HARRISON

SYMBIAN
OS C++
FOR
MOBILE
PHONES

TK 657Ø
.M6 H295
2ØØ3
Vol 3
Copy 1

WILEY

Symbian OS C++ for Mobile Phones

Volume 3

Application Development for Symbian OS v9

symbian

Richard Harrison
& Mark Shackman

Symbian OS C++ for Mobile Phones
Volume 3

Symbian OS C++ for Mobile Phones Volume 3

Richard Harrison, Mark Shackman

With

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

Reviewed by

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

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

TK 6570
.M6 H295
2003
vol. 3
Copy 1

Copyright © 2007 Symbian Software Ltd

Published by John Wiley & Sons, Ltd The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emailed to permreq@wiley.co.uk, or faxed to (+44) 1243 770620.

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

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

Other Wiley Editorial Offices

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

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

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

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

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

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

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

Library of Congress Cataloging-in-Publication Data

Harrison, Richard.
Symbian OS C++ for mobile phones / By Richard Harrison.
p. cm.
Includes bibliographical references and index.
ISBN 0-470-85611-4 (Paper : allk. paper)
1. Cellular telephone systems – Computer programs. 2. Operating systems (Computers) I. Title.
TK6570.M6H295 2003
621.3845'6 – dc21

2003006223

British Library Cataloguing in Publication Data

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

ISBN: 978-0-470-06641-6

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

Contents

About the Authors xi

Acknowledgements xix

Symbian Press Acknowledgments xxi

About this book xxiii

Glossary xxvii

1 Getting Started 1
- 1.1 Using the Emulator 1
- 1.2 Hello World – Text Version 6
- Summary 15

2 A System Introduction to Symbian OS 17
- 2.1 Symbian OS Basics 17
- 2.2 Framework Basics 30
- 2.3 APIs Covered in this Book 36
- Summary 39

3 Symbian OS C++ 41
- 3.1 Fundamental Data Types 41
- 3.2 Naming Conventions 43
- 3.3 Functions 49
- 3.4 APIs 51
- 3.5 Templates 55
- 3.6 Casting 56

3.7 Classes 57
3.8 Design Patterns 59
Summary 60

4 Objects – Memory Management, Cleanup and Error Handling 61
4.1 Object Creation and Destruction 62
4.2 Class Categories in Symbian OS 66
4.3 Error Handling 70
4.4 The Cleanup Stack 76
4.5 Two-Phase Construction 81
Summary 85

5 Descriptors 87
5.1 Overview 87
5.2 Anatomy of Descriptors 91
5.3 Literals 93
5.4 Stack Descriptors 93
5.5 Pointer Descriptors 95
5.6 Heap Descriptors 99
5.7 Narrow, Wide and Neutral Descriptors 108
5.8 Descriptors and Binary Data 109
5.9 Using Descriptors with Methods 110
5.10 Some Descriptor Operations 121
5.11 Correct Use of Descriptors 128
5.12 Manipulating Descriptors 138
Summary 148

6 Active Objects 151
6.1 The Asynchronous Service 151
6.2 Multitasking and Pre-emption 156
6.3 A More In-depth Look at Active Objects 158
6.4 How It Works 167
6.5 Active Object Priorities 171
6.6 Active Object Cancellation 176
6.7 Starting and Stopping the Scheduler 181
6.8 Understanding a Stray Signal 182
6.9 Other Common Active Object Errors 187
6.10 Implementing State Machines 190
6.11 Long-Running Tasks and Active Objects 194
Summary 200

7 Files and the File System 203
7.1 File-Based Applications 203
7.2 Drives and File Types 204

3.7 Classes 57
3.8 Design Patterns 59
Summary 60

4 Objects – Memory Management, Cleanup and Error Handling 61
4.1 Object Creation and Destruction 62
4.2 Class Categories in Symbian OS 66
4.3 Error Handling 70
4.4 The Cleanup Stack 76
4.5 Two-Phase Construction 81
Summary 85

5 Descriptors 87
5.1 Overview 87
5.2 Anatomy of Descriptors 91
5.3 Literals 93
5.4 Stack Descriptors 93
5.5 Pointer Descriptors 95
5.6 Heap Descriptors 99
5.7 Narrow, Wide and Neutral Descriptors 108
5.8 Descriptors and Binary Data 109
5.9 Using Descriptors with Methods 110
5.10 Some Descriptor Operations 121
5.11 Correct Use of Descriptors 128
5.12 Manipulating Descriptors 138
Summary 148

6 Active Objects 151
6.1 The Asynchronous Service 151
6.2 Multitasking and Pre-emption 156
6.3 A More In-depth Look at Active Objects 158
6.4 How It Works 167
6.5 Active Object Priorities 171
6.6 Active Object Cancellation 176
6.7 Starting and Stopping the Scheduler 181
6.8 Understanding a Stray Signal 182
6.9 Other Common Active Object Errors 187
6.10 Implementing State Machines 190
6.11 Long-Running Tasks and Active Objects 194
Summary 200

7 Files and the File System 203
7.1 File-Based Applications 203
7.2 Drives and File Types 204

7.3 File System Services 205
7.4 Streams 216
7.5 Stores 224
Summary 239

8 Interprocess Communication Mechanisms 241
8.1 Overview 241
8.2 Client–server IPC 247
8.3 Publish and Subscribe IPC 253
8.4 Message Queue IPC 257
8.5 Which IPC Mechanism Should You Use? 259
Summary 261

9 Platform Security and Publishing Applications 263
9.1 Releasing an Application 263
9.2 How Does Platform Security Work? 264
9.3 How Do I Support Platform Security? 266
9.4 Preparing an Application for Distribution 268
9.5 Overview of Symbian Signed 272
9.5 Installing a SIS File 279
9.6 List of Capabilities 280
Summary 281

10 Debugging and the Emulator 283
10.1 Using the Emulator 283
10.2 Emulator Debugging 289
10.3 Debugging on a Phone 306
10.4 Miscellaneous Tools 308
Summary 309

11 The Application Framework 311
11.1 Symbian OS Application Framework 311
11.2 S60 and UIQ Platform Application Frameworks 313
11.3 A Graphical Hello World 315
Summary 330

12 A Simple Graphical Application 331
12.1 Implementing the Game on S60 333
12.2 Differences for UIQ 3 355
Summary 366

13 Resource Files 367
13.1 Why a Symbian-Specific Resource Compiler? 367
13.2 Source File Syntax 368
13.3 Bitmaps and Icons 372

13.4 Updating the Resource Files 377
13.5 Application Registration Files 378
13.6 Localizable Strings 379
13.7 Multiple Resource Files 382
13.8 Compiling a Resource File 383
13.9 The Content of a Compiled Resource File 385
13.10 Reading Resource Files 388
Summary 389

14 Views and the View Architecture 391
14.1 The View Architecture 392
14.2 The `MCoeView` Interface 397
14.3 Introduction to the Example Application 398
14.4 Creating and Managing the Views 406
14.5 Implementing the `MCoeView` Interface 408
14.6 Command Menus 414
Summary 421

15 Controls 423
15.1 What Is a Control? 423
15.2 Control Types 424
15.3 Control Layout 429
15.4 Handling Key and Pointer Events 432
15.5 Observing a Control 442
15.6 Drawing a Control 444
15.7 Backed-up Windows 451
15.8 Backed-up-Behind Windows 452
15.9 Dimmed and Invisible Controls 454
Summary 455

16 Dialogs 457
16.1 What Is a Dialog? 457
16.2 Simple Dialogs 459
16.3 Complex Dialogs 462
16.4 Single-Page Dialogs 464
16.5 Multi-Page Dialogs 466
16.6 Dialog APIs 472
16.7 Stock Controls for Dialogs 475
16.8 Custom Controls in Dialogs 477
Summary 480

17 Graphics for Display 481
17.1 Drawing Basics 482
17.2 The `CGraphicsContext` API 486

17.3 Drawing and Redrawing 492
17.4 Drawing Controls 499
17.5 Sharing the Screen 502
17.6 Support for Drawing in `CCoeControl` 514
17.7 Special Effects 521
17.8 Window Server Features 526
17.9 Device- and Size-Independent Graphics 529
Summary 556

18 Graphics for Interaction 557
18.1 Key, Pointer and Command Basics 558
18.2 User Requirements for Interaction 559
18.3 Some Basic Abstractions 561
18.4 Processing Key Events 567
18.5 Processing Pointer Events 571
18.6 Window Server and Control Environment APIs 576
Summary 582

19 Plug-ins and Extensibility 583
19.1 System Services 583
19.2 What Is a Plug-in? 587
19.3 The ECOM Library 593
19.4 Plug-ins in Symbian OS 603
Summary 604

20 Communications and Messaging Services 605
20.1 Communications in Noughts and Crosses 605
20.2 Communication Between Controller and Transport 606
20.3 Serial Communications 609
20.4 Socket-based Communications 615
20.5 Messaging 632
20.6 Security 646
Summary 649

21 Multimedia 651
21.1 The Multimedia Framework 651
21.2 The Image Conversion Library 682
21.3 Camera API 699
21.4 Tuner API 706
Summary 712

22 Introduction to SQL RDBMS 715
22.1 Overview of RDBMS 715
22.2 SQL Basics 716

22.3 Symbian SQL Server Component Architecture 729
22.4 Symbian SQL Error Codes 751
Summary 753

Appendix: Developer Resources 755

References 761

Index 763

About this book

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

The book is broadly organized into four sections:

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

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

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

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

Who Is This Book For?

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

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

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

Conventions

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

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

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

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

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

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

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

We show commands typed at the command line like this:

```
abld build winscw udeb
```

Glossary

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

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

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

1

Getting Started

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

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

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

1.1 Using the Emulator

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

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

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

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

Launching the Emulator

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

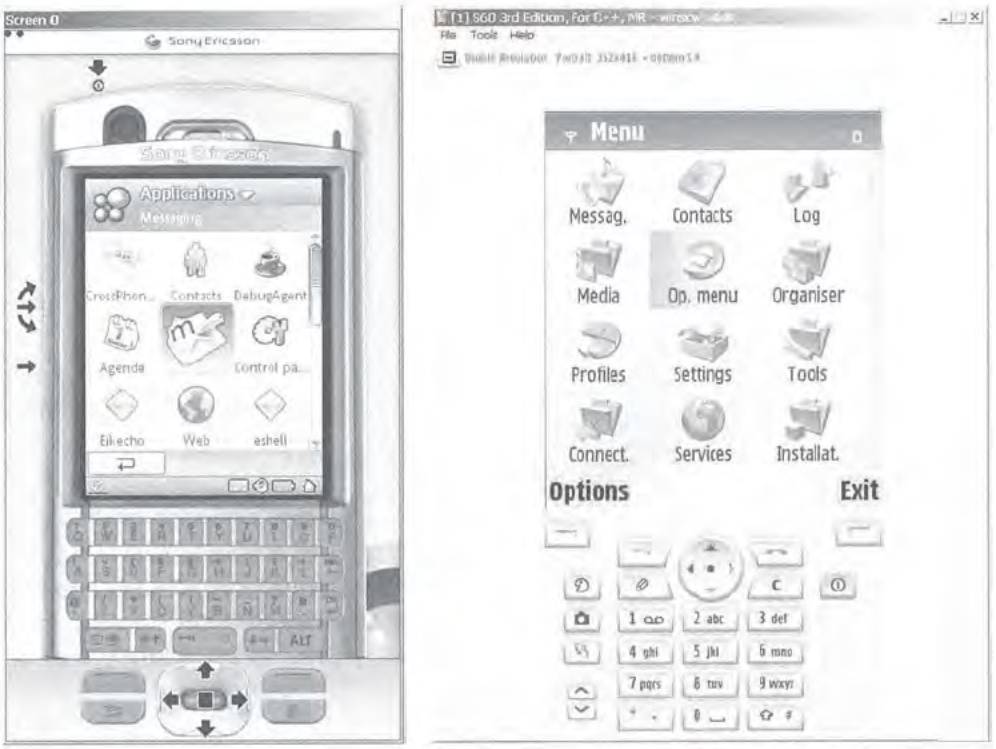
- launch the executable `epoc.exe`, which you'll find in the directory `\epoc32\release\winscw\udeb` underneath the directory in which the SDK is installed
- from the Start menu, select either Programs, UIQ SDK or Programs, S60 Developer Tools, 3rd Edition SDK and select Emulator from the appropriate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

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

GUI Style

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

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



(a) (b)

Figure 1.1 UIQ (with P990 extensions) and S60 emulators

Screen layout

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

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

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

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

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

Figure 1.2 UIQ screen layout

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

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

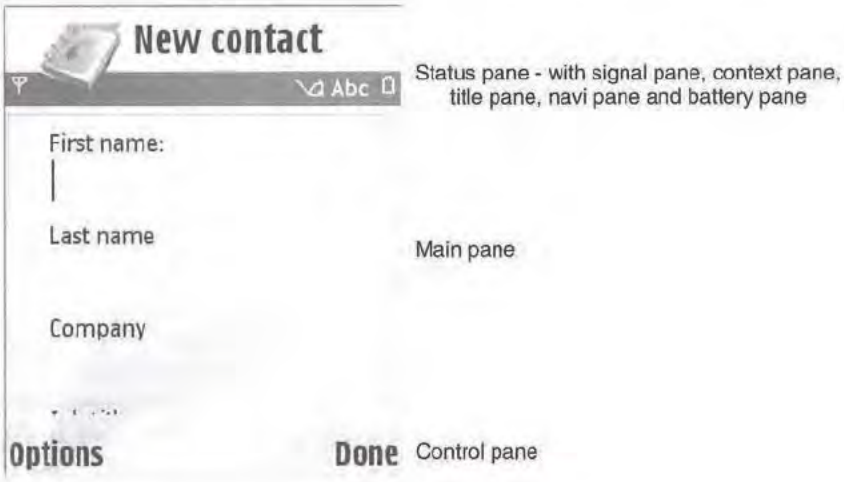


Figure 1.3 S60 screen layout

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

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

Menus

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

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

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

Figure 1.4 Menu structure in the Calendar application

Figure 1.5 S60 cascaded menu items

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

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

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

1.2 Hello World – Text Version

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

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

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

The Program: `HelloText`

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

```
// hellotext.cpp

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

LOCAL_D CConsoleBase* gConsole;

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

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

  // Call function
  MainL();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here's `ConsoleMainL()`:

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

  // Call function
  MainL();

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

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

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

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

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

with something like:

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

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

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

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

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

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

The Project Specification File

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

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

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

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

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

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

The Component Definition File

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

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

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

Building from the Command Line

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

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

```
bldmake bldfiles
```

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

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

```
abld build winscw udeb
```

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

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

The build is split into six phases:

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

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

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

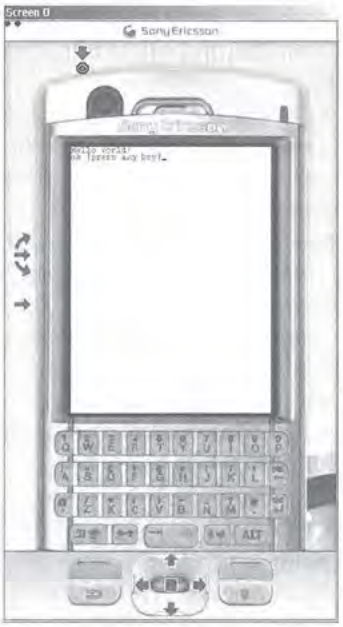


Figure 1.6 `HelloText` emulator output

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

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

Using Multiple SDKs

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

Building in the Carbide.c++ IDE

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

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

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

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

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

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

Summary

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

The topics we've looked at are:

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

2

A System Introduction to Symbian OS

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

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

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

2.1 Symbian OS Basics

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

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

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

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

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

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

About the Kernel

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

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

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

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

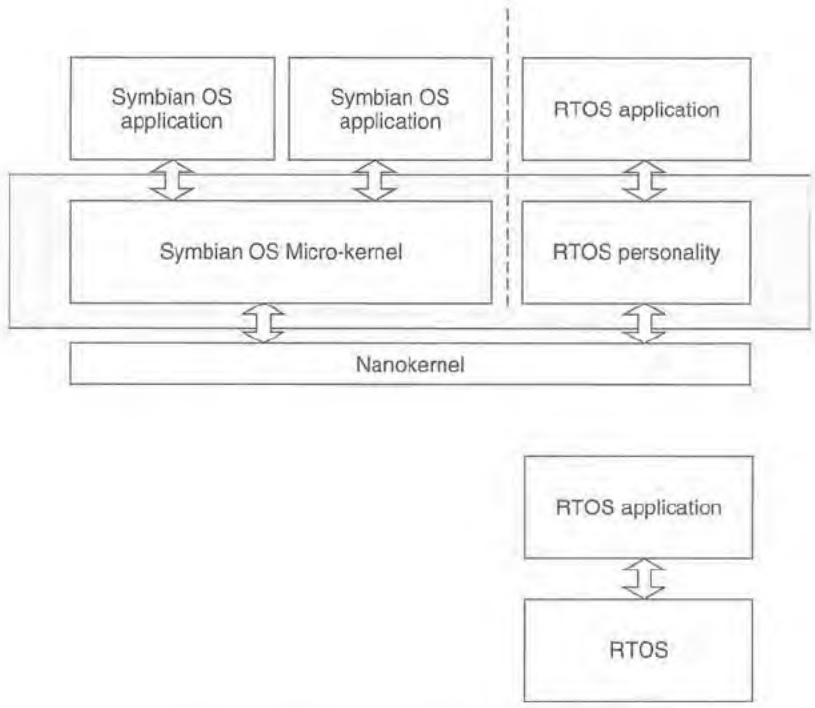


Figure 2.1 Symbian OS kernel architecture

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

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

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

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

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

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

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

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

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

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

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

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

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

Crossing from user side to kernel side

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

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

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

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

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

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

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

Kernel threads

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

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

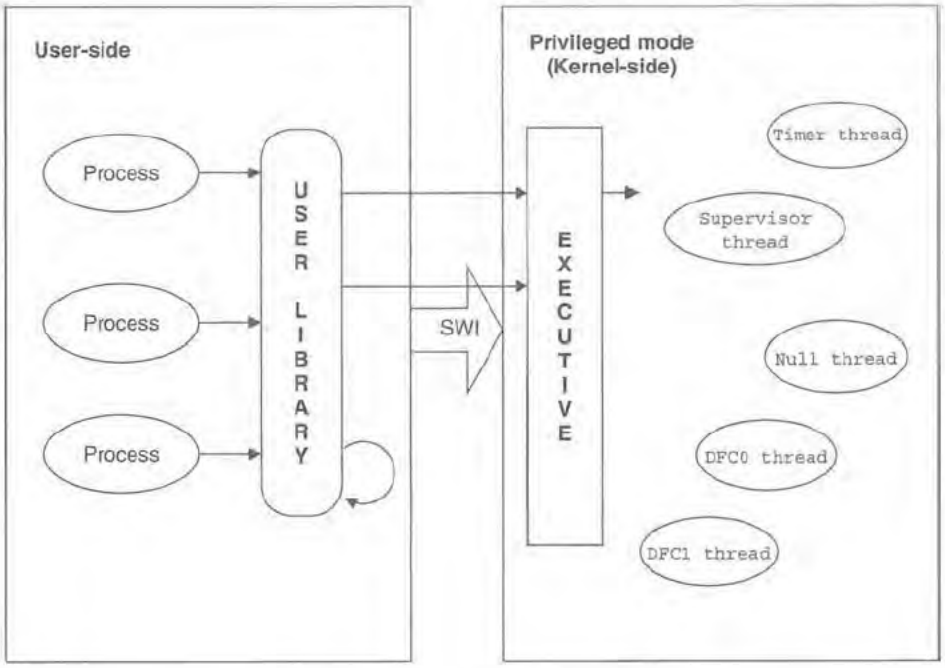


Figure 2.2 User-side processes and the kernel

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

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

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

Threading and Process Model

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

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

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

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

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

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

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

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

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

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

User Memory and Memory Management

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

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

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

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

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

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

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

Symbian OS Servers

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

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

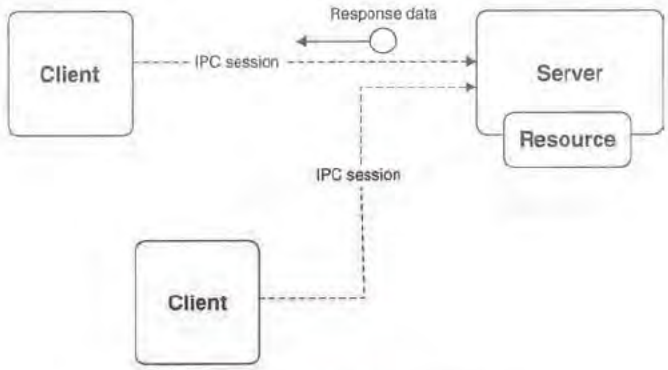


Figure 2.3 Client–server communications

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

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

Symbian OS Inter-Process Communication

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

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

Power Management

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

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

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

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

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

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

Timers

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

Executables and Shared Libraries

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

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

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

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

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

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

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

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

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

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

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

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

Writable Static Data Optimization

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

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

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

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

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

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

Files and the File Server

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

Platform Security

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

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

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

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

2.2 Framework Basics

Symbian OS C++

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

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

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

Concurrency and Active Objects

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

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

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

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

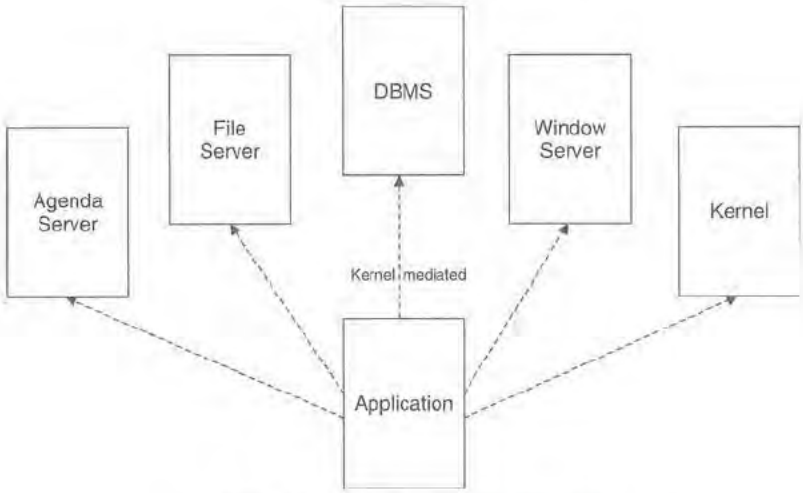


Figure 2.4 Typical concurrent processes

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

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

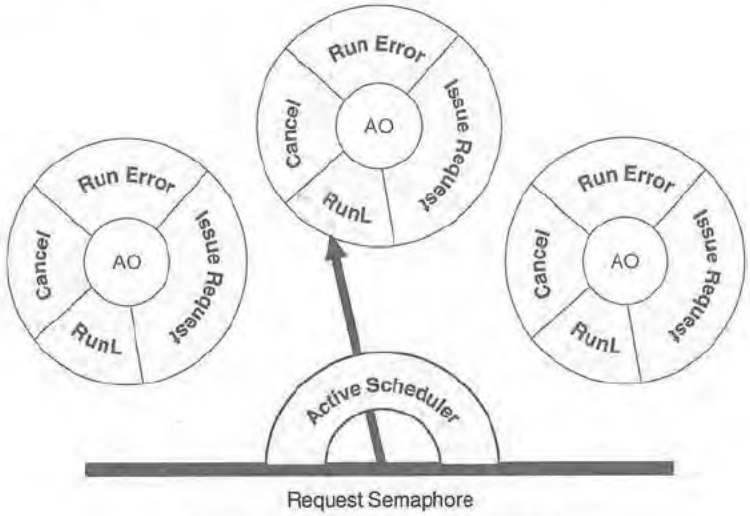


Figure 2.5 Scheduling of active objects

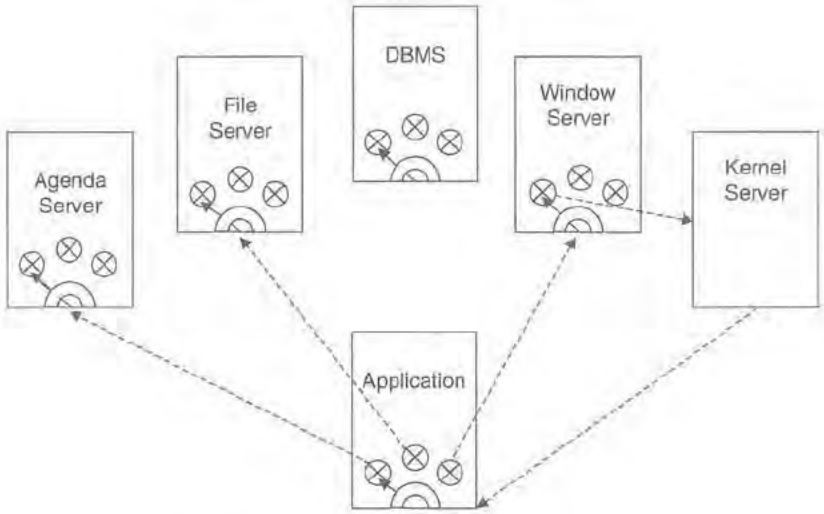


Figure 2.6 Active objects in concurrent processes

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

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

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

Event handling with active objects

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

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

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

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

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

Event-handling example

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

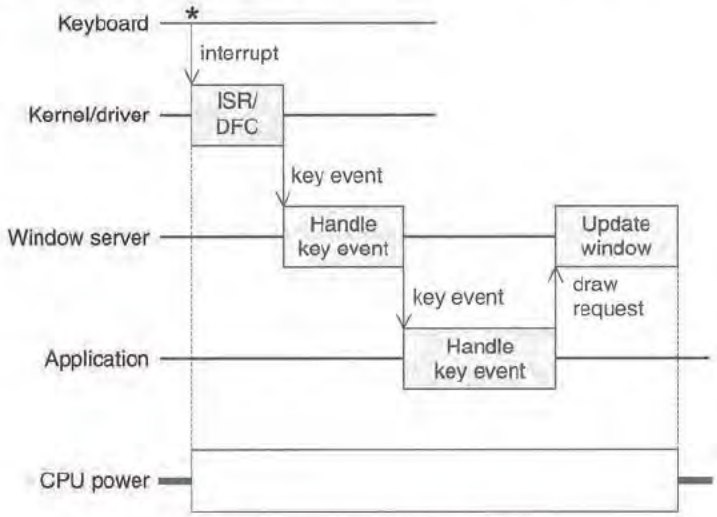


Figure 2.7 Events triggered by a key press

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

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

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

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

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

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

Symbian OS C++ Exception Handling

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

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

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

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

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

Here is a brief example using the cleanup stack:

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

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

  CleanupStack::Pop(tmpObject); // we know an exception was
                                // not thrown and we may remove
                                // the object from the stack
  ...
  // ...and pass ownership of that object
  // OR possibly even do the following instead
  CleanupStack::PopAndDestroy(tmpObject);
  }
```

So how does the cleanup stack work?

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

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

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

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

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

Symbian OS C++ Naming Conventions

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

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

The coding standard is discussed in detail in Chapter 3.

2.3 APIs Covered in this Book

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

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

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

These divide into several broad groupings:

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

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

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

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

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

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

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

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

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

Summary

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

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

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

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

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

9

Platform Security and Publishing Applications

This chapter deals with preparing your application for distribution. We discuss information related to building an application for, and running it on, a target machine, including giving your application an icon and building a distributable SIS file from a PKG file. There is also an introduction to platform security, so that you can successfully prepare your application for the Symbian Signed program.

9.1 Releasing an Application

In order to release your application, some important steps must be taken. These steps do not require you to add more C++ code, but are considered critical to ensuring the success of an application in the mass market. The extra work involved focuses on things such as preparing an icon for the application, building it as an installable application and authorizing it under the Symbian Signed program. The last step is especially important with the introduction of platform security in Symbian OS v9.

Symbian OS v9 is a major evolution. It is specifically geared to target mid-range phones in tens and hundreds of millions of units – offering even larger opportunities for developers to market their products. As such, it includes a significant number of new features and improvements to enable key emerging technologies required by advanced, open mobile phones for years to come. In order to provide comprehensive support for new functionality such as Digital Rights Management, Device Management and Enterprise-grade data handling, large changes have been made to the

very core of Symbian OS to support vital concepts such as data protection or 'caging' and restricting some API usage.

9.2 How Does Platform Security Work?

The platform security enhancements represent an evolution of the perimeter security model of Symbian OS v8 and help ensure the stability of the platform. They provide greater protection against malicious or badly implemented applications, by operating at the software level to:

- detect unauthorized access attempts to hardware, software or data which may lock up the phone or affect other software or the network
- prevent applications from acting in unacceptable ways, irrespective of whether these actions are intentional (i.e. malware) or unintentional. Such actions might include unauthorized access to hardware, attempts to read or write restricted data, etc.

This section gives you an introduction to the way that platform security is implemented, with a particular focus on ensuring that you know how to update your applications to conform to and work within the platform-security-enabled Symbian OS environment. Knowing how to do this will allow you to successfully go through the Symbian Signed process and release your application.

Capabilities

Platform Security provides protection by ensuring that only trusted applications can use certain functionality within Symbian OS. It introduces the idea of *capabilities*, which protect APIs within Symbian OS.

Not all of the APIs in the Symbian OS are capability-protected – in fact, only about 40% of Symbian OS APIs fall within this category; all other APIs still have unrestricted access. Of those APIs that are protected, many are used to access the low-level aspects on Symbian OS, so it is unlikely that you will need to use them.

If your application needs to use any of the APIs that are protected by a capability, it must request use of the capability. This is called *authorizing* your code. Once the authorization is obtained, the application is *authorized* and can use the protected functionality.

The Symbian Signed programme is your route to authorizing and releasing your applications before the user receives them so we'll take a quick look at this now, with more detail in Section 9.5.

Authorizing via Symbian Signed

There are a variety of capabilities (see Section 9.7) that are used to protect functionality within Symbian OS. To ease the process of authorizing applications, Symbian Signed groups them together into two sets for verification – basic and extended.

- The basic capabilities protect the APIs that: access confidential user data; provide data about the user's environment; and access network services (both locally and remotely). Being authorized for the basic capabilities ensures that users of your application are authorized for access to services and to data (for example, to send text messages, make Bluetooth connections to other phones and access user-confidential data such as contact information) without the user being asked explicitly for permission.
- The extended capabilities protect the APIs that provide lower-level access to system functions, such as changing the network configuration, in addition to the basic capabilities. Applications requiring authorization for the extended set via Symbian Signed have to undergo additional testing as these APIs require what is known as 'phone-manufacturer approval'.

Within the extended set are some particularly sensitive capabilities, designated 'Phone-Manufacturer Approved', which include DRM, DiskAccess, NetworkControl and AllFiles. These capabilities control access to areas of the phone or content on the device that manufacturers may want to restrict, and therefore Symbian Signed has additional criteria to be satisfied if your application needs any of these capabilities.

If you decide to follow the Symbian Signed route, Section 9.5 contains more details of what you need to do.

Authorizing without using Symbian Signed

Sometimes you don't actually need to authorize your code in advance to use the capability-protected functionality.

The Symbian Signed process does a good job in protecting the phone from errant applications and providing a number of routes to market, but there are times when getting an application signed may not be appropriate. In this situation, you can take advantage of the Unsigned-Sandboxed capabilities set. The unsigned-sandboxed capabilities consist of all functionality that doesn't require any capabilities and all capability-dependent functionality for which access may be granted by the user.

If the application is not signed, Symbian OS can ask the phone user directly when the application is installed if they are prepared to authorize the unsigned-sandboxed capabilities that the application declares in its

project definition (MMP) file (this is called a *blanket grant*). An unsigned-sandboxed application only has access to a subset of APIs. If your application only needs to access functionality in this set there is no need to submit it to Symbian Signed. That said, a phone manufacturer may still require you to 'sign' the application with a certificate that you can create using tools in the SDK.

If you prefer your application to be authorized and that the phone user never has to be asked for authorization to perform operations, then the application should be Symbian Signed. It is also worth noting that, depending on how you intend to distribute or sell your application, some channels may require an application to be Symbian Signed before they even consider it for distribution via their sales channels.

One-shot Grants

It's not possible for users to blanket-grant all capabilities – only a Symbian Signed application can have this authorization. Instead, if you need limited use of a particular capability (perhaps to send an SMS), you can use selected APIs on a 'one-shot grant'. To achieve this, because the access is being granted as a one-off, special case, you mustn't request these capabilities in your project definition file like other capabilities (otherwise installation will fail as they cannot be authorized by the user) – just ensure they are excluded.

If you subsequently want your application to be Symbian Signed, with capabilities you require being given a blanket grant, you only need to add the required capability to your project definition file and your application code can remain unchanged.

9.3 How Do I Support Platform Security?

In order to support the platform security architecture, a number of APIs were added, amended or deleted in Symbian OS v9. Information about these changes is available in the SDKs. If any of the APIs that you use in applications running under versions of Symbian OS before v9 are affected, you need to make the appropriate changes.

Project Definition Files

The capabilities that an application needs are defined at build time in the project's definition file. The `CAPABILITY` keyword is followed by a list of the capabilities that the executable requires.

```
TARGET       example.dll
CAPABILITY   LocalServices ReadUserData
```

A list of capabilities can be found in Section 9.7.

You can work out what capabilities need to be specified by checking the documentation – this shows which API calls are protected by which capabilities – and specifying all these capabilities in the MMP file. (You can also use the emulator to see if you've omitted any required capabilities, see Chapter 10.)

Instead of specifying capabilities in a list, the special capability name `NONE` can be used. This is the default value if the `CAPABILITY` keyword is not found in the MMP file.

There are a number of APIs whose capabilities do not necessarily have to be specified in the MMP file. If the relevant capability is not specified, authorization may still be granted on a one-shot basis. (This applies to both signed and unsigned-sandboxed applications, as discussed in Section 9.2.)

Secure IDs

In Symbian OS v9, each executable has a Secure ID (SID) which uniquely identifies it. SIDs are a specialized use of the UID3 value, which is specified in your application's MMP file with the keyword `UID`. The SID value, unless explicitly specified otherwise by the `SECUREID` keyword in the MMP file, defaults to the same value as the application's UID3. To avoid confusion, a `SECUREID` should not be specified in the application's MMP file. The SID value is not relevant for DLLs since the SID of a process is always that of its EXE.

A SID can be requested in the form of a UID from the Symbian Signed portal, ***www.symbiansigned.com***. This is an automated process and the UIDs are issued instantly. Note that the UIDs are divided into two ranges and only values from the protected range are valid SIDs. UIDs from the unprotected range are often used during development and testing. The Symbian OS software installation system checks that a file is using a UID from the protected range. If not, your application won't install.

Data Caging

Another feature of platform security is data caging. This means that each application, when installed, has its own private directory on the phone that is accessible only by a process with either the same UID as the application or with certain privileged capabilities. For example, your application cannot access configuration files of another application that the user has installed.

Each data-caged directory resides under the `\private` directory on the phone and is named according to the SID of the main application:

```
\private\<application SID>\...
```

The private directory can be created on any writable drive, however only certain drives can be described as secure. A secure drive must have media that is not removable and cannot be accessed by an external 'host' computer using a mass-storage file system connected via USB. For drives that do not satisfy these criteria, the data stored in the secure areas can easily be accessed by the computer system connected to them.

Other directories in the file system are also restricted by Symbian OS, regardless of drive:

- `\resource\...` contains resource files for installed applications. Resource files for an application can only be written to this directory at installation time by the Symbian OS software installer. No capabilities are required to read the resource files stored in this directory.
- `\sys\...` contains system files and installed binaries. Again these binaries can be written to only by the software installer. If an application is installed on media that can be accessed by an external computer then the binary files are protected by a hash checksum. Each time the binaries are loaded, the hash is checked to ensure the binary has not been modified in any way. If tampered with, a binary is not loaded.

Other areas of the file system, from the root directory upwards, are considered public and can be accessed with no capabilities.

9.4 Preparing an Application for Distribution

Having discussed Platform Security, we need to see what effect it has on distributing applications. To do this, we first show you how to build your application so that it can be installed on a phone and then go through the steps required to authorize your application via Symbian Signed.

Building the Application

Up to this point, we've done everything in the emulator. Now it is time to rebuild the application for ARM and install it onto a Symbian OS phone using the appropriate user interface.

Building the application for the target phone is simple. You can build it from the Carbide.c++ IDE, after selecting the ARM UREL target, or you can build it from the command line by typing:

```
abld build armv5 urel
```

This builds a releasable application, which is ready to be copied onto the phone itself. To install the application on to the phone, Symbian OS

provides a powerful, yet straightforward installation system that offers a simple and consistent user interface for installing applications, data or configuration information on to phones.

The basic idea is that end users install all the code, packaged in Symbian OS installation (SIS) files. These can be transferred to the phone from a PC using connectivity software, Bluetooth, infrared or even email and an Internet connection, as illustrated in Figure 9.1.

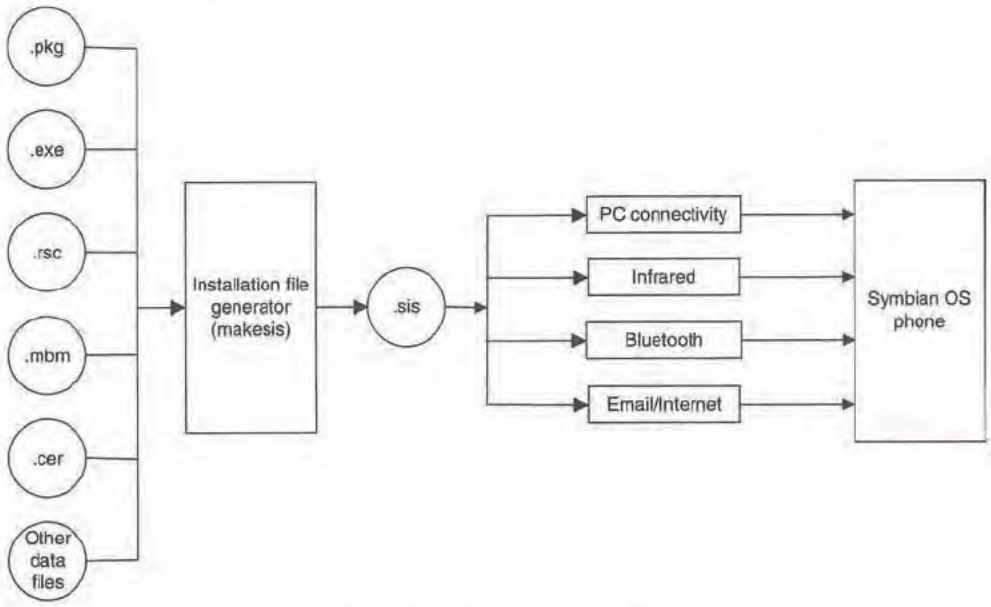


Figure 9.1 Installation methods

The SIS files are generally very small in size (at most a few hundred kilobytes) and are therefore very quick to transfer to the target phone. Three command-line tools are provided to enable you to create SIS files:

- the Installation File Generator, `makesis.exe`, creates the installation files from information specified in the package file
- the Certificate Generator, `makekeys.exe`, creates private–public key pairs and a certificate file; it is used by the Installation File Generator to digitally sign the installation files
- the Installation File Signer, `signsis.exe`, digitally signs the installation files with a Class 3 ID certificate such as an ACS Publisher ID issued by TC TrustCenter.

There are three steps involved in generating the SIS file.

1. Create a PKG package file that specifies all the files that go into the SIS file, and where they are to be installed on the target phone.

2. Run the `makesis` command line tool to generate the SIS file.
3. Sign the application using `signsis.exe`. If the application is to be submitted to Symbian Signed, you sign it with an ACS Publisher ID. If the application does not need to be Symbian Signed, certain phone manufacturers may require you to sign it with a user-generated certificate.

Producing the Package File

Before you create the SIS file, you need to specify information about the application in a package file. This is a plain text file but, because of its flexibility, the format is one of the most obscure of the Symbian OS tool control files.

For our example, we start with the `OandX.pkg` file from the Noughts and Crosses application that is described in Chapter 12. Bear in mind that the resulting SIS file cannot be installed on any phone based on Symbian OS v9, unless it is signed. Here is the content of `OandX.pkg` for an S60 phone:

```
; OandX.pkg
; Language - standard language definitions
&EN
; Vendor
:"Symbian Software Ltd."
%{"Symbian Software Ltd."}
; standard SIS file header
#{"OandX (S60)"},(0xE04E4143),1,0,0,TYPE=SA
; Supports S60 v3.0
[0x101F7961], 0, 0, 0, {"Series603rdEditionProductID"}
"\epoc32\data\z\resource\apps\OandX.rsc"-
    "!:\resource\apps\OandX.rsc"
"\epoc32\data\z\resource\apps\OandX_loc.rsc"-
    "!:\resource\apps\OandX_loc.rsc"
"\epoc32\data\z\resource\apps\OandX.mbm"-
    "!:\resource\apps\OandX.mbm"
"\epoc32\data\z\private\10003a3f\import\apps\OandX_reg.rsc"-
    "!:\private\10003a3f\import\apps\OandX_reg.rsc"
"\epoc32\release\armv5\urel\OandX.exe"-"!:\sys\bin\OandX.exe"
```

The body of this file should be obvious enough: all source files that will be packaged into the SIS file are listed, along with information about where they should be unpacked upon installation. The '`!`' drive specifier means 'the chosen installation drive'. If you require a file to reside on the `C:` drive of the phone, for instance, you can specify that instead, but this is rare, and should only be used if absolutely necessary.

The header is more interesting.

- The `&` line specifies the languages supported by this installation file: in this case, English only.

- The : and % lines specify the localized and non-localized vendor name.
- The # line specifies the application's caption to be used at install time, its final SID and its three-part version number – 1,0,0 is specified here, which will be displayed as 1.0(000), indicating major, minor and build.

This is the general form of a PKG file for a basic, single-language application. In fact, the PKG file format supports more options than this, including nested packages, multi-language installation files and required dependencies. All these are documented in the SDK.

One thing to note is that, because Symbian OS is so flexible and allows many different user interfaces to be developed, you should always ensure that your SIS file can only be installed on the user interface for which it was designed. If you don't, you – or, more importantly, your end-users – may experience problems.

To enable you to do this, each product is assigned a unique ID which is used by the application installation mechanism to ensure that only compatible applications can be installed on the phone. Also, each platform version is assigned a unique ID. It is assumed that a platform version is compatible with all earlier platform versions, but not with any later versions. You can specify the platform information by including in the package file a line of the following form:

```
[0x101F6300], 1, 0, 0, {"UIQ30ProductID"}
```

where:

- `[0x101F6300]` is the product and platform version SID
- `1, 0, 0` is the major version, minor version and build numbers of the platform (not the application)
- `{"UIQ30ProductID"}` is a feature-identification string.

Not all products support this mechanism: for those that do, you can find exact details of the SIDs, feature-identification strings and version numbers to use in the appropriate product-specific SDK. On products that support this mechanism, the installation software refuses to install any SIS file that does not contain the correct platform information.

Generating the Final SIS file

When all the source files and application information have been specified in the PKG file, you can create an installable `OandX.sis` file in your current directory by issuing the command:

```
makesis OandX.pkg OandX.sis
```

You then need to sign the SIS file:

```
signsis OandX.sis OandX_Signed.sis myCert.cer myCert.key password
```

where:

- `OandX_Signed.sis` is the filename for the signed version of the application
- `myCert.cer` is the certificate filename
- `myCert.key` is the certificate's private key
- `password` is the password of the private key.

The SIS file is now ready. If you need to have the application Symbian Signed, read Section 9.5, otherwise go to Section 9.6 to see how to install your application on a mobile phone.

9.5 Overview of Symbian Signed

Symbian Signed was introduced against a background in which, although open phones offer revenue opportunities, there are also associated threats such as fear of viruses, billing storms (where the user is billed illegally or without their knowledge), etc. With an open environment comes responsibility and Symbian Signed is intended to support that responsibility, hence the need for a testing and certification programme that ensures that you know the source and provenance of an application. It also means that the application has been tested to meet certain industry-agreed minimum criteria. Applications that are signed must then be tamper-proof in order to be trusted.

Symbian Signed is a foundation programme on which other licensees, network operators and partners can build if they have a specific need. As a result there are mechanisms to support additional tests where required, for example where:

- access to prototype devices is needed
- the application is going to be a signature application shipped with phones or has operator requirements it must fulfill
- the application makes extensive use of the network and so the network impact of the application needs to be assessed.

To date, over 18 000 SIS files have been signed, with more and more network operators around the world insisting that applications are Symbian Signed before they even consider selling and distributing them via their portals.

Test Criteria

In order to get your application Symbian Signed, it must pass the Symbian Signed test criteria (available at ***www.symbiansigned.com***). We recommend that you make these tests part of your usual development lifecycle for your application to minimize the chances of failure when it comes to testing against the Symbian Signed test criteria.

The test criteria are split into six categories as outlined in Figure 9.2.

Figure 9.2 Symbian Signed test criteria categories

The Identity category refers to the ACS Publisher ID that you can obtain from TC TrustCenter. The other five categories focus on functionality aspects of the application. For example, Memory tests ensure that your application starts and closes gracefully in low-memory situations. Phone Operation tests ensure that your application does not stop the phone from undertaking its primary function – making phone calls. Various other tests such as installation and uninstallation, verification of the use of correct SIDs and so on are also part of the test criteria. These all combine to help make the use of Symbian OS devices a pleasant experience.

To get your application Symbian Signed, you develop the application, test it and send it to a test house. If it passes the test criteria, your application is Symbian Signed (see Figure 9.3).

Signed or Unsigned?

It is optional for developers to put applications written for versions of Symbian OS before v9 through the Symbian Signed process. The benefits

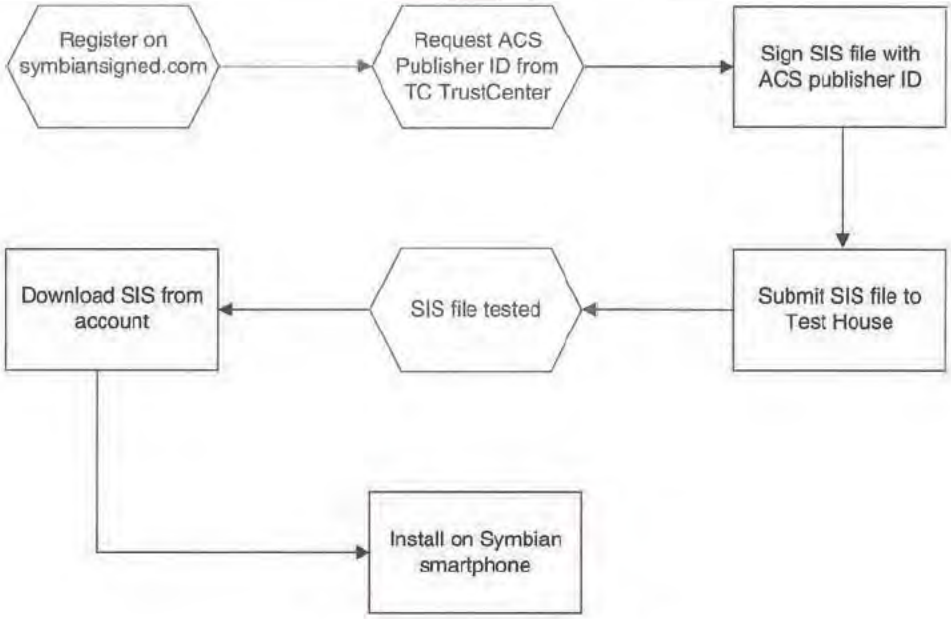


Figure 9.3 Application-signing process

of doing so are driven by market demand. Application robustness, quality and identification of the origin of the application are the primary drivers of this demand. Due to this, some phone manufacturers now insist that all applications delivered in a SIS file are Symbian Signed prior to distribution via their channels. Many network operators are also increasingly aware of the importance of testing and signing applications and require applications to be Symbian Signed prior to distribution via their portals.

If you are developing for Symbian OS v9, Symbian Signed becomes even more significant. This is because, if an application you are developing makes use of capabilities, the application must be Symbian Signed. If it is not Symbian Signed, a phone will not install the application. If an application doesn't use capabilities then it will install without signing – simple! Well actually, not quite that simple.

Some Symbian OS v9 phones may allow unsigned applications that use certain capabilities to be installed with no problems. An unsigned application, however, behaves slightly differently to a signed application when used on the phone. Let's say, for example, you create an application that makes use of the UserEnvironment capability (i.e., it uses Bluetooth APIs) and you do not have to get it signed as you know it will install on the phone you are developing for. When using this unsigned application on the phone, any functionality that utilizes Bluetooth results in a dialog box appearing on the phone's screen asking the user whether to allow the

application to use the Bluetooth functionality. This dialog appears every time the functionality is needed. A Symbian Signed application does not display these dialog boxes at all.

A further consideration is that some phone manufacturers may require these applications to be signed by the developer, sometimes called 'self-signed'. This means that a signing must take place using a certificate that you have developed using tools on the SDK (i.e. `makekeys.exe`) since, without this, the application will not install. It doesn't require visiting the Symbian Signed website or obtaining an ACS Publisher ID, so it is a small task in relation to the overall development effort, but it is something to bear in mind.

These policies are defined by the mobile phone manufacturers so it is recommended that you check with them first. For the sake of simplicity in this book, we have assumed that all applications you are developing for Symbian OS v9 phones will be Symbian Signed.

However, there's just one small catch – an application won't install on a phone until you've had it Symbian Signed, so how do you test it on the phone before submitting it for testing and signing? The solution is to use Developer Certificates.

Developer Certificates

Developer Certificates allow controlled capability access to specific phones. By controlled access we mean that the range of phones that can be used is limited to specific phones according to their IMEI and capabilities access is restricted to only the ones that are needed by the application. The level of access in terms of the number of IMEIs and capabilities required is determined according to identity of the developer (i.e. who you are) and whether the capability requires phone-manufacturer approval.

If you are happy to test your application on just one phone and it only uses the capabilities that are ticked in the first column of Table 9.1, then there is no need for an ACS Publisher ID. If you need to debug on more than one phone then you need an ACS Publisher ID. If you need to debug your application on more than 100 phones and/or you need any of the seven remaining capabilities, then you will need both an ACS Publisher ID and phone-manufacturer approval.

An ACS Publisher ID is a class 3 certificate, obtainable from organizations such as TC TrustCenter and costing US$200 per annum at the time of writing. It verifies that your organization exists and is bona fide.

All Developer Certificates are issued via the Symbian Signed web portal, once you have registered. If you need a developer certificate that requires phone-manufacturer approval, you simply need to follow the appropriate links on the site.

To request a developer certificate, you use a tool called DevCertRequest which can be downloaded from the Symbian Signed portal. You

Table 9.1 Developer certificate requirements

Number of IMEIs	**1**	**1–100**	**Phone Manufacturer Approved**
Identity Requirements	- Registration - IMEI	- Registration - IMEIs - ACS Publisher ID	- Registration - IMEIs - ACS Publisher ID - Phone Manufacturer Approved
Capability			
Local Services	√	√	√
Location	√	√	√
Network Services	√	√	√
Power Management	√	√	√
ProtServ	√	√	√
Read User Data	√	√	√
Surroundings DD	√	√	√
SW Event	√	√	√
User Environment	√	√	√
Write User Data	√	√	√
Read Device Data	√	√	√
Trusted UI	√	√	√
Write Device Data	√	√	√
All Files			√
CommDD			√
Disk Admin			√
DRM			√
MultimediaDD			√

Table 9.1 *(continued)*

Number of IMEIs	1	1–100	Phone Manufacturer Approved
Identity Requirements	- Registration - IMEI	- Registration - IMEIs - ACS Publisher ID	- Registration - IMEIs - ACS Publisher ID - Phone Manufacturer Approved
Capability			
Network Control			√
TCB			√

use this tool to enter the ACS Publisher ID, IMEI and capability information and create a Certificate Signing Request (CSR) file. This file is then uploaded to the Symbian Signed portal and the information contained in it is used to generate a developer certificate (CER).

The Signing Process

Now you have developed your application for phones based on Symbian OS v9, you must get it signed. This section describes how.

Preparation

You have written and tested your application successfully on the Symbian OS emulator. You have tested it on a Symbian OS v9 phone after signing it with a developer certificate and you are now confident that not only has it passed your own strenuous black-box and white-box testing, it has also passed all the Symbian Signed test criteria. What's the next stage?

ACS Publisher ID signing

Actually there isn't that much more left to do other than sign your application with your ACS Publisher ID and submit it to the website.

Run the `signsis.exe` tool to sign your SIS file, specifying your public ACS Publisher ID certificate, private key and password:

```
signsis OandX.sis OandX_Signed.sis myACS.cer myACS.key password
```

You should now have a SIS file that is signed and ready to be submitted for testing. You are almost there...

Submission

To submit the SIS file, visit the Symbian Signed website at ***www.symbian-signed.com*** and log in using the username and password you created when you first registered. Choose the Submit option and follow the steps to select the Test House and enter the required information (user details and application details). Finally submit the application and its supporting materials.

The following files are required to be included in the ZIP file you submit.

- the SIS file (signed with your ACS Publisher ID) to be tested by the test house
- the PKG file used to create the SIS file; this will be cross-referenced by the Test House to ensure correct target platform and specification
- a completed `Readme.txt` file, which should include any release notes and quick advice on how to use the application. Depending on the nature of the application, you might choose to supplement – or replace – this file with a user guide in PDF format.

After you have submitted your application, it is sent to the test house you selected. The test house verifies the validity of the ACS Publisher ID and the signature of the SIS file. If verification is successful, the test house examines your application and sends you a quote for the cost of the test run. You receive notification emails throughout this process. If you wish to query the quote you receive, you should contact the test house directly – additional details can be found on the website.

Once your application has successfully passed all of the tests conducted by the test house, the test house uploads your application to the Certificate Authority. The Certificate Authority removes the ACS Publisher ID, stores details of the application in its revocation database, re-signs the application against the Symbian root certificate and sends the signed application back to the test house. The test house then informs you that you can download your Symbian Signed application from the site, along with the test report.

Your application is ready for distribution and can display the Symbian Signed logo as shown in Figure 9.4.

Figure 9.4 Symbian Signed logo

9.5 Installing a SIS File

Once you have a working SIS file, it is easy to install it onto a Symbian OS v9 phone: all you need do is transfer it to the phone and launch the SIS file!

There are several transfer methods available – choose what is supported on your PC or phone and what is most convenient for you at the time:

- serial or USB connection – install the PC Connectivity software accompanying your phone and send files through a serial or a USB cable
- infrared – you need to enable infrared on both your PC and the phone and to run software that allows you to send files between two infrared ports; this is normally part of the PC Connectivity software accompanying your phone
- Bluetooth – you need to enable Bluetooth on both your PC and your phone
- email – you can send the SIS file to your phone as an email attachment if you activate the email account on your phone; as you will use the mobile phone network to send the file, you may incur costs
- Internet – you can download a SIS file straight from a website; you may incur costs when using the mobile phone network to connect to the website.

Once you have transferred the SIS file, it should appear on the phone as a 'received' file – simply select the icon to launch the file and begin the installation process. The program is then visible from the Application Launcher.

One of the benefits of installing applications using the SIS file format, rather than by copying files directly, is that you get an entry in this listing. This means that you can use it to remove the application; just tap on

Uninstall and select your application from the list of uninstallable programs. Then press Uninstall again to confirm. Go back to the Application Launcher and see that it is gone.

9.6 List of Capabilities

The unsigned-sandboxed and basic capabilities are:

- LocalServices, which grants access to local network services that usually do not incur a cost, such as Bluetooth
- NetworkServices, which grants access to remote network services that may incur a cost, such as text messages; this capability is granted on a one-shot basis if not previously authorized
- UserEnvironment, which grants access to live confidential information about the user and their immediate environment (e.g., audio, video or biometric data)
- ReadUserData, which grants read access to data that is confidential to the phone user (subject to confirmation)
- WriteUserData, which grants write access to data that is confidential to the phone user (subject to confirmation)

The extended capabilities essentially control access to services and data. They are defined at a general level, so that a server can present them to the end-user of a phone when installing an application, allowing the end-user some control over what an application may do.

- CommDD grants access to communications device drivers.
- MultimediaDD grants access to multimedia device drivers.
- PowerMgmt grants the right to power off unused peripherals, switch the phone into and out of standby state, and power the phone down.
- ReadDeviceData grants read access to confidential phone settings or data.
- WriteDeviceData grants write access to confidential settings that control the phone's behavior.
- DiskAdmin grants access to specific disk-administration operations.
- NetworkControl grants access or modification rights to network protocol controls.

Summary

In this chapter, we've looked at:

- key concepts in Platform Security
- authorizing applications
- implementing Platform Security in application and project files
- how to build an application for delivering to a phone
- how to get an application Symbian Signed.

21

Multimedia

This chapter describes the Multimedia Framework and the associated Multimedia APIs that are present in Symbian OS v9.1. These APIs can be used to play and record audio and video data, perform image processing and manipulation, and access any camera and radio tuner hardware present on a phone.

21.1 The Multimedia Framework

The Multimedia Framework (MMF) is a framework, based on ECOM plug-ins, that is used for playing back and recording audio and video clips. It allows phone manufacturers and third parties to add plug-ins to provide support for more audio and video formats. For the application programmer, it provides APIs that abstract away from the underlying hardware, thereby simplifying the code needed to record and play the supported formats. The streaming APIs, which bypass large parts of the MMF, provide a lower-level interface to allow streaming of audio data to and from the audio hardware.

The MMF is a lightweight framework that makes use of a worker thread to process the audio or video data. It makes extensive use of ECOM to allow for extensibility via plug-ins. Figure 21.1 shows its basic structure during audio- or video-clip playback or recording.

Applications use the client APIs to control the playback or recording of media. The underlying controller framework creates a new thread to process the media data and uses the Symbian client–server framework to communicate with this thread. The audio streaming APIs and the audio tone API interface directly to the MDA layer; they do not use a separate thread.

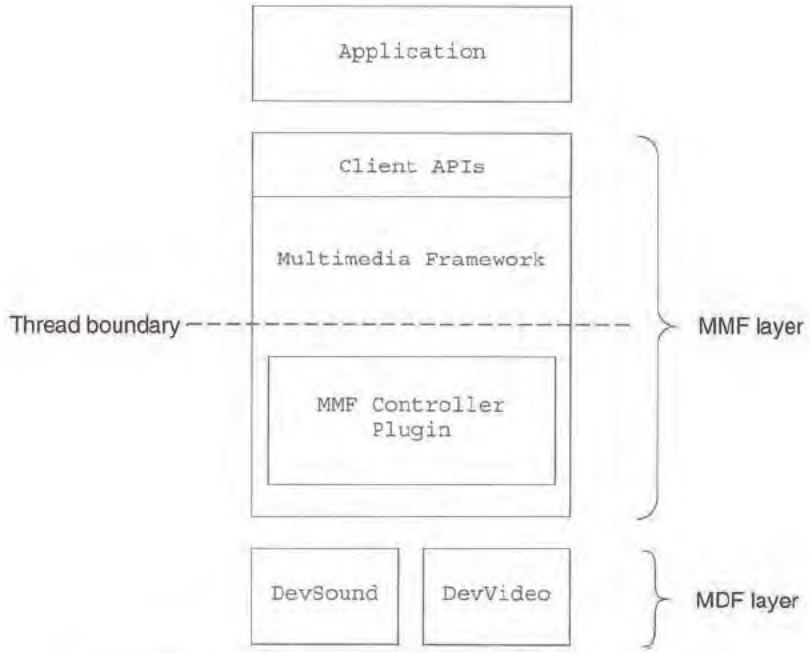


Figure 21.1 MMF configuration when using audio- or video-clip APIs

The main processing in the worker thread is done by a controller plug-in. In the case of playback, a controller plug-in is responsible for

- reading data from the source, typically either a file or a URL
- performing any processing required on the data
- writing the data to a sink, either the audio device or the screen.

In the case of recording, the data flow would be in the opposite direction.

The layer below the controller plug-in is the Media Device Framework, or MDF layer. This provides a hardware-abstraction layer that is tailored to each hardware platform on which the MMF runs. The MDF consists of DevSound, for audio input and output, and DevVideo for video.

The MDF also makes use of ECOM plug-ins that allow compressed audio or video formats to be written to, or read from, the MDF layer and allow the MMF to make use of any hardware acceleration present on a phone. These are known as hardware device plug-ins and are intended to be provided by device manufacturers to make use of specific hardware present on a phone.

The presence of these plug-ins means that it is often possible to read and write encoded audio formats directly to and from the MDF layer; most

Symbian OS v9.1 phones support adaptive multi-rate (AMR) encoding and decoding in the MDF layer and decoding of other popular formats such as MP3.

The audio policy forms a part of the MDF. It is responsible for allocating the available audio resources to applications that request them. Many phones are not capable of playing back more than one audio stream at a time and the audio policy must enforce this. The policy can pre-empt an audio playback or recording activity. This is essential if a phone call is received, for example, as any audio playback must usually be stopped before the call can begin.

The audio policy does not have a directly accessible API. You cannot directly request permission to play or record audio; the MDF layer does this for you when you attempt to play or record any such data.

Since the audio policy is part of the MDF layer, manufacturers customize it for each phone. It may therefore behave differently on different phones.

Support for Media Formats

Symbian OS v9.1 provides the 'Symbian Audio Controller' to support playback and recording of uncompressed Pulse Code Modulation (PCM) data. It provides playback and recording support for PCM data contained in Wav, Au and Raw files. It supports 8- and 16-bit PCM formats and the following PCM variants: IMA-ADPCM (Adaptive Differential PCM); Mu-law; A-law; and GSM 610. Conversion of these PCM variants to normal 16-bit PCM is performed in the MDF layer by a hardware device plug-in.

Phone manufacturers and third parties can supply support for other audio formats via MMF controller plug-ins. Most Symbian OS v9.1 phones provide support for playback of many popular formats such as MP3, advanced audio coding (AAC) and AMR. These compressed formats are generally much more useful due to decreased storage requirements. Since phones use the MMF for ring-signal playback and other system sounds, it is possible to use these controller plug-ins to play these sounds.

Symbian OS v9.1 does not provide any video plug-ins for playback or recording, hence these must be provided by phone manufacturers or third parties. Generally, video controller plug-ins are usually tightly integrated with the hardware on a particular phone in order to provide optimum performance and so are provided by the phone manufacturer.

Overview of APIs

The MMF client APIs can be broadly split into two types. APIs that deal with 'clips' are distinguished by the fact that no media data is passed

directly into the API: a filename or URL is passed in instead. The MMF controller is responsible for reading data from, or writing data to, the clip.

Streaming APIs do not deal with clips, but allow a client to pass audio data into the API, or read data directly from it. The word 'streaming' when applied to the MMF APIs is intended to mean that audio data is streamed into or out of the API. This should not be confused with streaming data over an Internet connection – something that can be done with the clip APIs, by passing in a URL.

The Observer pattern

For each API, there is a corresponding mixin class that must be implemented by a client of that API. When instantiating any of the MMF APIs, it is necessary to provide a reference to a class that implements the appropriate mixin class. This allows the MMF to notify the client of events that occur during the activity and of errors that occur.

You must have an active scheduler in your thread before using any of the MMF APIs, but this is not normally a problem since the application framework supplies every GUI application with its own active scheduler.

> Many of the MMF APIs feature the letters MDA in their name, for historic reasons. Prior to Symbian OS v7.0s, all multimedia services were provided by the Media Server, abbreviated to MDA. The names remain to provide backward source compatibility with applications written for these early versions of Symbian OS.

Audio clip APIs

The two audio clip APIs are `CMdaAudioPlayerUtility` and `CMdaAudioRecorderUtility`, which provide playback and recording, respectively, of audio clips. You can use any clip format for which a controller plug-in is present on the device.

`CMdaAudioPlayerUtility` provides functionality to play back clips, either in their entirety or from a defined playback window. Clips can be repositioned and paused during play back and the volume can be changed. You can also access any metadata present in a clip; for example, you may be able to access the artist name, album name and song title when playing back an MP3 file.

`CMdaAudioRecorderUtility` provides recording, editing and playback functionality of audio clips. It also allows you to access and edit metadata in an audio clip.

Note that `CMdaAudioPlayerUtility` provides a subset of the functionality of `CMdaAudioRecorderUtility`. However, it is slightly

easier to use and therefore recommended if you do not need to use the additional functionality of the latter.

Video clip APIs

The video clip APIs, `CVideoPlayerUtility` and `CVideoRecorderUtility`, provide playback and recording, respectively, of video clips. As with the audio clip utilities, they support any format for which a suitable controller plug-in is present in the device.

Audio tone API

The audio tone API, `CMdaAudioToneUtility`, provides playback of DTMF tone sequences, fixed tone sequences and custom tone sequences. The fixed tone sequences may be customized by a phone manufacturer, as may the format of the custom tone sequences.

Note that much of the MMF framework is not used when you use the audio tone API (see Figure 21.2). Specifically, a new thread is not created to process the audio data, so you must ensure that every active object `RunL()` in your thread finishes quickly to allow the tone utility to run in a timely fashion. Otherwise you may find that the tone playback breaks up or the hardware underflows, causing playback to terminate prematurely.

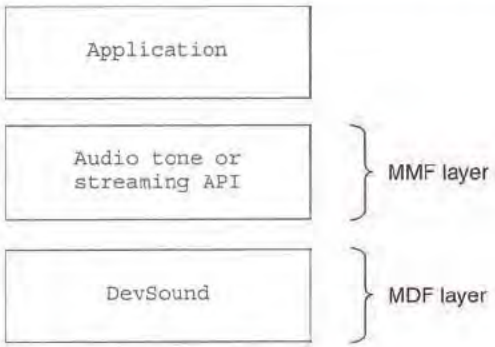


Figure 21.2 Structure of the audio tone and streaming APIs

Audio streaming APIs

The audio streaming APIs, `CMdaAudioInputStream` and `CMdaAudioOutputStream`, allow you to stream raw audio directly to and from the audio hardware.

`CMdaAudioOutputStream` is used where audio data is being received or generated as it is played, making the clip APIs unsuitable. Similarly, `CMdaAudioInputStream` is used in cases where you want to access the recorded data immediately.

There are no explicit video streaming APIs. However, the Camera API (see Section 21.3) can be used to achieve input streaming of video and direct screen access (see [Sales 2005, Chapter 11]) can be used to achieve output streaming.

DRM support

Symbian OS v9.1 has support for playback of DRM protected audio and video content by making use of the Content Access Framework (CAF). The abstract `TMMSource` class provides the basis of this support; the audio- and video-clip playback APIs contain overloads of their open methods that take a `TMMSource` argument. The following concrete subclasses of `TMMSource` are defined:

- `TMMFileHandleSource`, allowing a file handle to be provided
- `TMMFileSource`, allowing a file to be provided by name.

The `TMMSource` class and its subclasses also allow you to provide a DRM access intent, indicating what you intend to do with the DRM protected data, and a unique identifier to specify a content object within a file, if appropriate. The DRM intent that you specify may affect whether or not you are granted access to the content in question. These API methods work similarly for any non-protected DRM content.

In order to access any DRM protected content using the Symbian multimedia APIs, your process must have the DRM capability. For more information on the Content Access Framework, see the Symbian Developer Library.

Audio Clip Playback

Playback of audio files is achieved using the `CMdaAudioPlayerUtility` class, defined in `MdaAudioInputStream.h`. The class supports the playback of audio clips from files, descriptors or URLs. Please note that in order for playback from a URL to be supported, the controller plug-in in use must support this; many do not.

You do not need any capabilities to play audio using these APIs. However, if you want to play audio with a high priority, you need the MultimediaDD capability. This policy is intended to stop untrusted applications holding the audio resource, which could interfere with system functions such as ring signal playback or setting up a call. If you do not have MultimediaDD and you request a high audio priority, it may be silently downgraded. The exact behavior depends on the audio policy, which differs between phone manufacturers and models.

In order to use the class, you must implement the observer mixin class `MMdaAudioPlayerCallback`. When instantiating `CMdaAudioClipUtility`, you have to provide a reference to a class implementing this mixin; typically, the client itself does so.

```
class CAudioPlayer : public CBase, public MMdaAudioPlayerCallback
  {
  ...
  // from MMdaAudioPlayerCallback:
  virtual void MapcInitComplete(TInt aError,
      const TTimeIntervalMicroSeconds& aDuration);
  virtual void MapcPlayComplete(TInt aError);
  };

void CAudioPlayer::ConstructL()
  {
  iPlayer = CMdaAudioPlayerUtility::NewL(*this);
  }
```

To play back an audio clip from a file, you must first open the file using the `OpenFileL()` method. This locates a suitable controller plug-in to play the file you have given and calls back asynchronously to the `MapcInitComplete()` method. An error code is given; if this is `KErrNone`, you are free to start calling other methods in the utility or to initiate playback. If the error you receive is `KErrNotSupported`, it means that no suitable MMF controller plug-in could be found to open the file you provided. Other errors are possible, such as `KErrNotFound` if the specified file does not exist.

```
void CAudioPlayer::PlayFileL(const TDesC& aFileName)
  {
  // close the player in case it has previously been
  // opened or if we are already playing a clip.
  iPlayer->Close();
  iPlayer->OpenFileL(aFileName);
  }
```

To start playback, you call `Play()`. If you simply want to play the file, this can be done from the callback method.

```
void CAudioPlayer::MapcInitComplete(TInt aError,
    const TTimeIntervalMicroSeconds& /*aDuration*/)
  {
  if (aError == KErrNone)
    {
    iPlayer->Play();
    }
  else
    {
    // handle the error
    }
  }
```

In a slightly more complex application, you may not want to start playback immediately but simply to update your state, to indicate that further operations are now possible.

```
void CAudioPlayEngine::MapcInitComplete(TInt aError,
     const TTimeIntervalMicroSeconds& /*aDuration*/)
  {
  if (aError == KErrNone)
    {
    iState = EOpen;
    // notify the UI of our new state
    iObserver.MapeoStateChanged();
    }
  else
    {
    // notify the UI of the error that has occurred
    iObserver.MapeoError(aError);
    }
  }
```

Playback can then be initiated at a later time. Now this clip is open, we can access the metadata entries and other information associated with it.

The audio player utility can pause a clip during playback, by calling the `Pause()` method. When a clip is paused, the current playback position is preserved; we can also reposition the clip while it is paused, something that we cannot do while it is stopped. Use the `SetPosition()` method to do this.

Once playback of the clip has completed, we receive another callback:

```
void CAudioPlayEngine::MapcPlayComplete(TInt aError)
  {
  if (aError == KErrNone)
    {
    iState = EOpen;
    // notify the UI
    iObserver.MapeoStateChanged();
    }
  else
    {
    // handle the error
    }
  }
```

An error of `KErrNone` indicates that playback ended because we reached the end of the clip – this is the usual case. Other error codes are possible, for example if your playback is pre-empted by another audio playback.

To stop playback before the end of the clip is reached, simply call `Stop()`. This resets the clip but does not close it, so playback can be restarted simply by calling `Play()` again. Note that if you call `Stop()`, you do not receive a `MapcPlayComplete()` callback, so you must update any internal state.

```
void CAudioPlayEngine::Stop()
  {
  // it only makes sense to stop the clip if it is currently
  // playing or paused.
  __ASSERT_ALWAYS((iState == EPlaying) || (iState == EPaused),
                        Panic(EAudioPlayerEngineWrongState));
  iPlayer->Stop();
  iState = EOpen;
  // notify the UI
  iObserver.MapeoStateChanged();
  }
```

If you have previously set a play window by calling the `SetPlayWindow()` method, a call to `Stop()` resets the clip position to the start of the play window, not the start of the clip.

Notifications of Audio Resource Availability

When you are playing audio, you may be pre-empted by the system or by another audio playback. When this happens, it is possible to receive a notification from the system when the audio resource has become available again so that playback can be resumed.

> The behavior of the resource notifications depends on the internal audio policy implementation and therefore it varies from phone to phone. For example, on the Sony Ericsson P990 family of phones, notifications are only provided when the application is pre-empted by certain system events, such as a phone call, not if it is interrupted by another application such as the music player.

To do this, your class must derive from the `MMMFAudioResourceNotificationCallback` mixin, which contains the single method `MarncResourceAvailable()`. Then, register for the notification when you start playback.

```
void CAudioPlayEngine::Play()
  {
  ...
  iPlayer->Play();

  TInt err = iPlayer->RegisterAudioResourceNotification(*this,
                    KMMFEventCategoryAudioResourceAvailable);
  if ((err != KErrAlreadyExists) && (err != KErrNone))
    {
    // handle the error
    ...
    }
  ...
  }
```

The UID `KMMFEventCategoryAudioResourceAvailable` is used to indicate that it is audio-resource events that you are interested in. When your audio playback is pre-empted, you receive an error via the usual callback:

```
void CAudioPlayEngine::MapcPlayComplete(TInt aError)
  {
  if (aError == KErrNone)
    {
    ...
    }
  else if ((aError == KErrInUse) || (aError == KErrAccessDenied))
    {
    // Playback has been pre-empted by another audio activity,
    // or playback failed to start due to an ongoing audio
    // activity. Pause the clip so that we can resume when the
    // audio resource is available again.
    iState = EPaused;
    iObserver.MapeoStateChanged();
    }
  else
    {
    ...
    }
  }
```

In this example, the engine enters the paused state. Then, when the resource becomes available again, we receive another callback:

```
void CAudioPlayEngine::MarncResourceAvailable(TUid aNotificationEventId,
                                              const TDesC8& aNotificationData)
  {
  if ((aNotificationEventId == KMMFEventCategoryAudioResourceAvailable)
                                                  && (iState == EPaused)
                        && (aNotificationData.Size() >= sizeof(TInt64)))
  // aNotificationData may have 0 size if some error occurred.
    {
    // retrieve the resume position from the notification data:
    TPckgBuf<TInt64> resumePosition;
    resumePosition.Copy(aNotificationData);
    // seek to the appropriate position
    iPlayer->SetPosition(resumePosition());
    // and resume playing
    iPlayer->Play();
    iState = EPlaying;
    iObserver.MapeoStateChanged();
    }
  }
```

The notification data provided with the callback tells us what position in the clip we were at when the interruption occurred. We retrieve this

information and perform a seek before restarting playback to ensure that we resume from the correct position.

Note that although the example code here is given for `CMdaAudio-PlayerUtility`, identical functionality is provided in all the audio APIs.

Audio Clip Recording

Recording of audio clips is achieved using class `CMdaAudioRecorder-Utility` which is defined in `MdaAudioSampleEditor.h`. You can use this class to record audio in any format for which a suitable controller plug-in exists. The class contains a superset of the functionality of `CMdaAudioPlayerUtility` and so is often used for playback of clips. Note however that the class does not contain a `Pause()` method, unlike the player utility.

To perform audio recording, you need the UserEnvironment capability. High audio priorities are policed in the same way as with the player utility, using the MultimediaDD capability.

To use the audio recorder utility, you must derive from mixin class `MMdaObjectStateChangeObserver` which defines a single method:

```
class MMdaObjectStateChangeObserver
  {
public:
  virtual void MoscoStateChangeEvent(CBase* aObject,
      TInt aPreviousState, TInt aCurrentState, TInt aErrorCode) = 0;
  };
```

`MoscoStateChangeEvent` is called every time the utility changes state; the `aPreviousState` and `aCurrentState` parameters have values corresponding to `CMdaAudioClipUtility::TState`:

```
enum TState
  {
  ENotReady = 0,
  EOpen,
  EPlaying,
  ERecording
  };
```

When using the utility for audio playback, the usage is very similar to the player utility so it is not discussed again here. One point worth mentioning however is that you receive a callback to `MoscoStateChange` event following a call to `PlayL()` indicating that playback has started; no equivalent callback exists when using the player utility.

To perform audio recording, a controller is opened by the recorder utility. You can record to a descriptor or to a file; it is usual to record to a file due to the memory requirements of using a descriptor.

When you open the recorder utility, you must indicate which controller to use for the recording. You can do this either by specifying explicitly the UID of the controller to use or by letting the controller framework select the controller automatically based on the extension of the filename given. For example, if you open a file with the extension `.wav`, the Symbian audio controller is selected automatically.

Selecting a controller by UID

As stated earlier, all controller plug-ins are ECOM plug-ins. Hence we can use ECOM to query the controllers present on a phone and the formats that they support. Symbian provides some helper classes to make this job easier:

- `CMMFControllerImplementationInformation`
- `CMMFFormatImplementationInformation`
- `CMMFControllerPluginSelectionParameters`
- `CMMFFormatSelectionParameters`.

You can use these classes to query the controllers present on a phone and find out what media formats they support. They are all defined in the header file `mmf\common\mmfcontrollerpluginresolver.h`.

> The `mmfcontrollerframework.lib` library that you need to link against when using these classes is not present in the S60 3rd Edition SDK from Nokia. You must download an SDK extension plug-in separately before you can use the classes discussed here.

First, create an instance of `CMMFControllerPluginSelection-Parameters` and indicate what sort of controller you are interested in. Here, we specify that we are only interested in controllers that support audio:

```
CMMFControllerPluginSelectionParameters* controllerSelection =
            CMMFControllerPluginSelectionParameters::NewLC();

RArray<TUid> mediaIds;
CleanupClosePushL(mediaIds);

mediaIds.AppendL(KUidMediaTypeAudio);
controllerSelection->SetMediaIdsL(mediaIds,
    CMMFPluginSelectionParameters::EAllowOnlySuppliedMediaIds);
```

If we wanted to find out about video controllers, we could have specified `KUidMediaTypeVideo` instead of `KUidMediaTypeAudio`. In this example, we have specified that we are interested in controllers that support only audio, by giving the value `EAllowOnlySuppliedMediaIds`. Other options here are:

- `ENoMediaIdMatch` – do not perform matching on media ID
- `EAllowOtherMediaIds` – allow controllers that support the specified media ID in addition to others. This would show controllers supporting audio and video.

The next step is to specify the media formats we are interested in:

```
CMMFFormatSelectionParameters* formatSelect =
      CMMFFormatSelectionParameters::NewLC();

controllerSelection-> SetRequiredRecordFormatSupportL(*formatSelect);
```

We could have chosen a specific format at this point by calling `SetMatchToMimeTypeL()` and giving the MIME type of the audio format we are interested in. In this example, we list all controllers that support audio recording.

The final step is to retrieve the list of controllers that meet the criteria we have specified.

```
RMMFControllerImplInfoArray iControllers;
...
controllerSelection->ListImplementationsL(iControllers);
```

The array `iControllers` now contains a list of controllers that support audio recording, each one represented by an instance of `CMMFControllerImplementationInformation`.

Each controller can support any number of audio playback or recording formats. The list of formats supported can be accessed using the `RecordFormats()` and `PlayFormats()` methods. Each format is represented by an instance of `CMMFFormatImplementationInformation`.

Recording the sound

Once we have chosen the format we wish to record in, the controller can be opened by the utility. If we want to let the MMF choose a controller and format based on the extension of the filename we give, simply pass in this filename.

```
iRecorder->OpenFileL(fileName);
```

If we are specifying a format to record in by controller and format UID, we pass these in to the `OpenFileL()` method.

```
void CAudioRecordEngine::OpenL(const TDesC& aRecordFileName,
                        TRecordFormatParameters aParameters)
  {
  // in case it has previously been used:
  iRecorder->Close();
  iState = ENotReady;

  // Append a suitable extension to the filename we are given,
  const CDesC8Array& extensions =
      aParameters.iFormat->SupportedFileExtensions();
  if (extensions.Count()>0)
    {
    // just use the first one
    InitializeFileNameL(aRecordFileName, extensions[0]);
    }

  iRecorder->OpenFileL(iFileName,
     aParameters.iController->Uid(),  // controller UID
                          KNullUid,  // play controller UID (unused here)
        aParameters.iFormat->Uid()  // record format UID
                                 );
  }
```

In this example, we are appending a suitable file extension to the filename given. This step is not necessary but adds completeness.

As with the player utility, we expect a callback after we have called `OpenFileL()`.

```
void CAudioRecordEngine::MoscoStateChangeEvent(CBase* /*aObject*/,
             TInt /*aPreviousState*/, TInt aCurrentState, TInt aErrorCode)
  {
  switch (iState)
    {
    case ENotReady:
      // The OpenFileL call is complete
      if (aErrorCode == KErrNone)
        {
        iState = ERecordReady;
        // notify the UI
        iObserver.MareoStateChanged();
        }
      else
        {
        // notify the UI
       iObserver.MareoError(aErrorCode);
       }
    break;
    ...
    }
  }
```

The `aObject` parameter to this method indicates which object the state change relates to; in this example, since we only use a single recorder utility, we know that `aObject == iRecorder`.

If we want to set the parameters for the recording, now is the time to do it. The recorder utilities provides a number of methods allowing you to change the number of channels, bitrate and other settings. You should query which values are supported before setting them, for example by using the `GetSupportedBitRatesL()` method before calling `SetDestinationBitRateL()`.

Now the recorder is ready to begin recording.

```
void CAudioRecordEngine::RecordL()
  {
  __ASSERT_ALWAYS(iState == ERecordReady,
      Panic(EAudioRecorderEngineWrongState));
  iRecorder->RecordL();
  }
```

We receive another callback to `MoscoStateChangeEvent()` when the recording starts successfully; at this point, we move the engine into state `ERecording`. The recording continues until we stop it or an error occurs, such as if we run out of disk space or the descriptor we are recording into becomes full. If we call `Stop()`, we do not receive a callback notifying us of the state change.

Playing back the recording

The recorder utility is capable of playing audio clips as well as recording them. Once the clip is recorded, we can play it back. This can be done simply by making a call to `PlayL()` after the call to `Stop()`; however this requires that the same controller supports recording and playback of the format in use, since only one controller is used at a time. While this is true for the Symbian audio controller, it is often not the case for controllers that record in compressed formats.

So, to play the clip reliably, we need to close and reopen the controller. This gives the MMF the chance to load a new controller if required. In this example, we do it immediately after the recording finishes.

```
void CAudioRecordEngine::Stop()
  {
  __ASSERT_ALWAYS(iState != ENotReady,
      Panic(EAudioRecorderEngineWrongState));
  iRecorder->Stop();
  switch (iState)
    {
    ...
    case ERecording:
      OpenControllerForPlay();
```

```
      break;
    }
    iObserver.MareoStateChanged();
  }
...
void CAudioRecordEngine::OpenControllerForPlay()
  {
  iRecorder->Close();
  TRAPD(err, iRecorder->OpenFileL(iFileName));
  if (err == KErrNone)
    {
    iState = EPlayReady;
    // don't notify our observer yet -  wait until we get another
    // state change event notifying us that the player is now
    // ready to go
    }
  else
    {
    iState = ENotReady;
    iObserver.MareoError(err);
    }
  }
```

We then receive a callback to `MoscoStateChangeEvent()`, indicating that the utility is ready to begin playback.

Audio Streaming

The audio streaming APIs allow you to provide or receive audio data while it is being played or recorded respectively. This allows you to generate audio data for playback while it is being played and process audio data while it is being recorded.

When streaming, you are using a client utility API that interfaces directly to the MDF layer. Hence, no MMF controller plug-in is used and the subthread associated with controllers is not used either. The streaming takes place directly in the application's thread. This makes it important to ensure that the `RunL()` implementations for all the active objects in your application are kept short to avoid interrupting the streaming – see Chapter 6. If your application needs to perform processor-intensive tasks, it may be advisable to delegate usage of the streaming API to a separate thread.

Since the streaming APIs interface directly to the MDF layer, the audio formats that are supported by the streaming APIs are determined by the audio formats supported in each phone. All Symbian OS v9.1 phones support PCM formats. Many also support compressed formats such as AMR and MP3. To determine which formats a particular phone supports, you must use the `CMMFDevSound` interface.

Output streaming

Audio output streaming is performed using class `CMdaAudioOutputStream`. You must implement the corresponding mixin class `MMdaAudioOutputStreamCallback` in order to use it. The capability restrictions associated with this class are identical to the audio clip playback API.

As with the audio clip utilities, you must initialize the streaming utility after it has been instantiated.

```
iStream = CMdaAudioOutputStream::NewL(*this);
...
// open the audio stream. We get a callback to
// MaoscOpenComplete indicating success or failure.
iStream->Open(NULL);
```

The `TMdaPackage*` parameter is present only for historical reasons and is not used in Symbian OS v9.1. You should therefore just specify `NULL`.

Once the stream has been opened, you can configure it for the audio type that you intend to stream to it. In this example, we stream PCM data but most phones also support some compressed formats. In this example, the stream is configured from inside the callback method; this could be done any time between receiving the callback and starting streaming.

```
void CAudioStreamOutEngine::MaoscOpenComplete(TInt aError)
  {
  if (aError == KErrNone)
    {
    // set the data type to 16-bit PCM
    TRAP(aError, iStream->SetDataTypeL(KMMFFourCCCodePCM16));
    }
  if (aError == KErrNone)
    {
    // setup the sample rate and number of channels
    TRAP(aError, iStream->SetAudioPropertiesL(iSampleRateCaps,
                                              iChannelCaps));
    }
  if (aError == KErrNone)
    {
    // setup complete
    }
  else
    {
    // setup failed - handle the error
    }
  }
```

The `SetDataTypeL()` method is given a four-character code specifying the format of the audio data to be streaming. The four-character codes are specified in `mmffourcc.h`.

The parameters to the `SetAudioPropertiesL()` method are values from enumeration `TMdaAudioDataSettings::TAudioCaps`. Valid values for the sample rate setting are of the form `ESampleRateXxxxHz`. The number-of-channels setting should be either `EChannelsMono` or `EChannelsStereo`.

Once the stream has been set up successfully, streaming is started simply by writing some data to the stream:

```
iStream->WriteL(buffer);
```

`Buffer` is a descriptor filled with audio data of the appropriate type. Once you have written a buffer of audio data to the stream, you should not modify the data in the buffer, nor the buffer itself, until it is returned to you via the callback `MaoscBufferCopied()`.

You must supply a sufficiently frequent stream of data to the output stream to prevent it running out of data, known as an underflow. This can be done simply by writing a large number of buffers to the stream all in one go. They are queued and played in turn. However, it is usual to use two buffers – this allows for one buffer to be filled up while the other is being played by the audio stream. If you use a single buffer, the stream may run out of data while you are refilling it, leading to an underflow and a break in the audio. Using more than two buffers may reduce the risk of underflow when the system is heavily loaded, but clearly has a larger memory overhead.

When a buffer is returned to you, you should refill it and write it back to the utility if you wish to continue streaming.

```
void CAudioStreamOutEngine::MaoscBufferCopied(TInt aError,
                                              const TDesC8& aBuffer)
  {
  // the audio device has finished with the data
  // in a buffer. Refill it, and send it back to
  // the device.
  TInt bufferNum = KErrNotFound;
  if (aError == KErrNone)
    {
    // find the buffer that has just been emptied so that we
    // can refill it.
    for (TInt i=0; (i<KBufferCount) && (bufferNum == KErrNotFound); ++i)
      {
      if (&aBuffer == &iBuffer[i])
        {
        bufferNum = i;
        }
      }
      // we can be sure here that bufferNum != KErrNotFound
      // because the stream only passes back buffers
      // that we have given it.
      aError = FillThisBuffer(iBuffer[bufferNum]);
    }
```

```
if (aError == KErrNone)
  {
  // write the new data to the output stream.
  iStream->WriteL(iBuffer[bufferNum]);
  }
  // we get a KErrAbort for each queued buffer
  // after a call to Stop. Ignore these errors.
else if (aError != KErrAbort)
  {
  iStream->Stop();
  // handle the error
  }
}
```

In this example, we have a small array of buffers. We find the buffer that has just been emptied in this array, refill it and then write it back to the stream.

To stop streaming, you have two options. Either you can just stop writing buffers back to the stream when they are returned to you – in this case, you get a callback to `MaoscPlayComplete()` with an error of `KErrUnderflow` when all the data you have provided has been played completely. This method ensures that all the data is played fully.

If you want to stop the streaming immediately, you should call `Stop()` on the audio stream.

```
iStream->Stop();
```

In this case, you should expect the following callbacks:

- `MaoscBufferCopied()` with an error of `KErrAbort` for each buffer that you have written to the stream but which has not yet been returned
- `MaoscPlayComplete()` with an error of `KErrCancel`.

These callbacks happen synchronously from within the call to `Stop()`. By the time this call returns, you can be sure that streaming has terminated.

Choosing buffer sizes

When using the audio streaming APIs, you should take a little time to decide what size buffers you should use and how many of them. Some compressed formats dictate a fixed buffer size; for PCM, any size buffer can be used.

- If you want low-latency sound, such that the sound heard can react quickly (for example, to events on screen), you should use a smaller

buffer size. This carries the risk of underflow when the system is heavily loaded.

- If the reliability of the audio is important, larger buffers should be used. This reduces the risk of underflow occurring during the stream.

If you need reliable and low latency audio, you should consider having a dedicated high-priority thread to process the audio.

Some extra overheads are introduced when using the audio streaming utilities. The data is copied from the buffers in which you provide it into another buffer that is owned by the audio device. This introduces extra processing overheads and increases the latency slightly. For performance-critical applications, you should consider interfacing directly to the DevSound interface, which avoids these overheads.

Input streaming

The audio input-streaming interface, `CMdaAudioInputStream`, is similar in structure and usage patterns to the output streaming interface. To use it, you must provide a class that derives from `MMdaAudioInputStreamCallback`. As with audio clip recording, you need the UserEnvironment capability to perform audio input streaming.

As with the output-streaming API, you must perform the following sequence of events to start input streaming:

1. Instantiate the class `CMdaAudioInputStream`.
2. Initialize it by calling `iStream->Open(NULL)`.
3. Wait for a callback to `MaiscOpenComplete()`.
4. Configure the stream using the `SetDataTypeL()` and `SetAudioPropertiesL()` methods.
5. Start streaming by making a call to `iStream->ReadL()`.

You should provide an empty descriptor to the `ReadL()` method. As with the output stream, to increase the reliability of the stream and reduce the risk of an overflow, you may want to issue more than one call to `ReadL()` when the streaming starts.

When a buffer full of audio data has been recorded, it is provided to you via the `MaiscBufferCopied()` callback. You should then process the new data in the buffer and issue another read request if you wish to continue streaming.

To stop the input streaming, simply call `iStream->Stop()`. After this call, you receive a callback to `MaiscBufferCopied()` for each buffer that you had previously given to the utility via a call to `ReadL()`. Note that these buffers may not all be empty: any data that was recorded

between the last callback to `MaiscBufferCopied()` and the call you make to `Stop()` is provided to you at this time, to ensure that you do not miss any audio that was recorded immediately prior to when you stopped the stream.

The same tradeoffs that apply to output streaming in terms of buffer sizes also apply to input streaming. If you want access to the data very soon after it was recorded, you should use smaller buffers. For maximum reliability, use larger buffers.

Audio Tone Playback

Symbian OS v9.1 supports playback of basic, predefined tone sequences and DTMF strings. The class `CMdaAudioToneUtility` provides this support; the tones are generated as they are being played back and consist of one or more sine waves mixed together.

The predefined tone sequences that can be played back using the class are known as fixed sequences. They are defined by the phone manufacturer and cannot be changed. Each fixed sequence has a human-readable name which can be retrieved using the `FixedSequenceCount()` and `FixedSequenceName()` methods.

To use this class, you should implement the `MMdaAudioToneObserver` mixin. The usage is similar to the audio clip utility.

Once the clip has been instantiated, it must be initialized. Here, we initialize it to play a DTMF string contained in a descriptor:

```
iUtility->PrepareToPlayDTMFString(aDtmfString);
```

This results in a callback to `MatoPrepareComplete()`; playback can then be started.

```
void CAudioToneEngine::MatoPrepareComplete(TInt aError)
  {
  if (aError != KErrNone)
    {
    // handle the error
    }
  else
    {
    iUtility->Play();
    }
  }
```

The `CancelPrepare()` and `CancelPlay()` methods can be used to stop the utility during the initialization phase and during playback respectively. `CancelPlay()` is analogous to the `Stop()` methods seen in the other utility classes. When playback of the tone completes, you get a callback to method `MatoPlayComplete()`.

As with the audio streaming APIs, the audio tone utility interfaces directly to the MDF layer and does not use a subthread to process audio data. The utility must be able to receive timely `RunL()` calls during playback of the tone if it is to play successfully. In order to avoid an audio underflow during tone playback, you should ensure that every active object `RunL()` in your application completes promptly.

Video Playback

Video playback in Symbian OS is similar in principle to audio playback. An MMF controller plug-in is used, along with its associated subthread. Symbian OS does not provide any concrete video playback support, but video controller plug-ins are provided by phone manufacturers to support video playback.

No capabilities are required to play video. However, if you are playing a video with an audio track, the restrictions regarding the audio priority are the same as those that apply to the audio clip playback API.

Video playback is performed using the `CVideoPlayerUtility` class. You need to provide a class that derives from `MVideoPlayer-UtilityObserver` in order to use it.

When you instantiate this class, you must provide a number of parameters to allow the video playback to be configured:

```
iPlayer = CVideoPlayerUtility::NewL(*this,
                        EMdaPriorityNormal,
                EMdaPriorityPreferenceNone,
                     iCoeEnv->WsSession(),  // window-server session
                *iCoeEnv->ScreenDevice(),  // software screen device
                                 *window,  // display window
                             drawingRect,  // display-window dimensions
                          drawingRect );   // clipping area
```

The first parameter is the observer class deriving from `MVideo-PlayerUtilityObserver`; the next two are the priority and priority preference that are used when accessing the audio device, if the video in question has an audio track.

The window server session, software screen device and display window are the parameters required to set up direct screen access, which is used to render the video. See [Sales 2005, Chapter 11] for more information on direct screen access. The display window dimensions specify the rectangle on the screen in which the video is to be rendered. Note that the rectangle is relative to the origin of the entire screen – not the origin of the control or window. The clipping area – also relative to the entire screen – specifies the region of the video that should be rendered. It is possible to display just a portion of a video.

Once the utility is instantiated, we can then open a video file and await the callback:

```
iPlayer->OpenFileL(aFileName);
...

void CVideoPlayEngine::MvpuoOpenComplete(TInt aError)
  {
  if (aError == KErrNone)
    {
    iPlayer->Prepare();
    }
  else
    {
    // handle the error
    }
  }
```

The next phase that must be carried out before the video can be played is to call `Prepare()` on the utility. This allows the video controller plug-in to finalize its configuration for video playback. In this example, it is done immediately; it could be done any time after the open complete callback is received. Once the `Prepare()` call has been completed, we receive another callback.

```
void CVideoPlayEngine::MvpuoPrepareComplete(TInt aError)
  {
  TSize videoSize;
  if (aError == KErrNone)
    {
    TRAP(aError, iPlayer->VideoFrameSizeL(videoSize));
    }
  TSize controlSize = iCoeControl->Size();
  if (aError == KErrNone)
    {
    if ((controlSize.iHeight < videoSize.iHeight)
        || (controlSize.iWidth < videoSize.iWidth))
      {
      // video is bigger than the control so we need to scale it
      TReal32 scalePercentX = controlSize.iHeight * 100.0
                                / videoSize.iHeight;
      TReal32 scalePercentY = controlSize.iWidth * 100.0
                                / videoSize.iWidth;
      // scale both directions by the same amount so that
      // the aspect ratio is preserved:
      TReal32 scalePercent = Min(scalePercentX, scalePercentY);
      TRAP(aError, iPlayer->SetScaleFactorL(
                              scalePercent,
                              scalePercent,
                          // use anti-alias filtering if available
                                    ETrue));
      }
    }
  if (aError == KErrNone)
    {
```

```
    // Success! play the video
    }
  else
    {
    // handle the error
    }
  }
```

Once the callback indicating the completion of the `Prepare()` call is received, we can query the dimensions of the video clip that we have just opened. We then work out by what factor it should be scaled so that it fits in the area into which we are rendering it.

Now, to start the playback of the video, we simply call `Play()`:

```
iPlayer->Play();
```

We then receive another callback when the clip ends naturally, or when it is interrupted by the system. This could happen, for example, if the audio resource is pre-empted by another audio client.

Any metadata present in the video clip can be accessed in exactly the same way as with the audio-clip player utility.

Video Recording

Video recording can be achieved using the `CVideoRecorderUtility` class which is defined in `VideoRecorder.h`. It is similar in principle to audio recording, but slightly more complex. The callback class that you must implement to use `CVideoRecorder` is `MVideoRecorderUtilityObserver`. Video is recorded from the built-in camera on the phone.

Since the UserEnvironment capability is required to record audio and to use the camera, this capability is required in order to use the video recording API.

When recording video, you must select a format for the recorded data. Unlike the audio-recorder utility, you cannot let the MMF select the controller plug-in automatically based on the extension of a filename that you provide; instead, you must supply explicit code to find the UID of the controller to use.

```
CMMFControllerPluginSelectionParameters* controllerSelection =
            CMMFControllerPluginSelectionParameters::NewLC();

RArray<TUid> mediaIds;
CleanupClosePushL(mediaIds);

mediaIds.AppendL(KUidMediaTypeVideo);
controllerSelection->SetMediaIdsL(mediaIds,
    CMMFPluginSelectionParameters::EAllowOtherMediaIds);
```

Here, we indicate that we are interested in video controllers by specifying `KUidMediaTypeVideo`. When calling `SetMediaIdsL()`, we must specify `EAllowOtherMediaIds`, as most video controllers also support audio. Querying of the associated formats can now proceed in the same way as when querying audio controllers.

Once you have selected the UID of the controller and format that you wish to use for the recording, you can open the recorder utility using `OpenFileL()`, `OpenDesL()` or `OpenUrlL()`. Each of these methods requires the UID of a controller and format. You are also required to supply the handle to a camera from which to record. The handle can be obtained using the `CCamera::Handle()` method; see Section 21.3 for more information on `CCamera`.

After the call to an `OpenXxx()` method, you receive a callback to `MvruoOpenComplete()` indicating success or otherwise. Once the utility has been successfully opened, you can perform any configuration necessary. For this purpose, `CVideoRecorderUtility` provides a number of `SetXxx()` methods that can be called at this time:

- call `SetAudioEnabledL(ETrue)` if you wish the recorded video clip to contain an audio track
- use `SetAudioTypeL()` and `SetVideoTypeL()` to configure the specific audio and video encoding to use. You should check the supported types beforehand, using `GetSupportedAudioTypesL()` and `GetSupportedVideoTypesL()`.

Due to the complexity of video controllers, plug-in suppliers may also provide a number of custom commands to configure the controller.

Once the video recorder is configured, you must prepare it by calling `Prepare()`. This is another asynchronous method and result in a callback to `MvruoPrepareComplete()`. Once this callback has been received with an error of `KErrNone`, the recorder utility is ready to start the recording itself.

To begin the video recording, simply call `Record()`, which starts the capturing and encoding of video frames from the camera. Recording continues until you stop it, the recording destination becomes full or some other error occurs.

To stop the recording yourself, call `Stop()`. In this case, you do not receive any callback. If recording stops for some other reason, you receive a callback to the `MvruoRecordComplete()` method.

Other asynchronous events from the controller plug-in may be delivered to you via the `MvruoEvent()` callback method. These events are defined by the video controller plug-in; to interpret them you need to check the details of the controller plug-in in use.

Using DevSound

The DevSound API, `CMMFDevSound`, forms the interface audio component to the Media Device Framework (MDF) layer. It is the lowest-level audio API available on Symbian OS phones.

The DevSound implementation is replaced by phone manufacturers to interface to the audio hardware present on each phone; the interface remains unchanged between all Symbian OS v9.1 phones and should behave in the same manner.

The DevSound API can be used to query the capabilities of the audio hardware on any given phone, to determine which audio formats are supported. This determines which formats you can use with the audio input and output streams described earlier. You can also use it to perform high-performance streaming to and from the audio hardware.

Querying MDF capabilities

The `GetSupportedOutputDataTypesL()` and `GetSupportedInputDataTypesL()` methods indicate which audio formats are supported for audio output and audio input, respectively. Both return an array of four-character codes that identify an audio encoding format. All Symbian OS phones can be expected to support playback and recording of various PCM formats. Many also support encoded formats such as AMR. The header file `mmffourcc.h` defines the four-character codes for these audio formats.

The `Capabilities()` method indicates what audio parameters are supported by the sound device. It returns a `TMMFCapabilities` object; the members of this class are interpreted as follows:

- `iRate` is a bit field of values taken from enumeration `TMMFSampleRate`, indicating which audio sample rates are supported for audio input and output
- `iEncoding` represents which PCM encoding formats are supported, with values taken from enumeration `TMMFSoundEncoding`; this functionality has been superseded by the methods described above and should not be relied on
- `iChannels` indicates how many channels are supported by the audio device; it is a bit field of values from enumeration `TMMFMonoStereo`
- `iBufferSize` is the maximum size of an audio buffer that can be handled by DevSound.

The sample rates supported for audio input and output may differ; by default DevSound reports the capabilities supported for playback. To get the recording capabilities, you should initialize DevSound for recording

before calling this method. Note also that the number of channels and the sample rate may be constrained by the audio format in use: for example, when using AMR-encoded audio, only 8 kHz mono may be supported.

Audio playback and recording using DevSound

The DevSound API can be used for high-performance audio input and output streaming, avoiding the overheads associated with the streaming utilities. It also allows more control over the audio device. While a full description of how do to this is outside the scope of this book, a brief discussion is worthwhile.

Before streaming can begin, DevSound must be initialized using one of the `InitializeL()` methods. An observer class implementing mixin `MDevSoundObserver` must be provided.

```
IMPORT_C void InitializeL(MDevSoundObserver& aDevSoundObserver,
                          TFourCC aDesiredFourCC, TMMFState aMode);
```

The parameter `aMode` should be set to `EMMFStatePlaying` to perform playback or `EMMFStateRecording` for recording. Note that it is possible to play fixed tone sequences and DTMF strings directly using DevSound, but this is not recommended since it provides no advantage over the easier-to-use class `CMdaAudioToneUtility`.

Following an initialization call, you receive a callback to the `InitializeComplete()` method in your observer indicating the success or otherwise of the initialize operation.

You can now configure the parameters of your playback or recording using the appropriate methods. To begin the playback or recording activity, you should now call either `PlayInitL()` or `RecordInitL()`, which request permission from the audio policy and then begin.

During a playback operation, the following sequence of events occurs:

1. DevSound calls its observer's `BufferToBeFilled()` method, requesting some audio data.
2. The observer fills the buffer with audio of the appropriate type.
3. The observer passes the buffer back to DevSound via the `PlayData()` method. This can be done synchronously or asynchronously.

When DevSound requires more data, `BufferToBeFilled()` is called again. To end playback, the final buffer is marked as the last buffer by calling `CMMFBuffer::SetLastBuffer(ETrue)` on it.

During a recording operation, the sequence of events is similar. Full audio buffers are passed to the observer via the `BufferToBeEmptied()` method; the observer returns the buffers to DevSound via the `RecordData()` method.

DevSound may report errors to the observer at any time during playback or recording via the asynchronous callback methods `RecordError()` or `PlayError()`. Other events may be passed to the observer via the `SendEventToClient()` method. Event types that are passed into this method include the following:

- `KMMFEventCategoryPlaybackComplete` when a playback or recording activity is complete
- `KMMFEventCategoryAudioResourceAvailable` if you have requested a notification when the audio resource is available.

Writing Controller Plug-ins

The MMF is designed to allow extensibility via ECOM plug-ins. The controller plug-in is the mechanism by which support for new audio formats are added to the system. The following discussion should provide you with the information you need to start writing controller plug-ins; it is not intended to be a comprehensive description.

All MMF controller plug-ins derive from class `CMMFController`. They are ECOM plug-ins, which are described in detail in Chapter 19. The details of creating the ECOM plug-in itself are therefore omitted here.

The behavior of an MMF controller largely dictates the behavior of the client-side clip APIs. A badly behaving controller plug-in can cause a well-written client application to stop responding or to behave incorrectly. Some care must therefore be taken when writing controller plug-ins to preserve the expected behavior of the client-side APIs.

Defining the plug-in

The MMF requires that each MMF controller plug-in has at least one associated format, a type of file that the plug-in can read or write. A basic controller plug-in has a single format; a more complex plug-in can have any number. A format is presented as another ECOM plug-in to the system; however, the format is not instantiated by ECOM unless the associated controller does this explicitly; hence it is not necessary to actually write a format plug-in, but merely to describe it in the controller plug-in resource file.

The `opaque_data` section in a controller's resource file describes what media types and formats are supported. The section is split with tags of the following forms:

- <s> specifies the supplier of the controller plug-in
- <i> specifies the media type UIDs supported by the controller
- <p> specifies the ECOM interface UID of play formats supported by the controller
- <r> specifies the ECOM interface UID of record formats supported by the controller.

For example:

```
opaque_data = "<s>Symbian<i>0x101F5D07<p>0x101F0001";
```

The UIDs following an <i> tag are defined in `MmfDatasourcesink.hrh`. Valid UIDs are:

- `KUidMediaTypeAudio: 0x101F5D07`
- `KUidMediaTypeVideo: 0x101F5D08.`

An audio controller specifies only audio as a supported media type. A video controller would typically specify audio and video. In this case, there would be two separate <i> tags in the `opaque_data` section.

The format plug-ins must have interface UIDs matching those given in the <p> or <r> tags in the controller. The `opaque_data` section for a format plug-in is similar to that for a controller; the following tags are recognized:

- <s> specifies the supplier of the format;
- <i> specifies the media IDs supported by the format;
- <m> specifies the MIME type of the format;
- <e> specifies the file extension given to files of this format;
- <h> specifies header data to match to recognize a file of this format.

For example:

```
opaque_data =
        "<s>Symbian<i>0x101f5d07<e>.ogg<h>OggS*vorbis<m>application/ogg";
```

The <i> tags in the format would usually be the same as those in the controller. The <e> and <m> tags can be specified multiple times, if more than one file extension or MIME type is commonly associated with the format.

The format of the string following an <h> tag is the same format as that accepted by the `TDesC8::MatchF()` method. If more flexibility is required, you can write a separate recognizer plug-in that matches the files to a MIME type supported by the format plug-in. See Chapter 19 for more information on writing recognizer plug-ins.

Implementing the plug-in

As mentioned earlier, a controller plug-in derives from class `CMMFController`. You need therefore to derive from this class. To properly support the MMF client-side APIs, you also need to derive from a number of other classes, to support custom commands. Custom commands are those commands that need to be supported by some controllers, but not all.

An audio playback controller should implement the following custom command mixin classes:

- `MMMFAudioPlayDeviceCustomCommandImplementor`, allowing the volume and balance to be set and queried
- `MMMFAudioPlayControllerCustomCommandImplementor`, to support playback windows.

`MMMFAudioControllerCustomCommandImplementor` can also be implemented by an audio playback controller if it wishes to allow its client to query the properties of the audio clip, such as the bitrate and number of channels.

An audio recording controller should implement the following:

- `MMMFAudioRecordControllerCustomCommandImplementor`, for recording information to be provided
- `MMMFAudioControllerCustomCommandImplementor`, to allow the parameters of the recording to be queried and set.

The following should be implemented by video controller plug-ins:

- `MMMFVideoControllerCustomCommandImplementor` for querying and setting basic video information
- `MMMFVideoPlayControllerCustomCommandImplementor` in a video-playback controller
- `MMMFVideoRecordControllerCustomCommandImplementor` in a video-recording controller.

Additionally, `MMMFResourceNotificationCustomCommandImplementor` should be implemented by any controller that wishes to support audio resource notifications.

For each custom command mixin class implemented by a controller plug-in, a custom command parser should be added when the plug-in is constructed. Each custom command implementer interface has a corresponding custom command parser. For example:

```
void COggVorbisController::ConstructL()
  {
  ...
  CMMFAudioPlayDeviceCustomCommandParser* audPlayDevParser =
        CMMFAudioPlayDeviceCustomCommandParser::NewL(*this);
  CleanupStack::PushL(audPlayDevParser);
  AddCustomCommandParserL(*audPlayDevParser);
  CleanupStack::Pop( audPlayDevParser ); //audPlayDevParser
  ...
  }
```

This allows the MMF to find the custom command implementer and send the appropriate commands to it.

Any custom commands that are not implemented by a controller result in a `KErrNotSupported` error when a client calls the corresponding client-side API. In many cases, it may cause the client-side APIs to fail entirely if vital custom commands are not implemented.

Operation of the plug-in

Your controller plug-in is instantiated by the MMF when it is opened by a client. The subthread is created for you. After your controller has been opened, you can add one or more data sources and data sinks, via the `AddDataSourceL()` and `AddDataSinkL()` methods. Which specific sources and sinks are added depends on the media IDs that you support (as specified in the plug-in definition), and whether you are performing playback or recording. For example, an audio playback controller has a file, descriptor or URL data source and an audio output data sink. The principal job of the controller is to read data from the sources and write data to the sinks.

When a data source or sink is added, you can verify its type by checking its UID. These UIDs are defined in header file `MmfDatasourcesink.hrh`. Once all the appropriate data sources and sinks have been added, the controller is primed via the `PrimeL()` method. Here, you should prime each of your sources and sinks and perform any other initialization necessary to begin the playback or recording activity.

The `CMMFController::PlayL()` method is used to begin both a playback and a recording activity. The two are similar to the extent that your controller is reading from its sources and writing to its sinks. The fact that `PlayL()` is called to begin recording should not therefore be a source of confusion.

Platform security limitations

Prior to Symbian OS v9.1, third-party controller plug-ins could be added to phones to allow new formats to be supported in the built-in media player applications, and to allow those new formats to be used as ringtones and other system sounds. However, due to the capability-based DLL loading restrictions introduced in Symbian OS v9.1, many of these use cases are not available to the majority of third-party developers. A built-in media player application typically has a DRM capability and the process responsible for playing ringtones usually has MultimediaDD capability to allow it to play system sounds. These are both protected capabilities not easily available to third-party developers.

You should therefore note that when writing a new controller plug-in, it may not be usable in any of the built-in applications. However, it does not prevent a third-party media application to be written that supports all the formats available on the phone, including those provided in the phone's firmware and those added via third-party controller plug-ins.

21.2 The Image Conversion Library

The Image Conversion Library, or ICL, is a library based on ECOM plug-ins that supports encoding and decoding of a wide variety of image formats. ICL image encoder and decoder plug-ins for many formats are provided by Symbian. Other plug-ins may be added by phone manufacturers, to support more formats or to take advantage of hardware-accelerated encoding or decoding on a phone. Plug-ins can also be provided by third parties.

The ICL also supports scaling and rotating of bitmaps once they have been decoded, as well as displaying encoded images on the screen easily.

Overview of the API

The core ICL API consists of basic image encoding and decoding classes. These are used to convert between external, usually compressed, image formats and the native `CFbsBitmap` type. See Chapter 17 for more information about `CFbsBitmap`. The classes used to perform decoding and encoding are `CImageDecoder` and `CImageEncoder` respectively.

The encoding or decoding operation takes place asynchronously, allowing your application to continue responding to other events. This is achieved in one of two ways:

- By using an active object running in the same thread, the image conversion operation is split into numerous smaller tasks, each running within a separate call to the object's `RunL()`. Your application remains responsive during the decode operation, but calls to the

RunL() functions of any of the application's other active objects could be slightly delayed during the conversion operation.

- By using a dedicated subthread that the ICL can automatically create for the conversion process; no intensive processing takes place inside your application's thread, ensuring that it always remains responsive.

Note that image conversion plug-ins can indicate that they should always be executed in a separate thread. If this is the case, you cannot override this behavior.

If you simply want to display an image on screen, you can make use of the CImageDisplay class. The ICL also provides functionality to scale and rotate native CFbsBitmap s, using the CBitmapRotator and CBitmapScaler classes. These classes perform their functions using active objects in your application's thread. Unlike the image-decoding and -encoding classes, they do not have the facility to perform in a subthread.

Image Decoding

Decoding of images is achieved using the CImageDecoder class, defined in ImageConversion.h. It supports decoding of an entire image in one go, or 'progressive' or 'streamed' decoding, where the image data arrives in parts. This allows a partial image to be rendered before the entire image has been received.

> If you need only display an image on the screen, you should consider using CImageDisplay. This class has numerous advantages, such as automatic animation images with multiple frames, which make it better suited to displaying images on screen. However Nokia have excluded it from the S60 SDK.

When we instantiate an image decoder, we must provide all or some of the image data immediately. For example, if we are decoding an image from a file:

```
iDecoder = CImageDecoder::FileNewL(iFs, aFileName);
```

Here, iFs is an RFs instance. It should already be connected to the file server. The method leaves if the file cannot be recognized or if it is not a supported image file. The file can contain either a partial or a complete image.

There is also a DataNewL() method that allows you to instantiate a decoder for processing image data stored inside a descriptor. The initial

descriptor you provide can contain an entire encoded image or only as much as is available at the time. Construction of the decoder fails if there is not enough data to determine its format, unless you specify the format explicitly.

Decoding options

There are several overloads of the `FileNewL()` and `DataNewL()` methods that allow us to specify the image decoder options, the content access intent, the MIME type of the image, the UID of the image type and subtype, and the UID of the specific image decoding plug-in to use.

The image decoder options are defined by enumeration `CImageDecoder::TOptions`:

- `EOptionNone` – no options set; this is the default
- `EOptionNoDither` – do not perform any error-diffusion dithering when loading an image; when decoding a bitmap with a different display mode to the original image, dithering is used by default; you should use this option if you intend to further resize the image after it has been decoded
- `OptionAlwaysThread` – always perform the image decoding in a separate subthread; consider using this option if you are using the decoder within a GUI, or other, application that requires the `RunL()` functions of all active objects to be serviced promptly; it has extra overhead associated with creating a new thread, so should only be used when it is required
- `EOptionAllowZeroFrameOpen` – allows the creation of the image decoder even when not enough data is present to decode an entire image frame; this should be set when you intend to perform streamed decoding
- `EAllowGeneratedMask` – an image decoder normally only reports that transparency is possible if mask data is encoded with the image; however, some decoders have the ability to automatically generate a mask; setting this flag instructs the decoder to enable this operation, if it is available.

Several options can be combined using bitwise or.

If we specify the MIME type of an image, the decoder plug-in is selected on the basis of this rather than by recognizing the image type based on the image data itself. Similarly, we can indicate the format of the image by specifying the image type and subtype UIDs and the UID of the specific decoder plug-in to use. This means that the decoder plug-in can always be instantiated, even if no data is present at the time of creating the plug-in. The image subtype is usually set to `KNullUid`.

Image type and subtype UIDs for image formats supported by Symbian OS are defined in `imagecodecdata.h`. Note that other image formats can be added to the ICL by phone manufacturers or third parties, so the list of image UIDs here may not be complete.

Some image formats do not contain enough data in the header to be recognized on the basis of the encoded data itself. In these cases, it is necessary to specify either the MIME type or the image type when creating a decoder.

Decoder subclasses

When an image decoder is created, the plug-in can optionally provide a subclass of `CImageDecoder` that is returned to the client instead of `CImageDecoder` itself. This allows functionality specific to a particular image format to be provided to the client. For example, when decoding a Jpeg image, we get an instance of `CJPEGExifDecoder` allowing access to EXIF metadata stored within the image.

Basic decoding

If all the image data is available when the decoder is constructed, decoding is quite straightforward. First, we create a new instance of `CImageDecoder` providing the source data. In this example, we let the ICL automatically select an appropriate plug-in for us:

```
iDecoder = CImageDecoder::FileNewL(iFs, aFileName);
iFrameInfo = iDecoder->FrameInfo();
```

Once the decoder is created, we can be sure that there is enough data for at least one frame since we did not specify the option `EOptionAllowZeroFrameOpen`.

Before we can decode the image, we need to create a destination `CFbsBitmap` object to store the decoded image. Depending upon what we intend to do with it next, we have two options:

- create a destination bitmap with the same size as the frame we are decoding
- scale the image down during the decoding operation to save memory.

If the image is to be encoded using a different format for a conversion operation, we want to decode the entire image. If it is to be displayed on screen, we may want to scale it down. Note that very large images may have to be scaled down, if there is insufficient memory to store the entire decoded image in memory at once.

To scale the image down, we create a `CFbsBitmap` that is smaller than the encoded image. Some decoders support arbitrary scaling of an image during decoding; in this case, we can simply create the `CFbsBitmap` at the exact size that we wish the decoded image to be. In general though, decoders only support scaling down of images by a factor of 2, 4 or 8. `CImageDecoder` provides some methods to help work out what scaling factor should be used. When arbitrary scaling is not supported, we have to perform further scaling once the image has been decoded if we need to fit it into a predefined area. Many user interface components can do this for us, however.

```
if (iFrameInfo.iFlags & TFrameInfo::EFullyScaleable)
  {
  TRect fitRect(desiredSize);
  ShrinkToAspectRatio(iFrameInfo.iOverallSizeInPixels, fitRect);
  decodeImageSize = fitRect.Size();
  }
else
  {
  TInt reductionFactor = iDecoder->ReductionFactor(
     iFrameInfo.iOverallSizeInPixels, desiredSize);
  if (reductionFactor>3) reductionFactor=3;
  User::LeaveIfError(iDecoder->ReducedSize(
           iFrameInfo.iOverallSizeInPixels,
        reductionFactor, decodeImageSize));
  }
```

The flag `TFrameInfo::EFullyScaleable` indicates if the decoder supports arbitrary scaling of the image. If it does, we simply match the aspect ratio of the desired size to that of the image. If the flag `TFrameInfo::EConstantAspectRatio` is not set, this is not necessary as the image can be stretched during decoding.

If the decoder does not support arbitrary scaling, we use the `ReductionFactor()` and `ReducedSize()` methods to work out what size the decoder can scale to. The `ReductionFactor()` method gives us the factor by which we need to scale the image, as a logarithm to the base 2. Since the image decoder only supports scaling by factors of 2, 4 and 8, we must make sure that the reduction factor we use is not greater than 3 (since $2^3 = 8$). The `ReducedSize()` method is then used to calculate the final size for the bitmap. It is important that you use this method rather than calculating the final size yourself as different decoder plug-ins may use different rounding algorithms.

> Some third-party ICL decoder plugins supplied on Symbian OS phones support reduction factors greater than 3. If your desired reduction factor is greater than 3, you can attempt to call `ReducedSize()` anyway and only reduce it to 3 if this returns an error.

Once the destination bitmap has been created, we can call `Convert()` to start the conversion process. This takes a zero-based frame number as a parameter, for images that consist of more than one frame; it defaults to zero.

```
iBitmap = new(ELeave)CFbsBitmap();
User::LeaveIfError(iBitmap->Create(decodeImageSize,
                    iFrameInfo.iFrameDisplayMode));

iDecoder->Convert(&iStatus, *iBitmap);
SetActive();
```

Here, the destination bitmap has the same display mode as the source image. If we intend to display the image on the screen, it is better to use the same display mode as the screen. This allows the decoder to perform error diffusion if necessary.

When the decode operation has completed, the request is completed and our active object `RunL()` is executed, indicating success or an error. If the decoding operation is successful, the error code is `KErrNone` and we can display the image on the screen using any method that accepts a `CFbsBitmap`, or encode it as another format for a conversion operation. If we receive an error of `KErrUnderflow`, this means that there was not enough data to fully decode the image (see the section on progressive decoding).

Image masks

The presence of a mask in a frame is indicated by the setting of the `ETransparencyPossible` flag in the frame information. The mask can be decoded along with the image by using the overload of `CImageDecoder::Convert()` that takes a second `CFbsBitmap`. The mask bitmap must have the same size as the main bitmap; the display mode is usually different.

The display mode of the mask can be determined by the flag `EAlphaChannel` in the frame information. If this flag is set, then the mask is an 8-bit alpha blend; hence we should use display mode `EGray256` for the mask bitmap. Otherwise, the mask is a simple 1-bit mask and the mask bitmap should have display mode `EGray2`.

Frame information

An image can contain a number of frames, each of which is, essentially, a separate image. Multiple frames are only supported by some image formats. They are often used to achieve animation.

After the image decoder has been instantiated and header processing has completed, we can query the number of frames present using the

FrameCount() method. Information about each frame is returned by the FrameInfo() method that we saw earlier. In a Jpeg image that contains a thumbnail in the EXIF metadata, two frames are reported: the main image and the thumbnail.

Images that contain frames that are intended to be animated contain timing information in the iDelay member of TFrameInfo. This determines how long the frame should remain on screen, and the flags ELeaveInPlace, ERestoreToBackground and ERestoreToPrevious determine how to transition from one frame to the next. However, it should not be necessary to implement this functionality yourself since it is provided by the CImageDisplay class.

Further information about the frames in an image is represented by the CFrameImageData class and is retrieved using the FrameData() method. This provides access to data specific to various image formats, such as palettes, compression methods or quality settings. Each decoder can provide its own classes deriving from TFrameDataBlock or TimageDataBlock, which can be encapsulated within CFrameImageData; many are defined in imagecodecdata.h.

Determining the type of an encoded image

The CImageDecoder class provides two static methods for determining the format of an encoded image:

```
static void GetMimeTypeFileL(RFs& aFs, const TDesC& aFileName,
                                        TDes8& aMimeType);
static void GetMimeTypeDataL(const TDesC8& aImageData, TDes8& aMimeType);
```

These methods can be used to determine if an encoded image is recognized by the ICL with the available decoder plug-ins. Note that successful recognition of some image data by these methods usually implies that the image can be decoded. It is possible, though unlikely, to get false positives using these methods and the image data could be corrupted, leading to an error when you attempt to decode it.

Progressive decoding

Progressive decoding, also known as streamed decoding, allows you to start decoding an image before all of the data is available. This is useful when the image data is being downloaded over a network, for example. It enables you to render a partial image during downloading or display some frames of a multi-frame image before all frames have been received. Progressive decoding is more complex than basic decoding. The process is shown in Figure 21.3.

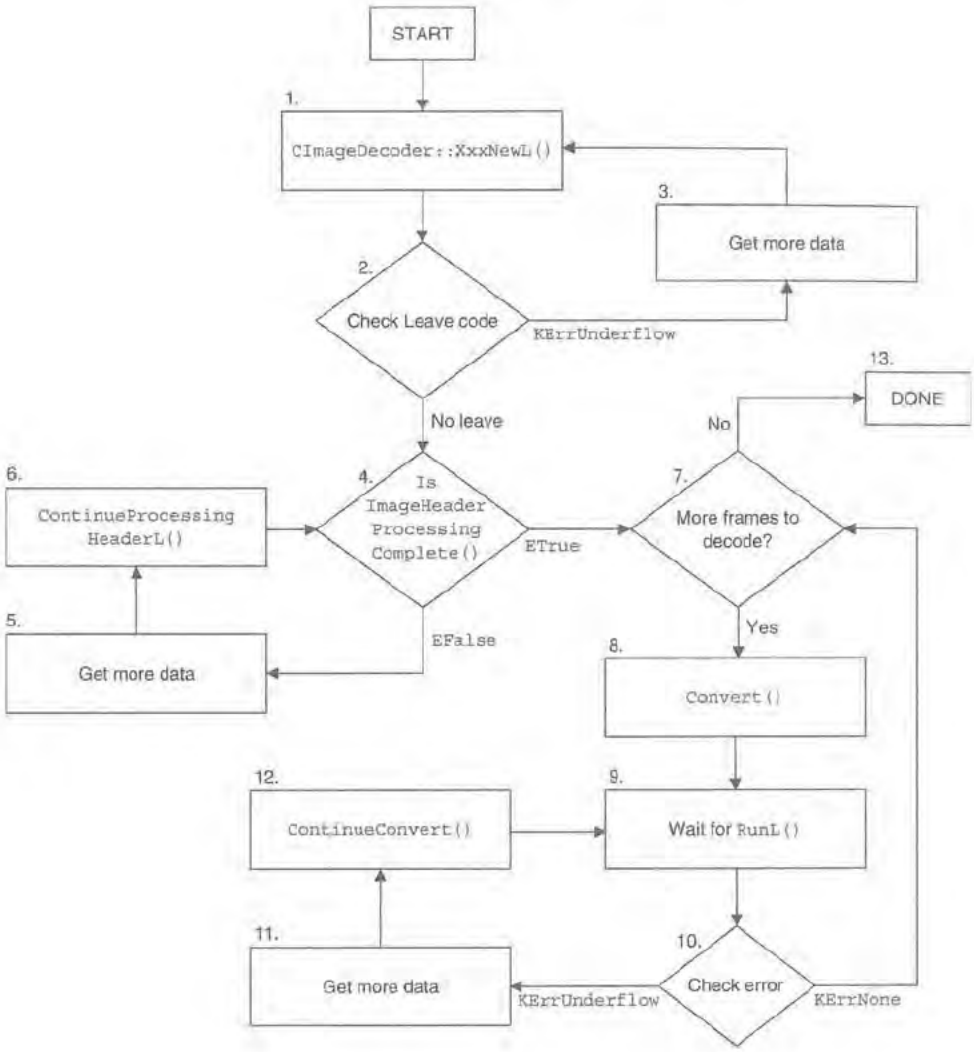


Figure 21.3 Progressive decoding of an image

The first step is to create the image decoder using either `CImageDecoder::FileNewL()` or `CImageDecoder::DataNewL()`. Whichever method you are using, it is assumed here that the data source does not contain the entire image, merely the first part of it, however small.

If you do not explicitly specify either the MIME type of the data or the type of the image, the construction of `CImageDecoder` fails if there is not enough data present to recognize the image format from the data provided. In this case, you should wait for more data to become available and then try again (Steps 2, 3 and 1).

Once the image data has been successfully recognized and a `CImageDecoder` has been instantiated, we must then continue to read data until header processing has been completed (Steps 4, 5 and 6). We continue appending data to the file or descriptor until `IsHeaderProcessingComplete()` returns `ETrue`; at this point, we can start attempting to decode the first frame.

The first time we reach Step 7, we know that the condition is true as we have not yet decoded any frames and an image is assumed to contain at least one frame.

When we call `Convert()`, we must then wait asynchronously for our `RunL()` (Steps 8, 9 and 10). In the normal case (i.e. when no errors are detected in the image data), we receive an error of either `KErrNone`, indicating that the frame was successfully decoded, or `KErrUnderflow` indicating that more data is required to decode the frame. If we receive an error of `KErrUnderflow`, we must wait for more data to become available; this data should be appended to the existing data source and then `ContinueConvert()` called to continue the decoding process (Steps 11, 12 and back to 9).

Once an entire frame has been decoded, we can either move onto the next frame if it exists (Steps 10, followed by 7, 8, 9 and 10 again) or conclude that the image processing is now complete (step 13).

In Steps 8 and 12, you must provide the same `CFbsBitmap` instance while decoding a single frame. You can use a new `CFbsBitmap` when moving onto the next frame, that is, when moving between Steps 10, 7 and 8.

The details of Steps 3, 5 and 11 depend upon your data source. It is likely that, since you are using progressive decoding, the data is arriving asynchronously and so these stages may involve an asynchronous wait for more data. In this case, the data is receiving concurrently with the decoding; you should be careful to coordinate the two activities properly. As new data arrives, it should be appended to the original file or descriptor.

There are two possible places in Figure 21.3 when the image could be rendered. If the frame being decoded does not have the flag `EPartialDecodeInvalid` set, then we can render a partially decoded frame between Steps 10 and 11. This would allow the user to see the portion of the image that has been received already, which is the main advantage of progressive loading of images.

In all cases, the frame can be rendered between Steps 10 and 7 when an error of `KErrNone` is received, as in this case the frame has been fully decoded.

Figure 21.3 does not show any error conditions, for the sake of clarity. However, there are a number of steps where your code need to consider error cases.

In any of Steps 3, 5 and 11, if no more data is available this means that the source data has become truncated. You should fail with `KErrUnderflow`; you can still render a partial frame if one exists.

If an error other than `KErrNone` or `KErrUnderflow` is received in Steps 2 or 10, again decoding has failed and you should not continue.

Buffered decoding

The class `CBufferedImageDecoder` can be used to simplify the process of progressive image decoding slightly. The class encapsulates the `CImageDecoder` and the descriptor in which the encoded image is stored. It can be created before any image data is present and it can be reused.

Its use is very similar to that of `CImageDecoder` when we perform progressive decoding. However, once we have passed some data into the buffered decoder, we no longer need to keep that data ourselves, as it has been appended to an internal buffer. The upshot of this is that the encoded image data is always copied, meaning slightly greater overheads.

The usage of the buffered decoder is as follows:

1. Call `OpenL()`, optionally specifying a MIME type or image type UIDs, as with `CImageDecoder`.
2. While `ValidDecoder()` returns `EFalse`, add more data to the decoder using the `ContinueOpenL()` method.
3. When `ValidDecoder()` returns `ETrue`, write data via the `AppendDataL()` method and then call `ContinueProcessingHeaderL()`. Continue making this sequence of calls until `IsImageHeaderProcessingComplete()` returns `ETrue`.
4. The frame information can now be queried using the `FrameCount()` and `FrameInfo()` methods. To begin decoding of a frame, create a `CFbsBitmap` of the appropriate size and then call `Convert()`.
5. The decoding process now takes place in the same way as with `CImageDecoder`, except that when more data becomes available we write it to the decoder using the `AppendDataL()` method.

To reuse a `CBufferedImageDecoder`, simply call `Reset()` and then start again.

Image Encoding

Encoding of images is done using class `CImageEncoder`, defined in header `ImageConversion.h`. The class is similar to `CImageDecoder` in the way that it is used, but is somewhat simpler. It encodes a

CFbsBitmap to a number of external formats, where the destination for encoded data can be either a memory buffer or a file; it does not support streamed, progressive or buffered encoding.

Clearly, when we are encoding an image, we must specify a destination image format to use. There are two ways of doing this:

- provide a MIME type for the encoded image; an image encoder plug-in is selected based on this MIME type
- specify the image type, subtype and encoder plug-in to use by their UIDs.

The set of formats that can be encoded into by CImageEncoder depends entirely on what image encoder plug-ins are present on the phone. As with decoder plug-ins, further encoding plug-ins may be added be phone manufacturers or third parties.

If you know in advance what image format you wish to encode, and it is supported by the release of Symbian OS that you are targeting, it is easiest to hard code the MIME type of the format. This allows the ICL to automatically select a suitable encoder plug-in for you.

Querying supported encoding formats

The ICL provides the ability to query what image formats can be encoded. Since all of the encoders are ECOM plug-ins, it would be possible to do this by interfacing directly to ECOM; using the ICL to query the available plug-in is far easier, however. Class CImageEncoder contains the following methods to achieve this:

- GetImageTypesL() returns all image types that can be encoded, together with short descriptions
- GetImageSubTypesL() returns the list of image subtypes associated with a given image type
- GetFileTypesL() returns the list of MIME types, that can be encoded, together with their associated filename extensions and image type UIDs.

Encoding an image

As with the decoder, an image encoder is instantiated using the File-NewL() or DataNewL() methods. The methods are similar to the equivalent ones in CImageDecoder; the main difference being that it is mandatory to specify the format.

When writing the image to a file, it is desirable to ensure that the file has the expected extension for the image type in use, as reported by GetFileTypesL(). Any existing file with the same name is overwritten.

If you are using the DataNewL() method, the destination for the encoded image is given as type HBufC*&. You should pass in a reference to a NULL pointer; a descriptor of a suitable size is created and the pointer is updated. Ownership of the descriptor is transferred to the caller.

The image encoder allows you to specify some options when it is created. The options are defined by enumeration CImageDecoder::TOptions. The only option available is EOptionAlwaysThread, which has the same meaning as the identically named option for the decoder.

```
iEncoder = CImageEncoder::FileNewL(iFs, fileName, mimeType);
```

Once the encoder has been instantiated, we can begin the encoding process. This is done using the Convert() method; we provide a TRequestStatus to signal when the encoding is complete, the source bitmap and, optionally, some frame image data represented by CFrameImageData. This can be used to configure the encoder.

```
iEncoder->Convert(&iStatus, *iBitmap);
SetActive();
```

We then wait for a RunL() indicating success or otherwise. You must ensure that the CFbsBitmap that you pass into the Convert() method is not deleted or altered until the conversion has completed.

The error code we receive in the iStatus is KErrNone if the encoding was a success; in this case, the data destination we provided when we instantiated the encoder contains an encoded image. Any other value indicates that there was some problem with the encoding process. Likely errors are KErrNotFound, if no suitable encoder plug-in could be found, or KErrNoMemory if the system runs out of memory.

Configuring the encoder

When we call convert on the encoder, we have the option of providing some frame image data. Encoders that support any configuration provide subclasses of TImageDataBlock or TFrameDataBlock that can be added to the CFrameImageData passed into the Convert() method. These classes are defined in imagecodecdata.h. For example, you can use class TBmpImageData to configure the bits per pixel of the Bmp encoder, or class TJpegImageData to configure the quality of the Jpeg encoder.

```
TJpegImageData data;
data.iSampleScheme = TJpegImageData::EColor444;
data.iQualityFactor = 75;
```

```
iFrameImageData* = CFrameImageData::NewL();
User::LeaveIfError(iFrameImageData->AppendImageData(data));

iEncoder->Convert(&iStatus, *iBitmap, iFrameImageData);
SetActive();
```

The size of the encoded image is the same as the source bitmap that you provide.

Displaying Images

The class `CImageDisplay` is intended to ease the process of decoding images for display. It uses an asynchronous callback interface, `MIclImageDisplayObserver`, to pass decoded images to its client; you must therefore implement this mixin to use the class.

In addition to basic decoding of an image, `CImageDisplay` can also perform the following operations:

- rotation of an image by 90°, 180° or 270°
- flipping of an image vertically or horizontally
- animation of images that support it, such as Gif or MNG images
- automatic rotation of images based on the header information; principally this is used to present the correct orientation of Jpeg images that contain EXIF rotation information as generated by many digital cameras
- scaling up or down to any arbitrary size
- clipping.

> Although `CImageDisplay` is part of Symbian OS v9.1, Nokia have excluded it from the S60 3rd edition SDK. It is however present in the UIQ SDK and on UIQ-based phones.

Basic use of `CImageDisplay` is slightly simpler than that of `CImageDecoder`. Once it has been instantiated, we call `SetImageSource()` to pass the image source into the class. The source is represented by class `TMMSource`; you can use `TMMFileSource` or `TDescriptorDataSource` to open an image from a file or a descriptor, respectively.

```
class CImageDisplayer : public CCoeControl,
          public MIclImageDisplayObserver
...
```

```
void CImageDisplayer::SetImageSourceL(const TDesC& aFileName)
  {
  ...
  User::LeaveIfError(
      iImageDisplay->SetImageSource(TMMFileSource(aFileName));

  iImageDisplay->SetOptions(CImageDisplay::EOptionMainImage |
                            CImageDisplay::EOptionAutoRotate);
  iImageDisplay->SetupL();
  ...
  }
```

Note that the filename or descriptor used when instantiating a class derived from `TMMSource` must persist for the lifetime of the `CImageDisplay` object. The `TMMSource` class itself does not need to.

After `SetImageSource()` has been called, we can set the options to use for displaying the image. In this example, we specify to use the main image rather than a thumbnail and to automatically rotate the image so that it has the correct orientation. The available options are as follows:

- `EOptionThumbnail` – use a thumbnail rather than the main image, if available; this saves memory and speeds up the decoding time
- `EOptionMainImage` – use the main image, even if a thumbnail is embedded in the image
- `EOptionRotateCw90` – rotate the image by 90° clockwise
- `EOptionRotateCw180` – rotate the image by 180° clockwise
- `EOptionRotateCw270` – rotate the image by 270° clockwise
- `EOptionMirrorHorizontal` – flip the image about the horizontal axis
- `EOptionMirrorVertical` – flip the image about the vertical axis
- `EOptionAutoRotate` – automatically rotate the image as dictated by the header data.

It is mandatory to specify either `EOptionThumbnail` or `EOptionMainImage`. Failure to do so results in the `SetupL()` call leaving with `KErrArgument`. Once we have set the options, calling `SetupL()` attempts to recognize the image data and find and load a plug-in to decode it.

Before we can display the image, we must set a size for the destination; this would usually be the size at which we intend to draw the image.

```
TSize sizeToUse;
const CImageDisplay::RImageSizeArray& recommendedSizes =
                iImageDisplay->RecommendedImageSizes();
```

```
if (recommendedSizes.Count() == 0)
  {
  // no recommended sizes -  use our controls size
  sizeToUse = Rect().Size();
  }
else
  {
  TRect myRect = Rect();
  TSize imageSize = recommendedSizes[0];
  ShrinkToAspectRatio(imageSize, myRect);
  sizeToUse = myRect.Size();
  }
iImageDisplay->SetSizeInPixels(sizeToUse);
iImageDisplay->Play();
```

Here, we scale down the recommended size – the size of the decoded image – to fit within the control, while maintaining the aspect ratio. Once the size has been set, we call `Play()` and then wait for a callback.

```
void CImageDisplayer::MiidoImageReady(const CFbsBitmap* aBitmap,
      TUint aStatus, const TRect& /*aUpdatedArea*/, TInt aError)
  {
  if (aError == KErrNone)
    {
    if (aStatus & CImageDisplayPlugin::EStatusFrameReady)
      {
      iLastBitmap = aBitmap;
      DrawNow();
      }
    ...
    }
  else
    {
    //  handle the error
    }
  }
```

If we don't want to animate the image, this is all the we need to do. To achieve the animation, however, a little more work is required.

```
void CImageDisplayer::MiidoImageReady(const CFbsBitmap* aBitmap,
      TUint aStatus, const TRect& /*aUpdatedArea*/, TInt aError)
  {
  if (aError == KErrNone)
    {
    ...
    if (!(aStatus & CImageDisplayPlugin::EStatusNoMoreToDecode))
      {
      // there's more to decode
      iImageDisplay->Play();
      }
    else if (iImageDisplay->ImageStatus() & CImageDisplay::EImageAnimated)
      {
      // we have completed a playthrough of the animation
      // play it again
```

```
      iImageDisplay->StopPlay();
      iImageDisplay->Play();
      }
    }
    ...
  }
```

The absence of flag `EStatusNoMoreToDecode` indicates that there remains more data to be decoded. This usually means further frames in an animation, but it could also be that we have not fully decoded the first frame. Calling `Play()` again prompts `CImageDisplay` to continue the animation (with the appropriate delay) or finish decoding the image.

When the animation has played through once, the flag `EStatusNoMoreToDecode` is set. If we wish the repeat the animation, we call `StopPlay()` to reset the animation to the beginning followed by `Play()` to play it again. We should only do this if the image is animated; if we were to call `Play()` again with a single frame image, it would cause it to be decoded again resulting in a infinite loop.

Please note that you cannot call `Reset()` or delete the `CImageDecoder` instance from within the `MiidoImageReady()` callback. Attempting to do so results in a panic.

As with many other parts of the multimedia subsystem, `CImageDisplay` is based on ECOM plug-ins. The image formats that are supported by the class are directly influenced by the available plug-ins. There is a generic ICL wrapper plug-in that ensures that all image formats supported by `CImageDecoder` are also supported by `CImageDisplay`. A specific plug-in for Jpeg images ensures that any rotation information contained within a Jpeg EXIF header is honored. Other specific plug-ins exist for other formats, such as Multiple-image Network Graphics (MNG) images.

An image display plug-in can optionally provide an extension interface allowing access to plug-in-specific functionality. The `CImageDisplay::ExtensionInterface()` method provides access to these interfaces, which are identified by UID. Extension interfaces provided in Symbian OS v9.1 include:

- `MExifImageDisplayExtension`, which gives access to Jpeg EXIF metadata and to control the quality of the Jpeg-scaling algorithm. This interface is identified by UID `KExifImageDisplayExtensionUid`
- `MGenIclImageDisplayExtension`, which gives access to some of the functionality of the underlying image decoder plug-in, if one exists. The UID `KUidGenIclImageDisplayPluginExtUid` identifies this interface.

The `CImageDisplay::ExtensionInterface()` method returns `KErrNotSupported` if you request an interface that is not supported by the current image display plug-in. You should be sure to check for this error before attempting to use the interface pointer provided.

Image Transformations

The classes `CBitmapScaler` and `CBitmapRotator`, both defined in `BitmapTransforms.h`, can be used to scale and rotate bitmaps represented by `CFbsBitmap`. They are similar to `CImageEncoder` and `CImageDecoder` in the way that they are used, but do not provide the option to perform the operation in a separate thread.

Both classes have two modes of operation: transforming a bitmap in-place or transforming a source bitmap and writing the result into a separate bitmap, leaving the source unchanged.

Rotating bitmaps

The class `CBitmapRotator` is used to rotate bitmaps. Once the call is instantiated, simply call the `Rotate()` method.

```
iRotator->Rotate(iStatus, *iSourceBitmap, *iDestBitmap,
          CBitmapRotator::ERotation90DegreesClockwise);
SetActive();
```

The destination bitmap should be instantiated, but `Create()` does not need to have been called. `CBitmapRotator` atomically creates the destination bitmap of the appropriate size and shape, depending on what rotation angle is applied. The rotation angle must be one of the values from enumeration `CBitmapRotator::TRotationAngle`.

You must ensure that both the source and the destination bitmaps remain in scope for the duration of the rotation operation.

When the rotation is complete, the `TRequestStatus` you provided is signaled. Note that the rotation operation itself requires many separate occurrences of the active object's `RunL()` internally; hence you must yield to the active scheduler after calling `Rotate()`.

Scaling bitmaps

Use of `CBitmapScaler` is very similar to that of `CBitmapRotator`. The `Scale()` method is called to initiate the asynchronous scaling operation.

```
iScaler->Scale(iStatus, *iBitmap, iNewSize, ETrue);
SetActive();
```

If, as here, you are using the in-place variant of the `Scale()` operation, the size of the new bitmap is specified using `TSize`. If you provide a destination bitmap object to write the scaled bitmap into, the size of that destination determines the scaling factor. You must have already called `Create()` on the destination bitmap, if you provide one.

You can change the quality of the scaled image by calling the `SetQualityAlgorithm()` method before scaling the image. You can choose between minimum, medium and maximum quality. The high-quality algorithm takes longer to process; minimum quality is faster.

The final parameter indicates whether or not to maintain the aspect ratio of the scaled bitmap. If it is set to `EFalse`, the bitmap is stretched to fit the destination size exactly. If it is set to `ETrue`, the bitmap is scaled down so that it fits inside the rectangle defined by the destination size – that is, such that the width or height of the final bitmap match the width or height of the requested destination size, but the other dimension is smaller if the aspect ratio differs.

Again, you must ensure that the source and destination bitmaps remain in scope for the duration of the scaling operation and you must yield to the active scheduler to allow the scale operation to take place.

21.3 Camera API

Symbian OS v9.1 provides the ECAM API to allow access to any onboard camera on the phone. It provides functionality to display a viewfinder and capture still images from the camera. It is also possible to capture video using the ECAM API; however we recommend that you use the video recording API outlined in Section 21.1 instead.

The class `CCamera` provides this functionality; it resides in `ECam.h`. To use this class, you must provide a class that derives from `MCameraObserver2` or `MCameraObserver`. It is recommended that you use `MCameraObserver2` where it is supported, as this class supersedes the older `MCameraObserver`. You must have capability UserEnvironment to use `CCamera`.

The implementation of the camera API is not supplied by Symbian since it is highly dependent on the hardware architecture of each phone. Each phone manufacturer provides their own implementation, using the same `CCamera` API. You may therefore find that the behavior differs slightly between phones and manufacturers.

> Some Symbian OS v9.1 phones from Nokia do not support `MCameraObserver2`; to work with these phones, you have to use the old `MCameraObserver` class.

Selecting a Camera

Before you attempt to use the camera on a Symbian OS phone, you should first check that one exists. The static `CCamera::CamerasAvailable()` method returns the number of cameras present. Many phones capable of video telephony have more than one; often one facing towards the user and one facing away.

To instantiate `CCamera`, simply call `CCamera::NewL()`, specifying the index of the camera you want to use as the `aCameraIndex` parameter. If you want to select a camera to use on the basis of its orientation or capabilities, you should instantiate a `CCamera` instance for each one and check the capabilities using the `CameraInfo()` method. This returns a `TCameraInfo` class, containing a variety of information about the camera in question.

The camera's orientation, `iOrientation`, can take one of the following values:

- `EOrientationOutwards` – the camera faces away from the user and so would usually be used to take pictures
- `EOrientationInwards` – the camera faces towards the user and is probably used for video telephony
- `EOrientationMobile` – the orientation of the camera can be changed by the user; in this case, you can't rely on knowing the present orientation of the camera
- `EOrientationUnknown` – the orientation of the camera is not known.

The member `iOptionsSupported` of `TCameraInfo` is a bit field indicating a number of options that a camera supports. The values for the bits are defined by enumeration `TCameraInfo::TOptions`. The following values are useful in selecting a camera:

- `EImageCaptureSupported` – indicating if still-image capture is supported by the camera
- `EVideoCaptureSupported` – indicating if video capture is supported.

Other information contained in this field is discussed later.

Setting up the Camera

Once we have selected a camera to use and instantiated the `CCamera` instance, the next stop is to reserve it. Reserving the camera, if successful, grants our application exclusive access to the camera until our application

releases it or is pre-empted by another application. Camera reservation is based on priority; higher-priority camera clients can pre-empt lower-priority ones. The priority is set when we instantiate `CCamera` object, using the priority parameter of the `NewL()` method.

To reserve the camera, simply call `Reserve()`. This is an asynchronous call and we expect a callback indicating success or failure.

```
  iCamera->Reserve();
  ...

void CCameraEngine::HandleEvent(const TECAMEvent& aEvent)
  {
  if (aEvent.iEventType == KUidECamEventReserveComplete)
    {
    if (aEvent.iErrorCode == KErrNone)
      {
      iCamera->PowerOn();
      }
    else
      {
      // handle the error
      }
    }
  ...
  }
```

If you are using `MCameraObserver` rather than `MCameraObserver2`, the corresponding callback is `ReserveComplete()`. In this example, we power up the camera immediately. If we were just reserving the camera with the intention of using it later, we would not do this.

A call to `PowerOn()` results in the camera itself being switched on. This results in greater drain on the battery, so it should not be done until necessary. After this call, we receive another callback.

```
void CCameraEngine::HandleEvent(const TECAMEvent& aEvent)
  {
  ...
  if (aEvent.iEventType == KUidECamEventPowerOnComplete)
    {
    if (aEvent.iErrorCode == KErrNone)
      {
      // success! Camera is now active.
      iState =  ECameraReady;
      ...
      }
    else
      {
      iCamera->Release();
      // handle the error
      }
    }
  ...
  }
```

The corresponding callback for `MCameraObserver` is `PowerOnComplete()`.

You'll have noticed that the `MCameraObserver2::HandleEvent()` callback method is used for multiple purposes. Each event type is given a UID; some event UIDs to expect are as follows:

- `KUidECamEventReserveComplete` – this is a response to a call to `CCamera::Reserve()`, indicating the success or otherwise of the camera reservation
- `KUidECamEventPowerOnComplete` – indicates the success or otherwise of powering on the camera hardware
- `KUidECamEventCameraNoLongerReserved` – this can occur at any time while we have reserved the camera and indicates that the reservation of the camera has been revoked, usually due to another application reserving it.

Displaying the Viewfinder

After we have successfully powered on the camera hardware, we can display a viewfinder. The camera subsystem can render the viewfinder for us, using direct screen access, or we can render the viewfinder ourselves, using bitmaps passed to us at regular intervals. See [Sales 2005, Chapter 11] for more information on direct screen access.

The option to use is determined by the capabilities of the camera. The `iOptions` member of `TCameraInfo` may have one of the following two bits set:

- `EViewFinderBitmapsSupported` – we have to render the viewfinder ourselves using bitmaps supplied by the camera
- `EViewFinderDirectSupported` – rendering of the viewfinder is performed for us using direct screen access.

You should always check which method is supported before attempting to render a viewfinder. It is possible that neither method is supported, in which case rendering a viewfinder is not possible. You may still be able to take still images or videos, however.

A viewfinder of either type can be stopped at any time by calling the `StopViewFinder()` method.

Direct screen access viewfinder

Rendering of a direct screen access viewfinder is the most efficient method. The camera subsystem takes care of the rendering and is usually

optimized to make use of any hardware acceleration present. To draw a direct screen access viewfinder, we use the `StartViewFinderDirectL()` method.

```
void StartViewFinderDirectL(RWsSession& aWs,
            CWsScreenDevice& aScreenDevice,
                    RWindowBase& aWindow,
                      TRect& aScreenRect)
```

The parameters are similar to those used to initialize video playback, which also uses direct screen access. The first three can be retrieved from an application GUI environment; `aScreenRect` is the rectangle, relative to the physical screen, in which the viewfinder is to be rendered. Once a successful call has been made to `StartViewFinderDirectL()`, the viewfinder is visible to the user. No further action is required by the application.

Bitmap-based viewfinder

When a bitmap-based viewfinder is active, the camera passes bitmaps to the application at regular intervals, and the application draws them on the screen as appropriate. The bitmaps are passed at a rate fast enough to ensure a smooth viewfinder display.

To start the viewfinder, make a call to `StartViewFinderBitmapsL()`. This method requires a single `TSize` parameter that determines the size of the bitmaps that you receive. It should be set to the size of the screen area into which you intend to draw the viewfinder.

Following this, bitmaps are passed to you via the `ViewFinderReady()` callback method (or `ViewFinderFrameReady()` if you are using `MCameraObserver`).

With `MCameraObserver::ViewFinderFrameReady`, the bitmap passed to you is simply a `CFbsBitmap` which can be drawn to the screen in the usual way.

If you are using `MCameraObserver2`, the bitmap is represented by class `MCameraBuffer`. The `CFbsBitmap` can be retrieved from this class using the `BitmapL()` method. An `MCameraBuffer` is capable of encapsulating several frames which may be encoded; for the sake of a viewfinder, you should expect a single, unencoded frame in the form of a `CFbsBitmap` instance.

Capturing Still Images

Before still images can be captured, you should query the capabilities of the camera in use. There are two parameters that need to be set based on the capabilities: the format of the captured image and its size.

Selecting the image format

The formats that are supported are determined by the member `iImageFormatsSupported` of class `TCameraInfo`. This is a bit field of values from enumeration `CCamera::TFormat`. Some cameras support a single format. Others may support a number of formats, in which case you choose the one most suitable for your application. Formats that may be supported include the following:

- `EFormatJpeg` and `EFormatExif`; these are encoded formats that need decoding before they can be displayed on the screen; the data arrives encoded in a descriptor
- `EFormatFbsBitmapColorXxx`; these are uncompressed formats in which the captured image is represented by a `CFbsBitmap` object
- `EFormatRgbXxBitRGBXxx` and `EFormatYUVXxx`; these are raw data formats, the characteristics of which are determined by the exact format; the data arrives in a descriptor.

The formats supported by a camera may be influenced by the resolution of that camera. High-resolution cameras are likely to present the image data in a compressed format, as the memory required to store the image in its uncompressed form would be too great.

Once you have selected a format to use, you can enumerate the capture sizes that are supported for that format. The number of image sizes supported is stored in the `iNumImageSizesSupported` member of class `TCameraInfo`. The sizes themselves are returned by the `CCamera::EnumerateCaptureSizes()` method.

```
RArray<TSize> iSizeArray;
...
for (TInt i=0; i<iCameraInfo.iNumImageSizesSupported; ++i)
  {
  TSize size;
  iCamera->EnumerateCaptureSizes(size, i, iFormat);
  iSizeArray.AppendL(size);
  }
```

The `iFormat` member contains a value from `CCamera::TFormat`, selected from `iImageFormatsSupported`.

Preparing for the capture

Before you can capture an image, you need to prepare the `CCamera` object using the `PrepareImageCaptureL()` method. This allows the camera subsystem to allocate any memory necessary and perform any other setup required to capture an image; it only needs to be called once

for many image captures. It can be done at any time before the image capture takes place; if images are being captured in response to user input, you should prepare the camera in advance to minimize the delay when the capture is performed.

```
iCamera->PrepareImageCaptureL(iFormat, aSizeIndex);
```

The parameter `aSizeIndex` is the index of capture size that you wish to use; it corresponds to the `aSizeIndex` parameter of the `EnumerateCaptureSizes()` method.

After a successful call to `PrepareImageCaptureL()`, the camera is ready to start capturing images. No asynchronous callback is received.

Capturing the image

Image capture is an asynchronous operation. It is initiated simply by making a call to `CaptureImage()`.

```
iCamera->CaptureImage();
```

You then receive a callback to the `MCameraObserver2::ImageBufferReady()` method.

```
void CCameraEngine::ImageBufferReady(MCameraBuffer& aCameraBuffer,
                                     TInt aError)
  {
  if (aError == KErrNone)
    {
    // process the image
    }
  else
    {
    // Handle the error
    }
  }
```

To cancel an image capture, you can call `CancelImageCapture()`. You must not call `CaptureImage()` again before you receive the `ImageBufferReady()` callback without canceling the ongoing image capture first.

The image is encapsulated by class `MCameraBuffer`. The class can store image data in a number of ways and can store multiple frames. In the case of still-image capture, it contains a single frame of the requested format.

If you requested an `EFormatFbsBitmapColorXxx` format when you configured the image capture, the `MCameraBuffer::BitmapL()` method returns a handle to the `CFbsBitmap` containing the image data.

For other formats where the data is presented in a descriptor, the data can be accessed using the `MCameraBuffer::DataL()` method. The interpretation of this data depends on the format that you requested.

If the format is `EFormatJpeg` or `EFormatExif`, the descriptor can be decoded to a `CFbsBitmap` using `CImageDecoder` as described in Section 21.2; alternatively, the data can written directly to a Jpeg file.

Some camera implementations are able to store the data in a shared chunk. In this case, the `ChunkL()` and `ChunkOffsetL()` methods return details of it. See [Sales 2005, Chapter 13] for more information on shared chunks.

Once you have finished processing an `MCameraBuffer` object, you must call its `Release()` method. This allows the memory to be freed or reused by the camera subsystem. Failure to do so results in memory leaks.

If you are working with a phone that only supports `MCameraObserver` instead of `MCameraObserver2`, the equivalent callback is `MCameraObserver::ImageReady()`. In this case, you are given a `CFbsBitmap*` and an `HBufC8*`; these can be considered equivalent to the objects returned by `MCameraBuffer::BitmapL()` and `MCameraBuffer::DataL()`, respectively. Only one of them is valid, as determined by the image format in use.

21.4 Tuner API

The tuner API allows an application to control a radio tuner on a phone, and play and record audio from it. Symbian provides the Tuner API, as defined in `tuner.h`, which phone manufacturers can implement on phones that have a radio tuner.

> At time of writing, the only phones that support the Tuner API are the Sony Ericsson P990 and W950, which allow control of the FM radio. Symbian OS phones made by other manufacturers that have an FM radio do not yet support the Tuner API.

Tuning

The `CMMTunerUtility` provides the basis for the Tuner API. This class allows you to perform tuning and acts as a factory for the other classes that form the Tuner API. To use it, you must derive from `MMMTunerObserver`.

Before instantiating a tuner utility, you should check that a tuner is present. The static `CMMTunerUtility::TunersAvailable()` method indicates how many tuners are present on the phone. If this method

returns a value greater than 0, you can call `CMMTunerUtility::NewL()`.

The static `CMMTunerUtility::GetCapabilities()` method can be used to retrieve the capabilities of a given tuner; if more than one tuner is present, you could use the capabilities to select a suitable tuner for your needs.

Before attempting to use a tuner, you should check that it is possible to do so currently. A tuner may, for example, require an external antenna to be attached. In addition, you should never attempt to use a tuner if the phone is in 'flight mode'.

You can determine the requirements of the tuner using the `iAdditionalFunctions` member of class `TTunerCapabilities`; this is a bit field of values from `TTunerCapabilities::TTunerFunctions`. You should check the bits `ETunerFunctionRequiresAntenna` and `ETunerFunctionAvailableInFlightMode`. For example:

```
TTunerCapabilities caps;
User::LeaveIfError(CMMTunerUtility::GetCapabilities(0, caps));
if (caps.iAdditionalFunctions &
    TTunerCapabilities::ETunerFunctionAvailableInFlightMode)
  {
  // we can use the tuner regardless of the flight mode setting
  ...
  }
else
  {
  TBool flightMode;
  User::LeaveIfError(iTuner->GetFlightMode(flightMode));
  if (flightMode)
    {
    // we cannot use the tuner until flight mode is enabled
    ...
    }
  else
    {
    // flight mode is disabled; we can use the tuner
    ...
    }
  }
```

The state of the external antenna can be checked using the `IsAntennaAttached()` method. An attempt to use a tuner when either the flight mode or the antenna state does not allow it results in an error of `KErrNotReady`.

Unlike the ECAM API discussed in Section 21.3, the tuner does not have explicit control over the power state of the tuner hardware; there is no `PowerOn()` method. This is because the Tuner API allows multiple clients to use the same tuner hardware concurrently. The hardware remains powered on while at least one client requires it.

If the flight mode and antenna states allow the tuner to be used, you can start using it by issuing a `Tune()` request:

```
iTuner->Tune(TFrequency(98500000), CMMTunerUtility::ETunerBandFm);
```

The class `TFrequency` is used to represent a radio frequency that can be tuned to. The unit of the frequency is Hertz; hence the value given above corresponds to 98.5 MHz in the FM band.

You should check the tuner capabilities to determine which bands are supported, which also determines whether you should use frequencies or channels; you can check the valid range of frequencies or channels for a given band using the `GetFrequencyBandRange()` and `GetChannelRange()` methods.

After making any tune or search call, you receive a callback indicating success or otherwise to the `MToTuneComplete()` method. You must not make any further tune calls until you have received this callback.

You can search for frequencies or channels that carry stations using an appropriate overload of the `StationSeek()` method.

Tuner notifications

There are a number of events for which you can request notification when using a tuner. Each notification category has an associated callback class and request method in `CMMTunerUtility`.

Tuner state change notifications can be requested using the `NotifyChange()` method; you must derive from class `MMMTunerChangeObserver` to receive these notifications. This provides the following notifications:

- changes of tuner state – that is, if the tuner hardware is powered up, if it is playing and if someone is recording from it
- changes to the currently tuned frequency or channel
- antenna attachment and detachment notifications
- flight-mode change notifications
- changes to the squelch setting.

If you derive from `MMMSignalStrengthObserver`, you can call its `NotifySignalStrength()` method to request notifications when the strength of the signal being received by the tuner changes. Similarly, you can request notifications when the audio reception changes between stereo and mono by deriving from `MMMTunerStereoObserver` and calling its `NotifyStereoChange()` method.

Control of the tuner

When you instantiate the tuner utility you have the option to specify a tuner access priority. This is used to arbitrate between multiple applications attempting to use the tuner at the same time. Only one application is granted control of the tuner at any given time; only the application that has control of the tuner can retune it and otherwise change its state.

When you call any method in the tuner utility that requires control of the tuner, a request for control is made for you. When control of the tuner is granted, you receive a callback to `MToTunerEvent()`:

```
void CRadioTuneEngine::MToTunerEvent(MMMTunerObserver::TEventType aType,
                                     TInt aError, TAny* /*aAdditionalInfo*/)
  {
  if (aType == MMMTunerObserver::EControlEvent)
    {
    if (aError == KErrNone)
      {
      // we have been granted control of the tuner
      iHaveControl = ETrue;
      }
    else
      {
      // some other tuner client has pre-empted us.
      iHaveControl = EFalse;
      }
    }
  else if (aError != KErrNone)
    {
    // handle the error
    }
  }
```

An event type of `MMMTunerObserver::EControlEvent` indicates that we have been granted or denied control of the tuner.

Once granted control of the tuner, we keep it until either we release it explicitly, by calling `ReleaseTunerControl()`, or we are pre-empted by another application with a higher priority for access to the tuner. If we do not wish to prevent other applications from tuning the tuner, we should release it whenever it is granted. As with audio priorities, high tuner priorities are protected by capability MultimediaDD.

Note that having control of the tuner does not prevent other applications playing or recording the tuner audio. Permission to do this is handled by the system-wide audio policy, in exactly the same way as for any other audio playback or recording. Other applications can also observe Radio Data System (RDS) data without requiring control of the tuner.

Closing the tuner

When you have finished using the tuner utility, you can call the `Close()` method to release any tuner resources that have been allocated to you. Doing so also affects any of the utilities that have been instantiated using the tuner utility described in the following sections: any ongoing tuner audio playback or recording is terminated and you receive no more RDS data notifications.

Before deleting your `CMMTunerUtility` instance you must ensure that any tuner utilities that you have created using the `GetXxxUtility()` methods have been deleted first. Failure to do so leads to undefined behavior – most likely a crash.

Tuner Audio Playback

Playback of audio from a tuner is performed using class `CMMTunerAudioPlayerUtility`. This class is instantiated by `CMMTunerUtility`.

```
iPlayer = iTuner->GetTunerAudioPlayerUtilityL(*this);
```

You must derive from the `MMMTunerAudioPlayerObserver` class in order to use it. The `CMMTunerUtility` object used to instantiate the tuner player utility (`iTuner`) must persist until after the player utility instance is deleted.

Use of the player utility is quite straightforward. First, you must initialize the utility using the `InitializeL()` method; you can specify an audio priority at this time. This audio priority has exactly the same meaning as in the standard audio-clip player described in Section 21.1. After initializing the utility, you receive a callback to `MTapoInitializeComplete()`.

```
iPlayer->InitializeL();
...
void CRadioTuneEngine::MTapoInitializeComplete(TInt aError)
  {
  if (aError == KErrNone)
    {
    iPlayer->Play();
    }
  else
    {
    // Handle the error
    }
  }
```

Playback continues indefinitely until it is stopped or some error occurs. If an error occurs during the playback, we receive a callback to the `MTapoPlayEvent()` method. The type of event is determined by the

aEvent parameter, which takes values from enumeration MMMTunerAudioPlayerObserver::TEventType. This is either ETunerEvent, indicating some problem with the tuner such as the antenna being removed, or EAudioEvent, which may mean that the audio policy has revoked our permission to play the audio.

The class CMMTunerAudioPlayerUtility also provides a number of methods to change the volume and balance of the tuner audio and some other methods common with the standard audio-clip-player utility.

Tuner Audio Recording

The CMMTunerAudioRecorderUtility class can be used to record audio from a tuner. It is very similar to the CMdaAudioRecorderUtility class described in Section 21.1, so it is not discussed in detail here. The main difference from the standard audio recorder is that you must derive from a different class to use it, MMMTunerAudioRecorderObserver. It is otherwise the same in all important respects; it does not contain any playback-related functionality.

The tuner audio recorder utility is instantiated using the CMMTunerUtility::GetTunerAudioRecorderUtilityL() method.

The tuner player and recorder utilities can be used simultaneously. However, before you attempt to play and record audio from the radio at the same time, you should use the the CMMTunerUtility::GetCapabilities() method to check that the ETunerFunctionSimultaneousPlayAndRecord capability is supported.

As with the tuner audio player utility, the CMMTunerUtility instance used to instantiate a CMMTunerAudioRecorderUtility must not be deleted before the recorder utility itself is deleted.

Using RDS data

Some phones that contain an FM radio also support Radio Data System (RDS) data access. This is used to broadcast small amounts of data along with the audio, containing details such as the name of the station. It can also be used to broadcast application-specific data in custom formats. The class CMMRdsTunerUtility allows applications to access the data being broadcast with a radio station. It can be used concurrently with the player and recorder utilities. The tuner utility used to instantiate CMMRdsTunerUtility must not be deleted before the CMMRdsTunerUtility itself.

The class contains a number of GetXxx() methods to access the RDS data. The most recently received RDS data is cached by the tuner system. However, since the data is received asynchronously over the air and may change at any time, it is more useful to receive notifications when it is received or when it changes.

To receive notifications for basic RDS data, you should derive from class `MMMRdsDataObserver`. Notifications can then be requested by calling its `NotifyRdsDataChange()` method. When you request these notifications, you can request a subset of RDS data for which you wish to receive notifications; for example:

```
iRdsTuner->NotifyRdsDataChange(*this, TRdsData::EStationName |
                                      TRdsData::EProgrammeType);
```

You receive a callback whenever the data in question changes, or becomes invalid because the tuner has been retuned.

```
void CRadioTuneEngine::MrdoDataReceived(const TRdsData &aData,
                                        TUint32 aValid, TUint32 aChanged)
 {
 if (aChanged & TRdsData::EStationName)
   {
   if (aValid & TRdsData::EStationName)
     {
     // PS has changed and is valid
     ...
     }
   else
     {
     // PS is no longer valid
     ...
     }
   }
 ...
 }
```

The RDS utility can provide notifications for all of the commonly used RDS data fields. You can also request access to raw RDS frames as they are received over the air if you wish to decode them yourself to access application-specific data. To do this, you should derive from class `MMMRdsFrameReceiver` and request the frames that you wish to receive by calling its `ReceiveRdsFrames()` method.

Summary

This chapter has provided an introduction to the range of Multimedia APIs that are available in Symbian OS, and how to use them.

- The Multimedia Framework supplies the ability to stream, or to record and play, audio and video data.
- The Image Conversion Library supports encoding and decoding of a wide variety of image formats.

- The ECAM camera API provides access to any onboard camera on the phone. It provides functionality to display a viewfinder and capture still images from the camera.
- The radio tuner API allows an application to control a radio tuner on a phone, and play and record audio from it.

Appendix

Developer Resources

Download the source code for this book at
developer.symbian.com/main/academy/press/books/scmp_v3.

Symbian OS Software Development Kits

Software development kits (SDKs) are based on a particular user interface (UI) for Symbian OS. Each distinct UI has an associated set of system applications for such tasks as messaging, browsing, telephony, multimedia and contact and calendar management. These applications typically make use of generic application engines provided by Symbian OS. SDKs are required to support the installation of third-party applications written in native C++. There are two UIs for Symbian OS v9, SDKs for which can be found at the websites listed here:

- Nokia S60 (***www.forum.nokia.com/S60***)
- UIQ (***developer.uiq.com/devtools_uiqsdk.html***)

For the independent software developer, the most important thing to know in targeting a particular phone is its associated reference platform. You can then decide which SDK you need to obtain. In most cases, you can target – with a single version of your application – all phones based on the same reference platform and Symbian OS version associated with this SDK. The Symbian OS System Definition papers give further details of possible differences between phones based on a single SDK (***www.symbian.com/phones***).

Getting an IDE for Development

To develop your code, you'll need an Integrated Development Environment (IDE), which contains an editor, compiler and linker. Two IDEs

are recommended – for non-commercial development, use Carbide.c++ Express Edition, which is a free download. Otherwise, Carbide.c++ Developer and Professional Editions are recommended, along with CodeWarrior. Support for Visual Studio .NET 2003 is also available using Carbide.vs.

- Carbide.c++ (***www.forum.nokia.com/carbide***)
- CodeWarrior (***www.forum.nokia.com/codewarrior***)

Getting a UID for Your Application

A UID is a 32-bit number. Every graphical application should have its own UID that allows Symbian OS to distinguish files associated with that application from files associated with other applications. UIDs are also used in other circumstances, such as to identify streams within a store and to identify one or more of an application's views.

To get a UID, register with and log on to the Symbian Signed website (***www.symbiansigned.com***). Request your UIDs from the My Symbian Signed tab – ten UIDs is a reasonable first request.

> Don't build different Symbian OS applications with the same application UID – even the same test UID – on your emulator or Symbian OS machine. If you do, the system recognizes only one of them and you will not be able to launch any of the others.

Symbian OS Developer Tools

As well as tools offerings from Symbian partners, Symbian Developer Network (***developer.symbian.com/main/tools***) provides a number of free, unsupported and open source tools.

Support Forums

Symbian Developer Network offers newsgroups and support forums:

- ***news://developer.symbian.com***
- ***http://developer.symbian.com/forum***

Symbian Developer Network partners also offer support for developers.

Intermediate and advanced courses require previous attendance at the OS Essentials course. The UI system creation course also requires previous attendance at the Application UI course.

Developer Community

These community websites offer news, reviews, features and forums, and represent a rich alternative source of information that complements the Symbian Developer Network and the development tools publishers. They are good places to keep abreast of new software and to announce the latest releases of your own applications.

My-Symbian

My-Symbian (***my-symbian.com***) is a website based in Poland and dedicated to news and information about Symbian OS phones. This site presents descriptions of new software for Symbian OS, classified by user interface. It also features discussion forums and an online shop.

All About Symbian

All About Symbian (***www.allaboutsymbian.com***) is a website based in the UK and dedicated to news and information about Symbian OS phones. The site features news, reviews, software directories and discussion forums. It has strong OPL coverage.

SymbianOne

SymbianOne (***www.symbianone.com***) features news, in-depth articles, case studies, employment opportunities and event information all focused on Symbian OS. A weekly newsletter provides up-to-date coverage of developments affecting the Symbian OS ecosystem. This initiative is a joint venture with offices in Canada and New Zealand.

NewLC

NewLC (***www.newlc.com***) is a collaborative website based in France and dedicated to Symbian OS C++ development. It aims to be initially valuable to developers just starting writing C++ applications for Symbian OS; in time it will cover more advanced topics.

infoSync World

infoSync World (***symbian.infosyncworld.com***) is a website based in Norway that provides features, news, reviews, comments and a wealth

of other content related to mobile information devices. It features a section dedicated to Symbian OS, covering new phones, software and services – mixed with strong opinions that infoSync is not afraid to share.

TodoSymbian

TodoSymbian (***www.todosymbian.com***) is a website written in Spanish. It provides news, reviews, software directories, discussion forums, tutorials and a developers' section.

Symbian OS Books

Development

Symbian OS C++ for Mobile Phones, Vol. 1, Harrison *et al.* John Wiley & Sons. ISBN: 0470856114

Symbian OS C++ for Mobile Phones, Vol. 2, Harrison *et al.* John Wiley & Sons. ISBN: 0470871083

Symbian OS Explained, Stichbury. John Wiley & Sons. ISBN: 0470021306

Developing Software for Symbian OS, Babin. John Wiley & Sons. ISBN: 0470018453

Symbian OS Internals, Sales *et al.* John Wiley & Sons. ISBN: 0470025247

Symbian OS Platform Security, Heath *et al.* John Wiley & Sons. ISBN: 0470018828

Symbian OS C++ Architecture, Morris. John Wiley & Sons. ISBN: 0470018461

Programming PC Connectivity Applications for Symbian OS, McDowall. John Wiley & Sons. ISBN: 0470090537

S60 Programming – A Tutorial Guide, Coulton *et al.* John Wiley & Sons. ISBN: 9780470027653

Management

Symbian for Software Leaders, Wood. John Wiley & Sons. ISBN: 0470016833

Symbian Academy

The Accredited Symbian Developer Primer, Stichbury and Jacobs. John Wiley & Sons. ISBN: 0470058277

Smartphone Operating System Concepts with Symbian OS, Jipping. John Wiley & Sons. ISBN: 0470034491