference to this WSD struct. The `Pls()` function can be supplied with a routine that initializes your global variables to whatever you choose. If you don't supply one, the struct is initialized with zeroes.

```
#include <pls.h>
TInt InitializeGlobals(Globals* pglobals)
{
    pglobals->gref = 0;
    pglobals->gerr = 0;
    // Initializer must return KErrNone to indicate success.
    // Else the Pls routine panics.
    return KErrNone;
}
struct Globals* GetGlobals()
{
    return Pls<Globals>(KDllSecureId, &InitializeGlobals);
}
```

3. Replace the definitions of your global variables with individual getter functions that invoke the getter defined in Step 2 and return a pointer to that variable in the struct.
4. Define macros that map the variable names to the getters defined in Step 3, so that the rest of the DLL code is opaque to all this and continues to use the WSD variables as usual.

```
#ifdef __WINSW__
// Define getters for each WSD variable
int* get_gref()
{
    return &(GetGlobals()->gref);
}
int* get_gerr()
{
    return &(GetGlobals()->gerr);
}
// Define macros to make the getters opaque to other code
#define gref (*get_gref())
#define gerr (*get_gerr())
#endif
```

13.4.2 The EWSD Library

To implement the above-described support, P.I.P.S. maintains an array of structures – one for each process in the system. Each process is identified by its PID.

```
struct TWsdNode WsdArray[KMaxNumberOfProcesses];
```

Each of these structures in turn hosts another array of structures – one for each WSD-bearing DLL loaded in this process. The DLL is identified by its UID.

```
struct TWsdNode
{
    TProcessId iPid;
    TInt iFirstUnusedSlot;
    struct TDllData iDllData[KMaxNumberOfLibraries];
};
```

In the structure corresponding to a particular DLL, we store a pointer to the `struct` containing WSD data (globals, above).

```
struct TDllData
{
    TUid iLibraryUid;
    TAny* iPtr;
};
```

Whenever a process invokes `Pls` (by de-referencing a WSD variable), the function first checks the outer array to determine if it has encountered this process before. If it hasn't, it reserves the process an entry in the array. It then checks the array of DLLs to see if a `struct` of WSD data has been stored already. If it has, it simply returns a pointer to this `struct`. If it hasn't, it allocates space for the WSD `struct` and saves the address as `Pls` for this DLL in this process.

The memory to store the WSD `struct` is allocated using the Windows `VirtualAlloc()` function. If a new `Pls` entry is to be allocated and there is no space in the PID array, the `SetPls()` routine checks the status of all processes already in the array and cleans up slots corresponding to dead processes.

13.5 Summary

P.I.P.S. is a fairly complex framework. In its approximately 150,000 lines of code, there are straightforward ports, emulations, wrappers, work-arounds and hacks. The Symbian platform was not designed to

be POSIX compliant and P.I.P.S. has had to work hard to present a mostly compliant layer around the native operating system. This chapter provides an overview of the various parts of the framework and brief insights into how they function. I hope that this knowledge is useful to you in your porting. Further information can be found at ***developer.symbian.org/wiki/index.php/P.I.P.S._is_POSIX_on_Symbian***.

14

Security Models

The organization cannot trust the individual; the individual must trust the organization.

Ray Kroc

The Symbian platform supports the development of native (C and C++) code by third parties, such as network operators and independent software vendors. Such code may be packaged up in an installation file called a SIS file, and then installed on the phone by the end user.

Before the introduction of platform security, the code could call any of the APIs exposed by the operating system, even those not explicitly published in the SDK.¹ So the user had to trust an application completely if they installed it. Anything they installed could potentially spend their money (by making premium-rate phone calls, for example), access personally or commercially sensitive data (calendar information and email messages may be commercially sensitive or simply personally private), or affect the behavior of other applications (for example, by changing system settings).

‘Platform security’ is the collective name for a group of technologies in the Symbian platform whose primary function is to control application access to data and system services. Platform security gives the user more control by allowing them to install applications which they trust in a limited way: the user can install an application and be confident that it only does the things it claims it needs to do. For example, a simple game may be refused network access or access to a user’s personal data.

We can distinguish three interrelated components of platform security:

- *The Capability Model:* Every process running on the device runs with a set of capabilities. When a client process requests a service from a

¹ Before introducing platform security in Symbian OS v9.1, some phone manufacturers attempted to stop unauthorized third-party developers from writing code that made phone calls by not including the relevant header files. Of course, they soon leaked out anyway.

server process, the server process can examine the client's capabilities and reject the request if it does not satisfy the server's security policy. Thus a given process cannot just do anything it wants: if it doesn't have the right capability, it can't make phone calls.

- *Process Identity*: Every independently certified EXE on the device has a globally unique secure identifier. Servers are able to examine the identifier of their client processes, so a server may know exactly which EXE is requesting a given service. Every independently certified EXE on the device may have a vendor identifier that securely identifies the organization that created it.
- *Data Caging*: Different parts of the file system are restricted, so only processes with specific capability sets or secure identifiers can read or write particular directories.

The technology outlined in this section is accompanied by a code-signing and certification scheme which determines the capabilities that may be granted to binaries² and assigns secure identifiers to EXEs. The definitive guide to platform security is Heath, C. (2006) *Symbian OS Platform Security*, John Wiley & Sons.

14.1 The Capability Model

Every binary in the system – including operating system code – contains a capability word embedded in the file. The structure and definition of this word is the same for EXEs and for DLLs, but its meaning and the operating system's use of it is completely different, as I explain in Sections 14.1.2 and 14.1.3.

14.1.1 Capability Word Structure and Definition

The capability word is a bitfield in which each bit represents a single capability. In practice, not all bits are used at the moment.

Each capability is logically independent of all the others: so having any one capability does not imply ownership of any other and there is no logical hierarchy of capabilities, thus no direct 'super-user' analog.

Capabilities are divided into *User capabilities* and *System capabilities*.

User Capabilities

User capabilities are designed to be meaningful to the end user. Depending on the specific security policy chosen by the manufacturer, the phone

² This chapter uses the term 'binary' to refer to any native executable file, including both EXEs and DLLs.

may present the user with the option of granting that capability to an application.

User capabilities are defined for activities that could cost the user money (such as using the network) or violate their privacy (such as accessing the address book).

The complete list of user capabilities is:

- `NetworkServices`: the ability to make phone calls, send emails, and so on
- `LocalServices`: the ability to use short-link network services, such as Bluetooth
- `ReadUserData`: the ability to read the user's private data
- `WriteUserData`: the ability to modify or create the user's private data
- `Location`: the ability to access to the device location
- `UserEnvironment`: the ability to access information about the user's environment, including the ability to record audio and use the camera.

System Capabilities

System capabilities are not expected to be meaningful to the end user, so it is not expected that users make decisions about them.

Some system capabilities allow access to services at a lower level than user capabilities, essentially providing backdoor access to activities already protected by a user capability. For example, direct access to communication device drivers could provide backdoor access to the network and direct access to application private data could provide backdoor access to user data. Additional capabilities are defined for activities that could affect the integrity of the system as a whole and for certain other quite specialized activities, such as Digital Rights Management (DRM).

A subset of system capabilities is given below; for the complete list, refer to the Developer Library or Heath (2006):

- `ReadDeviceData`: the ability to read system settings (e.g. IAP settings)
- `WriteDeviceData`: the ability to modify or create system settings
- `CommDD`: the ability to access communication device drivers
- `DRM`: the ability to access content protected by some form of DRM
- `AllFiles`: read access to all of the file system and write access to application private data

- TCB: read and write access to the part of the file system where binaries are stored.

Trusted Computing Base

Trusted Computing Base (TCB) is a standard term referring to that part of a system responsible for ensuring the security of all other parts of the system. In the Symbian platform, the Trusted Computing Base comprises:

- the kernel
- the file server
- the loader
- the secure installer.

TCB capability has only an indirect relationship with the Trusted Computing Base itself. Not all components which require TCB capability are actually part of the Trusted Computing Base: for example, an enterprise device management system may need TCB capability because it needs direct read and write access to binaries, even though it is not primarily responsible for maintaining the system security.

Because the capability bits are stored in binaries themselves, the TCB capability enables code possessing it to assign any set of capabilities to any binary on the device. So TCB-capable code is able to subvert the whole security model and must be trusted to the highest degree.

14.1.2 Use of EXE Capabilities

Assignment of Process Capabilities

When an EXE is launched, it and all the DLLs it links to are loaded into the EXE's process. The capability word is copied from the EXE header and assigned to the process.

Let's consider an example (see Figure 14.1). Suppose the EXE is an application called `MyApp.exe`. It links against an application engine DLL, `MyAppEngine.dll`. The engine in turn links against the telephony client DLL, `etel.dll`. `MyApp.exe` has two capabilities, `NetworkServices` and `ReadUserData`.

When `MyApp.exe` is launched, it and the two DLLs it links to are loaded into its process. The capabilities are copied from `MyApp.exe`'s header into the kernel-side representation of the process.³

³In reality, all binaries also link against `euser.dll`, but we leave this out for the sake of simplicity.

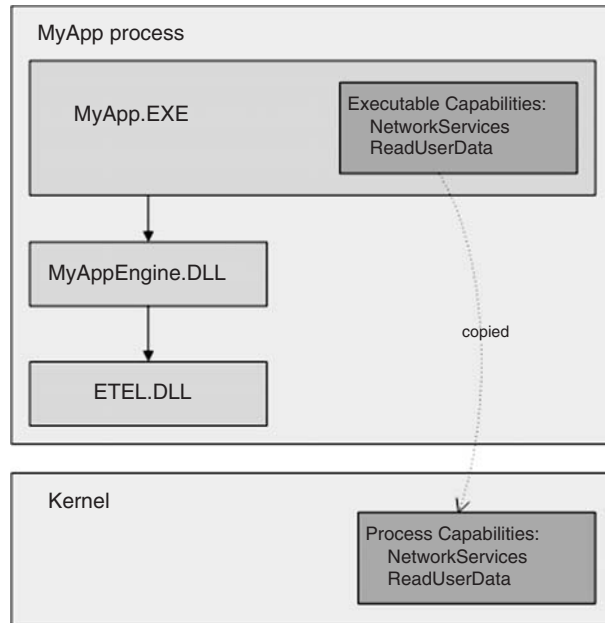


Figure 14.1 Process capabilities are copied to kernel at launch

So, the process capabilities:

- are always the same as the EXE capabilities⁴
- never change during the process lifetime.

Process Capability Checking at IPC Boundaries

Suppose MyApp wants to make a phone call. It calls a `Dial()` method on ETel, which sends a message to Symbian's telephony server via the IPC mechanism. The telephony server requires the `NetworkServices` capability for `Dial()` requests. So it looks at the capabilities for the calling process (this information is maintained by the kernel, so cannot be forged by the client), as shown in Figure 14.2, and denies the request if the capability does not appear.

What actually happens is that the capability set is accessible through the `RMessagePtr2` object passed to the server from the kernel. Servers have a lot of flexibility in how to use capabilities. They can require capabilities for a client's `Connect()` method to succeed or they can allow connections and police individual API calls. They can also accept

⁴ A possible change to this behavior is under consideration for a future release to enable greater flexibility for run-time environments but the current behavior would be maintained for existing binaries.

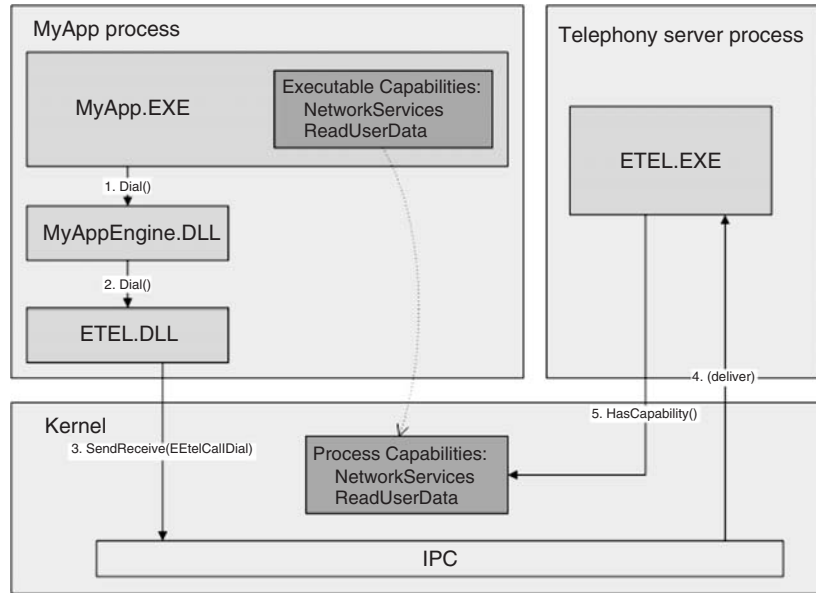


Figure 14.2 Process capability checking

or reject calls based on the arguments passed in as well as the capabilities: the file server in particular does this, as we will see. If a capability check fails, the server may complete the request with an error code, such as `KErrAccessDenied`, or panic the client.

Servers can implement a security policy for their APIs by deriving from `CPolicyServer` and defining a `CPolicyServer::TPolicy` object which specifies what check should happen when each IPC method is called.

14.1.3 Use of DLL Capabilities

DLLs also have capabilities, represented in the same way and referring to the same privileges. The capabilities of a DLL do not affect the capabilities of the process that loads it: process capabilities are entirely defined by the capabilities of the EXE.

DLL capabilities are intended to solve a different problem. We have seen that all code in the process runs at the same capability level. But any given binary cannot possibly know about all the other binaries it links to, both directly and indirectly. So how can it trust that the code it is linking to does not abuse the privileges derived from the EXE?

For example, suppose `MyApp` links to a DLL that claims to do some entirely innocuous stuff – string to integer conversion, for instance. This

DLL may have been supplied by a third party of whom the developer of MyApp may know nothing. When MyApp calls the innocuous DLL function, there is nothing to stop that DLL from making a premium rate phone call, because the telephony server only checks that the process has *NetworkServices*, as shown in Figure 14.3.

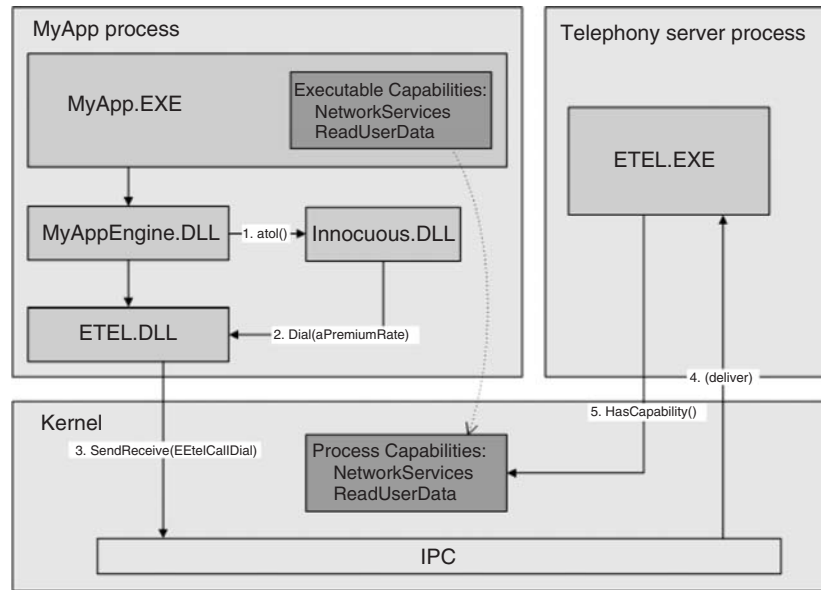


Figure 14.3 Code in a DLL runs with the capabilities of the calling process

The application engine thinks it's doing `atoi()`, but it has actually been subverted to spend the user's money. This issue is solved by DLL capabilities.

The rule for DLL capabilities is: a binary cannot load any DLL which has fewer capabilities than itself. This rule is enforced by the loader. Thus DLL capabilities mean that the DLL is trusted not to abuse the privilege it has been granted and so may safely be loaded into processes running with that capability level.

In Figure 14.3, if `Innocuous.dll` does not have the *NetworkServices* capability, then `MyAppEngine.dll` fails to load it. If `Innocuous.dll` does have *NetworkServices*, then it can be trusted to be loaded into processes running with *NetworkServices* and not do things such as making premium rate phone calls inside an `atoi()` function.

This means that the capability set for any DLL is the union of the capability sets of all processes in which they may ever need to run.

14.1.4 Working Out which Capabilities You Need

For EXEs, the capability set you need depends on what you want to do, in particular on which server APIs you need to call and what data you need to access.

There are not many capabilities and their definition is, in general, clear so you should start off with a good idea of which capabilities your application is likely to need.

If your thread fails a capability check, the server is likely to complete the request with an error code such as `KErrAccessDenied` (-46); alternatively, it may panic the client. In any case, for emulator builds, if the `epoc.ini` file under `/epoc32/data` in the SDK installation directory contains the line `PlatSecDiagnostics ON`, then the server writes to the debug output a diagnostic line that looks something like this:

```
*PlatSec* ERROR - Capability check failed - A Message (function
number=0x4000100a) from Thread
lbs-application.exe[10285a9b]0001::Main, sent to Server !PosServer,
was checked by Thread EPosServer.EXE[101f97b2]0001::!PosServer and
was found to be missing the capabilities: Location . Additional
diagnostic message: Checked by CPolicyServer::RunL
```

The diagnostic message tells us:

- the name of the thread whose request failed, in this case `lbs-application.exe[10285a9b]0001::Main`; the first part of the name is usually the name of the EXE
- the name of the server that rejected the request; in this case the positioning server, `!PosServer`
- the name of the thread in which the server was running, `EPosServer.EXE[101f97b2]0001::!PosServer`
- the name of the missing capability, `Location`
- the value of the enumeration for the IPC which failed, `0x4000100a`, which enables us to know exactly which operation failed; in this case, the enumeration name with the value `0x4000100a` is `ELbsPosNotifyPositionUpdate`, so the operation which failed is a request for a location update.

A simplistic approach to determining the capabilities you need would be to start with zero capabilities and keep adding them until you stop getting failures of this sort. It should be clear that sometimes messages

like this are indicative of more fundamental problems in the code, so you should start with some expectation of the capabilities you will need and pay attention to any surprising failures. For example, if you only want to read contact details, but the API you are using requires the `AllFiles` capability, then you should treat the error as an indication that you should be using a different API, not that you need the `AllFiles` capability. Especially if the missing capability is a very powerful one, such as `AllFiles`, or even if it is just an unexpected one, it is worth considering whether there is a better way to achieve the same result.

For DLLs, the capability set you need is the union of all the capability sets of all processes in which they may ever need to run. For an application engine, this would be the same as the application EXE, but for a more general-purpose DLL the set can be much bigger.

If your EXE has more capabilities than the DLL to which it is linking, the EXE fails to launch and the following diagnostic is written to the debug output:

```
*PlatSec* ERROR - Capability check failed - Can't load  
lbs-application.exe because it links to lbs-appengine.dll which has  
the following capabilities missing: AllFiles
```

This tells us:

- the name of the EXE, `lbs-application.exe`
- the name of the DLL with insufficient capabilities, `lbs-appengine.dll`
- the name of the missing capability, `AllFiles`.

Capabilities are specified in MMP files using the `CAPABILITY` keyword followed by a list of the names of the capabilities:

```
CAPABILITY ReadUserData WriteUserData NetworkServices
```

The special keyword `ALL` can be used to include all capabilities and this can be followed with the names of capabilities preceded by a minus sign to include all capabilities except those listed. The following line includes all capabilities except `TCB` and `DRM`:

```
CAPABILITY ALL -TCB -DRM
```

14.2 Process Identity

14.2.1 Secure ID (SID)

Every binary in the system – including Symbian platform and phone manufacturer code – contains a 32-bit *secure identifier* value, referred to as the SID. When an EXE is launched, the SID is copied from the EXE into the process.

- The EXE cannot change the SID.
- The process SID is always the same as the EXE SID.
- If the same EXE is launched multiple times, all the processes have the same SID.

Exactly as for capabilities, servers can find out the SIDs of their clients and use this to decide whether or not to service a request. SIDs are not used in DLLs. SIDs are divided into two ranges:

- the *protected* range, from 0x00000000 to 0x7FFFFFFF
- the *unprotected* range, from 0x80000000 to 0xFFFFFFFF.

EXEs which contain SIDs in the protected range must be signed by an approved authority (this is enforced by the software installer at installation time). That means code running on the device can be assured that protected SIDs were properly assigned by Symbian from the global SID space. Protected SIDs are globally unique and can be used to identify individual applications. Protected SIDs are supplied by Symbian Signed, and are specified in the MMP file using the SECUREID keyword. If no SECUREID statement is present in the MMP file, the UID3 is used instead; omitting SECUREID and using UID3 are functionally identical, but using SECUREID makes the intent clearer.

EXEs which contain SIDs in the unprotected range have no such constraint, so there is no guarantee of uniqueness.

If an application has a secure identifier in the protected range the system can protect it from other applications. This means that it can store private data in its own directory accessible only to other processes with the AllFiles capability, that other applications cannot impersonate it, and that other application installation packages cannot alter it.

14.2.2 Vendor ID (VID)

Each binary in the system *may optionally* also contain a 32-bit *vendor identifier* value, referred to as the VID. As for SIDs and capabilities, the EXE VID is copied into the process, cannot be changed by the EXE itself, and can be queried by servers.

VIDs are, of course, not unique: all EXEs provided by a single organization will share the same VID value. All VIDs are protected, so if an EXE contains a VID it must be signed by an approved authority. A VID can be used by a device manufacturer to implement a security policy in which only their own applications are allowed to use certain interfaces. In such a case, using a SID or even a list of SIDs is too constraining as they may not know in advance which of their applications may need to use it.

Vendor identifiers are less often useful to developers, but we could imagine the developer of a suite of applications using the VID to share user preferences across their different applications. VIDs may be obtained from Symbian Signed and built into an EXE using the `VENDORID` keyword in the MMP file.

14.3 Data Caging

Data caging is the term used to describe the practice of restricting access to certain parts of the file system. The most obvious use for this is to protect the binaries themselves. As capabilities, SIDs and VIDs are stored in the binaries, write access to them needs to be restricted.

Additionally, data caging protects read-only resources from accidental or intentional modification by unauthorized code and provides each EXE with its own private data area. Data caging is enforced by the file server on a directory basis. Table 14.1 shows the restrictions placed on

Table 14.1 Capabilities required to access top-level directories

Directory	Capabilities required for		Comments
	Read access	Write access	
<code>\sys\</code>	TCB	TCB	All binaries are stored under <code>\sys\bin\</code> . The loader loads binaries only from this location. Hashes of binaries stored on removable media (see Section 14.3.1) are stored in <code>\sys\hash\</code> .
<code>\resource\</code>	None	TCB	Read-only resources (e.g. bitmaps) are stored here.
<code>\private\</code>	AllFiles, or a SID equal to the subdirectory name	AllFiles, or a SID equal to the subdirectory name	Application private data is stored here. It contains a subdirectory for each EXE, of which the name is the SID of the EXE, e.g. <code>\private\101f7663</code> .

some top-level directories. Access to all other top-level directories is unrestricted.

14.3.1 Data Caging and Removable Media

The capability model enables data caging, because the file server controls access using capabilities. Data caging enables the capability model, because it protects the integrity of the binaries where capabilities are stored.

Removable media threaten the second of these assertions. If we allow binaries to be stored on removable cards, then we can no longer guarantee their integrity. If the card were to be inserted into a reader attached to a PC, we could directly alter the capabilities inside the binaries.

When binaries are installed to a removable drive, the installer calculates a hash of the binary and stores the hash on the internal drive, under `\sys\hash\`. Then when the loader loads any binary from a removable drive, it recalculates the hash of the binary and checks that it matches the hash retrieved from the internal `\sys\hash\` directory. If it doesn't, it fails to load.

- Any binaries on removable media which have not been through the installer are not run, because the hash is not found.
- Any binaries on removable media which have been altered after installation are not run, because the hash does not match.

14.3.2 Sharing Data Securely

If you need to keep data private, or simply have no need to share it, you can keep it in your private directory. If you need to share your data in a controlled fashion, you need to define a security policy for it. You could implement the policy by implementing a server, but in general you do not have to, because there are mechanisms provided by the operating system for sharing data with access control. In particular:

- You can define a security policy for a central repository key, or a group of keys. The policies can control read and write access based on a combination of secure identifiers and capabilities, and are defined in the INI file that defines the syntax of the keys themselves and other metadata. See the Central Repository How-To guide in the Symbian Developer Library for details on specifying a security policy for keys.
- Databases created using Symbian's DBMS component (`RDbNamedDatabase`) can be associated with one of a number of predefined security policies, although it is not possible to define new ones.

- Databases created using Symbian's SQL component (RSqlDatabase) can be associated with a user-defined security policy, encapsulated as a TSecurityPolicy object, to control who can read and write to the whole database or individual tables, and who can modify the database schema.
- Access to Publish and Subscribe properties (RProperty) can be controlled using customized security policies defined as TSecurityPolicy objects. Additionally, for new programs, the category value for a property must now be the same as the defining process's secure ID. This protects the properties a process defines and relies on, stopping other processes from taking them over: it is, in effect, data caging for properties.

The Symbian Press booklet on Data Sharing and Persistence gives an overview of these technologies; it can be found at developer.symbian.org/wiki/index.php/Data_Sharing_and_Persistence_with_Symbian_C++.

14.4 Code-Signing and Certification

14.4.1 Digital Signatures

A digital signature is a mechanism for associating some data with a private key and, by extension, with the owner of that key.

Digital signatures are an application of public-key cryptography. In public-key cryptography, a key pair is generated and a message encrypted using one of the keys in the pair can only be decrypted using the other key. In a digital signing scheme, the signer generates a key pair and keeps one key private, publishing the other one. To generate a signature, the sender creates a fixed-length hash of the message, encrypts the hash using the private key, attaches the signature to the original message and sends both to the recipient, as shown in Figure 14.4.

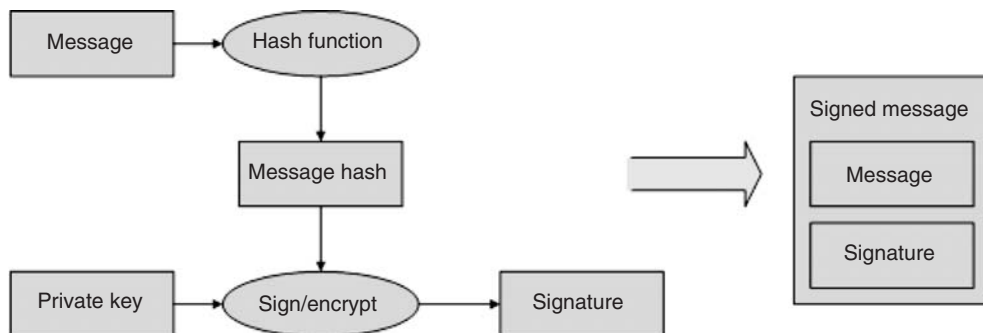


Figure 14.4 Digital signing

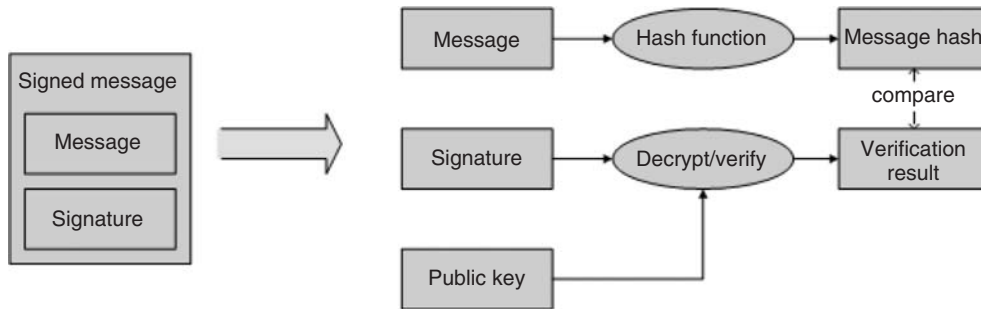


Figure 14.5 Digital signature verification

The recipient of the message can decrypt the signature using the public key and compare the result with the hash of the original message: if they match, the signature is valid, as shown in Figure 14.5. This tells the recipient:

- that the message has not been altered since the signature was generated
- that the signature was generated by someone with access to the private key corresponding to the public key used for verification.

14.4.2 Certificates and PKI

A public key infrastructure (PKI) is a system that enables the recipients of signed messages to obtain authentic copies of public keys in order to verify them. In a PKI, there is an organization called a Certification Authority (CA), which has a signing key pair. The public key is known to be authentic: this is generally because it is embedded in the devices or programs which are used by the recipients of signed messages.

The entity wishing to send signed messages generates a key pair, then applies to the CA, supplying the public key and some proof of identity. The CA verifies the identity and signs a data structure consisting of its name and the sender's name and public key. This data structure is a public key certificate for the message signer: it is an assertion by the CA that the public key belongs to the named entity.

Now the sender signs the message as before, but includes the certificate along with the message. The recipient extracts the public key from the certificate and verifies the signature on the message, then takes the CA public key embedded in the device and verifies the signature on the certificate. The recipient can consider that the message really does come from the individual named in the certificate, assuming that the following conditions hold:

- The recipient has a valid copy of the CA key, not a forgery.
- The CA is competent to identify and certify keyholders.
- The sender has kept his private signing key safe.

The CA's public key is also embedded in a certificate, but it is self-signed: that is, the signature on the certificate is generated using the CA's own private signing key (the counterpart of the public key in the certificate). Because the CA's certificate is the foundation of all trust in this model, it is known as a root certificate.

A good summary of the difficulties in implementing PKI effectively is given in Gutmann, P. (2002) 'PKI: It's Not Dead, Just Resting', available at www.cs.auckland.ac.nz/~pgut001/pubs/notdead.pdf.

14.4.3 Digital Signatures for Software Authentication

We can distinguish, very roughly, between two sorts of statement that a certified signature on some installable software may be expected to make:

- a statement about *identity*: the digital signature tells the user the name of the creator of the software
- a statement about *assurance*: the digital signature tells the user something about the trustworthiness of the software.

The advantage of the first of these is that statements about identity are relatively easy to make and they are the kind of statements for which commercial CAs are authoritative: this is what they do. The disadvantage is that they are not very helpful to end users, who are more likely to be seeking some assurance that the software will behave as they expect. The main disadvantage of statements of assurance is that evaluating the trustworthiness of a software package is very expensive and extremely difficult, especially when the software needs to be tested in a black-box fashion.

14.5 Certification and Platform Security

The Symbian platform distinguishes two main classes of application: trusted and untrusted.

Trusted applications are applications that have been signed either directly by an independent authority such as Symbian Signed, or indirectly with a key that has itself been certified by such an independent authority. The key criterion is that in some sense an independent authority has been able to exercise some control over the application.

Untrusted applications are applications that have been developed and deployed entirely outside the control of any independent authority. They may be unsigned or, more often, self-signed: that is, signed with a key pair whose accompanying certificate is signed with the same key, rather than by an independent authority.

The concept of trusted and untrusted applications intersects with platform security in two places: secure EXE identity and capability assignment.

14.5.1 Trusted Applications and Secure EXE Identity

System software components use secure EXE identifiers to decide whether to grant access to various resources: the most obvious system component and resource are the file server and the application's private data area, respectively.

If two EXEs were allowed to have the same SID, either accidentally or maliciously, one could impersonate the other and gain access to its private data files. Symbian Signed (or some other trusted authority) needs to check that EXEs only use the secure identifiers that have been assigned to their developers.

Any EXEs containing secure identifiers from the protected space must be signed by an approved authority. This is a consequence of the way secure identifiers are implemented in the Symbian platform.

14.5.2 Trusted Applications and Capability Assignment

The link between code signing and secure EXE identity is fixed by the implementation. However, the rules determining what, if any, code signing is needed for a given capability set are determined by the device manufacturer as a matter of policy.

We can distinguish four general approaches to answering the question of what an application is allowed to do:

- *an open policy*: the application is automatically granted the capability it asks for
- *a discretionary policy*: the system asks the user if they are prepared to grant the capability
- *a controlled policy*: the application must be granted the capability by an external authority
- *a closed policy*: the application is never granted access to the capability.

These policies represent different trade-offs between openness and control. To simplify,⁵ we could say that different stakeholders have different interests. Developers would like to minimize the barriers to getting their applications written and distributed. Users would like some assurance that the applications they install are reliable. Network operators and device manufacturers want stable devices so as to minimize their support costs and do not always trust users to make good security decisions or even, where applications such as DRM are concerned, to cooperate.

The approach taken by Symbian can be seen as a compromise in which:

- applications using only APIs which require no capabilities do not need to be trusted (*open*)
- untrusted applications which require capabilities from a specific subset may be installed if the user agrees (*discretionary*)
- applications which require capabilities outside that set must be trusted (*controlled*).

The controlled set is further subdivided, with more onerous verification requirements being imposed for capabilities considered especially powerful or sensitive, such as `DRM`, `AllFiles` and `TCB`. In theory at least, no capabilities are closed to third parties, although the most sensitive system capabilities (such as `DRM` and `TCB`) may be hard to acquire.

The certification options outlined here are subject to change in the future, although the underlying principles are likely to stay the same.

14.5.3 Untrusted Applications

Applications which do not need any capabilities and do not need the protection afforded by having a SID from the protected range do not need to be trusted. Additionally, untrusted applications are typically allowed the following set of capabilities:⁶

- `ReadUserData`
- `WriteUserData`
- `Location`

⁵ The reality is more complex, of course: device manufacturers have an interest in promoting third-party development and developers have an interest in promoting confident users.

⁶ Until S60 3.2, the `Location` capability was not available to untrusted applications. This is a good example of the volatility of the certification requirements.

- LocalServices
- NetworkServices
- UserEnvironment

There are two caveats to this:

- Untrusted applications are only allowed these capabilities at the user's discretion. At installation time, the installer asks the user whether they want the application to be installed and lists the capabilities it is requesting. If the user refuses, the application is not installed.
- There is no guarantee that all devices are configured to allow users to grant capabilities at their discretion.

Technically, untrusted applications may just be unsigned. But in practice, S60 mandates that all applications are signed, so for untrusted applications this means that they must be self-signed. For self-signed applications:

1. The developer generates a signing key pair.
2. The developer creates a certificate by signing the public key and their name with the corresponding private key.
3. The developer signs the application with the private key and distributes it along with the certificate.

The process of creating a key pair with a self-signed certificate and using it to sign a SIS file is further detailed at Forum Nokia ***wiki.forum.nokia.com/index.php/How_to_sign_a_.Sis_file_with_Self-Sign_Certificate***.

14.5.4 Certifying Trusted Applications

To gain access to a wider set of capabilities, applications need to be trusted and this means the developer needs to interact with Symbian's certification process.

Symbian Signed (***symbiansigned.com***) defines the certification process and is the main interface to it for developers. Further information about Symbian Signed can be found at ***developer.symbian.org/wiki/index.php/Complete_Guide_To_Symbian_Signed***.

Publisher ID

For an application to participate in Symbian's certification scheme, the developer's company must have a Publisher ID. This is a certificate issued

by a CA which binds the company to their public key. Only companies can obtain publisher IDs.

The Publisher ID is not used to sign applications for installation or to identify individual applications: the private key associated with it is used to sign messages from the developer to Symbian Signed, which uses it to verify that the messages really are from the developer in question.

Since the Publisher ID identifies the company rather than any individual application, the same Publisher ID can be reused for many different applications from the same company.

Express Signed

Under the Express Signed scheme, applications are considered trusted but are not independently tested. Thus the primary function of the process is to identify the developer rather than to make any statement about the application. The process is as follows:

1. Developers sign applications using the company's Publisher ID private key and submit them to Symbian Signed, along with a statement that they have performed a specified set of tests. As an audit measure, Symbian randomly selects applications and runs these tests against them: companies whose applications fail the tests may be banned from using the Express Signed scheme.
2. Symbian Signed verifies the developer's identity using the Publisher ID, then generates a new key pair specific to that application. It certifies this key pair against a root certificate present in the phone's ROM and uses the key pair to re-sign the application.
3. The developer downloads the re-signed application for distribution.
4. At installation time, the secure installer checks the signature on the application and checks that the certificate included with it is properly signed with the root certificate in ROM.

Applications signed under the Express Signed scheme may be granted capabilities only from the following set:

- LocalServices
- Location
- NetworkServices
- ReadUserData
- UserEnvironment
- WriteUserData

- PowerMgmt
- ProtServ
- ReadDeviceData
- SurroundingsDD
- SwEvent
- TrustedUI
- WriteDeviceData

Certified Signed

Under the Certified Signed scheme, application developers are identified and the applications are also independently tested, so the signature makes some kind of independent statement about the reliability of the application.

The Certified Signed process is very similar to Express Signed, except that the developer uploading the signed application to Symbian selects an independent test house from a list supplied by Symbian. Symbian passes on the application to the test house which runs a defined set of tests.

If the application passes the tests, Symbian re-signs the application with an application-specific key pair and certifies the key pair. At installation time, the secure installer works out that the application is Certified Signed and allows it a wider set of capabilities.

Applications signed under Certified Signed may be granted any capability. However, very sensitive capabilities – AllFiles, DRM and TCB – are also subject to device manufacturer approval.

14.6 Development Code

Code-signing presents special difficulties for development and testing: developers want to be able to debug code on the target device, meaning that they need to be able to install code on the device, but changing the code by fixing bugs breaks the signature.

For code under development, Symbian offers the Open Signed scheme, in which a relatively lightweight process allows applications to be installed on a limited number of devices identified by IMEI (the unique number assigned to every GSM or UMTS phone). Open Signed comes in two flavors, with and without a Publisher ID.

14.6.1 Open Signed Online

The Open Signed Online scheme operates without a Publisher ID:

1. The developer creates an unsigned SIS file.

2. The developer uploads the SIS file to Symbian Signed, together with the IMEI of the target phone and the set of capabilities needed.
3. Symbian generates a signed SIS file.
4. The developer downloads the signed SIS file for distribution.

The SIS can only be installed on a single phone: the one whose IMEI matches the IMEI sent to Symbian Signed. Applications signed under this scheme get access to the same set of capabilities as those available under Express Signed:

- LocalServices
- Location
- NetworkServices
- ReadUserData
- UserEnvironment
- WriteUserData
- PowerMgmt
- ProtServ
- ReadDeviceData
- SurroundingsDD
- SwEvent
- TrustedUI
- WriteDeviceData

14.6.2 Open Signed Offline

The Open Signed Offline scheme operates with a Publisher ID. The developer gets a Developer Certificate to re-sign the application when the code changes.

1. The developer generates a request for a Developer Certificate, signing it using the Publisher ID, and uploads it to Symbian Signed along with the list of IMEIs for the target phones.
2. Symbian Signed generates a Developer Certificate and associated key pair, which the developer can download and use to sign the application.

Applications signed under this scheme may be allowed access to the same set of capabilities as those available under Certified Signed, and the application may be installed on up to 1000 phones, identified by IMEI.

14.7 Tool Support

The `makekeys` command-line tool can be used to generate key pairs, certificates and certificate requests. To sign SIS files, you use the `signsis` tool. For details of the syntax of these tools, see the Symbian Developer Library.

Carbide.c++, the IDE for Symbian development, integrates signing SIS packages: select Project, Properties, Carbide.c++, Build Configurations and then the SIS Builder tab, as shown in Figure 14.6.

When you click the Add button, a dialog appears in which you can supply your private key and certificate file, as shown in Figure 14.7.

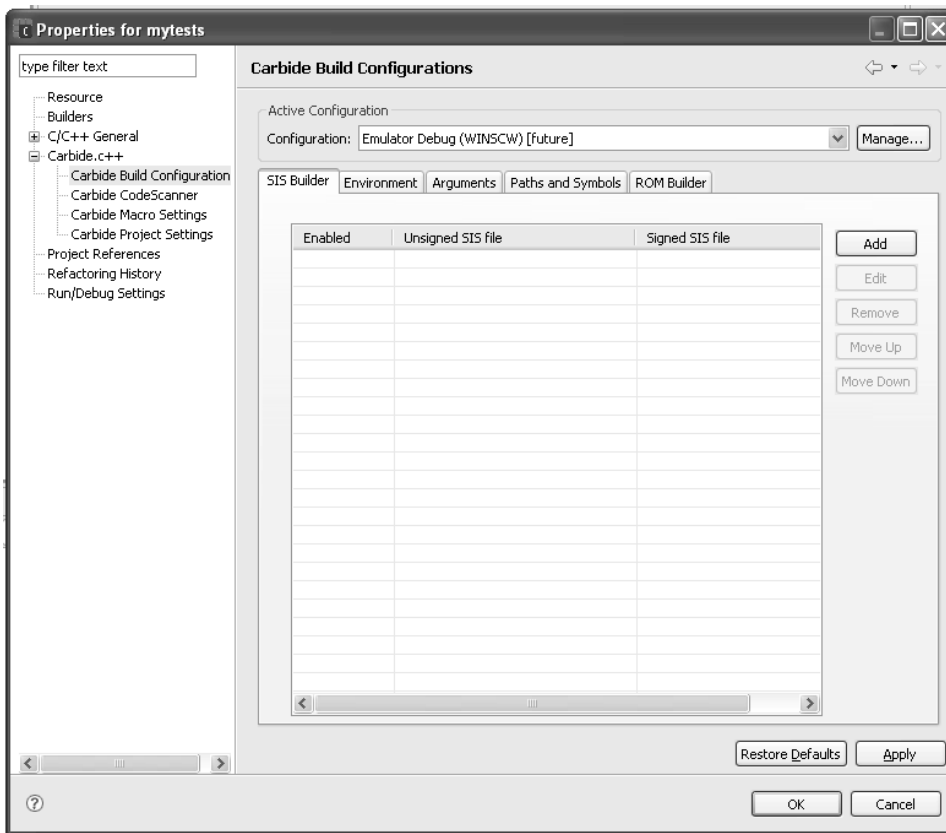


Figure 14.6 Carbide.c++ SIS Builder settings



Figure 14.7 SIS file signing options

14.8 Symbian Platform Security Compared with Other Models

14.8.1 Android

Application Identity

Although Android is based on Linux, it implements a very similar model to the Symbian platform, by treating every installed application as a different user with a locally unique Linux User ID. This User ID is Android's equivalent of the Symbian SID: all code running as that user has access to a single data cage insulated from the rest of the system, and runs with its own set of capabilities (called 'permissions' in Android).

The difference lies in how the two identifiers are managed. Symbian manages protected SIDs by assigning globally unique UUIDs to developers

and requiring that applications containing them are signed by an independent trusted authority. Most importantly, this means that no other process can impersonate an application by using its SID and thereby gain access to its private data area.

In Android's model, every application has to be signed, but applications may be (and, in general, are) self-signed. The signature is not used to verify that the application has been through a centralized certification process but to ensure that once an application has been installed on a device, new installation packages (such as upgrades) are seen as part of the same application only if they are signed using the same private key. There is a one-to-one mapping between a signing key pair and all versions and upgrades of the same application: applications signed with different keys are treated as different applications and installed as different users with their own data cages. Of course, this also prevents impersonation: once an application is installed, other packages can only access its data cage if they are signed with the same private key. Effectively, the signing key itself is the globally unique identifier.

The advantage of this approach is that developers may take advantage of secure identification without needing to be signed by a third party, and thus without needing to obtain a certificate. The disadvantage is that the identifier is not globally unique, so it is not possible for code to grant or reject access to system services based on an identifier value.

Capability Assignment

Capability assignment in Android is also remarkably similar to Symbian's approach. Code exposing services to applications can define a security policy for each service. This policy is called a permission and has an associated protection level which maps onto the four sorts of security policy identified in Section 14.5.2:

- A normal permission is granted to any application that asks for it (*open*).
- A dangerous permission is granted to any application that asks for it, if the user agrees (*discretionary*).
- A signed permission is granted only to applications signed with the same key as the key used to sign the service provider itself (*controlled* or *closed*, depending on the PKI).
- A signed or system permission is granted only to applications in ROM or applications signed with the same key used to sign the service provider (*controlled* or *closed*).

Applications implementing their own services can define their own permissions policy and the system services have a set of standard permissions with predefined policies.

One question which is not clear is whether system services marked as signed are controlled or closed. Unless Android intends to operate a certification scheme where the key used to sign system code is also used to certify third-party code, then these services are effectively closed.

14.8.2 Other Linux-based Platforms

Other platforms based on Linux adopt different security models and policies. Openmoko is completely open by philosophy: end users are encouraged not only to install applications but also to modify and flash new firmware.

Nokia's Maemo-based Internet tablets are sold as consumer devices and lots of functionality is protected by both a super-user password and a research and development mode setting. Access to this functionality requires the user to:

1. Install a terminal application or `ssh` into the device.
2. Find online the required password and instructions for accessing research and development mode.
3. Understand and carry out the instructions.

The idea is that users who have the technical capability to gain access to protected functionality also have the capability to fix the device if necessary.

14.8.3 iPhone OS

iPhone OS has a very different concept of third-party development to the Symbian platform. Subject to the specific security policy, third-party developers for the Symbian platform can create all kinds of programs: not only GUI applications, but servers exposing services to other applications; daemons running in the background and responding to events; plug-in components to generic system frameworks; device drivers; and so on.

In contrast, third-party development for iPhone OS is restricted to self-contained GUI applications which are completely isolated from each other. Developers cannot implement components that provide services to other programs. The security policy specifies that:

- All features in the public SDK are *controlled*.
- All other features are *closed*.

Every developer must obtain a certificate which identifies them as a registered iPhone developer: they use the corresponding private key to

sign their application and submit it to Apple for distribution. Apple verifies the developer's identity and reviews the application for conformance to the SDK agreement. If the application is judged to be conformant, Apple re-signs it and makes it available through the iTunes App Store. Applications can only be installed via the App Store: because Apple also controls the distribution channel, it can revoke applications easily if it later decides that an application has violated the agreement. So it is not possible for developers to develop and distribute applications without Apple's direct oversight.

The iPhone OS security model has no concept of capabilities: all properly certified applications can call any of the published API functions as long as they do not violate the SDK agreement. Because third-party applications cannot offer services to each other, there is never any need for developers to implement their own security policy.

Applications developed for the iPhone are assigned a unique identifier when they are installed and this is associated with a sandbox for the application which includes a private data area, like the private data cage supported in the Symbian platform. The iPhone does not implement any system services to enable applications to share data in a controlled fashion.

14.8.4 Windows Mobile

Like the Symbian platform, Windows Mobile gives device manufacturers a degree of flexibility in designing a security policy. The version of Windows Mobile 6 targeting high-end mobile devices, Windows Mobile Standard, supports a two-tier security model in which applications can run in Normal or Privileged mode:

- Applications running in Privileged mode can call almost all APIs⁷ and write to all system files and registry keys.
- Applications running in Normal mode cannot access a fairly small set of sensitive APIs, write to protected registry keys or system files.

Whether an application runs in Normal mode or Privileged mode depends on the certificate with which it is signed: a device which supports the two-tier model contains root certificates for Normal mode and for Privileged mode. Self-signed applications are treated as unsigned.

What happens to unsigned applications depends on two other security policy configuration items: whether they are allowed to execute at all and, if so, whether the user is prompted before they are allowed to execute. If they do execute, they execute in Normal mode in the two-tier

⁷ An additional subset of APIs is available only to the OEM.

model. The commonest configuration is to allow unsigned applications to run after prompting the user and getting their consent.

So the Windows Mobile approach uses a mixture of policies for different resources:

- A few resources are *closed* to third parties.
- An additional set, those available in Privileged mode, are *controlled* through mandatory certification.
- The remainder may be *open*, *discretionary* or *controlled* depending on the policy chosen for a device.

15

Writing Portable Code and Maintaining Ports

'Legacy code' often differs from its suggested alternative by actually working and scaling.

Bjarne Stroustrup

As I explained in Chapter 1, porting is an attractive option because it can eliminate the need to duplicate code. But with so many platforms, architectures and tools in the market, developers can find it difficult to port from one operating system to another.

The advantages of designing and writing portable code include:

- operating system flexibility
- significantly improved time to market for new platforms
- robust code: porting and testing code on different platforms helps in finding bugs.

Before thinking about how to design your portable code, you have to deal with something even more fundamental: getting the files on to your development host platform. If you target the Symbian platform, Windows Mobile and Linux, you'll probably end up editing your files on different platforms because you'll typically develop for the Symbian platform and Windows Mobile on a Windows machine and for Linux on a Linux machine. Because there is not yet a single-host, multiple-target, integrated development environment to deal with all these platforms, this chapter also covers how to manage your files while using different integrated development environments for writing cross-platform software.

15.1 Recognizing Portable Code

Before you write or change a single line of code, you should understand exactly what it is you're trying to do. Porting software is different from writing new software and you should take a different approach. Because there is no such thing as 100% portable code, you should be prepared to do some work and rewrite some parts of your code. Even when you design and write new software targeting multiple platforms, you should be prepared to completely rewrite parts of your application for different platforms. Your software should be as portable as is practical, but no more; therefore it is vital to establish a realistic portable baseline – part of the application that can be ported. You should rewrite the rest of your application specifically for each new platform.

The majority of the problems encountered when writing portable code are caused by assumptions made for one platform that become invalid once the code is moved to another platform. Assumptions about the following aspects of an application are common:

- performance
- resources
- features
- implementation.

15.1.1 Has Your Application Been Ported to Another Platform?

Many programmers claim their code is portable and should just compile and run on a new platform. Unfortunately, this is not true in the vast majority of cases, simply because there are too many things to keep track of mentally when dealing with portability issues. No matter how many assurances you are given that a piece of code is portable, always assume it is not. Until the code has been moved and tested on a new platform, you should treat it as non-portable code or 'portable friendly', which is a more reasonable description for software designed and written with porting in mind but which has not been ported to a new platform.

15.1.2 Are the Libraries Supported?

Today's software development is less about writing new code than it is about gluing together chunks of existing code. Check all the library dependencies in your project. If it depends on a group of proprietary libraries or APIs that are not available on other platforms, porting to a new platform will be difficult. Investigate whether your application can use alternative libraries or APIs. It is always a good idea to investigate

whether you can use open source libraries and APIs, because open source libraries allow for more support options. If practical, you can support it yourself and port the open source library to the new platform because you have access to the source code. Another advantage of open source libraries is that it is easier to evaluate an open source library than a proprietary one because open source libraries are usually freely available to download.

15.1.3 How Many New Compiler or Language Features Are Used?

Developers enjoy being clever with their code and if they can choose between writing code in a clear, concise manner or a cooler but more obtuse way, too often they choose the second way just to demonstrate their programming skills. There are downsides to this approach. One obvious disadvantage is in code maintenance. Another one is that these clever tricks often cause headaches when it comes to portability because they often rely on compiler-specific features or aspects of the programming language that may not be widely implemented. Every time you have to port an application, check how clearly and concisely it is written, and identify the use of new or experimental libraries and language features.

15.1.4 Do I Have to Deal with Varying Feature Availability?

Every time we try to port a piece of software to another platform, we face the question of how to deal with features available on one platform that are absent from another. Usually, a cross-platform application needs to address this dilemma. If you are in a situation where you have missing features, you might decide to implement or emulate a missing feature for your target platform. This approach can degrade performance, however. You may also find that you have agreed to tackle the implementation of a potentially huge piece of software (it is quite likely that a feature is missing for a very good reason!).

15.2 Design Strategies and Patterns

It is always a good idea to design and write portable code, but sometimes porting can't be done between different platforms or it is better to rewrite code than to port it. You should consider the performance requirements, size and purpose of the project, the number and specific type of platforms on which you intend to run the code, along with existing features on those platforms when designing and writing portable code.

Design work is always a complex process, and designing portable code is no different. Any project can raise many issues, including tools,

language compatibility and differences in underlying architectures. For example, many of the function calls made in the Symbian platform do not have a direct equivalent in Windows Mobile or Linux; such differences can mean that porting native Symbian C++ to Windows Mobile or Linux can be problematic, complex and time-consuming.

On the other hand, there are many similarities between platforms that you can exploit when designing portable code. For example, the Symbian platform, Windows Mobile and Linux all support POSIX. If you design your application to use POSIX as much as possible, you can reduce time and effort when you come to port between these platforms.

Writing code that ports easily to new platforms can add significant costs at the design stage, as well as long-term maintenance overheads. You need to decide whether it is worth the extra investment. For a large or complex project, the additional costs might be justifiable. If the project is fairly simple, however, it may be better simply to rewrite the code for new platforms. Whether or not you decide to write portable code, keep in mind that good software structure is essential for system extension and maintenance.

15.2.1 Writing Modular Code is a Good Way to Write Portable Code

When starting a new application or re-factoring an existing one, always try to create a modular architecture. It helps to have portable parts in modules that are separate from the parts that need to be re-written. For applications with a graphical interface, you should have a neat separation between the UI and the engine, as illustrated in Figure 15.1.

There are other factors to consider when designing modules. For example, software that is highly portable may not offer the best performance on every platform. Portable code is always a matter of proper and powerful abstraction, but just getting your code to compile on a range of platforms is only half of the battle. Sometimes, you also need to get your code to run fast on all targeted platforms. In this case, it is a good idea to separate speed-critical code into separate modules. You can then optimize this code on every platform.

Another advantage of writing modular code is that you can recycle it to other projects. A final point about modular code: aim to minimize connections to other modules, making the coupling between modules weak. By doing this, you'll be able to use different modules together in different configurations for your future projects.

15.2.2 Using Indirection Layers

Hiding low-level processing behind a common set of functions is key to designing and writing portable code. The goal is to avoid putting platform-specific code in the middle of your application. Any low-level

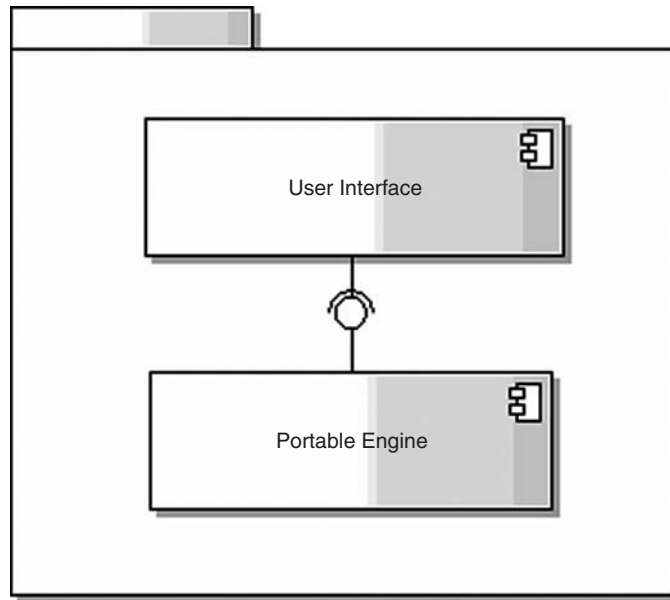


Figure 15.1 Separate the user interface from the engine

code is usually platform specific and should be hidden behind function calls.

But how low-level is too low-level? How you answer this question can have a significant impact on an application's performance because it can affect the specific implementation of a function. And how many restrictions should you impose on your code? For example, imagine you have to choose between the `DrawTriangle()` or the `DrawShape()` function in a graphic library. The first might enable you to overlay more common code, but the second might permit you to optimize platform-specific code. You have to decide between having very optimized code or having more common code across platforms. Experience with many platforms and environments can help you make better choices in this area. Even without this experience, try to think ahead about what the critical factors are for your layers.

If you have an application with a graphical interface, graphics code is the best place to start designing and implementing indirection layers. This is because different UIs can use the same engines and underlying indirection layers. You can write different UIs for the same platform and use the same engines and indirection layers, or you can write UIs that look the same on different platforms and use the same engines and indirection layers, as illustrated in Figure 15.2. Bear in mind that graphics and UI code are probably the pieces of code that you'll have to rewrite when porting from one platform to another.

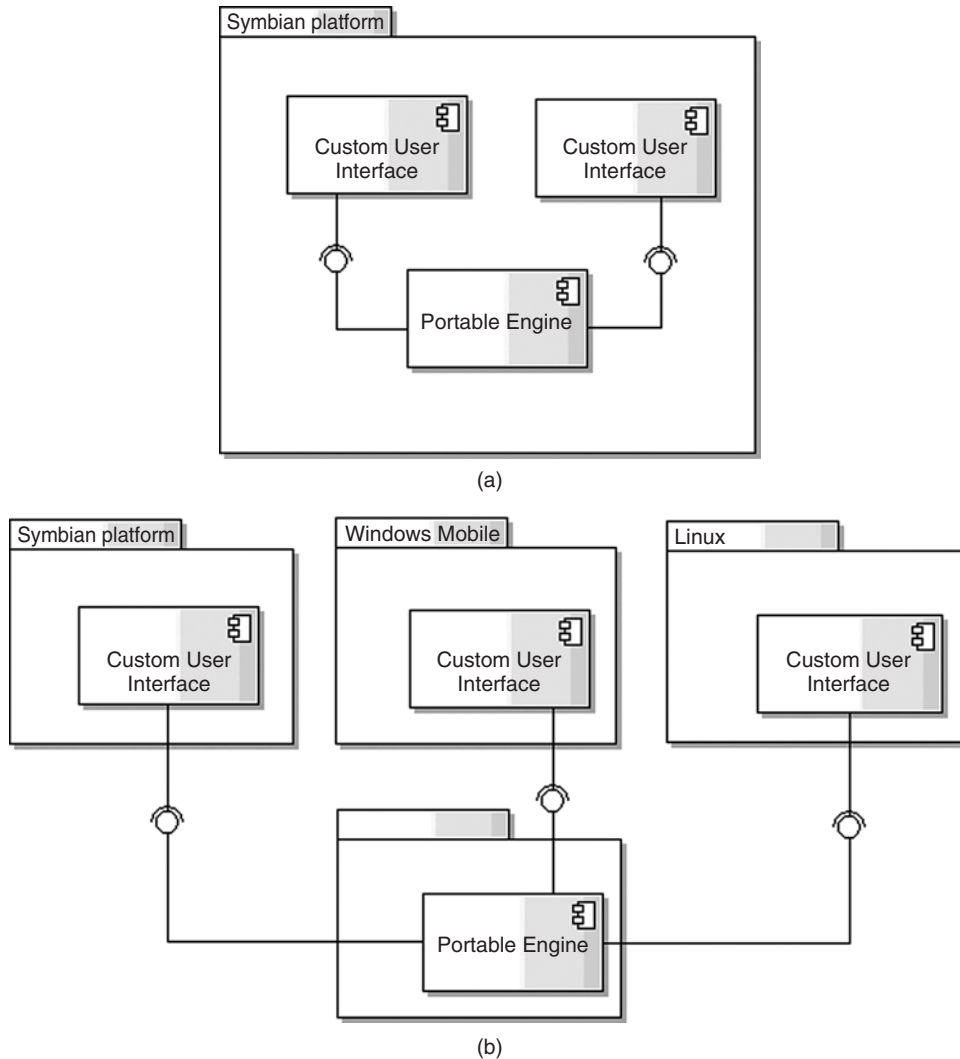


Figure 15.2 You can have a) different UIs on the same platform using the same engine or b) the same UI on different platforms using the same engine

15.2.3 Designing APIs

When you design indirection layers, you should decide on the set of routines, data structures, object classes and protocols they will expose. This brings us to another aspect of portability. Carefully designed APIs have a big impact not only on how easy it is to port your application to a new platform, but also on how easy it is to use different layers on the same platform. Good APIs are easy to learn and easy to use, even without

documentation. Also, good APIs are easy to extend. Don't try to put too many things into your API definition. Keep the interface simple and neat.

The first thing to consider, when designing an API, is namespace clashes. Some platforms might contain functions that have the same names as yours and this could force you into a painful round of find and replace. A simple way to isolate your functions is to prefix all your method names or class names with a given word. You should apply a similar scheme to name your exposed data types and structures as well, not just your API methods. Another technique is to use namespaces because they provide a way to avoid name collisions without some of the restrictions imposed by the use of classes and without the inconvenience of having nested classes.

You should hide system dependencies behind interfaces; this is the point at which you should focus your efforts on clean interfaces that provide enough information for future implementations. You should provide enough functionality and flexibility so that implementers on different platforms won't feel the need to go behind the scenes of your interface. Take, for example, a string-handling library. You would want to define a string type and a char type, along with a set of functions to handle strings, create them, copy them and calculate their length. Functionality has to be provided without any knowledge of how strings are stored.

But what if strings can't be stored in the same way on all your targeted platforms? When you come to port the code, you conclude that you have to rewrite your library for a new platform, which may be a sensible thing to do if your library is small enough. Another solution is to provide mapping functions to allow your library to map between strings and types on corresponding host platforms, allowing you to keep your library unchanged between platforms and port only the mapping functions. Doing this for all the different components can be a very complex task and, again, experience makes this a lot easier.

When designing an API, try to be consistent in the way you name the API functions and in the type and order of the parameters they take. Also, be consistent across your components, even if they use different technologies. For example, imagine you design a library to handle asynchronous operations and you split your library into multiple modules. Thinking that Symbian is the first platform on which your library will be implemented, you see that some of the modules can use active objects while others can use a callback notification mechanism. Try to think ahead and be consistent with the way you design your API interfaces. Active objects are Symbian-specific, so you might want to hide them, and design all your interfaces based on callback notifications. Try to be consistent with the words you use. The same word should mean same thing throughout the API.

If you provide system-specific functions, make it easy to guess their names. Make sure you mark them clearly as system specific so that

nobody wonders why they aren't provided on another platform. Also, keep in mind that some of the parameters can be ignored in some implementations. They should be clearly marked as such.

Make sure that no implementation of your API function has hidden side-effects upon which some users or platforms might rely for the high-level code to work properly. Each API method should do exactly what its name suggests. If, for some reason, you need to have a hidden side-effect, you should document it extensively to make porting to another platform as easy as possible. For example, if there are situations in which your method deletes a reference parameter that is passed in, you should document those situations and use cases. Make sure all state information and similar data are passed in as parameters or are accessed through pointers or references passed in as parameters (an object's `this` pointer is also a parameter, so object members' data also count as data accessed through parameters). Given the same input, the functions should produce the same output and should not depend on any hidden state information, for example, global or thread-local data.

15.2.4 General Design Guidelines

Here are some design practices and patterns to help you design a good API for your layers:

- Do not expose more than you want: A minimal API is one that has as few classes as possible and as few public members per class as possible. This makes the API easier to understand and maintain. An API should be as small as possible.
- APIs should be complete and provide all expected functionality.
- APIs should be intuitive. It's never a good thing to have to look up the documentation every two seconds to use an API.
- APIs should be easy to memorize: Choose consistent and precise naming conventions. Use recognizable patterns and concepts and avoid abbreviations. Be consistent with the meaning of words used. The same word should mean exactly the same thing across your layers.
- Do not abbreviate names: Even obvious abbreviations such as 'prev' for 'previous' don't pay off in the long run, because users must remember which words have been abbreviated.
- Choose appropriate names for classes: Identify a group of classes rather than trying to find a perfect name for each individual class. Make it clear from the name if there are classes that share common patterns of relationships, structural (such as associations) and non-structural (such as method invocation). For example, if you implement a UI

library, all your model-aware item view classes can be suffixed with View (CTreeView, CTableView).

- Choose appropriate names for functions and parameters: It should be clear from the name whether a function has side-effects or not. Parameter names are an important source of information to the programmer and it is worth spending some time giving good names to parameters.
- Choose appropriate names for Booleans, getters, setters and properties: Finding good names for getters and setters is always difficult. Be consistent on how you name them across your layers. If you can't see obvious names for Boolean types, check if defining them as enumerations can help.
- Implementation should not impact on an API: Implementation details confuse users and limit your freedom to change the implementation. Do not over-specify the behavior of your methods and classes. For example, if you identify some file-handling operations in your application, it doesn't mean you have to specify a new class for file handling. Specify new classes only if they are valuable enough. Also, when specifying new methods, remember not to specify implementation details. For example, if your method is implemented using vectors, you don't have to specify 'vector' in your function's name. Try to avoid leaky abstractions. A leaky abstraction is an unsatisfactory implementation of an abstraction, in which specific implementation details can be obstructive or counter-productive.
- Write documentation: An API is supposed to be read and understood by others. Try to be succinct, precise and cover every single function and class.
- Avoid long parameter lists: If you have long parameter lists, your API won't be usable without constant reference to its documentation because most programmers can't remember long parameter lists. You can avoid long parameter lists by using helper classes or structures to hold aggregates of parameters.
- Avoid return values that can cause exceptions: For example, return a zero-length array or an empty collection instead of NULL. You can also define objects that represent errors and return them instead of returning NULL, when returning empty objects is unsuitable for you. Instead of returning a value that can cause an exception, it is better to leave or throw an exception from your methods.
- Overload with care: Avoid ambiguous overloads. Just because you can, doesn't mean you should do it. Sometimes it's better to use a different name than overloading.

- A factory method is better than a constructor: It is more flexible to expose a factory method than to expose a constructor. Once a constructor is available as part of an API, it guarantees not only that an instance assignable to a given class will be created, but also that the instance will be of the exact class (no subclasses allowed) and that a new instance is created every time.

15.2.5 Adopting Patterns

Design patterns provide a way to share experiences so that a community can benefit and not continually reinvent the wheel. A design pattern typically captures a problem description, the context of the problem, a proposed solution to the problem and any anticipated consequences of using the solution. In order to have the broadest applicability (and therefore be useful to the largest audience), design patterns are often described in an abstract form and the implementation must be adapted to the specific situation. Design patterns are an amazing resource for developers.

The challenge comes in determining which design pattern to use and when. For example, you can choose the right design pattern, but there can be differences in how that design pattern is implemented on different platforms, increasing the time and effort necessary to maintain, debug and extend the implementation on various platforms. Keep in mind that design patterns are not a silver bullet.

The following sections give some examples of how design patterns can solve various problems in a porting context.

Adapter Pattern

The Adapter design pattern converts the interface of an existing class into an interface that a client understands. The Adapter pattern encourages the re-use of existing code by allowing classes to work together that normally could not because of incompatible interfaces. You should try to adopt the Adapter design pattern if your application depends on third-party toolkits or libraries. For example, you might want to use an alternative library but you discover that its interface is not compatible with what your application expects. If you don't have access to its source code, you should wrap its APIs. If you have access to its source code, you should decide whether it's suitable to change the library for each platform.

Façade Pattern

The Façade design pattern provides a unified and easy-to-use interface to a complex subsystem or set of interfaces. It provides a higher level

interface and decouples the client from the complex underlying systems. The Façade design pattern can be quite useful in two situations:

- Decoupling: programmers usually attempt to avoid excess coupling between modules and classes when designing applications. By using this pattern, you can create a small collection of classes that has a single class which is used to access the other classes. Also, you can use Façade when you interface with third-party components or libraries.
- Wrapping a collection of existing APIs: When porting code, you may find that some of the existing APIs are poorly designed or unsuitable. Because it is not always possible to change the initial code and integrate the changes, you can wrap those APIs.

Model–View–Controller Pattern

Model–View–Controller (MVC) is a classic design pattern often used by applications that need to be able to maintain multiple views of the same data. The MVC pattern separates objects into one of three categories:

- models – objects representing data or even activities
- views – some form of visualization of the state of the model
- controllers – facilities to change the state of the models.

The MVC pattern provides many advantages, including:

- Clarity of design: The public methods in the models stand as APIs for all commands available to manipulate its data and state. Just by looking at public APIs, it should be easy to understand how to control a model's behavior.
- Multiple views: The application can display the state of the model in a variety of ways.
- Ease of growth: Controllers and views can grow as models grow and, as long as you maintain a common interface, it is very easy to use older versions of views and controllers in your application.

I recommend Issott (2008)¹ for those who wish to pursue further reading on this subject. I also recommend Gamma *et al.* (1994)² which is a classic in the literature of object-oriented development.

¹ Issott, A. (2008) *Common Design Patterns for Symbian OS: The foundations of mobile software*. John Wiley & Sons.

² Gamma, E., Helm, R., Johnson, R. and Vlissides, J.M. (1994) *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley.

15.3 Strategies for Maximizing the Number of Portable Modules

15.3.1 Using User-Defined Data Types

Data types are at the lowest level of code portability. Not all data types are supported by all platforms and compilers, or are fully specified in the ISO 14882³ standard (also known as C++98). Don't think that `int`, `short` and `long` types will not cause problems just because they are standard in C++.

Check, for example, the `sizeof()` value returned from a `long` type in Symbian, Windows Mobile and Linux. The ANSI C++ standard specifies that a `long` is at least 32 bits but you can't rely on a `long` having a certain size. Imagine you have some code initializing a `long` variable to `0xffffffff` to set all its bits or set it to `-1`. If `long` is 32 bits, it works. If not, you have a problem that can sometimes be very hard to track down. Another problem can occur if you use these types in structure declarations, when you cannot expect structures to have a certain size. Both of these problems can be handled with user-defined data types, or simply by defining new symbols for each type the system needs. Data types can be abstracted through the use of `typedef`. You can roll the assumption that an integer is 32 bits into a type definition on platforms where this assumption is valid. For example:

```
#if __SYMBIAN32__
typedef TInt32 int32;
typedef TUint32 uint32;
#endif

#if _WIN32_WCE_
typedef int int32;
typedef unsigned int uint32;
#endif
```

Using your own data types, even if they default to the standard types, allows you to easily replace them with more suitable ones on another platform. Also, if a data type is not supported on a target platform, the normal operators and run-time functions probably do not support it either. In such cases, you should define a new function or macro to manipulate the data type.

Be very careful when working with floating-point types and floating-point operations because the ANSI C++ standard does not address the issues of floating-point operations definitively. Applications must not rely

³www.iso.org/iso/catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38110.

on the consistency and precision of floating-point operations, especially when comparisons with floating-point types are made. These may work on some systems but for other systems you'll end up with frustratingly elusive bugs. One way (though imperfect) to deal with floating-point types is to quantize any floating-point number to a limited fixed-point representation⁴ that should match on all targeted platforms.

Use compile-time asserts as much as possible to verify your assumptions. Define your own assert macro if necessary. For example, you can use some compile-time assertions to check you made the right guess about the data type's sizes:

```
COMPILE_TIME_ASSERT(int16, sizeof(int16) == 2);
COMPILE_TIME_ASSERT(uint16, sizeof(uint16) == 2);
COMPILE_TIME_ASSERT(int32, sizeof(int32) == 4);
COMPILE_TIME_ASSERT(uint32, sizeof(uint32) == 4);
```

COMPILE_TIME_ASSERT is a user-defined assertion and can be defined on every targeted platform using existing assert macros. For example, on the Symbian platform, you can define it as:

```
#define COMPILE_TIME_ASSERT (x,y) __ASSERT_COMPILE(y)
```

Configure the compiler to do as much argument checking as possible. This helps to reduce incompatibilities in the use of user-defined data types.

15.3.2 Considering Platform Independence

When developing portable systems, it is best to isolate all the platform-dependent code into separate modules that can be selected at compile time or at run time. These modules can then be rewritten for each target platform. This method is better than using conditional compilation for a given section of code because preprocessor directives affect code readability and complicate testing procedures.

Using separate compilation modules for isolating platform-dependent code explicitly identifies future porting work. When maintaining the code, you will make fewer errors if you have to work on the platform-dependent modules when you need to make changes, rather than working on a section of code in a platform-independent module that could have side effects on other platforms. Isolating platform-dependent code also helps to keep the platform-independent code portable when you decide

⁴ See Section 6.3.2. This may also improve performance significantly on most embedded platforms.

to target a new platform. Of course, there is an exception from this rule. If platform-dependent code is small (one or two functions), it may be better to use conditional compiles for given sections of code because this will make your build system slightly simpler.

Platform-dependent code includes any code that assumes the representation of data types, makes use of services provided by the operating system and so on. Generic interfaces or wrappers should be defined for platform-specific modules, so that calling code can remain portable. A good example of such a structure is the support of different communications paradigms. Communication-related code can be isolated in its own module and called using an interface so that it's identical, whether the code is implemented on the Symbian platform, Windows Mobile or Linux.

15.3.3 Supporting Multiple Libraries

As I said in Section 15.1.2, today's software development is more about putting together pieces of pre-existing code, than writing new code. This means library dependencies can have a great impact on the portability of your application. If you take time early in your project to support multiple alternative libraries that accomplish the same result, you'll have more options when porting to a new platform, even if the library vendor goes out of business or refuses to make his library available for the other platform. Investigate any open source alternatives you can support. Open source libraries allow for more support options. If practical and desired, you can self-support and port the open source library to the new platform, because you have access to the code. Another advantage of open source libraries is that they are easier to evaluate than proprietary ones because open source libraries are usually freely available to download. But keep in mind that open source libraries are not a silver bullet. Investigate how up to date are the libraries and whether they are actively maintained.

15.3.4 Building Your Own Bricks

Modern, general-purpose, embedded operating systems provide application and system programmers with a variety of programming interfaces. At the lowest level, every operating system provides system calls, system services or executive services – functions directly implemented in the operating system kernel and invoked via some kind of software interrupt or trap mechanism. System services can be wrapped in your chosen programming language and I show you how to wrap a `Mutex` class using C++. The `Mutex` class is straightforward and looks as follows:

```

class Mutex
{
public:
    Mutex();
    ~Mutex();
    void lock();
    void unlock();
private:
    Mutex(const Mutex&);
    Mutex& operator = (const Mutex&);
};

```

One thing you have to do is implement the `Mutex` class for all targeted platforms. But how should you organize these different implementations? First, you need a common header file that contains the definition of the `Mutex` class. Then, you need platform-specific header files along with their implementations. You want platform-specific `MutexImp` definitions to be used only through a common, generally defined `Mutex` class.

C++ has the solution to the problem: *private inheritance*. If class B is a private base class of class C, then only methods of C (and its friends) can access the public and protected members of B. Subclasses of C and other unrelated classes cannot access them. This is ideal for the intended purpose.

For every platform, a class `MutexImp` with the following interface is implemented:

```

class MutexImp
{
protected:
    MutexImp();
    ~MutexImp();
    void lockImp();
    void unlockImp();
};

```

All members of `MutexImp` are protected, so this class cannot be used on its own. It can, however, be used as a base class for `Mutex`. Private inheritance from `MutexImp` ensures that nothing but `Mutex` gets access to the members of `MutexImp`. The definition and implementation of `Mutex` therefore looks as follows:

```

class Mutex: private MutexImp
{
public:
    Mutex();
    ~Mutex();
};

```

```

void lock();
void unlock();
};

void Mutex::lock()
{
    lockImp();
}

void Mutex::unlock()
{
    unlockImp();
}

```

You might implement `MutexImp` for the Symbian platform like this:

```

class MutexImp
{
protected:
    MutexImp();
    ~MutexImp();
    void lockImp();
    void unlockImp();
private:
    RMutex iMutex;
};

MutexImp::MutexImp()
{
    TInt error=iMutex.CreateGlobal(KLitGlobalMutexName);
    if ( error == KErrAlreadyExists )
    {
        iMutex.OpenGlobal(KLitGlobalMutexName);
    }
}

MutexImp::~MutexImp()
{
    iMutex.Close();
}

void MutexImp::lockImp()
{
    iMutex.Wait();
}

void MutexImp::unlockImp()
{
    iMutex.Signal();
}

```

The code above, however, is not exception safe and does not handle errors. So how do we handle errors and exceptions in a unified manner? First, let's have a look at a typical `Mutex` usage:


```

Mutex *mutex = GlobalMutexInstance();
mutex->lock();

// execute some code that needs to be in a critical section

mutex->unlock();

```

As you can see, I use `GlobalMutexInstance()`, a factory method that creates and retrieves a single instance of `Mutex` across my application.

The code has a problem: if an exception is thrown from within a critical section, the mutex remains locked forever, leading to a hanging application next time the critical section is executed. Because most embedded operating systems support `try-catch`, the correct way to implement a critical section is as follows:

```

Mutex *mutex = GlobalMutexInstance();
mutex->lock();

try
{
    // execute some code that needs to be in a critical section
}
catch(...)
{
    mutex->unlock();
    throw;
}

mutex->unlock();

```

But what if one of your target systems does not support `try-catch`? You have to design your application to be able to handle errors and exceptions. Basically, you have to find a way to notify when an error occurs in one of your methods and you have to find a way to avoid orphaning heap-based objects and therefore creating memory leaks. To solve the first problem, you may decide to make your methods return an error code that is handled in calling application code. Another approach may be to install and register callback methods to handle errors. To solve the second problem, you may want to design and implement a collection class that stores and handles all heap-based pointers during their lifetime. The Symbian `CleanupStack` class is a good example of such a class. Another technique you can use is `auto_ptr`. Using `auto_ptr` allows you to have objects residing on the stack that store a pointer to a heap-allocated object. Since the `auto_ptr` objects reside on the stack, they are automatically destroyed when the enclosing scope is

exited, including the case of exceptions; their destructor is called before the exception propagates.

15.4 Configuration Management

Before thinking about portability, you have to deal with something even more fundamental: getting files onto your host platform. If you target the Symbian platform, Windows Mobile and Linux, you may even end up editing your files on different platforms. Text files created on different operating systems have to deal with multiple types of line endings. This becomes a problem when developing source code across a wide variety of host operating systems. You can overcome this problem by using the same set of tools across host platforms. For example, for editing you can use Emacs⁵ on both Linux and Windows.

15.4.1 Revision-Control Systems

Revision-control systems are designed to help with problems arising from editing files on multiple platforms, by converting your files to a canonical line-ending format. Revision-control systems also help you to track all the changes you make in your files. When choosing a revision-control system, it should be cross-platform and it should be able to handle the task of moving files on to a new platform as opposed to copying the files manually. Thankfully, you have some choices:

- Concurrent Version System

The Concurrent Version System (CVS) was originally a front end to an earlier system called RCS, which allowed simultaneous file editing. Despite its difficulties, CVS is the pre-eminent version-control system for cross-platform and open source developers today, mostly because of its availability on a huge number of platforms and its open source nature.

- Subversion⁶

Subversion is ‘the new CVS’. It is superior to CVS in nearly every way, addressing most of the common complaints about CVS. Subversion uses a database backend (Berkeley DB) instead of the file-oriented backend of CVS.

- Bazaar⁷

Bazaar is a distributed version-control system, sponsored by Canonical Ltd, which also manages the Ubuntu Linux distribution, available

⁵ See www.gnu.org/software/emacs.

⁶ See subversion.tigris.org.

⁷ See bazaar-vcs.org.

under GPL v2 or later. Bazaar is designed as a Python API with a plug-in system. Therefore it is easy to integrate it into an existing infrastructure. Bazaar can be configured to work as a distributed system that doesn't use a central server, to work with a central server or as a mixture of the two. It is a very reliable, portable and intuitive tool, although not as mature as Git or Mercurial.

- Git⁸

Git is a distributed version-control system that was developed for managing the Linux kernel source tree. It is very fast and has strong support for non-linear development, with support for rapid and convenient branching and merging. However, it is harder to learn than other version-control systems because it has a very large command set (version 1.5.0 provides around 139 commands) and requires frequent maintenance. Without frequent maintenance, aborting operations or backing out changes leaves useless garbage in the system database.

- Mercurial⁹

Mercurial is a distributed version-control system developed and maintained primarily by Selenic. Compared with Git, Mercurial has a strong focus on simplicity. Mercurial is very lightweight, easy to learn and use, very scalable and easy to customize. Mercurial is the system used by the Symbian Foundation, which has documented the usage on [*developer.symbian.org/wiki*](http://developer.symbian.org/wiki).

No matter which source-control system you choose, organizing source control can be a complex and challenging task. However you do it, make sure that everyone involved in the project, especially every developer checking in code, has a clear understanding of where everything goes. Develop guidelines for the project hierarchy within the source-control system. Make sure that all of the developers understand the organization and the fundamental reasons behind its design.

15.4.2 Common Problems

There are still some very common problems that a source-control system won't solve. Knowing about them will help you to keep your projects under control.

Badly Written Code

A source-control system cannot prevent developers from writing bad code. Every developer, including me, has written bad code at least once. The code seems to work on the surface but there are lurking problems.

⁸ See [*git-scm.com*](http://git-scm.com).

⁹ See [*www.selenic.com/mercurial/wiki*](http://www.selenic.com/mercurial/wiki).

Code reviews are designed to catch these things but how do you do a code review on code that changed even though no developer was assigned to it? This isn't a source-control problem, but you can use source control to help you solve it. You can use a source-control system to create a check-in differences report each day. You can review all of the check-ins from previous days and reverse incorrect or bad changes. However, if you are working on a big project and you have too many check-ins per day it is not feasible to review all the changes. You may not have enough time or you may not be the right person to review all the changes. You can solve this by creating branches for your work in progress. Developers should submit changes to their own working branches (or project branches) only. Integration of these branches with the main development or production branch should be kept under strict control, via a review process, an automated regression test suite or, preferably, both.

Developer Communication

When faced with conflicts within a single file, most developers manage to resolve them without too much effort, but if we look at a more general definition of 'conflict', we see that it can include problems too difficult to be solved without communication between developers. While a source-control system can help you to solve conflicts that are purely textual, it is useless when it comes to logical conflicts, for example, where one developer changes an API definition in file A and another developer changes some code that uses the old API in file B. Source-control systems can't help with this type of conflict. Make sure you develop the habits of reading specifications and talking to your peers. Also, integrate your changes as often as is sensible. If you integrate too often, you may end up submitting code that does not work properly and blocks your peers. If you integrate too infrequently, you may end up with a lot of conflicts that can be difficult to manage.

Configuration File Management

Configuration files are always a problem in source-control systems. Difficulties arise from the fact that configurations are stored for different platforms and sometimes for different phone models and, presumably, all developers work with them. When working with configuration files, you should always look for opportunities to consolidate configuration entries. In a multi-developer environment, it's easy to get two or more configuration values that really mean the same thing but apply to two different parts of the application. Also try to separate very specific platform

configurations. Consider creating separate configuration files for each configuration.

Source Integration

Once you have your source-control system organized, you should think about automated builds and continuous integration. Continuous integration is a software development practice whereby members of a team integrate their work frequently. Each integration is built by an automated system and tested. This approach helps teams to detect and fix integration errors as soon as possible. Automated build tools and continuous integration tools, including CruiseControl¹⁰, NAnt¹¹ and MSBuild¹², are designed to make the process of building software completely automatic. You don't have to have these in your toolbox for managing your build process because you can manage it using scripts. Either way, you should aim to have a simple, repeatable process that anyone can run. Remember that the build process should be extensively documented.

15.5 Summary

Designing and building portable software is not a trivial task. It requires experience, as well as careful planning and extra thought. However, software designed and implemented with portability in mind tends to have a cleaner design and improved maintainability.

Every time you design and build portable software, you should keep in mind at least the following points:

- Design and write modular code.
- Minimize connections between modules and make the coupling between modules weak.
- Design and write indirection layers and hide low-level processing behind a common set of functions.
- Design simple and neat APIs.
- Use design patterns as much as possible but always keep in mind they are not a silver bullet.

¹⁰ See cruisecontrol.sourceforge.net.

¹¹ See nant.sourceforge.net.

¹² See msdn.microsoft.com/en-gb/netframework/default.aspx.

- Be realistic when establishing a portable base line. You can maximize your portable modules by using simple techniques, such as defining your own data types, supporting multiple libraries or wrapping system services in your own classes and methods.
- Make sure you know all the dependencies your application has. Know exactly what libraries you use and on what platforms they are supported. Try to support multiple libraries and always investigate if open source libraries are suitable for you.

Appendix A

Techniques for Out-of-Memory Testing¹

This appendix describes some techniques for testing and fixing out-of-memory (OOM) robustness issues and memory leaks. OOM robustness means that when some memory allocation operation fails, the error is handled gracefully. The program should not crash. It should not produce incorrect results. It should not leak memory.

As a case example, I will be using Redland,² a set of open source libraries for processing resource description framework (RDF) data. RDF is a set of related World Wide Web Consortium (W3C) specifications.³ They were originally designed as a metadata model. Currently RDF is used for modeling all kinds of data, using various syntax notations. Usually the data is about web resources and RDF is the core of what is often called the ‘semantic web’.

In a project I was working on, I wanted to convert existing legacy data sources to RDF and run SPARQL queries on them. SPARQL is an SQL-like standardized query language for RDF data. Instead of writing everything from scratch, I decided to port existing open source components to Symbian OS and ended up porting Redland. It provided me with RDF parsers, serializers, storage and a query engine that supports SPARQL. Redland is written in C without any complex dependencies. Porting the core functionality was relatively straightforward using the P.I.P.S. libraries for Symbian OS. Redland’s Apache 2.0 and LGPL 2.1 dual license was also generous enough for my needs.

¹ This appendix is edited and extended from blogs.forum.nokia.com/blog/lauri-aaltos-forum-nokia-blog/2008/11/12/fixing-out-of-memory-issues-in-redland-rdf-libraries.

² www.librdf.org.

³ www.w3.org/RDF.

A.1 Why Test for Out-of-Memory Errors?

The importance of detecting and fixing errors caused by out-of-memory situations is heightened on the Symbian platform. This is because of some of its architectural design choices and its use in resource-constrained devices. Emphasizing graceful handling of low-memory situations is also reflected in the UNI-03 Symbian Signed test criterion.

Redland, like many open source projects, has been developed on Unix-like operating systems where memory is plentiful and supplemented by virtual memory. Programs are combined in shell scripting style where individual programs run only for a short while and resources of the process are then automatically freed by the operating system.

In contrast to desktop systems, Symbian devices have only a little RAM and no virtual memory. Therefore OOM errors are more likely to happen. Additionally in Symbian OS, the scripting architectural style is rarely used and program lifecycles are different from in Unix. For example, in my project, I use Redland in long-running background server processes. Leaking memory in such a process is a sure way to memory allocation failures and all kinds of errors.

Many native Symbian OS programs behave well in OOM situations. OOM errors are usually manifested by `KErrNoMemory` leaves (exceptions) that are handled in an appropriate trap harness. Resource cleanup is handled by `CleanupStack`. Symbian OS C++ programmers are familiar working with these concepts.

Standard C does not have exceptions. OOM checking and resource cleanup are all up to the programmer. When I started working on Redland, I sampled the code to see how it felt. I noticed it had OOM checks in some places but not consistently. Often the return value from `malloc()` and other allocation functions was not checked at all. The basic handling principle for fatal errors was simply to give up and `abort()` the program. Not very graceful! Redland obviously needed changes in it so that it could be used in a program targeting large-scale deployment. To support and focus the fixing work, additional tests were required – the test suite supplied with Redland only tested functionality in close-to-ideal scenarios and without checking for resource leaks.

A.2 Out-of-Memory Loop

There is a well-established testing technique on Symbian OS called the OOM loop. Its basic idea is a kind of fault injection. Allocation

failure injection is activated using the `__UHEAP_SETFAIL()` heap failure macro. The program being tested is then expected to either function properly or fail gracefully.⁴

I started by implementing some integration test cases that exercised the Redland libraries in a similar fashion. I was planning to use them in the actual program. I attempted to run the test functions in an OOM loop with lots of iterations, for example, using the `EDeterministic` heap failure mode to fail every k th allocation for $k = 1..2000$:

```
for ( TInt k = 1; k <= 2000; ++k )
{
    // Set memory allocation to fail after k allocs
    __UHEAP_SETFAIL( RHeap::EDeterministic, k );

    // Heap marker for detecting memory leaks
    __UHEAP_MARK;

    // Execute the test function
    TRAPD( err, DoTestL() );

    // Check for memory leaks
    __UHEAP_MARKEND;

    if ( !err )
    {
        // The test function did not leave
        // If it ran successfully,
        // we have found an upper bound for k in for this test function
        break;
    }
}
```

I didn't like the way the `__UHEAP_MARKEND` macro worked: if it detected a memory leak, it would simply kill the program with an `ALLOC` panic. When hunting down leaks, I wanted to have the option to experiment with the code dynamically while it was still running and not just to do some post-mortem analysis with tools, such as `HookLogger`, which, while useful, did not always help me enough to figure out what was going on in a complex, dynamic system. Therefore I changed the leak detection mechanism to use functions available in `RHeap` to query for allocated cell counts, before and after the test function, and the address of the first leaked cell, if any:

⁴ John Pagonis writes extensively about the OOM loop construct in his Symbian Developer Network technical paper at developer.symbian.com/wiki/pages/viewpage.action?pageId=432.

```

TInt allocsize;
TInt alloccount;
TInt alloccount2;
TUInt32 leaked;
RHeap &heap = User::Heap();

for ( TInt k = 1; k <= 2000; ++k )
{
    // Inject failures every k alloc
    __UHEAP_SETFAIL( RHeap::EDeterministic, k );

    // Store the current alloc count of user heap
    alloccount = heap.AllocSize( allocsize );

    // Set heap marker for detecting memory leaks
    heap.__DbgMarkStart();

    // Execute the test function
    TRAPD( err, DoTestL() );

    // Get current alloc count of user heap
    alloccount2 = heap.AllocSize( allocsize );

    // Expect 0 leaked cells
    leaked = heap.__DbgMarkEnd( 0 );

    // Reset heap alloc failures
    // Total reset sets the nesting level of all allocated cells to zero
    // so that previously leaked cells are no longer checked
    __UHEAP_TOTAL_RESET;

    // Issue a breakpoint if detected a memory leak
    if ( leaked )
    {
        RDebug::Printf( "leaked %d cells, first one at %p",
            alloccount2 - alloccount,
            reinterpret_cast< void * >( leaked ) );
        __BREAKPOINT();
    }

    if ( !err )
    {
        // The test function did not leave
        // If it ran successfully,
        // we have found an upper bound for k in for this test function
        break;
    }
}

```

The `abort()` issue mentioned in Section A.1 became a problem immediately: the test program terminated when I wanted to keep it running. To solve this problem, I created a porting layer DLL to replace the standard C library implementation of `abort()` with my own version that throws an exception:

```
extern "C" {  
  
EXPORT_C void abort()  
{  
    User::Leave( KErrAbort );  
}  
  
} // extern "C"
```

I had to link against this DLL before the standard C library so the linker would choose my version and not the standard version. Similar function replacement would have been possible without creating a new DLL, for example by using the C preprocessor string substitution functionality or compiling the replacement code directly into the project. A DLL was required later when I was adding failure detection and debugging support features directly to the allocators (this is explained in Section A.3). I decided to keep all standard library replacement functions in the same place.

Replacing program termination with throwing an exception allowed caller C++ code to catch the exception and deal with it properly. It also introduced a number of memory leaks: `abort()` normally just terminates the process. The C library implementation and the operating system may free some resources of the process (such as closing open files or freeing allocated memory) but it is not strictly mandatory. In any case, destructors and functions registered with `atexit()` are not called. Becoming more graceful is not easy and straightforward.

Now I was able to enter the following test-driven, bug-fixing loop:

1. Write new test code or extend old tests. Run all tests. Repeat until some of the tests fail.
2. Fix any problems discovered. Repeat until all tests pass again.
3. Go back to Step 1.

This way I discovered and fixed literally hundreds of bugs in the libraries. Most of them were relatively simple failures to check the return code of some potentially failing function, simple memory leaks and so on. Some bugs were a little more complicated, for example, requiring design-level clarifications to rules for passing object ownership.

A.3 Improved Heap Failure Tool

The OOM loop approach described in Section A.2 also had its issues:

1. It was hard to determine where to set heap failure limits i.e. the maximum value of k .

2. Not all allocation failures resulted in observable bugs but they still made the system under test run in a slightly inconsistent state.
3. Some complicated bugs were hard to debug because the allocation failure and the observed error were highly decoupled i.e. very far from each other.
4. Some integration test cases would detect errors in dependent libraries (e.g. the SQLite database or libxml2 parser). I was not interested in fixing them for the time being.

To counter these issues, I decided to not use the Symbian OS heap failure tool but to write my own. One way to implement a heap failure tool could be to write a subclass of `RAllocator` and pass it as a parameter to `RThread::Create()`. However, this wouldn't address issue 4 as all the libraries used in the same thread share the same allocator. Something else was needed.

Fortunately for me, the Redland libraries use only a small set of memory management functions: `malloc()`, `calloc()`, `realloc()` and `free()`. No other functions that allocate memory were used, for example `strdup()`. This made it easy to implement my own versions of these functions in the porting layer DLL that already contained the `abort()` replacement:

```
extern "C" {

EXPORT_C void *malloc( unsigned int size )
{
    return AllocWrapper( User::Alloc( size ) );
}

EXPORT_C void *calloc( unsigned int n, unsigned int count )
{
    return AllocWrapper( User::AllocZ( n * count ) );
}

EXPORT_C void *realloc( void *p, unsigned int size )
{
    return AllocWrapper( User::ReAlloc( p, size ) );
}

EXPORT_C void free( void *p )
{
    User::Free( p );
}

} // extern "C"
```

The `AllocWrapper()` function is the workhorse of my heap failure tool; it takes in a pointer to a block of allocated memory and may inject

a failure by freeing the block and returning a NULL pointer instead. I describe the function in more detail later.

The heap failure tool needs some state information so it knows when to inject a failure and when not to. I decided to store this state information in the DLL thread-local storage (TLS). I added an OOM counter that would keep track of all allocation failures, injected and real. I also added some DLL API functions to set the heap failure parameters and to query and reset the OOM counter. For debugging support, I wanted to start single-stepping in a debugger starting from the point of failure injection, so I added a state variable for that too:

```
struct TAllocState
{
    // OOM counter
    TInt iOomCount;

    // Heap failure mode
    RAllocator::TAllocFail iFailureMode;

    // Fail every iFailCount in deterministic failure mode
    // In random failure mode, fail approximately once every iFailCount
    // allocation
    TInt iFailCount;

    // Alloc counter
    TInt iAllocCount;

    // State information for pseudorandom generator
    TInt64 iRandomSeed;

    // Issue a debugger breakpoint on failed allocation
    TBool iBreakOnFailure;
};

// Access alloc state in DLL thread-local storage
inline TAllocState *AllocState()
{
    return reinterpret_cast< TAllocState * >( Dll::Tls() );
}

// Initialize the failure tool
// Must be called before calling any of the other functions
EXPORT_C TInt FailureToolInit()
{
    TAllocState *state;

    // Check for existing alloc state
    state = AllocState();
    if ( state )
    {
        return KErrAlreadyExists;
    }

    // Create new state
    state = reinterpret_cast< TAllocState * >( User::AllocZ(
        sizeof( TAllocState ) ) );
}
```

```

if ( !state )
{
    return KErrNoMemory;
}

// Set non-zero default values
state->iFailureMode = RAllocator::ENone;

// Store state in thread-local storage
Dll::SetTls( state );

return KErrNone;
}

// Clean up the failure tool
EXPORT_C void FailureToolFinish()
{
    TAllocState *state = AllocState();
    User::Free( state ); // ok to free NULL
    Dll::SetTls( 0 );
}

// Set failure mode
EXPORT_C void FailureToolSetAllocFail( RAllocator::TAllocFail aMode,
                                       TInt aCount )
{
    TAllocState *state = AllocState();
    state->iFailureMode = aMode;
    state->iFailCount = aCount;
    state->iAllocCount = 0;
    state->iRandomSeed = 0;
}

// Enable/disable breakpoints on alloc failures
EXPORT_C void FailureToolSetBreakOnFailure( TBool aBreakOnFailure )
{
    TAllocState *state = AllocState();
    state->iBreakOnFailure = aBreakOnFailure;
}

// Query and reset OOM counter
EXPORT_C TInt FailureToolOomOccured()
{
    TAllocState *state = AllocState();
    TInt count = state->iOomCount;
    state->iOomCount = 0;
    return count;
}

```

Now we can finally have a look at the `AllocWrapper()` failure tool implementation. I didn't feel the need to implement all heap failure modes supported by the native heap failure tool. I was happy with just the deterministic (`EDeterministic`) and pseudorandom (`ERandom`) modes.

```

// Test whether to fail an alloc in the current state
bool AllocShouldFail( TAllocState *aState )

```

```

{
    switch ( aState->iFailureMode )
    {
        // Fail pseudorandomly with 1/failcount probability
        case RAllocator::ERandom:
            return Math::Rand( aState->iRandomSeed ) % aState->iFailCount == 0;

        // Fail deterministically after failcount successful allocs
        case RAllocator::EDeterministic:
            return ++aState->iAllocCount % aState->iFailCount == 0;

        // Do not fail
        case RAllocator::ENone: // fall-through
        // Not supported modes, do not fail
        case RAllocator::ETrueRandom:
        case RAllocator::EFailNext:
        case RAllocator::EReset:
        default:
            return false;
    }
}

// Inject OOM failures
TAny *AllocWrapper( TAny *aMemory )
{
    TAllocState *state = AllocState();

    // Inject a failure?
    if ( AllocShouldFail( state ) )
    {
        User::Free( aMemory );
        aMemory = 0;
    }

    // OOM occurred, injected or real?
    if ( !aMemory )
    {
        // Increment OOM counter
        ++state->iOomCount;

        // Issue emulator breakpoint?
        if ( state->iBreakOnFailure )
        {
            __BREAKPOINT();
        }
    }

    return aMemory;
}

```

Finally I integrated this improved heap failure tool into the OOM loop from section A.2:

```

User::LeaveIfError( FailureToolInit() );

// ...

```

```

TInt allocsize;
TInt alloccount;
TInt alloccount2;
TUInt32 leaked;
TInt oom;
RHeap &heap = User::Heap();

// Debugging support
TBool die = EFalse; // set to true in debugger to terminate the OOM
// loop
TBool oombreak = EFalse; // set to true in debugger to enable
// breakpoint on OOM

for ( TInt k = 1; !die; ++k )
{
    FailureToolSetBreakOnFailure( oombreak );

    // Reset OOM counter
    FailureToolOomOccured();

    // Set memory allocation to fail after k allocs
    FailureToolSetAllocFail ( RHeap::EDeterministic, k );

    // Store the current alloc count of user heap
    alloccount = heap.AllocSize( allocsize );

    // Set heap marker for detecting memory leaks
    heap.__DbgMarkStart();

    // Execute the test function
    TRAPD( err, DoTestL() );

    // Get current alloc count of user heap
    alloccount2 = heap.AllocSize( allocsize );

    // Expect 0 leaked cells
    leaked = heap.__DbgMarkEnd( 0 );

    // Reset heap alloc failures
    // Total reset sets the nesting level of all allocated cells to zero
    // so that previously leaked cells are no longer checked
    __UHEAP_TOTAL_RESET;

    // Get OOM count + reset the counter
    oom = FailureToolOomOccured();

    // Issue a breakpoint if detected a memory leak
    if ( leaked )
    {
        RDebug::Printf( "[%d] leaked %d cells, first one at %p",
            k,
            alloccount2 - alloccount,
            reinterpret_cast< void * >( leaked ) );
        __BREAKPOINT();
    }

    if ( !err && !oom )
    {

```



```
// The test function did not leave and there was no OOM
// If it ran successfully,
// we have found an upper bound for k in for this test function
RDebug::Printf( "[%d] finished", k );
break;
}
}

// ...

FailureToolFinish();
```

This setup fully addresses the issues I listed at the beginning of this section:

1. A suitable upper bound for k was reached in deterministic failure mode when the test function ran without leaving and produced correct expected results, and no OOMs were registered.
2. I could query the porting layer state for OOM failure count using `FailureToolOomOccurred()` to see whether there had been any undetected OOM errors while running the test code.
3. On an injected allocation failure, I could set the heap failure tool to issue a debugger breakpoint. This way I quickly discovered the root causes for errors occurring much later in the test case.
4. Dependent libraries were not affected since they were not linked against the allocator functions in my heap failure tool.

The OOM counter was also useful in a non-testing setup. I could use it to invalidate the results of any operation to make sure the system was not running in an inconsistent state.

A.4 Summary

In this appendix, I have described some techniques for testing for out-of-memory issues. I started with the well established OOM loop technique with a heap failure tool and evolved them to a more sophisticated testing system that suited my needs when porting the Redland RDF libraries to Symbian OS.

The improved heap failure tool concept is not directly usable in every project – for example, the set of allocator functions that need to be instrumented in the failure tool may vary from project to project and adaptations are needed.

I exercised Redland with about 200 test functions which were run with failure count values ranging from 1 to 2000 or 10000, depending

on test function complexity. On average, the number of test case and failure injection pattern combinations was in the order of hundreds of thousands. With these tests I detected and fixed literally hundreds of OOM-related errors in Redland libraries. Of course, I have submitted all bug fixes back to the Redland open source project to benefit the whole community.

References

This list contains the books referred to in the text. An up-to-date list of the online references can be found on the book's wiki page at ***developer.symbian.org/wiki/index.php/Porting_to_the_Symbian_Platform***.

- Ahonen, T. (2008) *Mobile as 7th of the Mass Media*. Futuretext.
- Aubert, M. (2008) *Quick Recipes on Symbian OS*. John Wiley & Sons.
- Blanchette, J. and Summerfield, M. (2008) *C++ GUI Programming with Qt 4*, 2nd Edition. Prentice Hall.
- Feathers, M.C. (2004) *Working Effectively with Legacy Code*. Prentice Hall Pearson Education.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J.M. (1994) *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley.
- Greenfield, A. (2006) *Everyware: The dawning age of ubiquitous computing*. New Riders.
- Gutmann, P. (2002) 'PKI: It's Not Dead, Just Resting', available at ***www.cs.auckland.ac.nz/~pgut001/pubs/notdead.pdf***.
- Hayun, R. et al. (2009) *Java ME on Symbian OS: Inside the Smartphone Model*. John Wiley & Sons.
- Heath, C. (2006) *Symbian OS Platform Security*. John Wiley & Sons.
- Issott, A. (2008) *Common Design Patterns for Symbian OS: The foundations of mobile software*. John Wiley & Sons.
- Pulli, K. et al. (2008) *Mobile 3D Graphics with OpenGL ES and M3G*. Morgan Kaufmann.
- Rome, A., Wilcox, M., et al. (2008) *Multimedia on Symbian OS: Inside the Convergence Device*, John Wiley & Sons.
- Stichbury, J. (2004) *Symbian OS Explained*. John Wiley & Sons.
- Stichbury, J. et al. (2008) *Games on Symbian OS*. John Wiley & Sons.

Index

- abld build 36
- abld freeze 226
- abort() 74, 400, 402
- abstract base class 46
- active objects 31, 94–99, 381
- Active Template Library (ATL) 134
- Adapter pattern 384
- Alerts API 166
- AllFiles 130, 349, 355
- Alloc() 64
- ALLOC panic 45–46, 399
- AllocL() 64
- All-Tcb 117
- Android 11, 12, 17–18, 244–253, 369–371
- ANSI C++ standard 386
- APIs 68–71, 103–131
 - applications, portable 375–396
- ARM RealView Compilation Tools (RVCT) 34, 112
- ATL See Active Template Library
- auto_ptr 85, 143, 391–392
- AVRecorder 270
- Base Class Library (BCL) 237
- battery 10, 14, 43
- Bazaar 392–393
- BCL. See Base Class Library
- Binary Runtime Environment for Wireless (BREW) 253–254
- Blackberry 11, 17–18
- bld.inf 35, 36, 38, 115–116, 186, 211, 292, 293, 316
- bldmake 36, 211
- Bluetooth 127, 138
- Boost 107, 197
- BREW. See Binary Runtime Environment for Wireless
- C 21, 28, 40, 103–131, 214
- C++ 21, 28, 31, 50, 59, 61–66, 103–131, 144
 - abstract base class 46
 - with C 40
 - IDE 32
 - MFC 214
 - Mutex 388–391
- C classes 53, 79, 85, 87
- C Wrappers 339–340
- CActive 95, 97
- CActiveScheduler
 - Add() 95–96
 - Start() 141
- CAknAppUi 155
- calendar 186, 251
- callbacks 147, 324, 381
- camera 165, 185, 253
- Cancel() 98
- capabilities 228, 348–355
 - Android 369–371
 - trusted applications 362–363
 - untrusted applications 363–364
- CAPABILITY 37, 355
- capitalization guidelines 50–51
- Carbide.c++ 32, 38, 72, 147, 211, 214, 368–369
- CArray 58
- CArrayFixFlat 58

- CArrayFixSeg 58
- CArrayVarSeg 58
- CArrayXSeg 58
- catch 72
- CBase 53, 54, 82, 100–101, 230
- CBluetoothSocket 138
- CCalEntry 138
- CCirBuf 59
- CCoeControl 147, 215
- CConsoleBase 144–145
- CContactDatabase 138
- CContactItem 138
- CEikConsoleScreen 144–145
- certificates 233–234, 359–366
- character size 60
- CHearbeat 142
- circular buffers 59
- classes
 - API design 382–384
 - CActive 95
 - capitalization 50
 - descriptors 61–66
 - naming 51–55
- cleanup stack 78–86, 393, 400
 - Check() 81
 - PopAndDestroy (test) 80
 - PushL(TAny*) 82–83
 - PushL(TClean-upItem) 84
- CleanupClosePushL() 85
- Close() 54, 79, 84, 85
- CLR. *See* Common Language Runtime
- CLSI. *See* Local System Interface class
- CMdaAudioInputStream 154
- Cocoa 135, 261–266, 268
- code signing 227, 359–361
- COM. *See* Component Object Model
- Common Language Runtime (CLR) 236
- compilers 34, 39–40, 318–322
- Component Object Model (COM) 230–233
- Compressed Audio API 165
- COM:X 126
- Concurrent Version System (CVS) 392
- configuration management 392–395
- configure 35, 307
- const char* 257–258
- Container 197
- CountComponent-Controls() 149
- CPeriodic 142
- _cplusplus 40
- CPolicyServer 352
- Createpackage 286
- CreateWindowEx() 208
- CSendAPPUI 138, 139
- ctime 277
- CTimer 142
- CurrentTrust() 234
- CVS. *See* Concurrent Version System
- data caging 130, 357–359
- data types 41, 386–387
- databases 205, 222–223, 253
- DBMS 205, 358
- dependencies 30, 40, 381
- DEPLOYMENT 286
- descriptors 59–60, 68–71
 - classes 61–66
 - function parameters 66–68
 - P.I.P.S. 124
- destructors 85, 90, 98–100, 143
- digital signatures 359–361
- DirectX 136, 203
- DispatchMessage() 216
- DLL. *See* dynamic link library
- Document Object Model (DOM) 184
- DOM. *See* Document Object Model
- Draw() 150
- DrawNow() 153
- DRM 349, 355
- dynamic link library (DLL) 37, 40–41, 109, 352–353
 - COM 230
 - MFC 229
 - Microsoft Windows 209
 - RVCT 112
 - Windows 226
 - Windows Mobile 234
 - WSD 99–100, 111, 343–345
- E32Main() 109, 246, 248, 334
- E32USER-CBase 46 92
- E32USER-CBase 71 77
- E32USER-CBASE 90 92
- Eclipse Public License (EPL) 27
- ECOM. *See* Epoc Component Object Model
- ELeave 73, 75–76
- EPL. *See* Eclipse Public License
- Epoc Component Object Model (ECOM) 230–233
- EPOCALLOWDLLDATA 100
- EPOCHEAPSIZE 123
- epocheapsize 42
- epocstacksize 42, 92
- errno 128, 343
- error handling 71–93, 97–98, 128, 142–144
- eshell 119–120
- EventReady() 303
- exceptions 71–78, 383
 - C++ 74

- floating points 198
- trap handler 85
- EXE. *See* executables
- exec 124, 197, 265
- executables (EXE) 109, 118, 350–352
 - trusted applications 362
 - Windows 226
- exit() 335
- Exiv2 311, 316–318, 320–322
- Expat 311, 315–316, 318–319
- EXPORT_C 40–41, 111, 112, 119
- extensions 185–186, 281–286
- extern 111, 140, 298
- extern "C" 40
-
- fabs() 153
- Façade pattern 384–385
- FCL. *See* Framework Class Library
- FIFOs 124, 126
- File Table 341–342
- float 56
- floating point 56, 129, 198, 386–387
- Flush() 144
- fopen() 144
- FOPEN_MAX 121
- fork 23, 124, 197
- Framework Class Library (FCL) 237
- free() 123, 402
- FreeBSD 335
- functions 41, 381–382
 - C 51
 - capitalization 50
 - L 51
- fwrite() 144
-
- Garden 301
- GCC-E 34, 226, 297, 298
- Geospatial Data Abstraction Library (GDAL) 289, 300
-
- GetUserDefaultUI-Language API 229
- Git 393
- glib 106, 334
- glue code 109, 142, 334–335
- GNOME Mobile and Embedded 12
- GNU Public License (GPL) 27, 135, 312
- Google 11, 12, 17–18, 245. *See also* Android
- GPL. *See* GNU Public License
- GPS 6, 14, 309–331
 - Net60 Mobility Framework 236
 - selective availability 8
- graphical user interface (GUI) 37, 78, 379–380
- graphics 14, 162
- GROUP 126
- GSensor 282–284
- GSM 3, 9, 166, 193, 194, 252, 366
- GStreamer 137
- GTK+ 13, 135, 146–147, 203
- GUI. *See* graphical user interface
- Guitune application 145–157
-
- HandleCommandL() 155
- HandleCompletionL() 97
- HandleEventL() 97
- HandleResourceChange() 154
- HBufC 64, 67, 82
- HBufC* 213, 219
- heap 75–76, 122, 340–341, 399–407
- high-speed packet access (HSPA) 14
-
- HOME 337
- HSPA. *See* high-speed packet access
- HTC Dream 245
- HTTP 252, 263
- hybrid code 133–157
-
- IAPs. *See* Internet Access Points
- IDE. *See* integrated development environment
- IDispatch 221
- IDL. *See* interface definition language
- Images API 164
- ImageView 247
- IMEI 366
- IMPORT_C 40–41, 111, 112, 119
- indirection layers 378–380
- Input API 167
- input method editors 221
- int 57
- integrated development environment (IDE)
 - Build 39
 - C++ 32
 - SIS 39
 - Windows 210–211
- Intent 246
- IntentFilter 247
- interface definition language (IDL) 221, 230
- interleaving 129, 144
- Internet Access Points (IAPs) 127, 185
- inter-process
 - communication (IPC) 198–199, 223–226
- inter-thread communication (ITC) 223–226
- I/O APIs 251, 261–262
- ioctl() 127
- IOStreams 107, 197
- IPC. *See* inter-process communication

- IPC server 338–339
- iPhone 11, 14, 18–19, 254–270, 255–256
 - security 371–372
 - UI 266–269
 - UIKit 135
- ISO 14882 386
- ITC. *See* inter-thread communication
- Iterator 197

- JSON-C 311, 316, 319–320

- KERN-EXEC 46 92
- KErrAccessDenied 352, 354
- KErrNone 77, 97
- KErrNotSupported 111
- KErrXX error 73
- Keypad Capabilities API 167
- key_t 125
- KNullDesC 65

- _L 65–66
- _LEAVE_EQUALS_THROW_ 74
- leaves 45, 71–78
 - cleanup stack 84–86
 - PushL() 80
- LeaveScan 50, 72
- Length() 62–63, 69–70
- Li, Harry 35
- LIB. *See* static library
- libc 104, 115, 121, 124, 196, 333, 335
- libcrypt 196, 333
- libcrypto 196, 333
- libdl 129, 196, 333, 336
- libglib 196
- libm 115, 129, 196, 333, 336
- libpthread 105, 114, 196, 198, 333

- libssl 196, 334
- libtiff 297, 298
- LiMo Foundation 12–13, 17–18, 193–195
- linked lists 59
- Linux 11–13, 17, 32, 195–198, 204
 - database 205
 - security 204–205
 - UI 202–203
- Linux Phone Standards (LiPS) 12
- LiPS. *See* Linux Phone Standards
- literal descriptors 65–66
- LoadString() 229
- Local System Interface class (CLSI) 340
- LocalServices 129, 349, 364, 365
- Location 349, 354, 363, 365, 367
- Location API 192
- long 386
- long int 56

- M classes 54
- MACRO 286, 300
- Maemo 12, 13, 191–192, 371
- main() 109, 264, 334, 335
- MaiscBufferCopied() 149
- makekeys 368
- malloc() 122–123, 398
- Math class 55
- MaxLength() 62, 69–70
- MDI. *See* multiple document interface
- MEikCommandObserver 216
- Mem class 55
- memory management 60, 71–93
 - limitations and workarounds 122–124
- Mercurial 393
- message type modules (MTMs) 139
- messaging 139, 185, 252
- meta object compiler (MOC) 180, 183, 256
- MFC. *See* Microsoft Foundation Classes
- Microsoft Compact Framework 239
- Microsoft Foundation Classes (MFC) 134, 214, 216
- MID. *See* mobile information device
- middleware 289–307
- MiniWeb 311
- mkdir() 121
- mmap() 121, 317
- MMF 36–37
- MMP
 - bld.inf 316
 - GDAL 289, 290–300
 - GUI 37
 - libcrt0.lib 110
 - MACRO 300
 - SYSTEMINCLUDE 116
- MMP_RULES 286
- MOAP(S) 11, 33–34
- Mobile and Internet Linux Project (Moblin) 12, 192–193
- mobile information device (MID) 191
- Mobile Linux 11, 189–206, 190
- Mobile to Market 234
- Moblin. *See* Mobile and Internet Linux Project
- MOC. *See* meta object compiler
- Model–View–Controller (MVC) 268, 385
- modular code 378
- Mosaic 8
- Motorola 9, 10
- mousePressEvent() 268

- MTMs. *See* message type modules
- multimedia 136–138, 164–166, 204
- multiple document interface (MDI) 221
- `munmap()` 317
- Mutex 388–391
- MVC. *See* Model–View–Controller

- namespace clashes 381
- naming guidelines 51–52
- National Marine Electronics Association (NMEA) 314
- native application 133
- .NET 210, 239–240
- .NET Compact Framework 236
- Net60 Mobility Framework 236
- Netscape Communications Corporation 8
- networking 262–264
 - limitations and workarounds 126–127
 - Linux 204
 - mapping 252
- NetworkServices 129, 349, 350–353, 355, 364, 365, 367
- New File Service Client API 313
- `new(Leave)` 75
- `NewL()` 88–89
- NMEA. *See* National Marine Electronics Association
- Nokia 9–12, 21, 33–34
 - Open C 106–107
 - Sensor Plug-in 282
- NSArray 259
- NSInputStream 261, 262
- NSMutableString 257
- NSObject 257
- NSOperationQueue 265–266
- NSOutputStream 261, 262
- NSScanner 259
- NSSortDescriptor 261
- NSStream 261
- NSString 261
- NSURLCache 263
- NSURLConnection 262–263
- NSURLCredentialStorage 263
- NTT DoCoMo 11, 33–34

- Objective-C 135, 256
- OHA. *See* Open Handset Alliance
- `onCreate()` 247
- OOM. *See* out-of-memory
- `open()` 121, 126
- Open C 106–107, 120–131
 - `libpthread` 198
 - Linux 196–198
 - S60 106
- Open C++ 107–108
- Open C/C++ 21, 28
 - descriptors 60
 - interleaving 144
 - Linux 195–196
 - Microsoft Windows 207–209
 - plug-ins 161
 - RGA 160–161
 - STL 55
- Open Handset Alliance (OHA) 12, 244–253
- Open Signed 39, 366–368
- Open Signed Offline 367–368
- Open Signed Online 366–367
- OpenGL ES 136, 171–174, 236
- OpenKODE 169–176, 203
- OpenMAX 137, 175–176
- Openmoko 12, 13, 192, 371
- OpenGL 113–114
- OpenVG 174–175
- `operator[]` 58
- `operator new` 73, 75
- out-of-memory (OOM) 72, 123, 397–408

- `paintEvent()` 268
- Palm OS 11
- PAMP. *See* Personal Apache-MySQL-PHP
- `Panic()` 115
- panics 91–93. *See also specific panics*
- parameters 51, 381–382
- patterns 384–385
- payment systems 16
- PCM. *See* pulse code modulation
- `perm` 121, 126
- Personal Apache-MySQL-PHP (PAMP) 189–190, 205
- Phonon 137, 185
- PIM 138–139
 - mapping 251
 - Net60 Mobility Framework 236
- P.I.P.S. 21, 104–105, 333–346
 - heap 340–341
 - limitations and workarounds 120–131
 - S60 106
 - timers 127–128
 - UIQ 106
 - WSD 333
- PKI. *See* public key infrastructure
- Playback Rate 114
- platform independence 387–388
- `Pls()` 343–344
- plug-ins 33–34, 161

- `poll()` 141
- `Pop()` 81, 88–89
- `PopAndDestroy()` 81, 84
- `popen()` 124, 198
- portable code 375–396
- Portable Operating System Interface (POSIX) 21, 28, 103–131
- porting 2
 - analyzing code 28
 - applications 216–220, 271–288, 309–331
 - build files 115–118
 - build process 35–36
 - build system 34–35
 - compiling 39–41
 - complex applications 309–331
 - debugging 45–46
 - development environment 31–34
 - iPhone 255–256
 - Microsoft Windows 207–241
 - middleware 289–307
 - Mobile Linux 189–206
 - .NET 239–240
 - Objective-C 256
 - packing 38–39
 - process 23–47
 - projects 24–27
 - re-architecting 29–31
 - re-integrating 46–47
 - running and testing 44–45
 - simple applications 271–288
 - system requirements 31
 - UI 266–269
- porting layer 30–31
- `Position()` 150
- POSIX. *See* Portable Operating System Interface
- `posix_spawn()` 124
- `PowerMgmt` 366, 367
- `#pragma` 40
- prefixes 51–52
- private inheritance 391
- problems 41
- process identity 356–357
- processing power 42
- `PROT_EXEC` 121, 122
- `ProtServ` 366, 367
- P&S. *See* publish and subscribe
- `pthreads` 114, 123, 142, 144, 198, 345
- `Ptr()` 63, 149
- public key infrastructure (PKI) 360–361
- publish and subscribe (P&S) 248, 359
- Publisher ID 364–365
- pulse code modulation (PCM) 115, 146
- `PushL()` 80, 88–89
- `PWD` 335
- Pyramid 301
- Python 298–299, 393
- `QAbstractListModel` 268
- `QAbstractSocket` 264
- `QAuthenticator` 263
- `QBuffer` 262
- `QChar` 259
- `QCharRef` 259
- `QDataStream` 262
- `Q_DECLARE_PRIVATE` 283
- `QFile` 262
- `QIODevice` 262
- `QLineEdit` 328
- `QListView` 268
- `QMainWindow` 267
- `qmake` 292, 293, 305–306
- `QMenuBar` 280
- `QModelIndex` 268
- `QNetworkAccessManager` 262–263
- `QObject` 179, 180, 184, 257, 265
- `QObjectPrivate` 283
- `Q_OS_MAC` 280
- `Q_OS_SYMBIAN` 277, 304
- `Q_OS_UNIX` 277, 280
- `QPainter` 181
- `QReadWriteLock` 265
- `QScriptEngine` 183
- `QSoftMenuBar` 280
- `QSslSocket` 264
- `QString` 180, 257–259
- `QSvgWidget` 184
- `Qt` 13, 21, 135, 177–187, 271–286
 - Cocoa 256–266
 - descriptors 60
 - HTTP 263
 - I/O APIs 261–262
 - networking 262–264
 - S60 301–304
 - threads 264–266
 - views 267–269
 - widgets 267–269
- `QtCore` 178–181
- `QTcpSocket` 262, 264
- `QTextStream` 259
- `QThreadPool` 265
- `QThread::run()` 264
- `QThreadStorage` 265
- `QToolBar` 280
- `QTOPIA_PHONE` 278–279
- `QTreeView` 268
- `QtScript` 182–183
- `QtSql` 183
- `QtSvg` 183–184
- `QtWebKit` 184
- `QtXml` 184–185
- `QValidator` 328
- `QWidget` 181, 267–268
- `Q_WS_S60` 304
- R classes 53–54, 79, 85
- `RAllocator` 402
- RAM 42, 398
- Raptor 34
- `RArray` 57–58
- `R_AVKON_SOFTKEYS_EXIT` 155
- `RBase` 53
- `RBuf` 64–65, 67, 149

- RChunks 338–339
- RDA. *See* Remote Device Access
- RDbNamedDatabase 358
- RDF. *See* resource description framework
- ReadDeviceData 349, 366, 367
- ReadUserData 129, 349, 363, 367
- real-time graphics and audio (RGA) 60, 136, 137, 159–168
- Realview toolchain (RVCT) 226
- Receive() 201–202
- Redland 397–408
- RedrawReady() 303
- registry 221–222
- Release() 79, 84
- Remote Device Access (RDA) 246
- removable media data caging 358
- requests 96–97
- Research in Motion (RIM) Blackberry, 11, 17–18
- resource description framework (RDF) 397
- revision-control systems 392–393
- RFastLock 224
- RFCOMM 127
- RFile 54, 60, 144
- RFs::NotifyChange() 304
- RGA. *See* real-time graphics and audio
- RHandle 213
- RHashMap 59
- RHeap 122–123, 399
- RIM. *See* Research in Motion
- RMessagePtr2 92, 351
- RPointerArray 57, 58
- RProcess 234
- RProperty 248, 359
- RReadStream 216
- RSendAs 139
- RSendAsMessage 139
- RSessionBase 95
- RSockServ 340
- RSqlDatabase 359
- RThread 122, 123, 142, 144
- RTimer 127
- RunError() 97–98
- RunL() 97, 141
- runtime_error 119
- RVCT. *See* ARM RealView Compilation Tools; Realview toolchain
- RWindow 215
- RWindowGroup 303
- RWriteStream 216
- RWsSession 216, 304
- S60 60, 106, 161–162, 301–304
- Samsung GSensor 282–284
- SAX. *See* Simple API for XML
- scalable vector graphics (SVG) 183–184
- scope-resolution operator 55
- screen resolution 43–44
- SDKs 31, 33–34
 - C 214
 - Symbian Developer Library 41
 - Windows 210–214
- SDL. *See* Simple DirectMedia Layer
- Secure ID (SID) 130, 356
- security 117–118, 129–131, 347–373
 - Android 369–371
 - iPhone 371–372
 - Linux 204–205
 - Maemo 371
 - Openmoko 371
 - Windows 210
 - Windows Mobile 234, 372–373
- semget() 125
- Send() 200–201
- services 249
- SetActive() 97
- SetContainer-WindowL() 215
- setDefault() 280
- setgid() 121
- SetLength() 70
- SetMax() 70
- setpgid() 121
- setPls() 345
- setsockopt() 127
- setuid() 121
- setvbuf() 122
- shmget() 125, 339
- short int 56
- SID. *See* Secure ID
- SIGALRM 127
- SIGKILL 342
- SIGKILL/SIGQUIT 125
- signed char 56
- signing 233–236
- SIGSTOP 342
- Simple API for XML (SAX) 184
- Simple DirectMedia Layer (SDL) 168–169
- SIS 38, 39, 126, 229
 - Carbide.c++ 368–369
 - Open Signed Online 366–367
- Size() 69, 150
- SizeChanged() 149
- sizeHint() 268
- sockets 124, 127, 223
- soft input panels 221
- SoundStretch 115–120
- SoundTouch 36, 37, 115–120
- SourceForge 190
- SPARQL 397
- sparse files 122
- SQL 183, 359
- Standard Template Library (STL) 55, 107–108
- static library (LIB) 109

- st_atime 121
- std::bad_alloc 76
- stdcpp 333
- STDDLL 40–41, 112–113, 117, 336
- stderr 45, 119–120, 121, 334, 338
- STDEXE 112–113, 142, 336
- stdin 121, 334, 337
- stdio 126, 335, 337–338
- stdioserver 120, 126
- STDLIB 112–113
- STDLIBS-INIT panic 123
- stdout 45, 121, 126, 334, 337
- std::runtime_error 115
- STL. *See* Standard Template Library
- STLport 110, 293, 296
- st_mtime 121
- strcpy() 343
- string handling 59–60
- StringLoader 229
- struct 52, 82, 343–345
- Subversion 392
- suffixes 51–52
- SurroundingsDD 366, 367
- Suspend() 128
- SVG. *See* scalable vector graphics
- SwEvent 161, 366, 367
- Symbian Developer Library 41, 57
- Symbian Signed 234–235
 - certificates 364–366
 - Open Signed 39
 - testing 398
 - UID 38
- _SYMBIAN32_ 46, 294
- symlink() 122
- system() 124
- SYSTEMINCLUDE 116
- T classes 52, 79
- TAny 56–57, 213
- Target-Action 268
- TBool 57, 213
- TBuf 63–64, 67, 149
- TCB. *See* Trusted Computing Base
- TCB 117, 350, 355
- TCHAR 219, 322
- TCP/IP 6, 8, 223
- TDbIQue 59
- TDbIQueLink 59
- TDD. *See* test-driven development
- TDes 61–63, 67
- TDesC::Ptr() switch, 63
- telephony 139, 185, 251
- Temple 301
- Tempo 114
- test-driven development (TDD) 44
- text editor 32
- Themes API 167
- this 382
- thread local storage (TLS) 100, 335, 403
- threads
 - active objects 94–95
 - Cocoa 264–266
 - ECOM 230
 - heaps 122
 - Qt 264–266
- timestamps 121
- TInt 56, 61, 92
- TLeave 76
- TLitC 67
- TLocale 335
- TLS. *See* thread local storage
- TMPDIR 335
- TPoint 150, 213
- TPriority 95
- TPtr 63, 67, 153
- Trac 300
- transient server 337
- TranslateMessage() 216
- trap handlers 45, 72, 73, 76–78, 85, 90, 110, 143, 285
- TReal 56
- TRect 213
- TRequestStatus 93, 96, 127, 142
- TRgb 150
- True Type fonts (TTF) 164
- trusted applications 362–366
- Trusted Computing Base (TCB) 350, 355
- TrustedUI 366, 367
- try...catch 143, 391
- TSecurityPolicy 359
- TTF. *See* True Type fonts
- TUInt 56, 61, 213
- two-phase construction 86–91
- typedef 64, 386
- TZ 335
- Ubuntu Mobile 12
- _UHEAP_MARKEND 399
- _UHEAP_SETFAIL 72
- UI. *See* user interface
- UIAccelerometer 270
- uic 305
- UID 37, 117, 228
 - EXE 118
 - Symbian Signed 38
- UID3 356
- UIImagePicker 270
- UIKit 135, 267–269
- UIQ 10, 33–34, 106
- UITableView 268
- unsigned int 56
- untrusted applications 363–364
- USER 11 panic 60, 64, 92
- USER 42 panic 82, 90
- USER 129 panic 57
- user interface (UI) 33–34, 134–136, 266–269. *See also* graphical user interface
- DLL 29
- engine 29
- fragmentation 10–12

- games 136
- Linux 202–203
- reunification 12–13
- Windows Mobile
 - 238–239
- UserEnvironment 154,
 - 349, 364, 365
- User::Free() 82, 84
- User::Leave() 73, 74
- User::LeaveIfError()
 - 73
- User::Panic() 92
- User::ResetIn-
 - activityTime()
 - 155
- User::SetCritical()
 - 91
- User::SwitchHeap()
 - 123
- utility APIs 166–168

- VARIANT 230
- Vendor ID (VID) 356–357
- views 267–269

- Virtual Code API 168
- virtual memory 398
- Visual Studio 32,
 - 211–212, 237
- vsnprintf() 319

- wait() 124, 198
- waitpid() 124, 198
- wcelibex 312
- wchar 197
- wchar_t 320
- widgets 267–269
- Win32 293, 294, 295,
 - 297, 299, 318
- Windows 7, 207–241
- Windows Mobile 11,
 - 18–19, 211–212,
 - 223–226, 234
 - debugging 213–214
 - security 234, 372–373
 - UI 238–239
 - UIDs 228
- Windows Vista 31
- Windows XP 32

- WINSCW 34
- _WINSCW_ 284
- WndProc() 314
- World Wide Web
 - Consortium (W3C)
 - 397
- writable static data (WSD)
 - 99–100, 111,
 - 343–345
 - constructors 99
 - P.I.P.S. 333–334,
 - 343–346
- WriteDeviceData 349,
 - 366, 367
- WriteUserData 129,
 - 349, 363, 367
- WSD. *See* writable static data

- XLeaveException 74,
 - 85, 90

- zlib 294, 295, 296
- Zsh 119