
Multimedia on Symbian OS

Inside the Convergence Device

Lead Authors

Adi Rome and Mark Wilcox

With

**Kevin Butchart, John Forrest, Robert Heal, Kostyantyn Lutsenko,
James Nash**

Foreword

Andrew Till (Motorola)

Reviewers

**Fadi Abbas (Scalado), Phyllisia Adjei, Les Clark, Patrick
Cumming, James Curry, Brian Evans, Magnus Ingelsten
(Scalado), Amanda Kamin, Petteri Kangaslampi, Roman Kleiner,
Pamela Lopez, Anthony Newpower,
Mike Roshko, Jo Stichbury, Gábor Török**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



A John Wiley and Sons, Ltd., Publication

Multimedia on Symbian OS

Inside the Convergence Device

Multimedia on Symbian OS

Inside the Convergence Device

Lead Authors

Adi Rome and Mark Wilcox

With

**Kevin Butchart, John Forrest, Robert Heal, Kostyantyn Lutsenko,
James Nash**

Foreword

Andrew Till (Motorola)

Reviewers

**Fadi Abbas (Scalado), Phyllisia Adjei, Les Clark, Patrick
Cumming, James Curry, Brian Evans, Magnus Ingelsten
(Scalado), Amanda Kamin, Petteri Kangaslampi, Roman Kleiner,
Pamela Lopez, Anthony Newpower,
Mike Roshko, Jo Stichbury, Gábor Török**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



A John Wiley and Sons, Ltd., Publication

Copyright © 2008

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.

British Library Cataloguing in Publication Data

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

ISBN 978-0-470-69507-4

Typeset in 10/12 Optima by Laserwords Private Limited, Chennai, India

Printed and bound in Great Britain by Bell & Bain, Glasgow

This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

| | |
|---|--------------|
| Foreword | ix |
| About this Book | xiii |
| Authors' Biographies | xvii |
| Authors' Acknowledgements | xxi |
| Symbian Press Acknowledgments | xxiii |
| Code Conventions and Notations | xxv |
| 1 Introduction | 1 |
| 1.1 The Convergence Device | 1 |
| 1.2 Transformation of the Media Industry | 3 |
| 1.3 Symbian OS | 5 |
| 1.4 The Cutting Edge | 6 |
| 1.5 Evolution of the Multimedia Subsystem in Symbian OS | 8 |
| 1.6 A Peek into the Future | 12 |
| 2 Multimedia Architecture | 15 |
| 2.1 The ECOM Framework | 15 |
| 2.2 Platform Security | 16 |

| | | |
|----------|--|------------|
| 2.3 | The Content Access Framework | 21 |
| 2.4 | Multimedia Subsystem | 23 |
| 2.5 | Future Multimedia Support | 31 |
| 3 | The Onboard Camera | 35 |
| 3.1 | Introduction | 35 |
| 3.2 | Accessing the Camera | 36 |
| 3.3 | Camera Control | 39 |
| 3.4 | Displaying the Viewfinder | 43 |
| 3.5 | Capturing Still Images | 46 |
| 3.6 | Capturing Video | 49 |
| 3.7 | Error Handling | 52 |
| 3.8 | Advanced Topics | 53 |
| 4 | Multimedia Framework: Video | 55 |
| 4.1 | Video Concepts | 55 |
| 4.2 | Symbian OS Video Architecture | 57 |
| 4.3 | Client API Introduction | 60 |
| 4.4 | Identifying Video Controllers | 61 |
| 4.5 | Controlling Video Playback | 62 |
| 4.6 | Controlling Screen Output | 71 |
| 4.7 | Getting Video Information | 81 |
| 4.8 | Accessing Metadata | 83 |
| 4.9 | Controlling the Audio Output | 84 |
| 4.10 | Streaming Playback | 87 |
| 4.11 | Recording Video | 88 |
| 4.12 | Controlling the Video that Is Recorded | 93 |
| 4.13 | Controlling the Audio that Is Recorded | 97 |
| 4.14 | Storing Metadata | 100 |
| 4.15 | Custom Commands | 101 |
| 4.16 | Examples and Troubleshooting | 102 |
| 5 | Multimedia Framework: Audio | 105 |
| 5.1 | Introduction | 105 |
| 5.2 | Audio Input and Output Streams | 107 |
| 5.3 | Audio Player Utility | 117 |
| 5.4 | Audio Recorder Utility | 122 |
| 5.5 | File Conversion | 127 |
| 5.6 | Tone Utility | 129 |
| 5.7 | DevSound | 134 |
| 5.8 | Audio Policies | 135 |
| 5.9 | Priority Settings | 137 |
| 5.10 | Miscellaneous | 138 |

| | | |
|----------|---|------------|
| 6 | Image Conversion Library | 141 |
| 6.1 | Introduction | 141 |
| 6.2 | Decoding Images | 144 |
| 6.3 | Encoding Images | 174 |
| 6.4 | Displaying Images | 181 |
| 6.5 | Transforming Images | 185 |
| 6.6 | Native Bitmap Operations | 189 |
| 6.7 | Miscellaneous APIs | 193 |
| | | |
| 7 | The Tuner API | 197 |
| 7.1 | Introduction | 197 |
| 7.2 | Getting Started | 198 |
| 7.3 | Basic Use Cases | 201 |
| 7.4 | Future Technologies | 206 |
| 7.5 | Sample Code | 207 |
| | | |
| 8 | Best Practice | 209 |
| 8.1 | Always Use an Active Scheduler | 209 |
| 8.2 | Use APPARC to Recognize Audio and Video | 210 |
| 8.3 | Don't Use the Video Player to Open Audio Files | 212 |
| 8.4 | Know that MMF and ICL Cannot Detect Some Formats | 212 |
| 8.5 | Don't Use CMdaAudioOutputStream for Network Streaming | 213 |
| 8.6 | Use CMdaAudioPlayerUtility to Play Tone Sequence Files | 215 |
| 8.7 | Use CMdaAudioPlayerUtility to Play Clips | 215 |
| 8.8 | Don't Hardwire Controller UIDs in Portable Code | 216 |
| 8.9 | Set Controller Thread Priorities Before Playing or Recording | 217 |
| 8.10 | Recognize that Behavior Varies when Several Clients Are Active | 218 |
| 8.11 | Understand that the System is Based on Plug-ins | 219 |
| 8.12 | Use RFile and TMMSource Instead of Passing a File Name | 221 |
| 8.13 | Consider the Volume Control | 222 |
| 8.14 | Know When to Use Threaded Requests on ICL | 222 |
| 8.15 | Don't Have too many Clients | 223 |
| 8.16 | Understand the Multi-heap Problem | 224 |
| | | |
| | References and Resources | 229 |
| | | |
| | Index | 233 |

Foreword

It's no longer news that the world of technology has gone mobile. As Symbian celebrates its tenth anniversary, we can reflect on the massive global uptake of mobile devices over the past decade, which has brought us laptops, mobile phones, MP3 players, digital cameras and handheld video cameras, amongst others.

In the ten years since Symbian formed, we've also seen the way we use the Internet change immeasurably. It has evolved from a source of static information to dynamic, real-time content and a means of social interaction. We increasingly seek our entertainment online and we also use the Internet to instantaneously share content and engage with others. Entertainment is no longer just a passive, first-person experience. Traditional activities, such as watching TV or going to the movies, are being challenged by interactive experiences that put the consumer firmly in control, such as digital and web TV and streamed video services. We further extend the experience by using sharing sites such as YouTube; reading and writing reviews on blogs; holding discussions on forums; or accessing content online. Our experience can be personalized and shared in ways that were inconceivable when Symbian first opened its doors for business.

Once mobile technology and interactive entertainment became mainstream, the next step was for them to converge to yield entertainment on a mobile device. The demand for mobile entertainment has grown year on year. While it was once limited to passive consumption using handheld games consoles or portable music players, it is now far more dynamic. Mobile phones are a means of delivering high-quality multimedia, such as streaming music or mobile TV. Like a fixed camera or camcorder, mobile phones can be used to take pictures and videos, but they can also go far beyond these experiences, enabling seamless uploading directly

to blogs or sharing websites such as Facebook, MySpace or, more recently, Motorola's HelloMoto (www.hellomoto.co.uk) and Nokia's Ovi (www.ovi.com) services. As well as playing MP3s and high-quality ringtones, a phone can also play interactive, multimedia games, making the experience far more sophisticated through multiplayer modes or online communities. The converged device offers mobile connected entertainment, with the ability to make calls, send messages, and browse the web – combining mobility, the Internet and interactive entertainment in a small, yet powerful device.

Mobile multimedia is no longer something we are looking to evolve as a market opportunity; it is already a reality for millions of consumers around the world. In 2007, mobile entertainment application sales overtook feature applications such as PDF readers and calculator applications. Handango's Yardstick¹ data for sales of smartphone applications by category shows entertainment products – media players and mobile TV products – as the most purchased during 2007. This figure excludes games, which have traditionally been seen as the primary entertainment add-on by those purchasing after-market applications for their phones. The figure also excludes downloads of free applications such as mobile widgets to access online mobile services. In short, the time of the mobile multimedia developer has arrived.

This enthusiastic uptake in consumption of media-rich content and applications is, in part, down to a new generation of consumer – 'Generation C'. This generation has grown up in a world where the Internet, text messaging and on-demand services have been a natural part of the fabric of day-to-day life. Mobile technology, the Internet, interactive entertainment and social networking are basic factors in their lives. It's not an exclusive club, though; these technologies are adopted by a broad demographic across every continent.

The success of Nokia's Nseries phones, in particular, the uptake of the popular Nokia N95, confirms the appeal of the converged device. It offers functionality suited to high-quality multimedia entertainment, surpassing dedicated single-function consumer devices in many categories. Products from Motorola, Samsung and Sony Ericsson that have focused on particular types of mobile entertainment, such as photography or music, have cemented the Symbian smartphone as the consumer convergence platform of choice. At the time of going to press, Handango Yardstick data showed that there were more new mobile content titles released for Symbian OS than any other single smartphone platform in 2007 and just over 206 million Symbian smartphones shipped worldwide; it took eight years to ship 100 million units and only a further 18 months for the next 100 million.

¹The Handango Yardstick is a report about mobile applications. Yardstick data is available from the Press Room at corp.handango.com.

At Motorola, we are excited by the chance to use Symbian OS to deliver the best interactive mobile applications and experiences. The MOTORIZR Z8 introduced single-click usability for seamless media sharing, for example while photo-blogging. With the MOTO Z10 we further built on this, placing professional video-editing capabilities into handsets for the first time, allowing a user to quickly create a video blog with audio dubbing, transition effects and text overlays. The close relationship between Symbian, chipset manufacturers and UI vendors forms a powerful platform for phone manufacturers like us to build on. We have been able to use Symbian OS to create fully-featured devices which offer an intuitive user experience for multimedia creation and playback and for sharing and easily discovering new content. As an open platform for aftermarket developers, Symbian OS also has a strong developer ecosystem to ease the creation of applications. In the first quarter of 2008, Symbian reported that 9282 third-party Symbian OS applications had been released, a 24% increase on 31 March 2007.²

This book has been created by subject experts inside Symbian and third-party developers who have worked with Symbian OS. If you are planning to work with mobile multimedia, it demonstrates how to use Symbian OS to write a powerful multimedia application and how to manage the complexity and exploit the power of the platform. You can benefit from the combination of the authors' experience and knowledge while learning how to build mobile entertainment applications that ride the wave of convergence.

Andrew Till, Motorola, 2008.

²Symbian Fast Facts are available at www.symbian.com/about/fastfacts.html.

About this Book

This book forms part of the Symbian Press Technology series. It explains the services available to C++ developers wishing to create rich multimedia experiences for smartphones built on Symbian OS v9.1 and later.

What Is Covered?

Chapter 1 introduces the convergence trends that make mobile multimedia an exciting market sector in which to work. The history and evolution of multimedia on Symbian smartphones is covered, along with a look at what the near future holds in terms of new features and functionality. The first chapter avoids technical jargon wherever possible and should be of interest to anyone wanting to catch up with the latest technological developments in the multimedia sector in general, and on Symbian OS in particular.

The remaining chapters of the book are technical and targeted at developers working with Symbian OS v9.1 and later. Where the functionality discussed is not available in Symbian OS v9.1, but in later versions only, this is highlighted and the version where it was introduced is listed. Although many of the APIs described were available earlier than Symbian OS v9.1, we give no details of their use in earlier versions and we make no guarantees of source compatibility.

Devices built on Symbian OS v9.1 include those based on S60 3rd Edition (starting with the Nokia 3250, N80 and E60) and those built on UIQ 3.0 (starting with the Sony Ericsson P990, M600 and W950). All later

devices based on the same UI platforms should have both source and binary compatibility with these original devices (although, of course, they add further features). For S60 this includes, for example, the Nokia N95 and LG KS10 (based on S60 3rd Edition Feature Pack 1, using Symbian OS v9.2) as well as the Nokia N96 and the Samsung SGH-L870 (based on S60 3rd Edition Feature Pack 2, using Symbian OS v9.3). On the UIQ platform, this also includes the Motorola MOTORIZR Z8 and MOTO Z10 (based on UIQ 3.1 and UIQ 3.2 respectively, both using Symbian OS v9.2).

Chapter 2 looks at the architecture of the multimedia subsystem on Symbian OS, giving an overview of the structure, frameworks and component parts. It also provides some essential background material on other key platform features related to multimedia – the ECOM framework, platform security and the content access framework.

Chapter 3 provides a complete guide to the onboard camera API, which allows developers to access the digital camera hardware present on many Symbian smartphones. The steps required to query the presence of cameras on a device and reserve them for use are explained, followed by details of viewfinder, still image and video frame capture functions.

Chapters 4 and 5 are dedicated to the video and audio features, respectively, of the Multimedia Framework (MMF). The video chapter explains some important background concepts relating to digital video storage and delivery. It then goes on to describe the components of the MMF video architecture and their roles in playback and recording before examining the APIs available for client applications. The audio chapter delivers an example-based walkthrough of the various APIs available for playing and recording sounds, explaining which ones to use in different scenarios. Following that, there is a discussion of the lower levels of the audio infrastructure, upon which the client APIs are built.

Chapter 6 is an in-depth exploration of the Image Conversion Library (ICL), the part of Symbian OS used for any manipulation of still images or small animations. It covers encoding, decoding, scaling and rotation of images in any of the formats supported by the platform. Some of the use cases can be quite complex so there is plenty of example code available to clarify the details. Chapter 7 then provides an introduction to the Tuner API, which can be used to access radio tuner hardware on some Symbian smartphones. It discusses the availability and use of this fairly simple API, as well as some potential future extensions.

Chapter 8 discusses best practice for multimedia developers. It's packed with tips and tricks that could potentially save you hours of re-design or debugging later in your projects and is well worth an initial read, then bookmarking and referring to as you work on your multimedia project.

Who Is This Book For?

The core audience for this book is experienced application developers wanting an introduction and reference guide to the multimedia features of Symbian OS. You will also find this book useful if you are:

- porting multimedia code from other platforms
- working for a phone manufacturer or a partner company, creating new devices based on Symbian OS
- writing plug-ins to extend the multimedia functionality of new or existing devices.

All chapters but the first in this book are technical and assume a working knowledge of Symbian C++. The basic idioms and details of day-to-day working, such as how to get a working development environment or how to create ‘Hello World,’ are not included here. If you need more general information about developing on Symbian OS it can be found in other titles in the Symbian Press series (see the References and Resources section at the end of this book). In addition, there are a number of free papers and booklets available on the Symbian Developer Network website (developer.symbian.com).

While this book doesn’t cover the lowest levels of the multimedia subsystem in the detail required for the device creation community, it does provide a useful overview for those handset developers who are new to the area. Chapter 2, in particular, should be very valuable for those working on hardware abstraction and adaptation, as well as built-in multimedia applications.

Similarly, for those developing multimedia plug-ins, some of the implementation details are beyond the scope of this book, but an understanding of the relevant framework and its use by client applications will help you to deliver better components. In any case, where the details are missing from this book, you should find pointers to further material. Failing that, a list of other resources can be found in the References and Resources section.

Where appropriate, we’ve used a number of code snippets in each chapter; these have been tailored specifically to illustrate the topic in question. For readability, we have avoided using large chunks of example code. We have put the full code of the projects for Chapters 3 and 6 on the website for this book (developer.symbian.com/multimediabook). You will find further example code at the book’s page on the Symbian Developer wiki (developer.symbian.com/multimediabook_wikipage). On the

wiki page, you'll also find useful information about the book, such as links to documentation and tools, and an errata page. Please feel free to visit it regularly and to contribute.

The Symbian Foundation

In June 2008, as we prepared to take this book to press, Symbian celebrated its tenth anniversary, and announced that its assets would be divested into a not-for-profit organization called the Symbian Foundation (www.symbianfoundation.org).

The Symbian Foundation has been set up to create a complete, open and free mobile software platform for converged mobile devices, enabling the whole mobile ecosystem to accelerate innovation. To quote from the Symbian Foundation's website:

The Symbian Foundation platform will be available under a royalty-free license from this non-profit foundation. The Symbian Foundation will provide, manage and unify the platform for its members. The membership of the foundation is open to any organization, with the foundation's launch planned for the first half 2009 . . . The foundation will provide one unified platform with one UI framework with first release expected in 2009.

While some details of the multimedia services this book describes may change over time, we consider that much of the material it contains will be valid for the foreseeable future. To find the latest information and advice on how to use this book with the Symbian Foundation platform, we recommend you visit this book's wiki page at developer.symbian.com/multimediabook_wikipage.

Authors' Biographies

Adi Rome

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

Mark Wilcox

Mark has been playing with computers since his dad bought a ZX81 when he was four. He wrote his first multimedia application (a three-room text adventure with sound effects!) on an Acorn Electron when he was seven. Things went downhill from there and he ended up with a Masters degree in Mathematics followed by a brief stint as a research postgraduate in cybernetic intelligence.

In 2001, he decided it was time to get a proper job and started developing software for mobile phones. Since then he's been involved in handset development projects for Ericsson, Fujitsu, Nokia, Panasonic and Samsung. During that time he worked on everything from a GPRS stack, device drivers and a power management server to messaging applications and the code that draws the soft keys. However, it was while working as a Software Architect for Nokia's Multimedia business unit, developing their flagship Nseries products, that Mark developed his interest in the multimedia capabilities of Symbian OS.

Mark became an Accredited Symbian Developer and a Forum Nokia Champion in 2007.

John Forrest

John has been a Symbian OS engineer for a decade, working both inside and outside Symbian. He was a member of the original team for the development of both ICL and MMF, and has been a senior member of the Multimedia team ever since. Prior to that he was a lecturer at UMIST for many years, teaching programming, hardware logic, computer architecture and all things in between. John lives in North London with his partner and two boys.

Robert Heal

Rob joined Symbian in 2004 working as a software engineer for the Messaging team. He spent his first year in the role of defect coordinator, before moving on to act as technical lead on development projects. After three years, Rob moved to the Multimedia team to become the video technology owner, managing and providing technical leadership for the Multimedia video group.

Rob has a BSc in Software Engineering from Birmingham University. Since graduating he has worked in software development roles for a number of different companies, mostly in the telecommunications sector.

Rob lives in Chelmsford with his wife, Cheryl, and daughter, Lauren. He enjoys playing hockey and he firmly believes that his many years of experience in the game make up for his lack of skill and pace. His team mates may be of a different opinion.

Kostyantyn Lutsenko

Kostya received an MSc equivalent from the National Technical University (KhPI) of Kharkiv, Ukraine in 1999. He worked for several

telecommunications-related start-ups and then moved into Symbian programming in 2002. Kostya joined the Symbian multimedia team in 2004 where he was primarily working on the Image Conversion and Camera API subsystems. He likes optimizing graphics algorithms and taking part in orienteering competitions, equipped with a map and a compass.

James Nash

James joined Symbian in 2005 after completing a Computer Science degree at Warwick University. He has been part of the Multimedia team ever since he joined but has managed to get himself involved in several extra activities such as assisting a licensee on-site, performing technology demonstrations at trade shows and, most recently, contributing a chapter to this book.

When not at work playing with shiny gadgets, he enjoys drawing, watching films and cooking (as long as he doesn't need to wash up afterwards).

Authors' Acknowledgements

Adi Rome

Delivering a new life and this manuscript at the same time was very challenging. A special thank you to Mark for stepping in and getting involved when I needed it most. Further thanks to each and every author and reviewer for bringing their breadth of knowledge and experience to the book and so making it special and colorful.

It was a real pleasure for me to work with the Symbian Press team: Satu tracked every step of the project and made sure that all tasks were delivered on schedule and Jo's dedication, wide knowledge and open mind made her a wonderful person for brainstorming.

Finally, I would also like to thank my husband, Isaac, for his constant support and for putting up with my pregnancy hormones combined with the pressure of trying to complete the book before the birth of our first child, Eyal.

Mark Wilcox

I would like to thank Adi, both for her initial work on the book and also for deciding that delivering a manuscript on time wasn't sufficiently challenging and choosing to deliver a new life into the world instead, giving me the opportunity to get more involved with the project. I'd also like to thank the other authors, without whom there would be no book, and the reviewers for adding their unique insights. Special mention is due to Mike Roshko, who went beyond the call of duty to review several chapters outside of working hours at the last minute and also tracked down his colleagues to get them to answer queries.

The team at Symbian Press are a real pleasure to work with, eternally friendly and helpful. Satu's tact and skill at getting submissions and updates from busy contributors on schedule is the only reason you're able to read this at all. Jo's dedication to the clarity and accuracy of the text has been a real inspiration; her tireless efforts to find the right people to answer queries and fill knowledge gaps will benefit everyone who reads this and were very much appreciated.

Last, but not least, I would like to thank my wife, Lara, for putting up with the near constant presence of my laptop and supporting me throughout – despite my complete lack of ability to think and listen at the same time!

Symbian Press Acknowledgments

Symbian Press would like to thank everyone involved in the production of this book. Initiated by Freddie Gjertsen and Phil Northam, it was nurtured by Petteri Kangaslampi, James Nash and Brian Evans, before we handed it over to Adi Rome, our lead author. She brought order to our proposed contents, with assistance from Anthony Newpower and Kevin Butchart. With the help of Les Clark and Andreas Andersen, we drafted in authors from Symbian in the shape of Robert Heal, John Forrest, James Nash, and we expanded our author roster with the external expertise of Mark Wilcox and Kostya Lutsenko. Satu, as ever, managed this project with efficiency, tenacity and skill. The expert team at Wiley have steered us throughout. Thank you to Rosie, Sally, Colleen, Hannah, Claire, Brett and Shena.

A few weeks before delivering the manuscript, Adi delivered a son, Eyal, and handed completion of the project to Mark Wilcox, who effortlessly edited the chapters into their final form. We're eternally grateful to him for his contributions throughout the project, and to Adi for her initial work on the book. We'd also like to thank all the reviewers, who gave exemplary feedback under tight deadlines. Thanks Mike, Patrick, James, Roman, Brian, Petteri, Gábor, Pamela, Les and Anthony. We'd also like to acknowledge Magnus Ingelsten and Fadi Abbas from Scalado, for their review of Chapter 3, and Andrew Till of Motorola, for contributing the Foreword.

Thanks also go to Annabel Cooke for the images used in Chapter 1 and to Lane Roberts, for acting as official archivist, and casting his mind back in time to the days of the media server.

It's been a huge project, with many people involved. If we've missed thanking anyone here, we apologize. You know who you are, and thank you!

Code Conventions and Notations

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

```
This is example code;
```

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

If you are not familiar with them, the best way to get used to the conventions is to look at code examples in this book and those provided with your chosen SDK.

Capitalization

The first letter of class names is capitalized:

```
class TColor;
```

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

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

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

Prefixes

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

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

Parameters are prefixed with a lower case 'a' which stands for 'argument':

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

We do not use 'an' for arguments that start with a vowel: TBool aExampleBool rather than TBool anExampleBool.

Local variables have no prefix:

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

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

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

Constants are prefixed with 'K':

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

Enumerations are simple types and so are prefixed with 'T'. Enumeration members are prefixed with 'E':

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

Suffixes

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

```
void AllocL();
```

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

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

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

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

Underscores

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

Code Layout

The curly bracket layout convention in Symbian OS code, used throughout this book, is to indent the bracket as well as the following statement:

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

1

Introduction

1.1 The Convergence Device

Convergence has been one of the major technology trends of the last decade. Like all trends, its roots go back much further; in this case you would probably trace them back to the first combination of computing and communications technologies that led to the birth of the Internet. Human interactions that had previously taken place via the physical transportation of paper could now occur almost instantaneously to any connected computer in the world. Increases in computing power, digital storage capacity and communications bandwidth then enabled more complex media – such as voice, music, high resolution images and video – to enter this digital world. Today, near instant global communications are available in multiple media, or multimedia, as they're commonly described.

At the same time this was happening, communications were shifting from fixed wires to mobile radio technologies, allowing people to connect with one another from anywhere. Computing has become increasingly portable, freeing users to work and play on the move. Imaging and video converged with computing to give us digital cameras and camcorders. Music and video distribution and storage have gone first digital and then portable. Global positioning technology combined with portable computing has brought us personal navigation devices. Almost inevitably, miniaturization and integration have led to the development of the ultimate convergence device – the multimedia smartphone.

The term 'smartphone' was first applied to devices which combined the features of a mobile phone and a Personal Digital Assistant (PDA). As technology has advanced, that functionality is now available in some fairly low-end models. In response, the definition of what constitutes a smartphone has changed over time. High-end models are now differentiated with additional features, including hardware-accelerated graphics and video, multi-megapixel cameras, GPS and more sophisticated and

intuitive input methods – smartphones are getting smarter! Symbian currently defines a smartphone as:

a mobile phone that uses an operating system based on industry standards, designed for the requirements of advanced mobile telephony communication on 2.5G networks or above.

The key distinction between a smartphone and a less capable ‘feature phone’ is the operating system. A smartphone operating system is capable of supporting the installation of native applications after the device has been purchased. Examples of smartphones include devices based on Symbian OS and Microsoft Windows Mobile as well as RIM’s BlackBerry devices and Apple’s iPhone.

Smartphones are often referred to as the Swiss Army knives of the consumer electronics world – they have multiple functions, as Figure 1.1 shows. Arguably, the screen size of current smartphones isn’t ideal for web browsing or viewing pictures and videos, the input methods are limited and the quality of camera optics and speakers are constrained by the size of the device. However, there are massive advantages in cost and in having all of these functions available at any time in a single device. Unlike the humble Swiss Army knife, the smartphone is programmable – enabling new synergies and combinations of features not

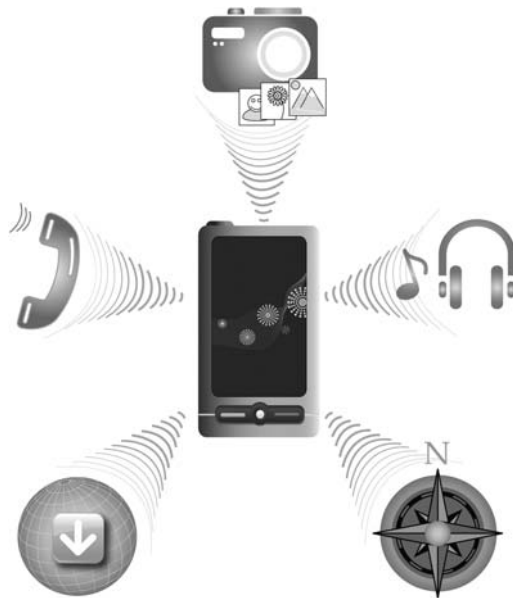


Figure 1.1 Some of the functions of a typical convergence device: taking still and video images, playing music and radio, navigating by GPS, browsing the web and, of course, making voice calls and SMS

possible on separate devices. For example, you can discover, purchase and download new music and video content direct to your device; a blogger, social network user or Internet journalist can capture, compose, upload and view content with a single gadget – no need to transfer between devices or convert between formats. Better still, because most smartphones are open to third-party software, you can customize your device and add new features throughout its life as new technologies and services become available.

When discussing the importance and utility of the smartphone, it is certainly worth considering the developing world. When current smartphone users evaluate devices their perceptions are shaped by their experiences with the Internet on a PC, video on their televisions and DVD players, music on their home stereos and possibly photography on a high-end camera or camcorder. In contrast to that, there are many hundreds of millions of people whose first experiences of the Internet have been, or will be, on a mobile phone. As devices become cheaper and today's latest model is replaced with something more powerful it seems very likely that, for many of those same people, their first experiences of television, photography and other applications will be on today's smartphones. Perhaps some will even be able to share their world and perspectives with a wider audience for the first time.

Although it's easy for most users to take new technology for granted after a very short time, it isn't only engineers, enthusiasts and those who've skipped earlier generations of technology who can get excited about the multimedia capabilities of smartphones – mobile multimedia is big business too!

1.2 Transformation of the Media Industry

The media industry is familiar with disruptive technologies coming along and changing its business models. In the early 1980s in the USA, film companies fought to suppress the spread of mass-market video recorders, fearing that copyright violations would damage their business. They failed, but in the years that followed they found that video recordings of their products had become a major source of income. In the twenty-first century, the convergence of technologies described above permits an unlimited number of copies of media content to be produced and distributed globally for very little cost. Traditional media distributors are challenged and again the content owners are understandably concerned about copyright infringement. Newspapers and magazines are being replaced by websites. CD and DVD sales are being replaced by Internet streaming, downloads and peer-to-peer file sharing. Combined with this, the new technologies have also seeded a very rapidly growing trend in social networking and consumer-generated content.

The primary tools for creating and consuming this content are convergence devices. Sales of camera phones exceeded those of dedicated digital cameras in 2003. The world's largest mobile phone manufacturer, Nokia, is also the largest manufacturer of digital cameras and music players. In Japan, 90% of digital music sales are on mobile devices, while South Korea has become the first market where digital sales have overtaken physical music sales.

According to *PriceWaterhouseCoopers Global Entertainment and Media Outlook 2007–2011*,¹ digital and mobile spending in the sector will rise to \$153 billion in 2011, by which time spending related to the distribution of entertainment and media on convergent platforms (convergence of the home computer, wireless handset and television) will exceed 50% of the global total. It should be noted that smartphones are not the only 'convergent platforms', although they do account for a large volume of such devices. To provide some more mobile-specific figures, Juniper Research predicts total mobile music revenues – including ringtones and ringback tones² – to rise from around \$9.3 billion in 2007 to more than \$17.5 billion by 2012, with much of that growth coming from full track downloads. Additionally, Juniper expects mobile TV, broadcast and streamed, to experience spectacular revenue growth from just under \$1.4 billion in 2007 to nearly \$12 billion by 2012. The projected rise in spending on downloads of music to mobile devices is illustrated by Figure 1.2, with figures taken from the 2008 Netsize Guide.³

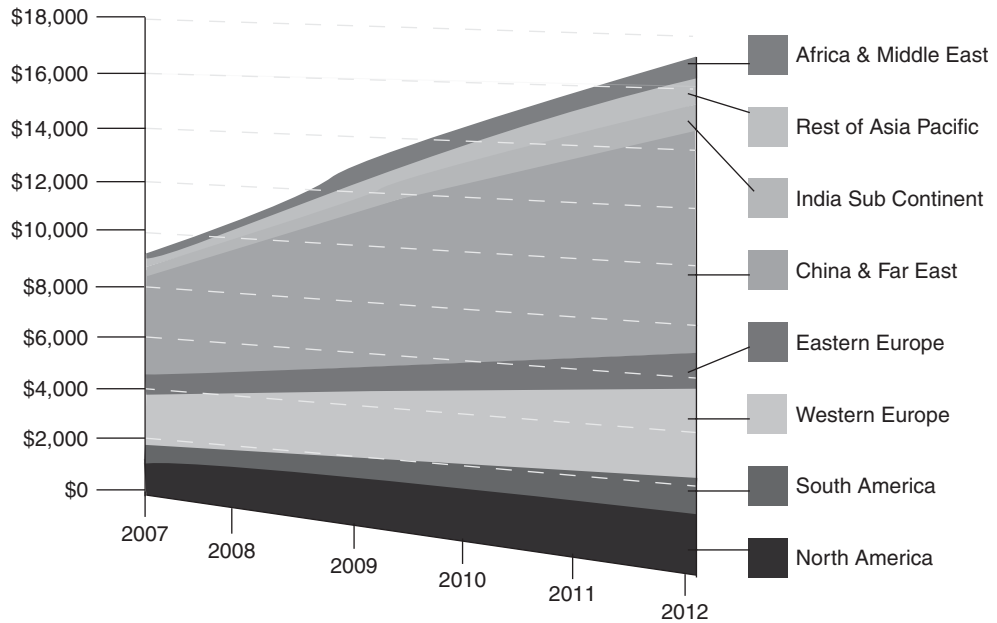
A lot of the revenue generated will belong to the content owners but the shifting landscape, combined with the rapidly increasing number of smartphones,⁴ will produce plenty of opportunities to develop new services and business models. New companies and brands can grow and market share is likely to shift between established brands as everyone competes for the attention of the mobile consumer. One great weapon in that competition is the ability to provide an optimal experience for the end user with a customized client or player application. To this end, Symbian empowers developers by providing the means to extend and enhance the features of the latest smartphones with applications and utilities. The part of Symbian OS concerned with multimedia content creation and consumption, known as the multimedia subsystem, is the subject of the rest of this book.

¹See www.pwc.com/extweb/ncpressrelease.nsf/docid/E042C329AE028974852573010051F342.

²In Asia, ringback tones are provided by a popular subscription service: users select the music that is heard by callers while they're waiting for an answer.

³See www.netsize.com.

⁴Cumulative sales are forecast to pass the 1 billion mark by 2011 according to both the Yankee Group and Canalys (www.symbian.com/about/fastfacts.html).



Source: Juniper Research

Figure 1.2 Total user-generated revenues (\$M) from mobile music (forecast for 2007–12)

1.3 Symbian OS

Symbian creates and licenses Symbian OS, the market leading, open operating system for mobile phones. It's not the only smartphone operating system you can target with multimedia applications and services but it is by far the most popular. According to Canalys,⁵ in 2007, Symbian had a 67% share of the market for converged device operating systems (Canalys combines smartphones with other wireless handhelds in their statistics). For comparison, next in the list was Microsoft with 13%.

Symbian does not make smartphones itself, but works with its licensees, which number the world's largest handset manufacturers. Those that shipped Symbian smartphones in Q4 2007 are Fujitsu, LG Electronics, Motorola, Mitsubishi Electric, Nokia, Samsung, Sharp and Sony Ericsson. As of June 2008, these licensees have shipped over 235 different models based on Symbian OS. Sales of these smartphones are accelerating rapidly; it took eight years to ship the first 100 million Symbian smartphones and only a further 18 months to ship the next 100 million. With global mobile phone shipments now well over 1 billion a year

⁵See www.canalys.com/pr/2008/r2008021.htm.

and Symbian OS providing new features to enable manufacturers to target mid-range, as well as high-end, products, that growth looks set to continue for some time to come.

For handset manufacturers, Symbian OS enables rapid integration with new device platforms, via a large ecosystem of hardware and software product and service provider partners. An application developer can get access to the new capabilities of the platform using the open APIs provided by Symbian and its partners. This powerful combination has kept Symbian OS smartphones at the forefront of mobile multimedia since the turn of the millennium.

Furthermore, as we prepare to take this book to press in June 2008, Symbian is transforming into the Symbian Foundation, to create an open and complete mobile software platform, which will be available for free. The Symbian Foundation will unify Symbian OS and its UI platforms, S60, UIQ and MOAP(S), to create an open software platform for converged mobile devices, enabling the whole mobile ecosystem to accelerate innovation. Further information is available from **www.symbianfoundation.org**.

1.4 The Cutting Edge

The first year of the new millennium was also the year the first ever Symbian smartphone went on sale. The Ericsson R380, as it was called, was the size of a standard phone but the keyboard folded back to reveal a touchscreen. It combined a mobile phone with a number of personal organization tools, including an address book, a calendar, email, voice memo and a note pad. These had previously been found in PDAs such as the Psion Series 5, which was built on an earlier version of Symbian OS. However, the R380 was 'closed' which means that no additional applications could be installed to the phone. The operating system supported the installation of aftermarket software, but it was not permitted on this device.

In 2001, we saw the launch of the first open Symbian smartphone, the Nokia 9210 Communicator. It was basically just a PDA and a mobile phone in a single package, aimed at high-end business users. Apart from a web browser, it shipped with very little multimedia functionality. However, thanks to the open, programmable nature of Symbian OS, third-party application developers rushed to fill the void and, within a few months, music and video players were available for the device. The same year also saw the launch of the first smartphone with a camera, the Nokia 7650. The digital camera and smartphone combined into one

device was marketed as an ‘imaging phone’. For reference, 2001 was also the year that Apple launched the first iPod.

The second half of 2002 saw Archos release the first portable media player (PMP) and this was closely followed by the launch of the Sony Ericsson P800 with a touchscreen, camera and support for many popular multimedia formats as well as streaming video from the Internet direct to the mobile device. Around the same time, the first PDA with built-in personal navigation functionality via GPS was released. In 2003, Motorola launched their first Symbian smartphone, the A920, which included built-in assisted GPS and fast 3G network access, as well as rich multimedia functionality.

In 2004, we had increasing processing power, storage space and camera resolutions in our Symbian smartphones, as well as built-in 3D graphics engines. It’s worth pausing to note that the cutting edge of graphics, although related to multimedia, is usually driven by game development. There is a separate graphics subsystem in Symbian OS and a separate book⁶ that covers it – it is not discussed further in this book.

There was a landmark for mobile multimedia in 2005 with the launch of Nokia’s Nseries range,⁷ described as ‘multimedia computers’. With camera resolutions ranging from two to five megapixels in the latest models, many with branded Carl Zeiss optics and some having auto-focus and even optical zoom capability, these devices are credible alternatives to a dedicated digital camera for all but the serious photography enthusiast or professional. The range targets different segments with devices optimized for music, still photography, video and games. Figure 1.3 shows a range of some of the most recent Symbian smartphones.

At the time of writing, the current flagship device, the Nokia N96, boasts HSDPA for download speeds of up to 3.6Mbps, WiFi, a five-megapixel camera with auto-focus and flash, built-in GPS, DVB-H TV broadcast receiver, TV out, support for all of the popular media formats and much more.

Similarly, other manufacturers are keen to promote the multimedia capabilities of their handsets. For example, Sony Ericsson is leveraging established brands, Walkman and Cybershot, from Sony’s music player and camera businesses to enhance the perception of its mobile phone offerings. Motorola is actively marketing the advanced video capabilities of its MOTORIZR Z8 and MOTO Z10 smartphones, emphasizing their capability to capture and edit video footage without the need for a PC. Motorola also bundles full-length feature films on removable storage with those products in some markets.

⁶Games on Symbian OS by Jo Stichbury *et al.* (2008). See developer.symbian.com/gamesbook for more information.

⁷See www.nseries.com for more details.



Figure 1.3 A range of Symbian smartphones (June 2008)

Smartphone multimedia functionality will continue to advance in the years ahead with new hardware in the form of advanced graphics and video accelerators, audio data engines, and faster processors with multiple cores. These will enable high-definition video output, rich games experiences, more advanced audio effects and video editing as well as longer playback times, to rival existing dedicated players. Of course, to keep Symbian smartphones on the cutting edge of mobile multimedia, the multimedia subsystem in Symbian OS has had to keep evolving.

1.5 Evolution of the Multimedia Subsystem in Symbian OS

Having explained the background against which Symbian OS has developed, we're going to take a closer look at its multimedia subsystem and how it has evolved, in order to better understand its current design and use. Readers who are either familiar with the history or not interested in the technical details can safely skip ahead to Section 1.6 for a peek into the future!

If you're not already aware of the history of Symbian OS releases, then we'd recommend taking a look at a visual timeline which you can

find on the wiki site for this book at [**developer.symbian.com/multimedia-book_wikipedia**](http://developer.symbian.com/multimedia-book_wikipedia).

In order to explain why many radical improvements were made to the multimedia subsystem in Symbian OS v7.0s and then further improvements in the later versions, it is necessary to give an overview of the subsystem in previous versions of the operating system and the problems they faced.

1.5.1 The Media Server

In early Symbian OS releases, v6.1 and v7.0, all multimedia processing was performed through the Media Server. This was a standard Symbian OS server; it operated in a single thread using an event-driven framework (an active scheduler with multiple active objects) that provided all multimedia functionality.

The server supported the playback and recording of audio, along with the encoding, decoding and manipulation of still images. Symbian OS shipped with support for an array of audio and image formats, which could be extended by writing proprietary plug-ins.

In order to use the server, a client could either explicitly instantiate a connection to the server or allow the client APIs to provide a connection automatically. Each client would supply an observer class to the server, which would allow the server to communicate by passing messages back to the calling application or library.

The server kept a list of clients and concurrently cycled through the multimedia requests. This meant that a number of different clients could use the server at the same time, and, for example, enabled an application to play back audio whilst simultaneously decoding images for display. Although this sounds ideal, the practical issues in producing this kind of behavior were complicated and fraught with difficulties.

If, for example, an application actually did try to use two parts of the server's functionality at the same time, the latency created by having one process controlling both could make the system virtually unusable. For instance, an intensive task, such as decoding an image, would interfere with any real-time multimedia task. The situation was made worse when poorly written third-party plug-ins were used (they were often converted from code not originally written for Symbian OS and failed to fit the co-operative multi-tasking model). The plug-in framework itself was also extremely complicated to write for, which did not improve the situation.

Handling the demands of high-performance multimedia applications such as streaming video, CD-quality audio, mobile commerce or location-based services, coupled with the fact that connecting to the server could take, in the worst case, several seconds, meant improvements were necessary.

1.5.2 The Multimedia Framework: The Beginning of a New Era

In 2001, Symbian began to write an entirely new multimedia subsystem that would successfully allow different areas of the subsystem to be used simultaneously and would provide a lightweight framework, redesigned from the ground up for mobile media, with powerful enhancements such as multiple threads, format recognition, streaming, and a plug-in media component library. With a new foundation of base media classes and an extensible controller framework, licensees and third-party developers could now undertake far more effective multimedia application development for Symbian OS.

The new architecture was not based on a central server and instead split up the various multimedia sections so that each application could use the functionality it needed independently. It also used multiple concurrent threads, avoiding the side effects that were seen in the Media Server.

The new subsystem retained many of the same client interfaces as the Media Server, but took advantage of a new plug-in-resolving methodology, known as ECOM (which is discussed further in Chapter 2). An architecture based on plug-ins provides the flexibility for additions to be made to the built-in functionality, which allows greater extensibility by creating support for a wide range of plug-in implementations. This translates into an open multimedia platform, enabling the implementation of specialized proprietary components, by both licensees and third-party developers.

The new subsystem was so successful that it was quickly integrated into Symbian OS v7.0s, which was just beginning its development lifecycle. As development of that release progressed, the audio and video parts of the subsystem evolved into what is now known as the Multimedia Framework (MMF). The MMF is a multithreaded framework geared towards multimedia plug-in writers. While it retains a subset of the original application programming interfaces (APIs) from v6.1 and v7.0, it also provides numerous enhancements.

The basic structure of the MMF consists of a client API layer, a controller framework, controller plug-ins and lower-level subsystems that will be implemented by a licensee when a new smartphone is created (which are generally hardware specific).

As Figure 1.4 shows, still-image processing is handled by a separate part of the subsystem, the Image Conversion Library (ICL), while still-image capture and camera viewfinder functionality are handled by the onboard camera API (ECam). Neither of these are part of the MMF. Video recording, despite effectively being the rapid capture and processing of multiple still images, is typically implemented using an MMF camcorder plug-in,⁸ which drives the necessary encoding process and controls

⁸On smartphones with high-resolution video recording capabilities, the capture and encoding is often performed on a dedicated hardware accelerator.

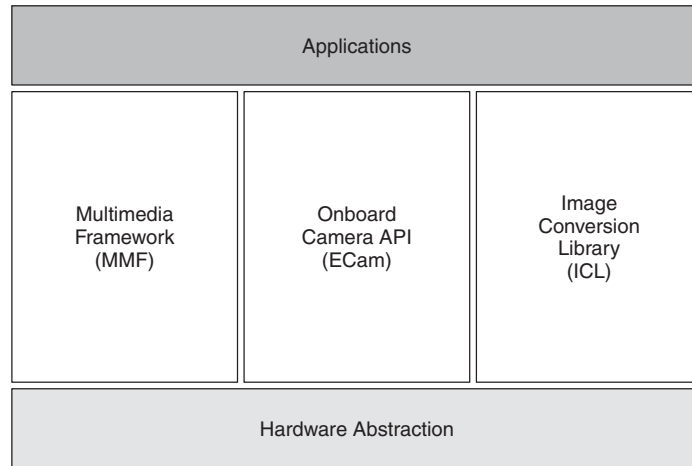


Figure 1.4 Symbian OS v7.0s Multimedia subsystem

transfer of the video stream from the camera and microphone to a file. Such an implementation was chosen to extend the ability of the camcorder application to use the pool of available encoding formats via the MMF.

In Symbian OS v8, the multimedia framework provides audio recording and playback and audio streaming functionality. Audio format support provided by Symbian includes WAV, AU, RAW (in various formats), PCM, μ -Law, A-Law, etc. Licensees typically add their own support for popular formats, such as MP3 and AAC, often with implementations optimized for their particular hardware.

Support is also provided for video recording, playback and streaming. The framework allows developers to write efficient and powerful plug-ins, using a range of re-usable software components that represent files, memory buffers, sockets, audio devices, screens, etc. In addition, a MIDI client API is available. The lower level of the subsystem, known as the Media Device Framework (MDF) was added in Symbian OS v8.0; it includes support for audio, video and MIDI as well as providing transparent support for hardware acceleration. Figure 1.5 shows the interaction between the MMF and the MDF.

The release of Symbian OS v9 marked a major turning point, with the introduction of an enhanced security architecture. This architecture provides the platform with the ability to defend itself against malicious or badly implemented programs (malware).

In the multimedia subsystem, process boundaries were added to allow enforcement of the platform security constraints. For example, there is now a client–server boundary in the middle of the hardware abstraction layer to restrict priority requests for the audio hardware to clients with the appropriate privilege.

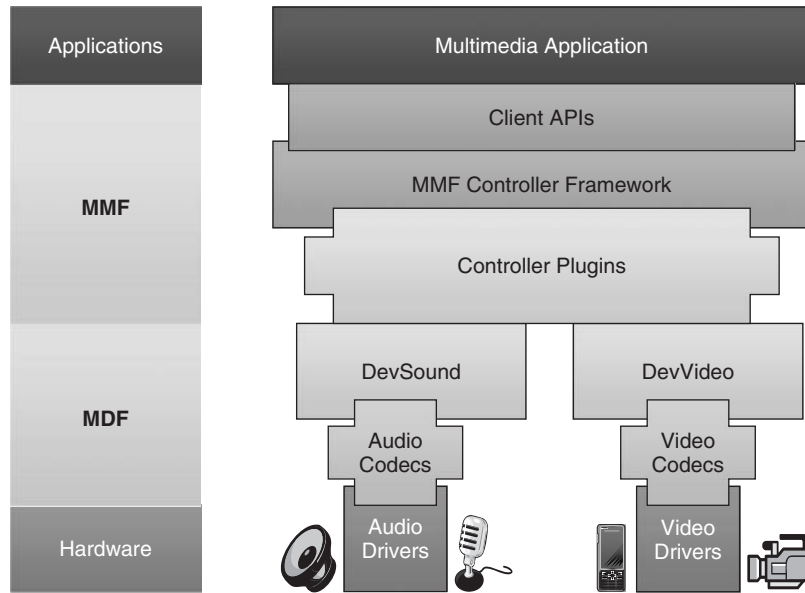


Figure 1.5 Multimedia Framework and Media Device Framework

The multimedia subsystem on the latest versions of Symbian OS, v9.4 and v9.5, provides support for rich multimedia capabilities: audio and video recording; playback and streaming; cameras with high resolutions; audio adaptation, routing and policy; use of audio, video and graphics hardware acceleration; digital TV and FM/digital radio support.

1.6 A Peek into the Future

The global smartphone market has never been so exciting. With over 200 million Symbian smartphones shipped, high smartphone sales growth in developing markets, and increasing mass market requirements, Symbian's addressable market is broadening across segments and regions.⁹ This rapid growth in shipments driven by a shift towards mid-range handsets, along with ever-increasing consumer demand for high-end multimedia features, leads the future development of Symbian OS.

The multimedia subsystem's future development will be driven by a number of factors:

- Power management – multimedia features consume lots of power and this is only increased by trends towards higher screen resolutions and

⁹See www.symbian.com/about/fastfacts.html to get the latest figures.

lower bulk memory costs enabling higher quality content consumption. These issues can be mitigated by intelligent design of software subsystems and hardware components. The subsystems need to be scalable so that they take advantage of hardware acceleration where it is present but work purely in software on mid-range devices. Examples of this are the Media Device Framework, described in Section 2.4, Section 4.2 and Section 5.7, and the new graphics architecture, ScreenPlay.¹⁰ Future smartphones are likely to be powered by multi-core processors. Symbian is adding support for symmetric multi-processing (SMP), which will improve battery life by accessing multiple cores only while running demanding high-end multimedia applications and powering them down when they are not in use. Symbian continue to optimize performance in these areas.

- High-bandwidth networks – cellular networks with LTE and super 3G as well as WiFi and WiMAX access will enable large quantities of high quality content to be transferred over the air for immediate consumption. Symbian is enabling super-fast content downloads on these networks with a new IP networking architecture, FreeWay.¹¹
- The home network – a totally connected home where your content may be stored on your PC, but played on your stereo in another room, all orchestrated by your mobile device. This functionality is enabled by universal plug and play (UPnP) technology. The mobile device is also able to originate and consume media. As the number of connected devices in the home grows, the demands on the user interface and media playback capabilities of mobile devices will increase significantly.
- User-generated content – sites such as YouTube demonstrate the interest in this feature, and mobile devices are ideal for capturing this content because you rarely forget to take your phone with you. Mobile social networking is also likely to drive the creation and consumption of multimedia content. To enable this, users are likely to demand better on-device multimedia editing tools.
- Open standards – the Khronos group is defining a number of open standard, royalty-free APIs to enable authoring and playback of dynamic media on a wide variety of platforms and devices. Symbian and several of its licensees are members of the group and are likely to adopt the standards as and when they become available.¹²
- More memory – currently a lot of multimedia applications are limited by the available RAM on the device. This often restricts the size

¹⁰See www.symbian.com/symbianos/os_screenplay.html for more information.

¹¹See www.symbian.com/symbianos/os_freeway.html for more information.

¹²Chapter 2 has more information on upcoming support for Khronos standards in Symbian OS.

and quality of images and video content that can be manipulated. In addition to the gradual improvement in device specifications, Symbian is making a major improvement in system RAM usage with the delivery of demand paging.¹³ Code on a Symbian smartphone is stored in NAND Flash memory and must be copied into RAM before it can execute. Demand paging means that only the required 'pages' of code within a DLL are kept in RAM; previously the entire DLL would need to remain loaded. This should free up more memory for application developers, enabling more advanced content manipulation.

Now, we invite you to continue with us to the next chapters of the book, where we take a closer look at the multimedia subsystem, its architecture and each of its building blocks.

¹³For details of the demand paging system and its benefits, see www.iqmagazineonline.com/article.php?crumb=SEARCH&issue=23&article_id=706.

2

Multimedia Architecture

The current multimedia architecture was introduced in Symbian OS v7.0s and has been extended and improved in subsequent releases. This chapter provides an overview of the architecture and its underlying functionality, followed by a description of the separate parts of the multimedia subsystem and the associated multimedia APIs available.

Among other things, the Symbian OS multimedia APIs can be used to: play and record audio and video data; perform image manipulation and processing; access the camera hardware; access the radio tuner hardware.

However, before getting into the Symbian OS multimedia architecture, we take a quick look at the following Symbian OS concepts that are commonly used while writing a multimedia application:

- ECOM – a framework which allows an easy way to write extensible applications
- platform security – a significant addition to Symbian OS v9, providing a defense mechanism against malicious or badly written code
- the Content Access Framework (CAF) – a framework which provides applications with a common method of accessing content wherever and however it is stored on the device.

2.1 The ECOM Framework

The ECOM framework is the Symbian OS plug-in framework. It is intended to be synonymous with Microsoft's Component Object Model (COM) architecture, with the first letter coming from the original name of Symbian OS – EPOC32 – by convention with other Symbian components, such as ESock, ETel and EKern.

A plug-in is a way to extend the functionality of an application by allowing different implementations of a defined interface to be provided.

Many Symbian OS system services use ECOM plug-ins; for example, the messaging architecture uses ECOM plug-ins to extend support for new message types and the socket server uses plug-ins to provide support for different networking protocols. Of more relevance to this discussion, the Symbian OS multimedia framework (Section 2.4.1) uses ECOM plug-ins to enable the playback and recording of different audio and video formats, while the image conversion library (Section 2.4.3) uses ECOM to provide support for image formats, such as JPEG, PNG and TIFF.

Using a framework that allows the loading of custom modules means that Symbian OS can be extended by phone manufacturers and third parties. Support for different multimedia formats can be added as necessary, independent of the general multimedia implementation supplied by Symbian OS.

On early versions of Symbian OS, different multimedia formats were handled through the use of generic polymorphic interface DLLs and a multimedia framework that was able to discover and instantiate each available implementation. ECOM provides a further abstraction and the multimedia subsystem no longer takes responsibility for determining which plug-ins are available, deferring instead to ECOM.

ECOM provides a single mechanism to:

- register and discover interface implementations
- select an appropriate implementation to use
- provide version control for the plug-ins.

In fact, an ECOM plug-in is just a particular type of polymorphic Symbian OS DLL, which is identified by a unique identifier built into the binary through the Symbian OS tool chain. Further detail about the role of ECOM within Symbian OS and how to write an ECOM plug-in can be found within the Symbian Developer Library documentation, for example, in the System Libraries documentation within the Symbian OS Guide. Your chosen SDK also provides an example ECOM plug-in implementation, in the `syslibs` directory of the Symbian OS example set.

2.2 Platform Security

Platform security was introduced in Symbian OS v9.1 in order to preserve the open nature of the platform, while protecting users from security threats (such as those typically seen with desktop computers). The platform security model prevents applications from gaining unauthorized access to Symbian OS services, the hardware, or system and user data.

The Symbian OS platform security architecture is based on a trust model. One of the most fundamental concepts in platform security is

the definition of the *unit of trust*. Symbian OS defines a process as the smallest unit of trust. The phone hardware raises a processor fault if access is made in a process to an address not in the virtual address space of that particular process. Symbian OS can trust that a process cannot directly access any other virtual address space, because the hardware prevents it. Thus hardware-assisted protection provides the basis of the software security model.

Platform security prevents software running on Symbian OS from acting in an unacceptable way, intentionally (by malware) or unintentionally. It is a system-wide concept which has an impact on all developers, whether they work on applications, middleware or device drivers.

On Symbian OS, platform security controls what a process can do and restricts its activities to those for which it has the appropriate privileges. The system is divided into tiers of trust and these tiers combine themselves in pairs to form the device's system software and other software that runs on top of the underlying platform, as follows:

- Trusted Computing Base (TCB) – the most trusted part of Symbian OS. Among other things, it ensures the proper operation of platform security by controlling the lowest level of the security mechanism. It includes the operating system kernel, kernel-side device drivers, the file server, the native software installer and the memory management unit (MMU). Only a few user libraries which are needed by the file server are included in the TCB. Any code that belongs to the TCB is reviewed very carefully.
- Trusted Computing Environment (TCE) – trusted software including the rest of Symbian OS and some provided by other suppliers, such as the UI platform provider and the phone manufacturer. Functionality exposed by the TCE is usually implemented in server processes that have only the required privileges. This restricts access to low-level operations and prevents misuse by other processes.

Application software is not usually part of the TCE. In order to utilize system services, it accesses the APIs provided by servers in the TCE. For example, a music player application would not normally communicate directly with the hardware and therefore does not need the same privilege as parts of the multimedia framework and audio device drivers. The application uses the APIs provided by the multimedia framework to access services within the TCE. Privileges for audio hardware access are typically enforced across the audio server process boundary. (The audio server hosts DevSound,¹ see Figure 2.4.) Alternatively a device manufacturer may prefer to enforce them at the driver process boundary.

¹ DevSound is part of the Media Device Framework, described in Section 2.4.2.

- Other trusted (signed) software – Most application software lies outside of the TCE tier, however this software often needs certain privileges to use the services that the TCE provides. It is important that any application requesting a service be considered trustworthy before the request is granted. A measure of this trust is that the software has been signed by a trusted certification authority to grant it the privileges it needs. More information about signing can be found on the Symbian Signed portal, ***www.symbiansigned.com***.
- Other software – unsigned or self-signed, and therefore untrusted, software. A user can install software to his Symbian smartphone that is untrusted (meaning that it is not signed or, more commonly, that it is ‘self-signed’). This means that it has not been signed by one of Symbian’s trusted authorities so the identity of the author has not been verified: it does not necessarily mean that the software is malicious or worthless. The software can be installed and run on the phone but cannot perform any actions which require security privileges, except where those privileges are grantable by the user. An example of software that could be self-signed is a Solitaire game that does not perform any actions that access sensitive user data or system-critical data or services.

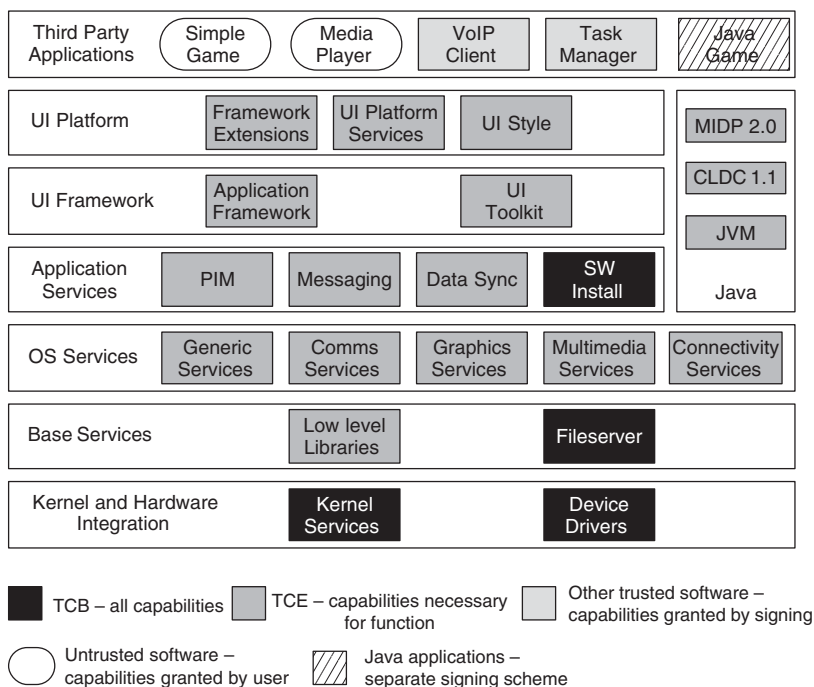


Figure 2.1 Tiers of trust

Figure 2.1 shows the components of a Symbian OS smartphone platform and the four tiers of trust: the TCB, the TCE, other trusted software and the rest of the Symbian OS platform. Note that the applications shown are chosen as typical examples of applications that do and don't require signing – some games and media players may use sensitive capabilities and would therefore need to be signed.

2.2.1 Capabilities as Privileges

Platform security is built around the use of capabilities to represent access privileges. A capability is a statement of trust. Every executable (EXEs and DLLs) is tagged with some capabilities at build time and at run time. Every process is granted a set of capabilities. The capabilities are assigned based on which APIs a process needs and is authorized to use.

The kernel holds a list of capabilities for every running process and a process may ask the kernel to check the capabilities of another process before deciding whether to carry out a service on its behalf. For installable software, the installer acts as a gatekeeper and validates that the program is authorized to use the capabilities it was built with. The authorization can be a digital signature or user permission. There are four families of capabilities:

- User-grantable capabilities relate to security that the user can comprehend and make a choice about; for example, a user can decide whether to install software that accesses his personal data or not.
- System capabilities allow a process to access sensitive operations. They protect system services, device settings and some hardware features. Installable software which needs system capabilities should be granted by a trusted certification authority such as Symbian Signed.
- Restricted capabilities require a higher level of trust and protect, for example, the file system, communications and multimedia device services. They are restricted because of the potential to interfere with core functions of the device, such as making calls or playing tones to alert the user. The publisher of installable software that requires restricted capabilities must be identified to Symbian Signed via a Publisher ID.
- Device manufacturer capabilities (TCB, AllFiles and DRM) are the most sensitive. They either protect the integrity of the platform security system or could expose the device manufacturer to legal liabilities. For this reason, capabilities in this group may only be granted with the approval of the device manufacturer.

In addition to protecting access to critical services on the phone, platform security also provides a data caging mechanism, which allows

an application to store data in a folder reserved for it that can not be accessed by other applications (except for applications trusted with the `AllFiles` capability).

2.2.2 Implications for Multimedia Developers

The majority of multimedia application developers only need user-grantable capabilities and their applications don't need to go through an official signing process – that is they can be self-signed. For example, to read media files from any public directory on the device `ReadUserData` is required and to modify a playlist requires `WriteUserData`. These are likely to be the only capabilities required for a playback application. It can even be possible to request the system to play content protected by digital rights management, without the application requiring the `DRM` capability.² In addition, any application that wants to record multimedia content requires the `UserEnvironment` capability, which is also user-grantable.

However, the situation is very different for developers attempting to extend the multimedia framework via `ECOM` plug-ins. As we described in Section 2.1, an `ECOM` plug-in is a DLL. Platform security imposes the following rule: a process can only load a DLL if that DLL has been trusted with at least the capabilities that the process has.

For example, if an application is trusted with the `DRM` capability, then any DLL loaded by that application would execute with that capability (since it is the application process that is trusted and the DLL runs within it). Therefore the DLL must also be trusted with `DRM` otherwise it would represent a security hole that could allow a developer of untrusted software to access the unencrypted contents of protected files, simply by being loaded into a more trusted process. The same argument applies to other capabilities, not just those relating to multimedia, so plug-ins for any frameworks that could be used by a number of applications need to be trusted with a broad set of capabilities (in practice, often all but the `TCB` capability).

Plug-in developers should be aware that the rule does not work in reverse. A DLL trusted with a large number of capabilities may be loaded by an application with none at all. However, in that situation, the plug-in code executes with no capabilities, since the process is not trusted with any. The DLL needs to be able to handle the fact that any API calls it makes that require capabilities may fail, typically with `KErrPermissionDenied`.

²This would involve requesting another process which does have the capability to play the content for you. S60 provides the `CDrmPlayerUtility` for this purpose from 3rd Edition onwards. The class is defined in `drmaudiosampleplayer.h` and you need to link against `drmaudioplayutility.lib` to use it. For more information, see the relevant SDK documentation. Section 2.3 discusses DRM on Symbian OS in more detail.

More information about platform security can be found in *Symbian OS Platform Security* by Craig Heath *et al.* (2006) and in the Symbian Developer Library documentation found online at developer.symbian.com or in your chosen SDK.

2.3 The Content Access Framework

One of the key benefits of platform security for content owners and distributors is the provision of a solid foundation for Digital Rights Management (DRM) systems. DRM systems are very important to the entertainment industry for secure information distribution, as they allow content owners to specify and control the usage policy for their content. For example, DRM prevents one person from buying a music album and then giving away or selling the music.

Device manufacturers have historically implemented their own support for proprietary or industry-standard DRM solutions (e.g. OMA DRM v1.0). This made life difficult for application developers, who were either unable to access certain content or had to use different methods to do so on different platforms. Fortunately, from Symbian OS v9.1 onwards DRM-protected audio and video content are supported via the Content Access Framework (CAF). The Content Access Framework provides a common interface for applications to access content regardless of whether the content is protected or not. The landscape of DRM changes constantly and any particular Symbian smartphone can implement different DRM schemes.³ CAF allows the media player to be oblivious to the particular protection scheme by providing a common API layer.

Internally, the CAF behaves as a switch between different agents, known as Content Access Agents. Each agent is an ECOM plug-in that implements one or more DRM schemes. CAF determines the corresponding agent dynamically for each file.

Symbian OS also provides a default agent for reading unprotected files. This way, media players can use the same APIs without detecting DRM presence at run time.

Since CAF agents are responsible for proper enforcement of the DRM scheme, they are likely to be separated into a client and a server, where the server ensures integrity of access rights. In this way, only the server process needs to be trusted with the DRM capability, while the client process never has direct access to unencrypted content. Due to the possible client–server split, if the media file is stored in the player’s private data cage, it should be passed to the CAF by file handle; otherwise the agent’s server might not be able to open the file. Figure 2.2 summarizes the top-level CAF architecture.

³At the time of writing, the most widespread DRM schemes supported by Symbian smartphones are OMA DRM v1, OMA DRM v2.0/2.1 and WMDRM.

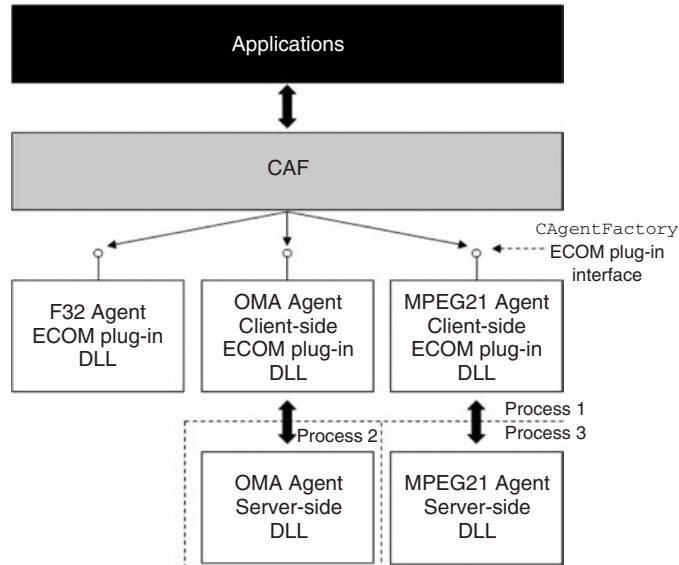


Figure 2.2 The Symbian OS CAF Architecture

Having presented the general framework for DRM support in Symbian OS, we move on to interfaces specific to multimedia content. The abstract class `TMMSource`⁴ provides a lightweight encapsulation of a DRM-protected content source. The audio and video clip playback APIs use `TMMSource` as an argument in their `Open()` file methods. There are two subclasses of `TMMSource` that are commonly used:

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

DRM schemes (e.g. OMA DRM v2.0) may allow several media tracks to be embedded in a single archive file. In this case, the particular content ID can be supplied to these classes in order to choose the track to be decoded.

To enforce the protection of the content, the agent must know what the client intends to do with the content once it has been supplied with the decoded version of that content.

The `TMMSource` class and its subclasses allow the developer of a multimedia player to provide a DRM access intent (play, execute, pause, etc.), that indicates what the software intends to do with the data. This affects whether or not it is granted access to the content.

⁴`TMMSource` is defined in `mm\mmcaf.h` and to use it you need to link against `mmcommon.lib`. More information about this class can be found in the Symbian Developer Library documentation.

DRM schemes may also adjust user's consumption rights. For example, an audio file may have a play count that allows the user to listen to the track a limited number of times. The intent needs to be carefully chosen by the audio player, so that rights are executed exactly when necessary. In order to access DRM-protected content encapsulated by one of the `TMMSource` subclasses, your process must have the DRM capability (a device manufacturer capability – see Section 2.2).

For more information about DRM support in Symbian OS and the Content Access Framework, see the Symbian Developer Library and the SDK documentation.

2.4 Multimedia Subsystem

The Symbian OS multimedia subsystem provides functionality for

- audio recording and playback
- video recording and playback
- still image conversion
- camera control
- FM radio receiver support.

While some of the features are provided by Symbian OS licensees and not by Symbian, the multimedia subsystem provides the basic frameworks to support them. Furthermore the multimedia subsystem provides cross-platform compatibility for client applications and hardware acceleration plug-in writers to reduce the cost of porting applications and codecs between different platforms.

The multimedia subsystem is a modular subsystem, constructed from discrete parts, each of which contributes to the whole. This enables a high degree of configurability across a wide range of target hardware. Figure 2.3 provides an overview of the different components that comprise the multimedia subsystem.

The four boxes at the top of the diagram represent multimedia applications. These can be provided by the UI platform, by the device manufacturer or by a third-party developer. The boxes underneath them represent the components of the multimedia subsystem. The image conversion library and the majority of its plug-ins are provided by Symbian OS. Device manufacturers may replace some of the plug-ins with their own optimized versions (see Chapter 6 for more details). Symbian provides the multimedia framework and whilst it is possible for anyone to write plug-ins for it, they are generally supplied by the device manufacturer.

Now let's take a closer look at each multimedia component.

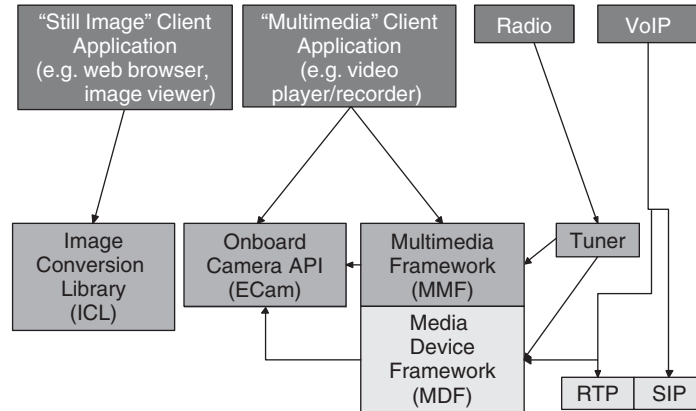


Figure 2.3 The multimedia ecosystem

2.4.1 The Multimedia Framework (MMF)

The multimedia framework (MMF) uses ECOM plug-ins to provide a multithreaded framework for handling audio and video playback and recording. It allows phone manufacturers and third parties to add plug-ins to extend the audio and video capabilities, for example adding support for new formats.

The MMF consists of the following basic layers:

- Client-side utility APIs provide applications with a simple interface to use functionality provided by plug-ins or for low-level audio streaming.
- The controller framework resolves selection and launching of plug-ins and also facilitates message passing between applications and plug-ins.
- The controller plug-ins are responsible for managing the data processing while moving data from the source (a file or URL) to the sink (or destination, hardware or a file). This includes writing or parsing the file format and, for video controllers, multiplexing or de-multiplexing the audio and video streams from or to the codecs.

The MMF is based on a client–server architecture, linking client-side utility APIs with the server-side controller plug-in interfaces. Applications use the client APIs to control the playback or recording of the media and the underlying controller framework creates a new thread to process the media data.

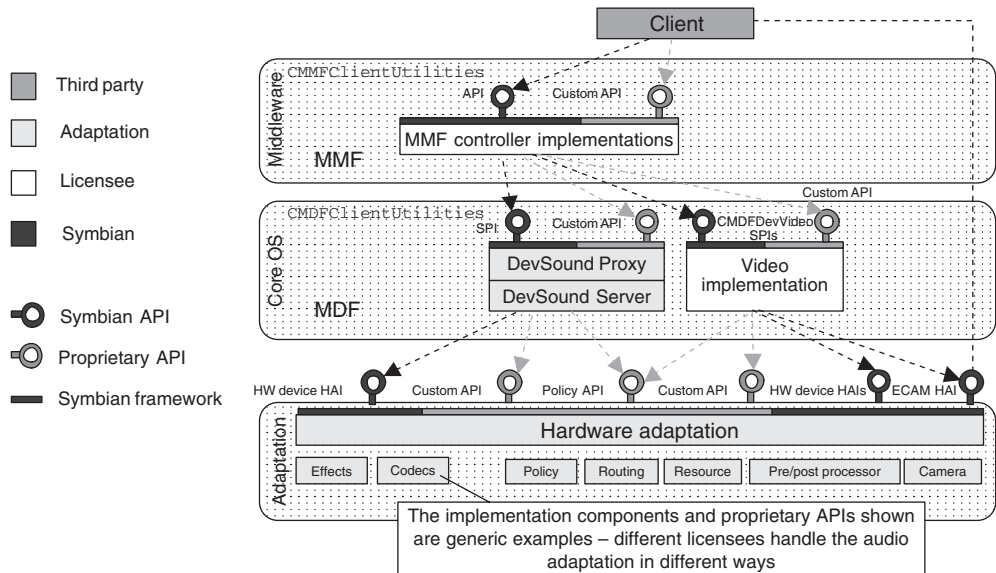


Figure 2.4 MMF architecture

How the MMF works

To have a better understanding of how it all works, see Figure 2.4. Now, let's have a look at a simple example – playing an audio file while no other application is using or attempting to use the audio device. The following sequence of actions occurs:

1. The user starts a player application and chooses to play an MP3 file.
2. The play request is forwarded to the controller framework by the audio client utility that the player application invokes.
3. The controller framework loads a controller plug-in that handles MP3 files.
4. The controller plug-in reads the data from the source (a file, memory or a URL).
5. The controller plug-in performs any processing required to extract buffers of compressed⁵ audio data from the source and pass them to a data sink (in this case, DevSound) continuing until the application stops the playback or the end of the file is reached.

⁵In some devices, the data may be decoded by an MMF codec plug-in before being passed to DevSound for playing. This is no longer recommended, but the codec plug-ins for some formats may still be available for use by third-party developers on some devices.

6. A codec in the DevSound layer decodes the audio to a raw PCM (pulse-code modulation) format that can be played by the audio hardware.

In the case of audio recording, the data flow would be in the opposite direction. The user chooses to record to a file and either the user or application selects an appropriate compressed format, such as Adaptive Multi-Rate (AMR). As before, a client utility is invoked, which causes the controller framework to load the appropriate plug-in. The plug-in then sets up DevSound for recording. The audio hardware converts the incoming sounds to PCM and they are then encoded to the compressed format within DevSound. The compressed data is passed back to the plug-in, one buffer at a time, where it is written to a file with any necessary headers added.

For the application programmer, the MMF provides APIs that abstract away from the underlying hardware, thereby simplifying the code needed to play and record the supported formats.

MMF Client-side APIs

The MMF provides a client API consisting of several interfaces that simplify the interaction with the underlying plug-ins, enabling the application writer to easily manipulate audio and video features. The client APIs all use the Observer design pattern. You have to both instantiate the utility class and implement an observer interface to receive asynchronous events from it. The features made available by the client utility APIs are:

- Audio playing, recording, and conversion – An interface consisting of three classes (and their respective observer classes); `CMdaAudioPlayerUtility`, `CMdaAudioRecorderUtility`, and `CMdaAudioConvertUtility`. These classes provide methods to create, play, and manipulate audio data stored in files, descriptors, and URLs.
- Audio streaming – An interface consisting of two classes; `CMdaAudioInputStream` and `CMdaAudioOutputStream`. These classes are a thin wrapper over the DevSound layer that provide extra buffer management functionality for client applications.
- Tone playing – An interface consisting of a single class `CMdaAudioToneUtility`. This class provides methods for playing and configuring single and sequenced tones as well as DTMF (Dual Tone Multi-Frequency) strings.
- Video playing and recording – An interface consisting of two classes `CVideoPlayerUtility` and `CVideoRecorderUtility`. These classes provide methods to create, play, and manipulate video clips with or without audio tracks stored in files, descriptors, and URLs.

Note that these interfaces are generic and developers cannot rely on all of the functionality being available on a particular device. For example, playing audio from a URL is generally not supported and recording audio or video to a URL has not been implemented by any manufacturer either. Additionally, the existence of two audio codecs on a device is not sufficient to ensure that the `CMdaAudioConvertUtility` can convert from one format to the other. Developers should always check the supported functionality on their target devices before planning to use a specific function.

The client APIs can be roughly split into two types; the first type are APIs that deal with clips. In this case, the MMF controller is responsible for reading the media data from or writing the data to the clip. The clip may be passed to the APIs by filename, URL or directly in a descriptor filled with data.

The second type are streaming APIs that do not deal with clips. They allow a client to pass audio data into the API or read data directly from it, bypassing the controller framework. The audio data is streamed into and out of the API, it is not 'streaming' in the sense of streaming data over an Internet connection.

MMF Controller Framework

The controller framework is a framework for specific multimedia plug-ins called controller plug-ins. A controller plug-in's main role is to manage the data flow between the data sources and sinks. For example, an audio controller plug-in can take data from a file – the source – and output it to a speaker – the sink – for playback. It could take data from a microphone source and save it into a file sink.

It's possible for a controller plug-in to support multiple media formats. For example, a single audio controller plug-in may support WAV, MP3, and AMR formats. This is possible since each controller may use multiple codecs, either as separate plug-ins or via direct support for them in the Media Device Framework (see Section 2.4.2). More typically each format has a separate controller plug-in.

More information about the MMF controller framework can be found in Chapters 4 and 5.

2.4.2 Media Device Framework (MDF)

The Media Device Framework (MDF) is the layer below the controller plug-ins. It provides a hardware abstraction layer that is tailored to each hardware platform on which the MMF runs. It consists of `DevSound` for audio and `DevVideo` for video and includes:

- interfaces for finding, loading and configuring codecs

- functionality for encoding, decoding, pre-processing and post-processing audio and video
- an audio policy that resolves the access priority for simultaneous client requests to use the sound device (for example, to determine what happens if the phone rings while we are playing music); the audio policy is often customized by the device manufacturer for each hardware platform, so it may behave differently on different devices
- a hardware device API that interfaces the low-level hardware devices including optional hardware-accelerated codecs residing on a DSP or other dedicated hardware.

The MDF contains codecs in the form of hardware device plug-ins; they are mainly provided by the device manufacturers to make use of the specific hardware present on the phone. Despite the name, hardware device plug-ins may be implemented entirely in software when no suitable hardware is available. You can read more about DevVideo and DevSound in Chapters 4 and 5, respectively. Details of the hardware device plug-ins and implementation of the adaptation layer are beyond the scope of this book, because they are the domain of device manufacturers only.

2.4.3 Image Conversion Library (ICL)

The image conversion library (ICL) provides support for still image decoding, encoding, scaling and rotation. It provides facilities to convert single and multiframe images stored in files or descriptors to CFbsBitmap objects. The library also provides facilities to convert single-frame images from CFbsBitmap objects to files or descriptors in a number of formats.

The ICL is a library based on ECOM plug-ins. It allows third parties and phone manufacturers to add plug-ins to support more formats or to take advantage of hardware-accelerated codecs available on the device.

Symbian provides a comprehensive set of ICL plug-ins to facilitate some of the more common image formats, such as JPEG, GIF, BMP, WBMP, TIFF, PNG, WMF, Windows icons and over-the-air bitmaps. Device manufacturers may choose to remove, replace or add to the Symbian supplied plug-ins but most are usually left as standard (see Chapter 6).

Figure 2.5 shows the ICL architecture. The client application does not access the plug-in directly as all communication is done via the ICL client APIs – the CImageDecoder, CBufferedImageDecoder, CImageEncoder and CImageTransform classes.

The ICL framework provides a communication layer between the client APIs and the actual ICL plug-in, and all plug-in control is performed via this layer. The ICL framework can automatically load the relevant ICL

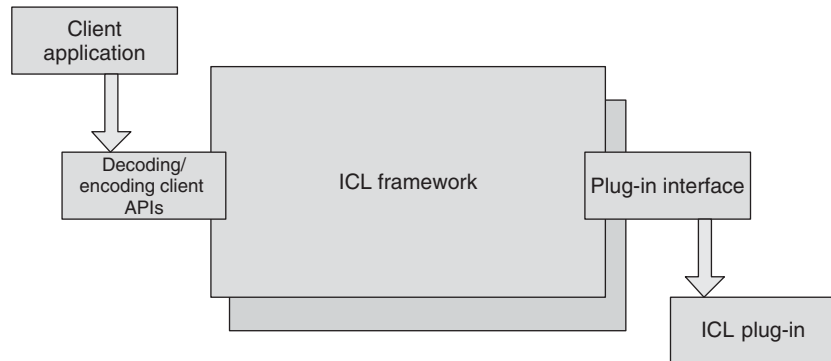


Figure 2.5 ICL architecture

plug-in needed for the chosen image processing, identifying an image for the purposes of plug-in assignment by its type (BMP, GIF, JPEG, etc).

Chapter 6 gives a detailed explanation of the ICL and how it can be used. Client usage is discussed for common file formats, image encoding, image decoding, displaying images and transforming images.

2.4.4 Onboard Camera API (ECam)

The Onboard Camera API (ECam) is a common API that provides the means to control any digital camera on a mobile phone and to request and receive specific image data from it.

It enables capturing of single or multiple frames and configuration of the camera's settings, such as orientation, image format, capture size, zoom, exposure, white balance, etc.

Symbian OS provides the onboard camera API definition to ensure source compatibility. The actual implementation of the underlying camera software is not supplied by Symbian OS as it is highly dependent on the hardware architecture of each mobile device. Each handset manufacturer provides a separate camera implementation, providing access to it through the onboard camera API, CCamera API. This gives application developers a consistent interface regardless of the actual device on which the camera is present.

The onboard camera API can be accessed directly by the client application for still image capture, viewfinder and settings control. It can also be accessed indirectly, via the MMF, which uses it to capture video frames before encoding them. For still image capture it is fairly straightforward: the client application requests image capture directly through the onboard camera API. In the case of video capture, the client application usually accesses the camera API directly, to control the viewfinder and video capture settings, but also indirectly, by sharing a handle to the camera with the MMF for video recording.

If you are writing an application that uses the camera of the mobile phone, I invite you to look at Chapter 3, which contains more information about the onboard camera API, with detailed explanations and examples of how to use it.

2.4.5 Tuner

The Tuner component provides an API to control tuner hardware for FM and Radio Data System, if it is present on the Symbian smartphone. For example, it can be used to scan for a specific radio station and play and record received audio signals. From v9.1 of Symbian OS, Symbian provides the Tuner API definition. Device manufacturers can choose to implement the Tuner API on smartphones that contain a radio tuner.⁶ So far it has only been used for FM radio but could also support digital radio broadcast services as well.

The Tuner component uses the MMF controller plug-ins for formats and audio data types and the MDF (more specifically DevSound) for playback and recording functionality. Figure 2.6 shows how the Tuner component interacts with the MMF Controller Framework and other components.

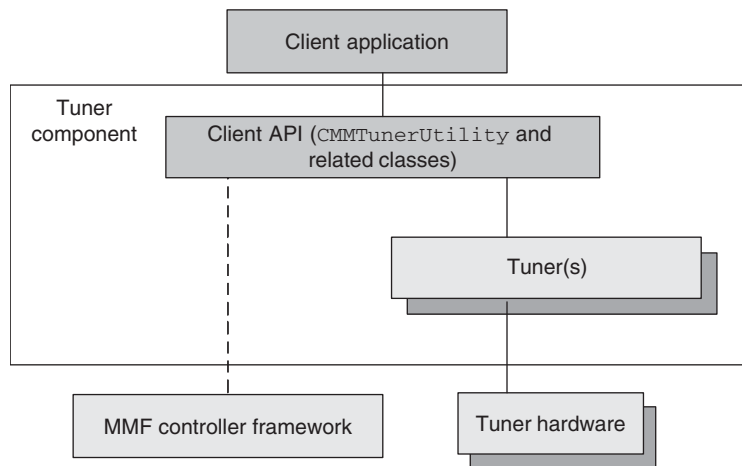


Figure 2.6 The tuner and related components

A tuner device converts radio signals from an antenna into audio signals and allows the user to select a specific broadcast channel, such as a radio station.

Chapter 7 contains more detailed information about the Tuner component and its API.

⁶S60 currently uses a proprietary tuner implementation and the API is not provided in a public SDK. UIQ 3 makes the Tuner APIs public and the necessary class definitions can be found in `tuner.h`. You must link against `tuner.lib` to use them.

2.4.6 RTP and SIP

The Real-time Transport Protocol (RTP) and Session Initiation Protocol (SIP) shown in Figure 2.3 are not counted as part of the multimedia subsystem. However, RTP is the basis for various streaming media and mobile TV services and both enable convergence between multimedia and telephony.

They are used by VoIP applications. For example, SIP is used by applications such as Gizmo and Truphone. It's also supported by Google Talk.

RTP provides end-to-end network transport services for data with real-time characteristics, such as interactive audio and video. It is built on top of the User Datagram Protocol (UDP). There are two closely linked parts of the standard: RTP carries data that has real-time properties and Real-time Transport Control Protocol (RTCP) provides feedback on the quality of service being provided by RTP. These are also commonly used in conjunction with Real-Time Streaming Protocol (RTSP) to provide streaming multimedia services.

More information about RTP and SIP in Symbian OS can be found in the SDKs.

2.5 Future Multimedia Support

In this section, we take a look at what the future holds for multimedia support on Symbian OS. Some features are not yet visible to a third-party application developer, as noted in the appropriate sections, and some may never be made accessible. However, being aware of their existence contributes to a better understanding of the multimedia subsystem.

2.5.1 Metadata Utility Framework (MUF)

Symbian OS v9.5 adds a Metadata Utility Framework (MUF) to the multimedia subsystem. The MUF is independent of the existing MMF layer and provides faster metadata access to any media file. It defines a generic scheme of metadata fields for the most common media file formats and allows the metadata client to select them as required. The MUF supports album art and both DRM-protected and non-DRM-protected files. It does not extract metadata for DRM-protected files without rights.

The purpose of the MUF is to provide direct and faster access to metadata and to support simple 'Media File Parser' plug-ins (ECOM plug-ins that parse the media files and extract the metadata synchronously or asynchronously). There is a client utility class that provides a simple application-level interface to communicate with the MUF, for use by parser plug-in writers.

More information on the MUF is available in the Symbian OS v9.5 documentation.

2.5.2 OpenMAX

OpenMAX is a collection of standards defined by the Khronos Group.⁷ Khronos is a member-funded industry consortium focused on the creation of open standard APIs to enable playback and authoring of media on a variety of platforms. Symbian is a member of the group, as are several licensees and partners.

OpenMAX consists of three layers. The lowest layer is the ‘development layer’ (OpenMAX DL), which defines a common set of functions to enable porting of codecs across hardware platforms. Generally a silicon vendor would implement and optimize these functions for their specific hardware.

The next layer up is the ‘integration layer’ (OpenMAX IL), which serves as a low-level interface to the codecs to enable porting of media libraries across operating systems. This layer would be expected as part of the operating system and indeed Symbian has added support for OpenMAX IL audio from Symbian OS v9.5 onwards. On future releases of Symbian OS, it is possible that OpenMAX IL will not only be used for audio but also other multimedia codecs and any other processing unit such as sources, sinks, audio processing effects, mixers, etc. However, just because support for the standard is provided by the OS, it doesn’t mean that it will be adopted immediately by device manufacturers. It is likely that the standard will only be taken into use on future hardware platforms.

The highest layer of abstraction is the ‘application layer’ (OpenMAX AL). This defines a set of APIs providing a standardized interface between an application and multimedia middleware. This specification has not yet been finalized.

OpenMAX IL is not expected to be exposed to third-party developers in a public SDK in the near future. For further details of OpenMAX and other multimedia standards support on Symbian OS, see *Games on Symbian OS* by Jo Stichbury *et al.* (2008), for which more information can be found at developer.symbian.com/gamesbook.

2.5.3 New MDF Architecture

In conjunction with the adoption of OpenMAX IL, Symbian is restructuring the MDF to take full advantage of the modular ‘processing unit’ model, improve performance and make hardware adaptation easier for licensees. The new architecture is divided into three planes – management, control and data. The first two planes are responsible for setting up

⁷More information about the Khronos group can be found at www.khronos.org.

and controlling the data processing chain, which stays entirely within the hardware adaptation layer. This aligns the multimedia architecture with Symbian's new networking architecture, FreeWay.

The existing DevSound and DevVideo interfaces will be retained for compatibility. However, in order to take advantage of performance improvements, MDF clients (such as video controller plug-ins) will have to use new multimedia host process (MMHP) APIs. The MMHP management APIs will provide functionality to query, select, configure and connect processing units, while control APIs will be used to implement the standard play, pause, stop, etc. functions. The full MMHP implementation, including video support, is targeted at Symbian OS v9.6. There is an interim step for audio in v9.5, known as the Advanced Audio Adaptation Framework (A3F), which will continue to be supported in later versions for compatibility. It is unlikely that any of these APIs will be exposed to third-party developers in the near future but their adoption by licensees will be a key enabler for the delivery of high-performance multimedia applications.

2.5.4 OpenSL ES

OpenSL ES is another standard from the Khronos Group. It is the sister standard of OpenGL ES in the audio area. It has been designed to minimize fragmentation of audio APIs between proprietary implementations and to provide a standard way to access audio hardware acceleration for application developers. OpenSL ES is also a royalty-free open API and is portable between platforms, like OpenMAX. In fact, OpenMAX AL and OpenSL ES overlap in their support for audio playback, recording and MIDI functionality. A device can choose to support OpenMAX AL and OpenSL ES together or just one of them.

OpenSL ES supports a large set of features, but these features have been grouped into three 'profiles': phone, music and game. Any device can choose to support one or more of these profiles depending on its target market. OpenSL ES also supports vendor-specific extensions to add more functionality to the standard feature set of a profile.

The OpenSL ES API is identical to OpenMAX AL, but adds support for more objects (for example, listener, player and 3D Groups) and interfaces (for example, 3DLocation, 3DPlayer, 3DDoppler, 3DMacroscopic) for 3D audio support.

At the time of writing, the OpenSL ES standard has not been finalized but Symbian have announced their intention to implement it when it is available.

2.5.5 Mobile TV

A number of standards have evolved for broadcasting television to mobile devices. The broadcast industry already has its own set of technologies

and business models which are likely to be used in the early implementations of mobile TV. They tend to involve proprietary middleware and conditional access solutions. As such, third-party developers are not likely to be able to access any mobile TV functionality via public APIs. Symbian is enabling the integration of mobile TV tuners via a common hardware abstraction layer for device integrators and middleware vendors.

In addition to broadcast services, mobile TV can also be delivered via 3G mobile networks. There are a number of different systems based on both proprietary and open standards, such as Multimedia Broadcast Multicast Service (MBMS). They can involve streaming, progressive download, or download and play. The multimedia framework can be used in the implementation of such services but specific details are beyond the scope of this book. For more information on mobile TV, turn to Section 7.4.2.

Links to the most up-to-date resources connected with this book can be found on the wiki page at ***developer.symbian.com/multimediabook_wiki***.

3

The Onboard Camera

In the previous chapters, we learnt about the multimedia subsystem and its components. From this chapter onwards, we take a closer look at each component. This chapter introduces you to the Onboard Camera component.

The chapter begins with an introduction to the Onboard Camera API. After understanding the need for a common API, we learn how to access and control the camera, through this API, to use the camera's full capabilities. We then look at capturing still images and video frames and we finish by looking at some advanced topics, such as multiple camera clients.

3.1 Introduction

As cameras became more widespread on phones, different APIs were introduced by the phone manufacturers to allow developers to access the camera functionality. Fragmentation began to occur; meaning that different camera handling code had to be written depending on the phone platform and the individual device's capabilities.

To resolve this, Symbian defined a common Camera API, the ECam API, which was first released in Symbian OS v7.0s. In each subsequent release of the operating system, additional functionality has been added to cover the variety of camera hardware available, and to provide a common API for developers to work with irrespective of the hardware available on any particular Symbian smartphone.

The ECam API provides functionality to display a viewfinder and capture still images from the camera. It is also possible to use it for capturing video; however we recommend that you use the video API outlined in Chapter 4. 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 its own

implementation, using the same ECam API. You may therefore find that the behavior differs slightly between phones and manufacturers.

3.2 Accessing the Camera

The ECam API is an open, extensible, generic API that provides functionality to control the digital camera on the phone and to request and receive specific image data from it. Figure 3.1 shows where the ECam API is placed in the multimedia subsystem and the other multimedia components it interacts with.

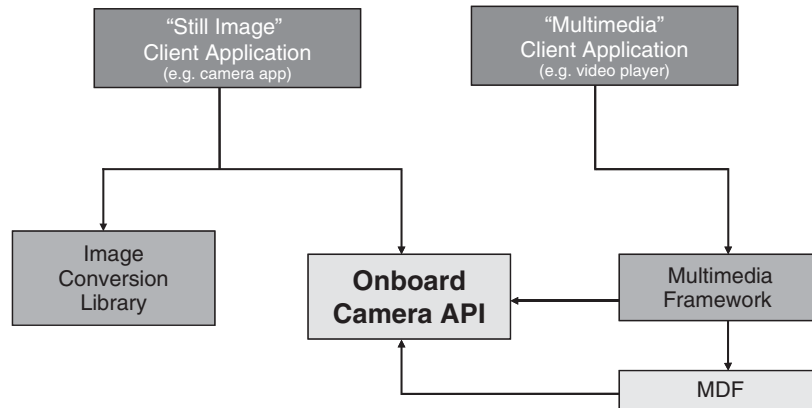


Figure 3.1 The multimedia subsystem

Further information about the Multimedia Framework can be found in Chapters 4 and 5 of this book; the Image Conversion Library is discussed in Chapter 6.

The onboard camera component provides images to its clients. A client may be an application with image requirements, for example a camera application, or an MMF video controller being used by a camcorder application (for information about video, look in Chapter 4). Clients have specific priorities and may pre-empt others in their interactions with the camera. When a camera client needs to share the camera with another component, it creates a handle to the camera and passes this handle to the component (see Section 3.8).

Platform security requires that applications have `UserEnvironment` capability to create a `CCamera` object. Client applications should include the `ecam.h` header file and link against `ecam.lib`.

To access the camera, use `CCamera::NewL()` to create the object. Upon creation of the `CCamera` object, the method `NewL()` might leave;

common leave error codes at this point are:

- `KErrPermissionDenied(-46)` if the application does not have the `UserEnvironment` capability.
- `KErrNotSupported(-5)` if:
 - camera functionality is not supported. Some emulators do not support camera functionality and you must test on the device.
 - `MCameraObserver2` observer is given, but only the older `MCameraObserver` is supported (these callback interfaces are discussed below).

After the camera object is created the camera is reserved for your use by calling `CCamera::Reserve()`.

A client must implement a callback interface so that the camera can notify it when events occur, such as when an image has been captured or a buffer of video data is ready. The recommended callback interface to implement is `MCameraObserver2` since it provides extra functionality such as allowing image capture without buffer copy and it unifies the presentation to the client of the images, video and viewfinder data. However, as `MCameraObserver2` is still not widely implemented on S60 devices, we recommend also implementing the older `MCameraObserver` interface to provide compatibility with existing S60 smartphones.

Before you attempt to access and use the camera on a Symbian smartphone, 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 initiate `CCamera`, simply call `CCamera::NewL()` which accepts an observer parameter of `MCameraObserver` or `MCameraObserver2`,¹ specifying the index of the camera you want to use as the `aCameraIndex` parameter.

```
// Determine the number of available cameras
TInt camerasAvailable = CCamera::CamerasAvailable();
TInt cameraIndex = 0; // range from 0 to camerasAvailable-1
TInt priority = 0; // priority between -100 and 100
// Try to create camera with MCameraObserver2
// Note the current class must be derived from both observers so we
// can just dereference the "this" pointer
TRAPD(error, iCamera = CCamera::NewL(*this, cameraIndex, priority));

if (error == KErrNotSupported)
{
    // if MCameraObserver2 is unsupported, try using MCameraObserver
    iCamera = CCamera::NewL(*this, cameraIndex);
}
```

¹ S60 3rd Edition FP2 SDKs use two separate functions for the different observer interfaces, `NewL()` and `New2L()`.

```

    }
else
{
    User::LeaveIfError(error);
}

```

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 method returns a `TCameraInfo` class that contains a variety of information about the camera in question, such as the camera's orientation, `iOrientation`, and the options supported, `iOptionsSupported`.

The `iOrientation` member 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 cannot rely on knowing the present orientation of the camera.
- `EOrientationUnknown` – the orientation of the camera is not known.

Something that is not covered by these enumerations, unfortunately, is that the camera can be mounted in portrait or landscape orientation with respect to the viewfinder. If your camera application is not in the same orientation as the main camera application of the device, the camera API will rotate the image – but you may find that the resolution you can capture is restricted.

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

- `EImageCaptureSupported` indicates that the camera can capture still images.
- `EVideoCaptureSupported` indicates that the camera can capture video.

```

// member of class
TCameraInfo info;
// Retrieve camera information

```

```

iCamera->CameraInfo(info);
// Camera orientation
TCameraInfo::TCameraOrientation orientation = info.iOrientation;
// Supported modes
TBool imageCaptureSupported = info.iOptionsSupported &
    TCameraInfo::EImageCaptureSupported;
TBool videoCaptureSupported = info.iOptionsSupported &
    TCameraInfo::EVideoCaptureSupported;

```

3.3 Camera Control

A client may not always be able to gain control of the camera. If more than one client is requesting control, then a priority calculation is used to determine which client should have it. Higher priority clients take ownership from lower priority clients when they contend for use of the camera.

There are two parts that make up a client's priority value:

- Clients may declare themselves to have a priority from -100 to 100 . The default priority is zero.
- Clients with `MultimediaDD` capability take priority ahead of clients without that capability, regardless of the declared priority values.

A camera implementation may use additional prioritization decisions, such as prioritizing a particular process ID. However the API does not assume any camera policy for this, as it is implementation-specific.

The `CCamera::Reserve()` call is unsuccessful if a higher priority client is already using the camera: the `MCameraObserver2::HandleEvent()` callback returns an error.

If a client is superseded by another, it receives a `MCameraObserver2::HandleEvent()` notification that the camera is no longer available. It should expect no further callbacks for outstanding capture requests or notifications for pending custom extended operations (histograms, snapshots, etc.) and further attempts to use the camera without a successful `CCamera::Reserve()` call will fail.

3.3.1 Power Control

Once a camera object is available, the client application needs to switch on the camera power, using `CCamera::PowerOn()`. The callback interface is notified when power on is complete. Once the power-on callback is received, the camera is ready to be used and the client can start setting up the viewfinder or capturing images.

As continuous camera use can drain the battery, the camera should be switched off whenever it is not in use, by calling `CCamera::PowerOff()`.

In addition, the viewfinder should have a timeout since this, in particular, will drain the battery.

```
void CCameraAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        // An example AppUI command to start the camera example
        case ECommandCameraOn:
        {
            // Reserve the camera
            iCamera->Reserve();
        }
        break;
    }
}

//Handle event method from MCameraObserver2
void CCameraAppUi::HandleEvent(const TECAMEvent& aEvent)
{
    if (aEvent.iEventType == KUidECamEventReserveComplete)
    {
        if (aEvent.iErrorCode == KErrNone)
        {
            iCamera->PowerOn();
        }
        else
            //report error to user
    }
    else if (aEvent.iEventType == KUidECamEventPowerOnComplete)
    {
        if (aEvent.iErrorCode == KErrNone)
        {
            // camera now ready for use
        }
        else
            //report error to user
    }
}

// from MCameraObserver
void CCameraDemoAppUi::ReserveComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        iCamera->PowerOn();
    }
    else
    {
        // handle error
    }
}

void CCameraDemoAppUi::PowerOnComplete(TInt aError)
{
    if (aError == KErrNone)
    {
        // camera now ready for use
    }
}
```

3.3.2 Basic Image Settings

The `CCamera` class allows access to the camera's settings. It provides functions that allow you to select the image format and to alter brightness, contrast, zoom, exposure, flash and white balance levels of the camera image. Before using such functions though, you should test that the camera supports what you want. To do this, get a `TCameraInfo` object using the `CameraInfo()` function. The object's `iOptionsSupported` member is a bitfield of flags that describe which options are supported by the camera.

Image Format

Before a client application captures still or video images it can first specify the required image format. There may be complicated dependencies between frame sizes and formats, so the required format must be specified as follows:

1. Select the format from those available from either the `TCameraInfo::iImageFormatsSupported` or `TCameraInfo::iVideoFrameFormatsSupported` bitfields, for still or video images respectively.
2. Select the required size using either `CCamera::EnumerateCaptureSizes()` or `CCamera::EnumerateVideoFrameSizes()` for still or video images respectively. Note that not all possible sizes are guaranteed to be supported for any given format. Unsupported sizes are returned as (0,0).
3. For video capture, select the frame rate using `CCamera::EnumerateVideoFrameRates()`. Again, not all rates are guaranteed to be supported, as dependencies may exist in the camera between the format, size, exposure mode and rate.

Brightness

To alter the camera image brightness:

1. Check if brightness control is supported, by testing if the `TCameraInfo::EBrightnessSupported` flag is set in the options.
2. Set the brightness using `CCamera::SetBrightnessL()`.
3. The brightness should be in the range -100 to $+100$. To set the brightness automatically, use the flag `CCamera::EBrightnessAuto`.

```
TBool brightnessSupported = info.iOptionsSupported &
    TCameraInfo::EBrightnessSupported;
```

```
if (brightnessSupported)
{
    iCamera->SetBrightnessL(CCamera::EBrightnessAuto);
}
```

Contrast

To alter the camera image contrast:

1. Check if contrast control is supported, by testing if the `TCameraInfo::EContrastSupported` flag is set in the options.
2. Set the contrast using `CCamera::SetContrastL()`. The contrast should be in the range -100 to $+100$. To set the contrast automatically, use the flag `CCamera::EContrastAuto`.

```
TBool contrastSupported = info.iOptionsSupported &
    TCameraInfo::EContrastSupported;
if (contrastSupported)
{
    iCamera->SetContrastL(CCamera::EContrastAuto);
}
```

Zoom

To alter the camera's zoom level:

1. Test if zoom is supported, and for what range of values, by reading the minimum and maximum zoom values from the data members in `TCameraInfo`. A value of zero means zoom is not supported. A step of one in the zoom value corresponds to the smallest zoom change available. The camera zoom increases linearly with the zoom value until the maximum zoom is reached. A separate set of values is available for (optical) zoom and for digital zoom. `TCameraInfo` also has members `iMinZoomFactor` and `iMaxZoomFactor` that contain the actual zoom factor when at minimum (non-digital only) and maximum zoom.
2. Set the zoom using `CCamera::SetDigitalZoomFactorL()` or `CCamera::SetZoomFactorL()`.

```
TCameraInfo info;
iCamera->CameraInfo(info);
// setup optical zoom
TInt minZoom = info.iMinZoom;
TInt maxZoom = info.iMaxZoom;
```



```

if (minZoom != 0) // if it is supported
{
    // set a value from minZoom to maxZoom
    iCamera->SetZoomFactorL(maxZoom);
}

// set up digital zoom
TInt maxDigitalZoom = info.iMaxDigitalZoom;
if (maxDigitalZoom > 0) // if it is supported
{
    // set a value from 0 to maxDigitalZoom
    iCamera->SetDigitalZoomFactorL(maxDigitalZoom);
}

```

White Balance

White balance presets are given by the `CCamera::TWhiteBalance` enum. Supported white balance options are given in `TCameraInfo::iWhiteBalanceOptionsSupported`. `EWBAuto` is the default and is always supported. To alter the white balance to compensate for tungsten lighting, use:

```
iCamera->SetWhiteBalanceL(CCamera::EWBTungsten);
```

Exposure

Exposure presets are given by the `CCamera::TExposure` enum. Supported exposure presets are given in `TCameraInfo::iExposureModesSupported`. `EExposureAuto` is the default and is always supported. To alter the exposure for nighttime shots, call:

```
iCamera->SetExposureL(CCamera::EExposureNight);
```

Flash

Flash presets are given by the `CCamera::TFlash` enum. Supported flash settings are given in `TCameraInfo::iFlashModesSupported`. `EFlashNone` is the default. To alter the flash mode, use:

```
iCamera->SetFlashL(CCamera::EFlashAuto);
```

3.4 Displaying the Viewfinder

After we have successfully powered on the camera hardware, we can display a viewfinder.

The viewfinder can, if the camera implementation supports it, transfer frames from the camera directly to the display memory at a location of the client's choosing. Alternatively, if supported, the application developers may implement a viewfinder function, in which case the client is passed the viewfinder image as a bitmap at regular intervals.

The option to use is determined by the capabilities of the camera. The `iOptions` member of `TCameraInfo` may have one or both of the following two bits set:

- `EViewFinderBitmapSupported` – we have to render the viewfinder ourselves using bitmaps supplied by the camera.
- `EViewFinderDirectSupported` – the frames are transferred from the camera directly to the display memory at the location the client chooses.

It is possible that neither method is supported, so you should always check which method is supported before attempting to render a viewfinder.

You can specify the portion of the screen to which viewfinder data is to be transferred. This is specified in screen co-ordinates and may be modified if, for example, the camera requires the destination to have a certain byte alignment. The size of image returned by the viewfinder may not be the exact size requested if this conflicts with the requirement for the viewfinder to show the unclipped view in the ratio required for video capture. In this case, the camera returns the best match.

3.4.1 Direct Screen Access Viewfinder

Rendering of a direct screen viewfinder, where the frames are transferred from the camera directly to the display memory at a location of the client's choosing, is the most efficient method. The rendering is done by the camera subsystem and is usually optimized to make use of any hardware acceleration present. To draw a direct screen access viewfinder, we use the following method:

```
virtual void StartViewFinderDirectL(RWsSession& aWs,
                                   CWScreenDevice& aScreenDevice,
                                   RWindowBase& aWindow,
                                   TRect& aScreenRect);
```

The first three parameters 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.

Note that the direct screen access handles the visible region of the screen – focus loss won't stop the viewfinder if it is partially visible.

This viewfinder mode can be more efficient than the bitmap-based mode as it allows optimization of the path between the camera hardware and the LCD controller (e.g. often the YUV to RGB conversion and scaling can take place in hardware). However, until recently, this mode has not been supported by S60 devices.

3.4.2 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. This will generally be less efficient than using the direct screen viewfinder (when available), but it allows the client to process the image for a computer vision application or game. (For example, one could use it with Nokia Research Center's free computer vision library² to do feature recognition or motion detection.)

To start the bitmap-based viewfinder, we use the following method:

```
virtual void StartViewFinderBitmapsL(TSize& aSize);
```

This method requires a single `TSize` parameter that determines the size of the bitmap that we receive. It should be set to the size of screen area into which we intend to draw the viewfinder.

Following this, bitmaps are passed to us via one of the following callback methods;

- `ViewFinderReady(MCameraBuffer& aCameraBuffer, TInt aError)` – This method is called when we use `MCameraObserver2`. The bitmap is represented by class `MCameraBuffer` and a `CFbsBitmap` to display can be retrieved from this class using the `BitmapL()` method. The `MCameraBuffer` is capable of encapsulating several frames which may be encoded; for a viewfinder, you should expect a single, unencoded frame in the form of a `CFbsBitmap` instance.
- `ViewFinderFrameReady(CFbsBitmap& aFrame)` – This method is called when we use `MCameraObserver`. The bitmap passed is simply a `CFbsBitmap` which can be drawn to the screen in the usual way. Note that, when using this method, the viewfinder frame must be drawn in the callback or the bitmap must be copied for later use. Once the callback returns, the API reuses the bitmap. Therefore if we try to use the bitmap after that stage, we see tearing on the viewfinder.

² research.nokia.com/research/projects/nokiav/index.html.

```

TRect screenRect = iAppView->Rect();
TSize size = screenRect.Size();
TCameraInfo info;
iCamera->CameraInfo(info);
if (info.iOptionsSupported & TCameraInfo::EViewFinderDirectSupported)
{
    iCamera->StartViewFinderDirectL(iCoeEnv->WsSession(),
                                   *iCoeEnv->ScreenDevice(),
                                   *iAppView->DrawableWindow(),
                                   screenRect);
}
else if (info.iOptionsSupported &
         TCameraInfo::EViewFinderBitmapsSupported)
{
    iCamera->StartViewFinderBitmapsL(size);
}

```

3.5 Capturing Still Images

Image capture involves transferring the current image from the camera to the client. Before a still image can be captured, we 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.

3.5.1 Selecting the Image Format

The formats that are supported are determined by `TCameraInfo::iImageFormatsSupported`. This is a bitfield of values from the enumeration `CCamera::TFormat`. When using cameras that support a number of formats, we need to choose the format most suitable for our application.

```

CCamera::TFormat iFormat;
TInt iSizeIndex;
// Initial camera querying to get supported capture sizes
void CCameraDemoAppUi::SelectCaptureModeL(CCamera::TFormat aFormat)
{
    iFormat = aFormat;
    //iSizeArray is defined elsewhere as RArray<TSize>
    for (TInt i=0; i<iInfo.iNumImageSizesSupported; ++i)
    {
        TSize size;
        iCamera->EnumerateCaptureSizes(size, i, iFormat);
        iSizeArray.AppendL(size);
    }
    // SelectBestSize() looks through the list of supported sizes for the
    // most appropriate for the application e.g. does it want the highest
    // resolution photo or does it want one that matches the display size?

```

```

    iSizeIndex = SelectBestSize(iSizeArray);
    iCamera->PrepareImageCaptureL(iFormat, iSizeIndex);
}
// called to capture the image
void CCameraDemoAppUi::CaptureImage()
{
    iCamera->CaptureImage();
}

```

The following formats may be supported:

- `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 and the ICL framework is used to decode the data (see Chapter 6).
- `EFormatFbsBitmapColorXxx`: these are uncompressed formats in which the captured image is represented by a `CFbsBitmap` object.
- `EFormatXxBitRGBXxx` 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 will be too great.

Once we have selected a format to use, we can enumerate the capture sizes that are supported for that format. The number of image sizes supported is in `TCameraInfo::iNumImageSizesSupported`. The sizes themselves are returned by `CCamera::EnumerateCaptureSizes()` method.

3.5.2 Preparing for Image Capture

Before we can capture an image, we 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 must be called at least once before requesting images. If images are being captured in response to user input, you should prepare the camera in advance to minimize the delay when capture is performed.

```

iCamera->PrepareImageCaptureL(aFormat, 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. Note that:

- Image settings should be used only for capturing images; for video capture, the camera should be prepared separately (see Section 3.6).
- It is valid to prepare the camera for image capture and for video capture in any order preceding the image capture.
- Image capture cannot take place while video capture is active.

3.5.3 Capturing an Image

Image capture is an asynchronous operation. It is instantiated simply by making a call to the `CaptureImage()` method:

```
iCamera->CaptureImage();
```

After we call the above method, we receive a callback to the `MCameraObserver2::ImageBufferReady()` method:

```
void CCameraAppUi::ImageBufferReady(MCameraBuffer& aCameraBuffer,
                                     TInt aError)
{
    if (aError == KErrNone)
    {
        // use the image
    }
    else
    {
        // handle error
    }
}
```

Note that the viewfinder is not automatically stopped when an image is captured. It is the responsibility of the client to stop the viewfinder and display the captured image.

We may want to capture more than one image; in that case, we can call the `CaptureImage()` method again, however we must not call `CaptureImage()` again before:

- We receive the `ImageBufferReady()` callback.
- We cancel the ongoing image capture by calling `CancelImageCapture()`.

The image is encapsulated by the class `MCameraBuffer`, which is also used when receiving images from the viewfinder. The `MCameraBuffer`

class can store image data in a number of ways and can store multiple frames. When capturing a still image, this class stores a single frame of the requested format.

As mentioned above, we can choose the image format of the image we are capturing. If we requested an `EFormatFbsBitmapColorXxx` when we configured the image capture, the `MCameraBuffer::BitmapL()` method returns a handle to the `CFbsBitmap` containing the image data. If we requested an `EFormatJpeg` or an `EFormatExif`, the descriptor can be decoded to a `CFbsBitmap` using `CImageDecoder` (see Chapter 6) or the data can be written directly to a JPEG file.

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 we requested.

Note that once we have finished processing an `MCameraBuffer` object, we must call its `Release()` method to avoid memory leaks. This releases the memory used and allows it to be reused by the camera subsystem.

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.

If we are working on a phone that only supports `MCameraObserver` and does not support `MCameraObserver2`, the equivalent callback is:

```
MCameraObserver::ImageReady(CFbsBitmap* aBmp, HBufC8* aData, TInt aError)
```

The `CFbsBitmap` and `HBufC8` pointers 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. In this case, when we finish using the pointer, we need to delete it in order to avoid a memory leak.

3.6 Capturing Video

Video capture involves transferring image frames from the camera to the client. Before video can be captured, we should query the capabilities of the camera in use. There are three parameters that need to be set based on the capabilities: format, size, and rate. You can also set the number of buffers to use and the number of frames per buffer.

3.6.1 Selecting the Video Format, Size and Rate

The video formats that are supported are determined by `TCameraInfo::iVideoFrameFormatsSupported`. This is a bitfield of values from the

enumeration `CCamera::TFormat`. When using cameras that support a number of formats, we need to choose the format most suitable for our application. For example, you would want to choose an uncompressed format if you wished to do some sort of processing on the video frames or compress them to another format, but might choose JPEG compressed frames if you wanted to stream them to a PC or another device. Video formats are read in the same way as the still image formats (see Section 3.5).

Once we have selected a format to use, we can enumerate the video capture sizes that are supported for that format. The number of frame sizes supported is in `TCameraInfo::iNumVideoFrameSizesSupported`. The sizes themselves are returned by the `CCamera::EnumerateVideoFrameSizes()` method.

```
TInt format = iInfo.iVideoFrameFormatsSupported;
// look for a format that the application can understand
// (in this example we just show one format, YUV422)
if (format & CCamera::EFormatYUV422)
{
    CCamera::TFormat format = CCamera::EFormatYUV422;
    for (TInt i=0; i<iInfo.iNumVideoFrameSizesSupported; ++i)
    {
        TSize size;
        iCamera->EnumerateVideoFrameSizes(size, i, format);
        iSizeArray.AppendL(size);
    }

    // Look for the most suitable size from the list; in this
    // example, we just choose the first one in the list
    TInt sizeIndex = 0;

    // iRateArray is defined elsewhere as RArray<TReal32>
    for (TInt i=0; i<iInfo.iNumVideoFrameRatesSupported; ++i)
    {
        TReal32 rate;
        iCamera->EnumerateVideoFrameRates(rate, i, format, sizeIndex);
        iRateArray.AppendL(rate);
    }
}
```

Once we have enumerated the video capture sizes, we can enumerate the video capture frame rate supported for the selected format and size. The number of frame rates supported is in `TCameraInfo::iNumVideoFrameRatesSupported`. The supported frame rates are returned by the `CCamera::EnumerateVideoFrameRates()` method.

3.6.2 Preparing for Video Capture

Before we can capture video, we need to prepare the `CCamera` object using the `CCamera::PrepareVideoCaptureL()` method. This allows

the camera subsystem to allocate any memory necessary and perform any other setup required to capture video.

```
iCamera->PrepareVideoCaptureL(aFormat, aSizeIndex, aRateIndex,
                              aBuffersToUse, aFramesPerBuffer);
```

- If video is not supported, then the function leaves with the error `KErrNotSupported`.
- Video settings should be used for capturing video only; for image capture, the camera should be prepared separately (see Section 3.5).
- Video capture cannot take place while a still image capture is active.

3.6.3 Capturing Video

Video capture is an asynchronous operation. It is instantiated simply by making a call to `CCamera::StartVideoCapture()`.

```
iCamera->StartVideoCapture();
```

The camera then fills the buffers as frames become available. Once a buffer has been filled, you receive a callback to `MCameraObserver2::VideoBufferReady()` with the buffer in an `MCameraBuffer` object. If an error occurs with video capture, then the client is notified by `MCameraObserver2::VideoBufferReady()` passing an error, in which case no valid frame data is included.

If the error is fatal to the process of capturing video, such as the camera being switched off, then video capture stops and outstanding buffers are deleted by the camera object.

If the camera runs out of frame buffers, then there will be no callbacks until you call `MCameraBuffer::Release()`, after which `MCameraObserver2::VideoBufferReady()` will start being called again.

You can use multiple sequences of `CCamera::StartVideoCapture()` and `CCamera::StopVideoCapture()` calls following a single call to `CCamera::PrepareVideoCaptureL()`.

In the callback, you need to extract the data from the `MCameraBuffer` object, as a descriptor, a bitmap or a handle to a kernel chunk. In all three cases, the camera retains ownership of the data. Once the data has been used, the buffer should be released by calling its `Release()` function, which indicates that the camera may reuse or delete it (or call `MFrameBuffer::Release()` with `MCameraObserver::FrameBufferReady()`).

The buffer format returned depends on the format selected. Either the image data is available as a bitmap (owned by the Font and Bitmap Server) or it is available as a descriptor and chunk (the descriptor refers to the memory in the chunk). An attempt to access an inappropriate format will cause the function to leave.

For the purpose of synchronization, a buffer provides the index of the starting frame in the buffer and the elapsed time (timestamp) since the video capture started. It is assumed that the frames within the buffer have been captured sequentially at the requested frame rate. Their individual timestamps can be calculated as a function of their index, capture rate and first frame time latency. In cases where considerable jitter is expected or observed, it may be better to have a single frame in a buffer.

```
void CCameraAppUi::CaptureVideo()
{
    iCamera->StartVideoCapture();
}
//MCameraObserver
void CCameraAppUi::FrameBufferReady(MFrameBuffer* aFrameBuffer,
                                     TInt aError);
//MCameraObserver2
void CCameraDemoAppUi::VideoBufferReady(MCameraBuffer& aCameraBuffer,
                                         TInt aError);
```

3.7 Error Handling

The permitted camera calls depend on the state of the camera. Illegal calls are made if:

- We programmed incorrectly. We may repeat a call unnecessarily, such as by calling `Reserve()` when the camera is already successfully reserved.
- The camera has been seized by another client and we have not received a notification, if we are using `MCameraObserver` rather than `MCameraObserver2`.

If we make an illegal call using a method that can leave, the function leaves with the error code; for example, calling `PrepareImageCaptureL()` while video is being captured or `PrepareVideoCaptureL()` while an image is being captured will leave with `KErrInUse`.

If we use a method which has a callback, the error code is returned in the callback; for example if we call `CaptureImage()` or `StartVideoCapture()` without a successful previous call to `PrepareImageCaptureL()` or `PrepareVideoCaptureL()`, respectively, we get a callback with the error `KErrNotReady`.

If we call the `Reserve()` method and a higher priority client is in control of the camera, we get a callback with the error `KErrAccessDenied`.

If we make illegal calls that cannot leave and have no callback, they are simply ignored; for example, `StopVideoCapture()` and `PowerOff()`. This makes these methods safe to use in the destructors of classes, which reduces the risk of the camera being left on accidentally.

More information about the error codes can be found in the method descriptions and `ecamerrors.h` file.

3.8 Advanced Topics

3.8.1 Secondary Clients

Multiple clients may share the camera if the secondary clients create their camera objects using the `CCamera::NewDuplicateL()` factory function. This can only be called with the handle of an existing camera object, read using `CCamera::Handle()`, and lets more than one client make `CCamera::Reserve()` calls at once. This is usually done by MMF video controllers, for instance, which are passed the handle of the camera that the camera application is using to display the viewfinder.

The secondary client (using the `CCamera::NewDuplicateL()` call) assumes the priority of the primary client (the one which generated the camera handle using `CCamera::NewL()`). Each client calls `CCamera::Reserve()` and must call `CCamera::Release()` once they no longer wish to use camera functionality. Even if the primary client finishes, a secondary client retains camera control until it calls `CCamera::Release()`. If a higher priority client gains control, all the clients receive individual `MCameraObserver2::HandleEvent()` notifications.

3.8.2 Innovative Applications Created for Symbian Smartphones

There have been many innovative uses of the camera on Symbian smartphones. A few examples are as follows:

- The Mosquitoes game from Ojom used the camera as a live backdrop for its game play. This created an extremely novel gameplay experience.
- RealEyes3d (www.realeyes3d.com) have a product that uses motion tracking with the camera to create an intuitive input device for application navigation.

- Bar-code readers have been created from image processing on camera viewfinder frames. In Japan, 2D bar codes have been used to contain web links to information about products, such as previews for music or videos. Further information about the use of bar codes can be found in Chapter 6 of *Games on Symbian OS* by Jo Stichbury *et al.* (2008) (***developer.symbian.com/gamesbook***).

Nokia's Computer Vision Library is freely available from ***research.nokia.com/research/projects/nokiacyv/*** and has some open source demonstration applications, which makes it a good starting point for creating innovative applications for S60 smartphones.

4

Multimedia Framework: Video

In this chapter, we look at the Symbian OS video architecture and how you can use it. We begin with details about some general video concepts and look at the various levels of the software architecture. We then describe the client-side APIs that allow you to perform various video playback and recording operations.

4.1 Video Concepts

Before explaining the Symbian OS video architecture, let's get familiar with a few important video concepts.

A video is a series of still images which can be displayed one after another in sequence to show a scene which changes over time. Typically, the images have an accompanying audio track which is played at the same time as the images. The term 'video' is often taken to mean this combination of images and associated audio.

4.1.1 Delivering Video to a Device

There are two main ways that a video can be delivered to a mobile device:

- The video data could be stored in a file that is downloaded to the device. When playback is requested, the file is opened and the data is extracted and played.
- The video data could be streamed over a network to the device, which plays it as it is received.

4.1.2 Video Recording

On Symbian smartphones with a built-in camera and microphone, it is possible to record a video. The video and audio data can be recorded to a file for subsequent playback or could be streamed off the device for storage on a server elsewhere.

4.1.3 Video Storage Formats

Video and audio data stored in a file, or received over a streaming link, is often encoded in order to reduce the size of video files or the amount of data that must be streamed. The job of encoding and decoding the data is handled by a device or program called a codec. Encoding and decoding video is a very processor-intensive operation, so the codec may reside on a separate dedicated processor or on a general-purpose Digital Signal Processor (DSP) to ensure timely processing of the video without impacting the performance of the rest of the system.

There are a number of common encoded data formats that you may come across. For video image data, you may see MPEG2, H.263, MPEG4 and H.264. Each of these formats define a set of methods that can be used to encode the image data. Generally the newer formats, such as H.264, provide better levels of compression than older formats, such as MPEG2, especially as they define sets of levels and profiles which mean different encoding methods can be used depending on the type of image data being encoded.

For audio data you may come across formats such as MP3, AAC and AMR. MP3 is a widely used format for music files. AAC is intended to be its successor and provides better compression at lower bit rates. AMR is specifically designed for encoding speech data.

The encoded video image and audio data for a video are often stored or streamed together in a container. For video files, there are a number of different container formats that are commonly used, such as MP4, 3GP, and AVI. They differ in how the encoded data is stored within the file and in what encoded data formats they can store. MP4 and 3GP keep the information about the size and position of the encoded data units separate from the encoded data itself, allowing easier access to specific units. AVI groups all the information together.

Video image and audio data types are often described using a four-character code, known as a 'FourCC code'. For instance, video image data encoded using the XVID codec would have a FourCC code of 'XVID'.

In addition, the container file format and the video image data format can be described using a notation called a MIME type. For instance, the AVI container file format is described by the MIME type `video/msvideo` and the MPEG4 video image data format has the MIME type `video/mp4v-es`.

4.2 Symbian OS Video Architecture

Figure 4.1 shows an overview of the Symbian OS video architecture. In this section, we briefly describe each of the architectural layers.

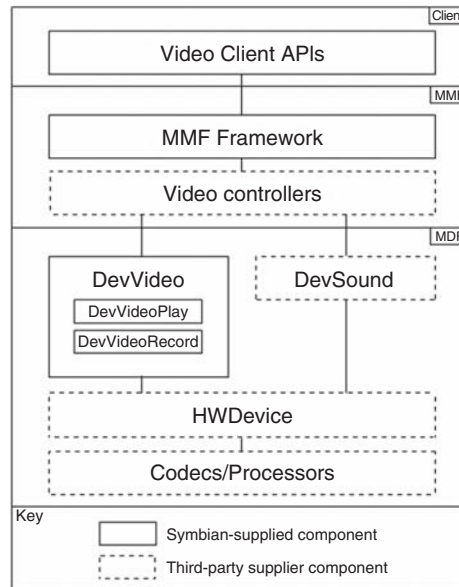


Figure 4.1 The Symbian OS video architecture

4.2.1 Client APIs

Access to the video functionality is provided through the client APIs. Applications use the APIs directly to perform the required video operations. One important point to note is that even though the client APIs may support an operation, there is no guarantee that the lower layers of the architecture will do so. Most of the functionality that performs video operations is handled by the video controllers (see Section 4.2.3) and some controllers may not handle certain operations. The client APIs are described in more detail in the following sections.

Video Playback

The client APIs support playback from file, stream, and descriptor. So you can open a file by specifying the file name or handle, specify a URL to stream from, or pass a buffer that contains the video data. Other basic video operations such as play, pause, and stop are also supported. There are methods which allow you to manipulate how the video is displayed on the screen and to perform operations such as rotating the video. There

are also methods which allow you to get information about the video, such as frame rate and frame size.

Video Recording

As with video playback, you can record to a file, to a stream, or to a descriptor. You can specify a file name, a stream URL, or a buffer. In addition you are required to give the handle of the camera that you would like to record from. See Section 3.6 for details about preparing the camera for recording. You can start, stop and pause the recording. There are also methods which allow you to modify various characteristics for the stored video such as the frame rate, or audio sample rate.

4.2.2 MMF Framework

The MMF framework provides methods to identify and load the video controller that should be used to playback or record a video. The framework also provides the mechanism for passing client requests to the controller and receiving responses and errors from the controller.

4.2.3 Video Controllers

While this book does not cover how to write a video controller, it is important to understand what they do. Video controllers are normally supplied preinstalled on a mobile phone, although, because the controllers use a plug-in mechanism, it is possible for developers to write and install their own controllers if they would like to.

A video controller controls the flow of video data from an input to one or more outputs. For example, a controller might open an MP4 file containing video and audio, use a video codec to decode the video frames and then write them to the display, while using an audio codec to decode the audio frames and then play them through the loudspeaker.

Each installed video controller advertises the container formats it can deal with. So for instance you might find one controller that can open and play only AVI files, while another controller, such as RealPlayer, might be able to support a whole range of container formats. This allows the correct controller to be selected based on what type of file or stream is being used.

For video playback from file, the chosen controller is responsible for opening the file and parsing any metadata in order to determine the types of video and audio data streams; the relevant video and audio codecs are then loaded to decode those streams. The controller extracts the video and audio frames into buffers and passes them to the relevant codecs for decoding. For video, the controller may then pass the decoded frames

onto the display device; alternatively, the codec may do this itself and just return the buffer once it is done.

For video recording to file, the controller takes the video frames from the camera and passes them to a video codec for encoding. On some devices, an external hardware accelerator may encode the video frames before they are passed to the controller. At the same time, the audio from the microphone is encoded by an audio codec. The controller then writes the encoded video and audio frames into the container file.

Another important role for a video controller is to ensure that the synchronization of audio and video is correct. For playback, the video frames must be played at the same time as the associated audio frames. For instance, if a person in the video is speaking, the audio and video frames need to match up to ensure lip synchronization is correct.

From Symbian OS v9.3 onwards, Symbian provides one reference video controller that can be used for test purposes. If you want to use this for testing, you should first check that the SDK you are using actually contains this controller. The controller handles playback from file and recording to file for the AVI container format. Within the file, video data is expected to use the XVID codec and audio data is expected to use the PCM format.

4.2.4 DevVideo

DevVideo is used by the video controllers to access video decoding, encoding, pre-processing, and post-processing functionality. It can be used to find, load, and configure codecs and performs buffer passing between the controllers and codecs. It is implemented by two separate classes, DevVideoPlay for playback and DevVideoRecord for recording.

4.2.5 DevSound

DevSound is used by the video controllers to access audio decoding and encoding functionality. It performs a similar role to DevVideo.

4.2.6 HWDevice

The HWDevice layer defines the interface that DevVideo and DevSound use to communicate with the codecs. The implementation of the codecs themselves is specific to the hardware and platform, so the HWDevice layer is needed to ensure a consistent interface is available for DevVideo and DevSound to use.

4.2.7 Codecs

The video codecs are responsible for converting between compressed and uncompressed video data. As with the controllers, each codec advertises

what type of video data it can process. Once a controller has determined the type of video data that it needs to handle, it can find and load the relevant codec. At this level of the architecture sit the audio codecs as well, performing the same job as the video codecs but for audio data.

For test purposes, an open source XVID video codec is available for download from the Symbian Developer Network website (***developer.symbian.com***). Full instructions on building and installing the codec are provided on the website. This codec is compatible with the reference AVI controller described in Section 4.2.3.

As with controllers, codecs with their associated HWDevice are loaded as plug-ins so it is possible for developers to write and install their own codecs if they would like to. One thing to remember though is that developers do not generally have access to hardware acceleration devices on the phone and this may mean that the codec's performance is disappointing.

4.3 Client API Introduction

For playback, there are two classes that contain all of the APIs that can be used, `CVideoPlayerUtility` (found in `videoplayer.h`) and `CVideoPlayerUtility2` (found in `videoplayer2.h`). For either class, you must link against `mediaclientvideo.lib`.

`CVideoPlayerUtility` can be used with any video controller, whereas `CVideoPlayerUtility2` is only available from Symbian OS v9.5 onwards and can only be used with video controllers that support graphics surfaces (see Section 4.6.3 for more information about graphics surfaces). If you use `CVideoPlayerUtility2` then it only uses controllers that support graphics surfaces. If it cannot find a controller that supports graphics surfaces for a particular video, then an error is returned when trying to open that video.

`CVideoPlayerUtility2` is derived from `CVideoPlayerUtility` which means that the APIs defined in `CVideoPlayerUtility` are also available when using `CVideoPlayerUtility2`; however some of the APIs return an error code because the operations are not supported when graphics surfaces are being used.

For recording, the `CVideoRecorderUtility` class, found in `videorecorder.h`, contains all the APIs that can be used.

Most of the calls to the APIs result in operations being requested on the controllers. Certain operations that you request may not be supported on the particular controller that you are using, in which case an error code is returned, typically `KErrNotSupported`.

Some of the API calls are synchronous and some are asynchronous. Where an API is synchronous it means that once the call to the API has returned, that operation has been fully completed. Where an API is asynchronous it means that the requested operation may not be completed immediately. In this case, a callback is made to an observer function that you supply when instantiating the relevant client API class. In normal operation, the call to the API returns and then at some point in the future your callback function is called to show that the requested operation has been completed. You should note however, that under some circumstances it is possible that the callback function is called before the call to the API returns.

The order of some of the API calls is also important. Some API calls will not work unless a previous operation has been completed. The majority of the API methods will not work unless the video open operation has completed, and in the text of the book we will highlight any known exceptions to this. However as behavior can differ between controllers, it is possible that an API method implemented by one controller may require an operation to have completed, while another controller may not. Where an API method is called before an operation it requires has completed, an error will be returned directly from the API method call or via your callback observer.

In the following sections, we go into more detail about each of the specific API classes and the methods that they provide. It is assumed that the information in this book will be used in conjunction with the Symbian Developer Library documentation, which can be found in every SDK or online at ***developer.symbian.com***. This book shows how the API methods can be used, but the documentation should be used for detailed descriptions of each of the individual parameters and the return values for the API methods.

Where the following sections show a particular method, then that method is normally available through both the `CVideoPlayerUtility` class and the `CVideoPlayerUtility2` class. There are a few exceptions to this, such as where the `CVideoPlayerUtility2` class has extra functions or where different versions of the same function exist between the two classes, and these are indicated in the relevant sections.

4.4 Identifying Video Controllers

Some of the client API methods allow you to specify the UID of the video controller that you want to use for the playback or recording operation.¹

¹Note that this is essential for recording but not good practice for playback – see Section 8.8 for details.

If you want to find a list of the controllers that exist on the phone then you can use some code similar to this example. It uses the MMF framework to access the list of controllers, and then retrieves the play formats that are supported:

```
CMMFControllerPluginSelectionParameters* cSelect =
    CMMFControllerPluginSelectionParameters::NewLC();
CMMFFormatSelectionParameters* fSelect =
    CMMFFormatSelectionParameters::NewLC();

RArray<TUid> mediaIds;
mediaIds.Append(KUidMediaTypeVideo);
cSelect->SetMediaIdsL(mediaIds,
    CMMFPluginSelectionParameters::EAllowOtherMediaIds);
// Indicate that we want to retrieve record formats
cSelect->SetRequiredRecordFormatSupportL(*fSelect);
// Populate an array of controllers that can record
RMMFControllerImplInfoArray controllers;
CleanupResetAndDestroyPushL(controllers);
cSelect->ListImplementationsL(controllers);
TInt numControllers = controllers.Count();
for (TInt i = 0; i < numControllers; ++i)
{
    RMMFFormatImplInfoArray& playFormats = controllers[i]->PlayFormats();
    if (playFormats.Count() > 0)
    {
        // If you just want the first controller that can play then you can
        // get its UID and the UID of the format now using code like this...
        // iControllerUid = controllers[i]->Uid();
        // iFormatUid = playFormats[0]->Uid();
        // break;
        //
        // Alternatively you may want to request the supported container file
        // MIME types, if you want a particular controller, like this...
        // const CDesC8Array* fileExtensions = &playFormats[0]->
        //                                     SupportedMimeType();
    }
}
CleanupStack::PopAndDestroy(3, cSelect);
```

This example is incomplete; the commented out code shows examples of what you can do with it. For instance, it shows how you would get the UID for a particular controller.

4.5 Controlling Video Playback

Figures 4.2 and 4.3 show the user interactions for the basic play and pause/stop sequences for video playback of a file. They show the API method calls from the user and the callbacks from the `CVideo-PlayerUtility` class. In the following sections, we look at these API method calls in more detail.


```
static CVideoPlayerUtility2* NewL(MVideoPlayerUtilityObserver& aObserver,
                                TInt aPriority,
                                TMdaPriorityPreference aPref);
```

The `NewL()` factory methods of `CVideoPlayerUtility` and `CVideoPlayerUtility2` share some common parameters:

- The observer parameter, `aObserver`, is a class that provides the set of callback functions for asynchronous API calls.
- The parameters `aPriority` and `aPref` are used to decide which client should take priority when two separate clients require access to the same sound device, for instance where a video is being played on the loudspeaker and a telephone call is received which requires a ringtone to be played. These parameters are explained in Section 4.9.1.

The `CVideoPlayerUtility::NewL()` factory method also requires a number of parameters that provide information about the display to be used. These are explained in Section 4.6.1.

The `CVideoPlayerUtility2::NewL()` routine does not require any display-related parameters to be passed to it. This is because the display information is set by a separate call to the `AddDisplayWindowL()` method as described in Section 4.6.1.

The observer parameter to the `NewL()` methods is a reference to a class that is derived from the `MVideoPlayerUtilityObserver` class. The `MVideoPlayerUtilityObserver` class defines a set of methods that the derived class must implement. These methods are then called when certain asynchronous operations are completed.

```
class CMyVideoPlayerObserver : public CBase,
                              public MVideoPlayerUtilityObserver
{
public:
    void MvpuoOpenComplete(TInt aError);
    void MvpuoPrepareComplete(TInt aError);
    void MvpuoFrameReady(CFbsBitmap& aFrame, Tint aError);
    void MvpuoPlayComplete(Tint aError);
    void MvpuoEvent(const TMMFEvent& aEvent);
};
```

`MVideoPlayerUtilityObserver` provides methods that are called on the completion of an open operation, a prepare operation, a request to fetch an individual frame and the play operation. Each of the callbacks includes an `aError` parameter which can be checked to see if the operation completed correctly or not. In addition, a general event reporting

callback, `MvpuoEvent`, is defined which supports things such as error handling. The event consists of a UID which defines the event type and an associated error code. Normal system-wide errors have the UID `KMMFErrorCategoryControllerGeneralError`; in addition, video controllers may define their own event types with their own specific UIDs. Note however that the authors of a video controller on a phone may choose not to make public the information about the events that their controller can generate.

4.5.2 Opening a Video

There are three methods that are available for opening a file containing a video:

```
void OpenFileL(const TDesC& aFileName, TUid aControllerUid = KNullUid);
void OpenFileL(const RFile& aFileName, TUid aControllerUid = KNullUid);
void OpenFileL(const TMMSource& aSource, TUid aControllerUid = KNullUid);
```

You can specify a file name, a handle to an already opened file, or a `TMMSource`, which allows access to a DRM-protected source. There is one method that allows you to open a video contained in a descriptor buffer:

```
void OpenDesL(const TDesC8& aDescriptor, TUid aControllerUid = KNullUid);
```

There is one method that allows you to open a video for streaming:

```
void OpenUrlL(const TDesC& aUrl, TInt aIapId = KUseDefaultIap,
              const TDesC8& aMimeType = KNullDesC8,
              TUid aControllerUid = KNullUid);
```

You are required to specify the URL that you want to stream from, an access point that is used to make the connection to that URL, and the MIME type of the video. Current Symbian smartphones generally require that the URL you supply is an RTSP URL.

All the open methods allow you to specify the UID of a specific video controller that you want to use. Normally the video controller is chosen based on the type and contents of the specified file but, by specifying the UID, you can override this decision and use a controller of your choice. This mechanism can also be used where the file does not have a recognizable extension but you know what the file format is.

Opening a video is an asynchronous event, so you must wait for the `MvpuoOpenComplete()` method of your observer to be completed before proceeding. If the open operation fails then the initial API call

can leave with an error or an error code can be passed to the observer method. You need to handle both these scenarios when using these APIs.

4.5.3 Preparing a Video

Once the video has been opened successfully, you need to prepare the video to be played:

```
void Prepare();
```

This tells the video controller to prime itself, ready to start playing. The controller allocates any buffers and resources required for playback at this point.

Preparing a video is an asynchronous event, so you must wait for the `MvpuoPrepareComplete()` method of your observer to be completed before proceeding.

4.5.4 Playing a Video

Once the prepare operation has completed, you can start playing the video (see Figure 4.3):

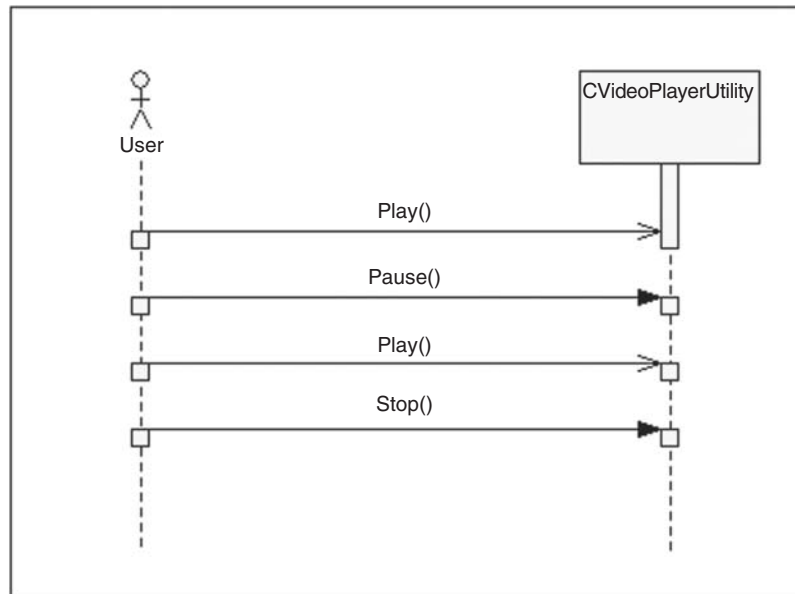


Figure 4.3 Pause and stop sequence


```
void Play();  
void Play(const TTimeIntervalMicroSeconds& aStartPoint,  
          const TTimeIntervalMicroSeconds& aEndPoint);
```

Calling `Play()` with no parameters causes playback to start from the beginning of the video and finish at the end of the video. You can also specify start and end time parameters, which cause playback to start and end at the designated points in the video.

Video playback is an asynchronous event. You know video playback has completed when the `MvpuoPlayComplete()` method of your observer is called. If playback finishes early due to an error then the error code is passed to the observer method.

4.5.5 Pausing a Video

You can pause the current video playback operation:

```
void PauseL();
```

This is a synchronous operation. It leaves with an error code if the pause operation fails. Call `Play()` to restart the playback from the paused position.

4.5.6 Stopping a Video

You can stop the current video playback operation:

```
TInt Stop();
```

This is a synchronous operation and returns an error code if a problem occurs while stopping the video.

The Symbian Developer Library documentation, found in each SDK, states that if you stop a video while it is playing, the `MvpuoPlayComplete()` method of your observer will not be called, however as this is controlled by the video controller it is safer to assume that you may receive this call.

4.5.7 Closing a Video

You can close the currently opened video:

```
void Close();
```

4.5.8 Setting the Video Position

It is possible to change the position that the video is playing from using the `SetPositionL()` method:

```
void SetPositionL(const TTimeIntervalMicroSeconds& aPosition);
```

If you call this method before the video has started playing, it changes the position that playback will start from. The controller may not support changing the video position if the video is already playing.

You can get the current playback position for the video using the `PositionL()` method:

```
TTimeIntervalMicroSeconds PositionL() const;
```

4.5.9 Trick Play Modes

Trick play refers to operations such as fast forward, reverse playback, and slow motion playback.

Prior to Symbian OS v9.4, there were two ways that trick play could be implemented:

- The video controller could supply a custom command² that allowed these modes to be used (see Section 4.15 for more information on custom commands).
- The `SetPositionL()` method (see Section 4.5.8) made it possible to jump forward or backward through the video by changing the position that the video is playing from, which emulates fast forward and reverse playback.

From Symbian OS v9.4 onwards, the following API methods allow trick mode functionality to be used. You can query the supported play rates

²A custom command is an extension that allows a developer to access functionality that is not available using a specific API method call.

for the video controller using the `GetPlayRateCapabilitiesL()` method:

```
void GetPlayRateCapabilitiesL(TVideoPlayRateCapabilities& aCapabilities)
                                const;
```

This returns a `TVideoPlayRateCapabilities` structure which contains a set of public flags to indicate the capabilities:

- `iPlayForward` indicates if fast and slow forward play is possible.
- `iPlayBackward` indicates if fast and slow backward play is possible.
- `iStepForward` indicates if stepping forward is possible.
- `iStepBackward` indicates if stepping backward is possible.

Where play forward and play backward modes are supported, you can set the play velocity:

```
void SetPlayVelocityL(TInt aVelocity);
```

The `aVelocity` parameter specifies the play velocity as a percentage of normal playback velocity. The meaning of the various velocities is shown in Table 4.1.

Table 4.1 Play Velocity Values

| Range | Meaning |
|-----------|--------------------------------|
| > 100 | Fast forward playback |
| 100 | Normal speed forward playback |
| 1 to 99 | Slow motion forward playback |
| 0 | No playback |
| -1 to -99 | Slow motion backward playback |
| -100 | Normal speed backward playback |
| < -100 | Fast backward playback |

Even if the controller supports play velocity settings, it may not support the exact velocities you try to set. In this case, it sets a velocity as close

as possible to the one you requested. To find out the exact play velocity being used at any time, you can call the `PlayVelocityL()` method:

```
TInt PlayVelocityL() const;
```

The `StepFrameL()` method allows you to step forward or backward a selected number of video frames. The method can only be called once the video has been paused (see Section 4.5.5).

```
void StepFrameL(TInt aStep);
```

The `aStep` parameter indicates how many frames to step. A negative value steps backward, a positive value steps forward.

4.5.10 Enabling Video Playback

From Symbian OS v9.4 onwards, it is possible to specify whether the video track in a video should be played or not. This can be set by calling `SetVideoEnabledL()`. If you specify that video is not enabled, then only the audio track is played:

```
void SetVideoEnabledL(TBool aVideoEnabled);
```

You can find out whether the video is enabled or not:

```
TBool VideoEnabledL() const;
```

4.5.11 Frame Snapshot

It is possible to try to take a snapshot from the currently playing video using the `GetFrameL()` methods:

```
void GetFrameL(TDisplayMode aDisplayMode);  
void GetFrameL(TDisplayMode aDisplayMode, ContentAccess::TIntent aIntent);
```

You need to specify the display mode that specifies the color format of the returned snapshot. When accessing DRM content, you also need to specify your intent such as whether you are peeking the content for internal use or whether you intend to display it to the user.

This is an asynchronous method. The result in the form of a `CFbs-
Bitmap` is returned through the `MvpuoFrameReady()` method in your observer.

4.6 Controlling Screen Output

4.6.1 Setting the Display Window

We saw in Section 4.5.1 that the `CVideoPlayerUtility::NewL()` method takes a number of parameters related to the display. The `CVideoPlayerUtility` class also has a method, `SetDisplayWindowL()`, which you can use to change those parameters later if you want to.

```
void SetDisplayWindowL(RWsSession& aWs, CWScreenDevice& aScreenDevice,
                      RWindowBase& aWindow, const TRect& aWindowRect,
                      const TRect& aClipRect);
```

Both the `NewL()` method and the `SetDisplayWindowL()` method take a window server session, a screen device, and a window parameter.

In addition there are two rectangles specified. The `aWindowRect` (or `aScreenRect` for the `NewL()` method) specifies the video extent (see Section 4.6.4). The `aClipRect` specifies the clipping region. The coordinates for these rectangles are interpreted relative to the display position.

`SetDisplayWindowL()` is not supported when using `CVideoPlayerUtility2`; it leaves with the `KErrNotSupported` error code. `SetDisplayWindowL()` can be called before the video open operation is performed.

For `CVideoPlayerUtility`, you can specify the initial screen to use if multiple displays are supported on the phone:

```
TInt SetInitScreenNumber(TInt aScreenNumber);
```

You should do this before attempting to open the video. If you don't call this method, then the screen used is determined by calling `GetScreenNumber()` on the screen device specified in the call to `NewL()`. `SetInitScreenNumber()` is not supported when using `CVideoPlayerUtility2`; it leaves with the `KErrNotSupported` error code. `SetInitScreenNumber()` can be called before the video open operation is performed.

To add a display window when using `CVideoPlayerUtility2`, you can use one of the `AddDisplayWindowL()` methods:

```
void AddDisplayWindowL(RWsSession& aWs, CWScreenDevice& aScreenDevice,
                      RWindowBase& aWindow, const TRect& aVideoExtent,
                      const TRect& aWindowClipRect);
void AddDisplayWindowL(RWsSession& aWs, CWScreenDevice& aScreenDevice,
                      RWindowBase& aWindow);
```

Both these methods have the same window parameters as the `CVideoPlayerUtility::NewL()` method. In addition, one of the methods allows you to specify the video extent rectangle and the clipping rectangle. If you use the method that does not specify the rectangles, then the video extent and clipping rectangle default to the same size as the window you have specified. Note that, in contrast to `CVideoPlayerUtility`, the video extent and clipping region coordinates should be specified relative to the window position.

The `AddDisplayWindowL()` methods can only be called after the video open operation has completed. This is because the `AddDisplayWindowL()` methods can only be used where the chosen video controller supports graphics surfaces, and the video controller must have been opened in order to determine that information.

It is possible to add multiple windows using these APIs provided you specify that they are on different displays i.e. different `CWssScreenDevice` instances. The same video is played in each window. This can be used on Symbian smartphones with multiple displays.

If you want to remove a display window when using `CVideoPlayerUtility2`, you can use the `RemoveDisplayWindow()` method:

```
void RemoveDisplayWindow(RWindowBase& aWindow);
```

4.6.2 Direct Screen Access

One method that a controller or codec can use to send output to the display is called direct screen access. This allows drawing to the screen without having to use the window server. This is the default method of screen access prior to Symbian OS v9.5.

A controller that wants to use direct screen access will start and stop it automatically as required. It is, however, possible for you to start and stop the direct screen access explicitly if you wish to do so.

```
void StopDirectScreenAccessL();  
void StartDirectScreenAccessL();
```

It can be useful to do this if you know that a lot of changes are about to be made to the windows on the UI and you want to temporarily turn off the direct screen access while this happens to ensure that the video does not flicker due to the constant redraws while the windows are moved around.

The direct screen access methods are not supported when using `CVideoPlayerUtility2` and they leave with `KErrNotSupported`.

The direct screen access methods can be called before the video open operation is performed.

4.6.3 Graphics Surfaces

From Symbian OS v9.5 onwards, support has been added for graphics surfaces. These have the advantage that they allow the user interface for the phone to be displayed above a video while it is playing. They also allow specific graphics hardware acceleration devices to be used to provide improved performance.

Controllers or codecs that have been written to support graphics surfaces always use that method to access the screen. This means that at the client API level there is no requirement to ask to use graphics surfaces. There are however some client APIs that only work when graphics surfaces are being used and some APIs that won't work when graphics surfaces are supported.

More details about graphics surfaces are available in the Symbian Developer Library documentation for Symbian OS v9.5 in the Graphics section of the Symbian OS Guide.

4.6.4 Video Extent

The video extent (also known as the screen rectangle or window rectangle) defines the area on the screen where the video is displayed. Figure 4.4 shows a video extent that is fully enclosed within the window. This means the whole of the original video picture is displayed.

It is possible to set the video extent to be fully or partially outside the current window. In this case, only the area of the video within the intersection of the video extent and the window is shown.

It is also possible to set the video extent to be a different size to the original video picture. If you set the video extent to be bigger, then you can choose whether you want to scale the video picture to fit into the video extent or whether you leave the video picture at the same size, in which case not all of the video extent will be covered. If you set the video extent to be smaller, then once again you can scale the video picture to fit into the extent or you can leave the video picture at the same size in which case the part that falls outside the video extent would not be displayed.

Figure 4.5 shows a larger video extent that is not fully enclosed within the window. Here only the parts of the video that appear in the intersection

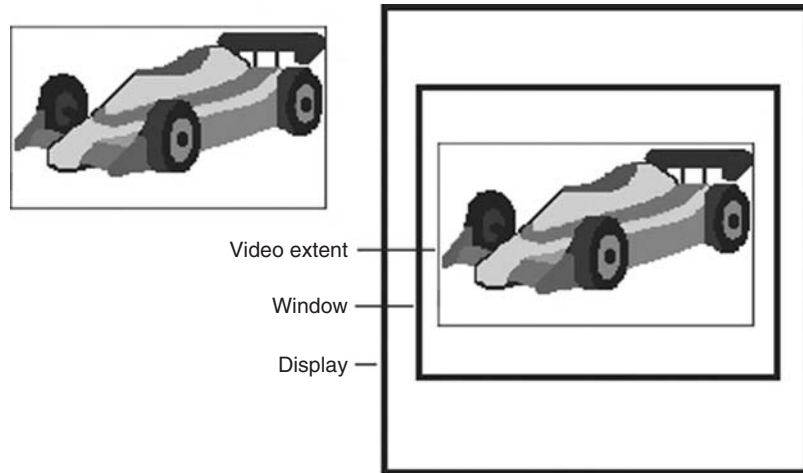


Figure 4.4 Video extent fully within window

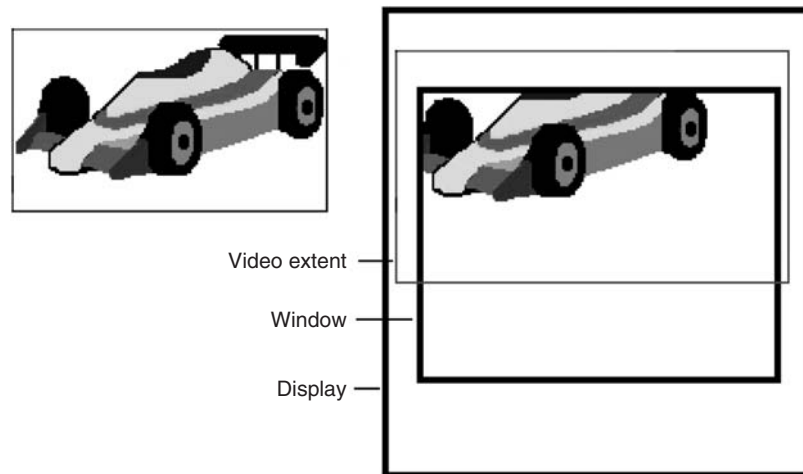


Figure 4.5 Large video extent partially outside window

of the window and the video extent are shown. Note also that the video picture has not been scaled, so part of the video extent is left unused.

Figure 4.6 shows a larger video extent partially outside the window, but in this case the original video picture has been scaled to fit the video extent. The operation to scale the original video picture into the video extent is performed first and then the parts of the scaled picture that lie outside the window are removed.

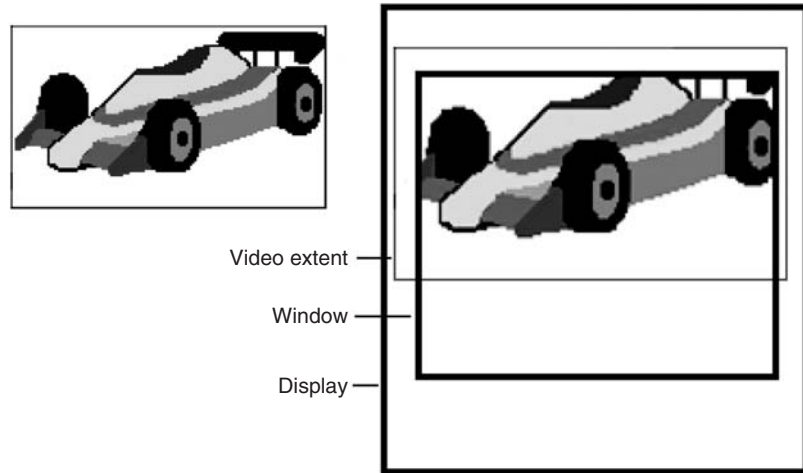


Figure 4.6 Scaling into video extent

For clarity, the figures show the device screen or display with a single window placed on top of it. It is, however, possible to have more than one window on the display at the same time.

For `CVideoPlayerUtility` we have seen already how the video extent is set using a parameter to the `NewL()` or `SetDisplayWindowL()` methods. For `CVideoPlayerUtility2`, the video extent can be set through the `AddDisplayWindowL()` method, but in addition a specific `SetVideoExtentL()` method also exists.

```
void SetVideoExtentL(const RWindowBase& aWindow,
                    const TRect& aVideoExtent);
```

The `aWindow` parameter allows you to specify which window to apply the video extent to.

The `SetVideoExtentL()` method can only be called after the video open operation has completed.

4.6.5 Clipping Region

The clipping region allows you to specify that only a small part of the video extent should be shown. An example of this is shown in Figure 4.7, where a specific clipping region has been applied and only the part of the video extent that appears in the clipping region is shown. As before, the mapping of the video picture into the video extent is performed first and then the clipping region is applied.

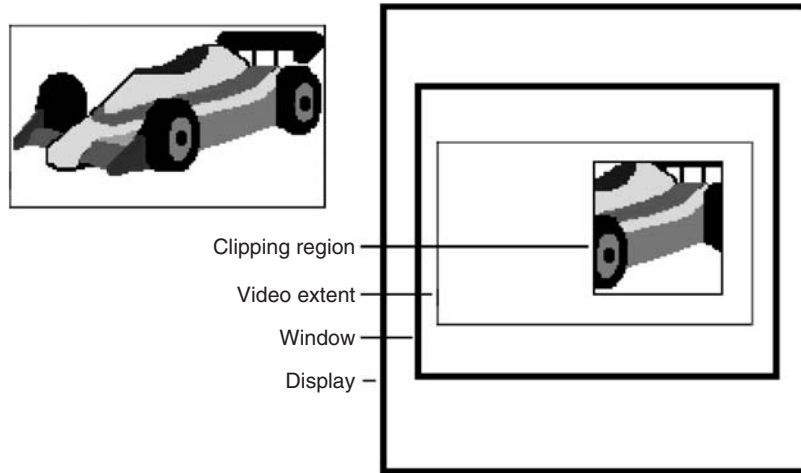


Figure 4.7 Clipping region

For `CVideoPlayerUtility`, we have seen already how the clipping rectangle is set using a parameter to the `NewL` or `SetDisplayWindowL()` methods. For `CVideoPlayerUtility2` the clipping rectangle can be set through the `AddDisplayWindowL()` method. There is also a specific `SetWindowClipRectL()` method:

```
void SetWindowClipRectL(const RWindowBase& aWindow,
                       const TRect& aWindowClipRect);
```

The `aWindow` parameter allows you to specify which window to apply the clipping region to. The clipping rectangle must be completely contained within the specified window.

4.6.6 Cropping

Cropping allows you to specify that only a part of the original video picture should be displayed in the video extent. This is shown in Figure 4.8. Only a small part of the original video picture is selected for display by specifying a cropping region. The figure shows that this region has then been scaled to fit into the video extent. You could also just choose to copy the cropped region into the video extent without scaling it.

While cropping and clipping may seem quite similar, the difference is that the crop is done on the original video whereas the clip is done on the video extent. You should also remember that the cropped region is displayed with its origin set to the top left corner of the video extent. There is normally no difference in performance between the two operations, so if you just want to display a non-scaled portion of the video starting at

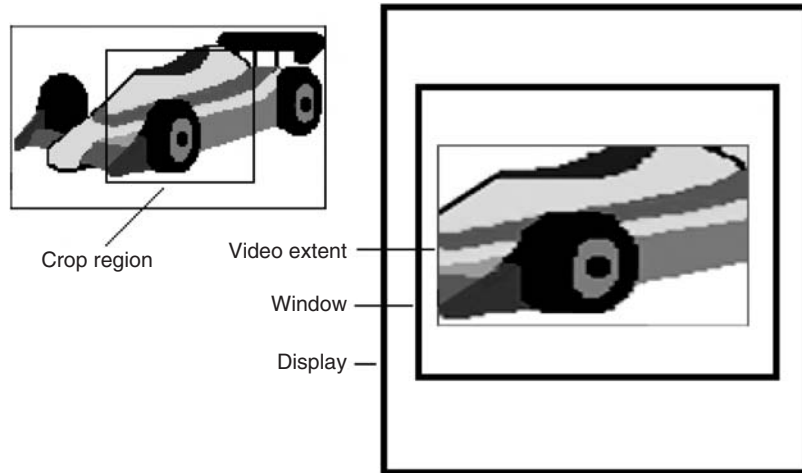


Figure 4.8 Cropping and scaling

its top left corner then you could do this equally well using cropping or clipping.

The crop region for a video can be set using the `SetCropRegionL()` method to specify the crop rectangle:

```
void SetCropRegionL(const TRect& aCropRegion);
```

The crop region applies to the video. If the video is being displayed in multiple windows, then the results of the crop are seen in all the windows.

The current crop region being used can be retrieved using the `GetCropRegionL()` method:

```
void GetCropRegionL(TRect& aCropRegion) const;
```

On return, the `aCropRegion` parameter contains the coordinates of the crop rectangle.

4.6.7 Scaling

The scale factor of the video can be changed using the `SetScaleFactorL()` method:

```
void SetScaleFactorL(TReal32 aWidthPercentage, TReal32 aHeightPercentage,
                    TBool aAntiAliasFiltering);
```

You can specify what percentage to scale the width and height of the video. The original image is assumed to have scale percentages of 100, so if you set the scale factor percentage to 50 then you would shrink the image to half its original size and a scale factor percentage of 200 would double the size of the image. If you change the width and height by differing amounts, then you may distort the image.

You can also request that anti-alias filtering is used to try to improve the appearance of the scaled image and remove distortion; however, the majority of video controllers always use anti-alias filtering regardless of what value you set.

If the video is being displayed in multiple windows, then the results of the scaling are seen in all the windows.

You can retrieve the current scale factor information using the `GetScaleFactorL()` method:

```
void GetScaleFactorL(TReal32& aWidthPercentage,
                    TReal32& aHeightPercentage, TBool& aAntiAliasFiltering) const;
```

`CVideoPlayerUtility2` has an extra `SetScaleFactorL()` method which allows you to specify the scale factors on a per-window basis:

```
void SetScaleFactorL(const RWindowBase& aWindow,
                    TReal32 aWidthPercentage, TReal32 aHeightPercentage);
```

The anti-alias filtering option is not offered in this case because the scaling is handled by the graphics composition engine and it always uses some form of anti-aliasing.

There is also a `GetScaleFactorL()` method that works on a per window basis:

```
void GetScaleFactorL(const RWindowBase& aWindow,
                    TReal32& aWidthPercentage, TReal32& aHeightPercentage);
```

From Symbian OS v9.4 onwards, some new auto-scaling methods are available:

```
void SetAutoScaleL(TAutoScaleType aScaleType);
void SetAutoScaleL(TAutoScaleType aScaleType,
                    TInt aHorizPos, TInt aVertPos);
```

These methods allow you to specify that a video should be automatically scaled to fit the window in which it is being displayed. Where the video is being displayed in multiple windows, the scale options apply to all the windows. The `aScaleType` parameter specifies the type of scaling that should be performed:

- `EAutoScaleNone` indicates that no scaling should be performed.
- `EAutoScaleBestFit` indicates that the video should be scaled to fit the extent but the pixel aspect ratio should be maintained,³ and no part of the video should be clipped in order to fit it into the extent. This may result in areas around the edge of the extent not being filled with video, in which case they are filled with the background window color.
- `EAutoScaleClip` indicates that the video should be scaled to completely fill the extent, while maintaining the pixel aspect ratio. If the extent and video pixel aspect ratios do not match, this may result in part of the video being clipped.
- `EAutoScaleStretch` indicates that the video should be scaled to completely fill the extent, without the pixel aspect ratio being maintained. If the extent and video pixel aspect ratios do not match, this may result in the video being distorted.

The first version of the `SetAutoScaleL()` method always centers the video picture within the extent. The second version allows you to control the horizontal and vertical position of the video relative to the extent.

You can set the horizontal and vertical positions to one of the pre-defined values defined by the `THorizontalAlign` and `TVerticalAlign` enumerations. These allow you to specify left, right, top, bottom, and centered alignment. Note that the scaling operation takes place before alignment is performed, so the alignment operations only have an effect when the selected scaling operation does not result in the whole video extent being used by the scaled image.

³The pixel aspect ratio describes the ratio between the number of pixels in the horizontal direction and the number of pixels in the vertical direction. Where the horizontal and vertical resolutions are the same, the pixels are square and the pixel aspect ratio is 1:1. If the vertical resolution were to be twice that of the horizontal resolution, it would mean the pixels were twice as wide as they are high and the pixel aspect ratio would be 2:1. When scaling a video to fit a specified area on the screen, the resulting image may appear to be distorted if the pixel aspect ratio is not maintained.

You can also set the alignment parameters to a numeric value which represents a pixel offset from the top left corner of the extent. The pixel offset can be both negative and positive. As before, the scaling operation is performed before the alignment. If you specify numeric alignment parameters that result in part of the scaled image being outside the video extent, the image is clipped so that only the part within the video extent is shown.

`CVideoPlayerUtility2` has an additional set of `SetAutoScaleL()` methods which allow the scaling to be set on a per-window basis:

```
void SetAutoScaleL(const RWindowBase& aWindow, TAutoScaleType aScaleType);
void SetAutoScaleL(const RWindowBase& aWindow, TAutoScaleType aScaleType,
                  TInt aHorizPos, TInt aVertPos);
```

It should be noted that the `SetScaleFactorL()` and `SetAutoScaleL()` methods cannot be used at the same time. If a `SetScaleFactorL()` method is called, then any scaling requested by a previous call to `SetAutoScaleL()` is removed and the new scale factors are applied. Similarly, a call to a `SetAutoScaleL()` method causes any scale factors set by a previous `SetScaleFactorL()` call to be removed before the automatic scaling is applied.

4.6.8 Rotation

You can specify that the original video picture should be rotated before it is displayed in the video extent. This is useful if the video orientation

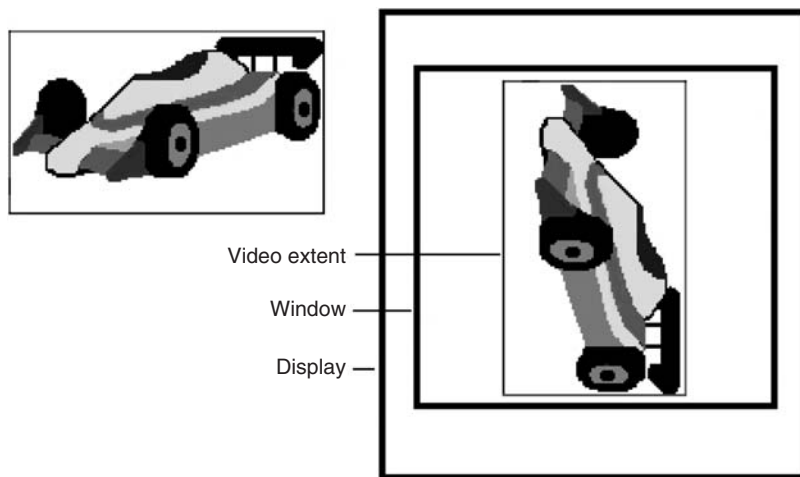


Figure 4.9 Rotation

is not the same as the orientation of the display you want to use. You can rotate the picture 90°, 180°, or 270°. In Figure 4.9, you can see an example of 90° rotation.

The orientation of the video within a window can be set with the `SetRotationL()` method:

```
void SetRotationL(TVideoRotation aRotation);
```

If the video is being displayed in multiple windows, then the results of the rotation are seen in all the windows.

The current orientation being used can be retrieved using the `RotationL()` method:

```
TVideoRotation RotationL() const;
```

For `CVideoPlayerUtility2`, a second `SetRotationL()` method exists which allows the rotation to be specified on a per-window basis:

```
void SetRotationL(const RWindowBase& aWindow, TVideoRotation aRotation);
```

This version of the method can only be called after the video opening operation has completed.

A second `RotationL()` method also exists that allows the rotation for a specific window to be retrieved:

```
TVideoRotation RotationL(const RWindowBase& aWindow);
```

4.6.9 Refreshing the Frame

You can ask for the current video frame to be refreshed on the display using `RefreshFrameL()`. This is useful if the video is paused and you need to force it to redraw.

```
void RefreshFrameL();
```

4.7 Getting Video Information

Frame Rate

The frame rate describes the number of images in the video per unit of time. For instance, a frame rate of 15 frames per second (fps) indicates that the video contains 15 separate images for each second of that video.

For videos on mobile phones, frame rates ranging from 8 fps up to 30 fps are typical. For reference, a modern console video game would run at 30–60 fps and a video telephony call would have a frame rate of 10–15 fps.

To get the frame rate of the currently open video, use the `VideoFrameRateL()` method:

```
TReal32 VideoFrameRateL() const;
```

The frame rate is a floating-point number specifying the number of frames per second. Generally, the controller returns a value that is specified in the metadata of the file or stream being played, which represents the frame rate you would expect if the video was decoded without any frames being dropped for performance reasons.

Frame Size

The frame size describes the size of each image within the video. This is typically indicated by giving the width and height of each image in pixels. So for instance a frame size of 640×480 indicates that each image has a width of 640 pixels and a height of 480 pixels.

The maximum video frame size that can be handled on a mobile phone depends on the codec and whether it runs on its own processor. Typical frame sizes for videos on mobile phones range from QCIF (176×144) up to VGA (640×480). Advances in phone hardware should soon allow high definition frame sizes, such as 1280×720 , to be used.

To get the frame size of the currently open video, the `VideoFrameSizeL()` method can be used:

```
void VideoFrameSizeL(TSize& aSize) const;
```

MIME Type

The MIME type of the video image data in the currently open video is available by calling the `VideoFormatMimeType()` method:

```
const TDesC8& VideoFormatMimeType() const;
```

Duration

The duration of the currently open video is available by calling the `DurationL()` method:

```
TTimeIntervalMicroSeconds DurationL() const;
```


Bit Rate

The bit rate describes how much information is contained within a video stream per unit of time. It is generally given as a number of bits per second (bps), kilobits per second (kbps), or megabits per second (Mbps).

The bit rate is important when streaming video over a network because the network medium can only transmit data at a certain maximum bit rate. Current mobile phones use various network media that run at various data rates, from the lower speed GSM and CDMA networks through to the high speed HSDPA and WLAN networks.

The frame size and frame rate of the video have a direct effect on the bit rate, as does the amount of compression applied by the encoding codec. The encoding codec uses a variety of techniques to compress a video in order to meet the bit rate requirements of the network medium over which it is being streamed. Lower bit rates mean more of the original video information is lost during the encoding and the video is of lower quality when it is played.

The bit rate is also used when encoding a video into a file in order to control the size of that file. So a lower bit rate means a smaller file size at the expense of lower video quality on playback.

Table 4.2 shows typical bit rates for various data bearers along with an indication of the video frame sizes and rates that you might expect on a phone that supports that medium.

Table 4.2 Data Bearer Video Capacities

| Medium | Bit rate (kbps) | Frame size and rate |
|--------|-----------------|--------------------------|
| GPRS | 48–64 | QCIF (176 × 144), 15 fps |
| 3G | 128–384 | QCIF (176 × 144), 30 fps |
| | | QVGA (320 × 240), 15 fps |
| WiFi | 500–1000 | QVGA (320 × 240), 30 fps |

To get the bit rate for the currently open video use the `VideoBitRateL()` method.

```
TInt VideoBitRateL() const;
```

This returns the average bit rate in bits per second.

4.8 Accessing Metadata

A video may contain extra items of information such as the video title or author. This information is held in a form called metadata. The metadata

format depends on the type of video being played. If the controller supports it, it is possible to read these items of metadata from the video file.

You can read the number of metadata entries using the `NumberOfMetaDataEntriesL()` method:

```
TInt NumberOfMetaDataEntriesL() const;
```

This tells you the maximum number of metadata items you can read from the file. If the controller does not support metadata then either the return value will be zero or the method will leave with `KErrNotSupported`.

Each item of metadata can be read using the `MetaDataEntryL()` method and specifying the index of the entry that you want.

```
CMMFMetaDataEntry* MetaDataEntryL(TInt aIndex) const;
```

The returned `CMMFMetaDataEntry` item allows you to access the metadata item's name and value:

```
const TDesC& Name() const;
const TDesC& Value() const;
```

Once you have finished with the `CMMFMetaDataEntry` item you need to delete it.

4.9 Controlling the Audio Output

4.9.1 Audio Resource Control

In the `CVideoPlayerUtility::NewL()` method we saw there were two parameters: `aPriority` and `aPref`. These are used to control access to an audio resource such as a loudspeaker.

The parameter, `aPriority`, is used in the situation where two clients require access to the same sound device such as the loudspeaker. While a video is being played, its audio track uses the speaker. If, during the video playback, the calendar client wishes to play an audible tone to indicate an upcoming appointment, the priority field is used to determine whether that tone can be played. Normally the preferred client is the one that sets the highest priority, however you should note that a client that has the `MultimediaDD` capability takes preference over a client that does not, even if that second client sets a higher priority. This behavior is explained in more depth in Chapter 5.

The parameter, `aPref`, is used in the situation where the audio track in a video is using a sound device, and some other client with a higher priority wants to use it. In this situation, the audio track in the video would either have to stop using the sound device, or might be mixed with the other client's audio. The `aPref` parameter is used to determine whether that is acceptable, or whether the whole video playback should stop and return an error code.

Methods exist to change these parameters after the call to `NewL()` and to read the values back:

```
void SetPriorityL(TInt aPriority, TMdaPriorityPreference aPref);
void PriorityL(TInt& aPriority, TMdaPriorityPreference& aPref) const;
```

If you try to play a video but the audio resource is already being used by a higher priority client, or you are playing a video and the audio resource is lost to a higher priority client, then your observer method `MvpuoPlayComplete` is called with the `KErrInUse` error code. At this point, you could indicate to the user in some way that the playback has been halted and wait for the user to restart the playback. This is normally acceptable because the audio resource will have been lost for an obvious reason such as an incoming phone call and, once the phone call has completed, the user can easily tell the video application to resume playback.

From Symbian OS v9.2 onwards, if you decide that you would like to automatically restart the video playing once the audio resource becomes available, then you can register for an audio notification:

```
TInt RegisterAudioResourceNotification(
    MMMFAudioResourceNotificationCallback& aCallback,
    TUid aNotificationEventUid,
    const TDesC8& aNotificationRegistrationData);
```

You need to pass a callback class which will receive the callback. To do this you need to derive your own class from the `MMMFAudioResourceNotificationCallback` class. That defines one method that you need to override in your own class:

```
class CMyAudioResourceNotification : public CBase,
    public MMMFAudioResourceNotificationCallback
{
public:
    void MarnResourceAvailable(TUid aNotificationEventId,
        const TDesC8& aNotificationData);
};
```

The call to `RegisterAudioResourceNotification` also takes a parameter `aNotificationEventUid`. This should be set to

KMMFEventCategoryAudioResourceAvailable. aNotificationRegistrationData is not used and you do not need to specify it, as a default parameter is supplied.

RegisterAudioResourceNotification() can be called before the video open operation has been performed. When the audio resource becomes available, the MarncResourceAvailable() method is called with the aNotificationEventId parameter set to KMMF-EventCategoryAudioResourceAvailable.

If you want to start playing the video again, it is important that you call the WillResumePlay() method. This tells the audio system that you want to take the audio resource back.

If the WillResumePlay() method returns KErrNone, this means you can now start playing the video again.

If you register for the audio resource notification, then you can cancel the registration later using the CancelRegisterAudioResourceNotification() method:

```
TInt CancelRegisterAudioResourceNotification(TUId aNotificationEventId);
```

The aNotificationEventId parameter should be set to KMMF-EventCategoryAudioResourceAvailable. CancelRegisterAudioResourceNotification() can be called before the video open operation has been performed.

4.9.2 Volume

You can set and get the volume for the audio track for the currently open video:

```
void SetVolumeL(TInt aVolume);
TInt Volume() const;
```

The volume is an integer value from zero, which indicates that the sound is muted, up to the maximum volume available. To obtain the maximum volume available for the audio track you can use the MaxVolume() method.

```
TInt MaxVolume() const;
```

4.9.3 Balance

The playback balance can be set with SetBalanceL() and retrieved using Balance():

```
void SetBalanceL(TInt aBalance);  
TInt Balance() const;
```

The range of values for balance runs from `KMMFBalanceMaxLeft` for maximum left balance, through `KMMFBalanceCenter` for centered balance, to `KMMFBalanceMaxRight` for maximum right balance.

4.9.4 Bit Rate

In the same way that the current video bit rate could be retrieved, the audio bit rate (in bits per second) can also be retrieved:

```
TInt AudioBitRateL() const;
```

4.9.5 Audio Type

The type of audio track in the open video can be retrieved:

```
TFourCC AudioTypeL() const;
```

This returns the FourCC code of the audio data.

4.9.6 Enabling Audio Playback

From Symbian OS v9.4 onwards, it is possible to specify whether the audio track in a video should be played back or not. This can be set by calling `SetAudioEnabledL()`. If you specify that audio is not enabled, then only the video image data is played.

```
void SetAudioEnabledL(TBool aAudioEnabled);
```

You can also find out whether audio is currently enabled or not.

```
TBool AudioEnabledL() const;
```

4.10 Streaming Playback

We have seen in previous sections some of the client API methods that are specific to video streaming. There are some other important things that need to be considered when streaming is supported.

We have seen that the `OpenUrlL()` method on `CVideoPlayerUtility` or `CVideoPlayerUtility2` is used to open a video for

streaming. What types of streaming are supported depends on the controllers that are installed. You may find that a controller can support on-demand streaming, live streaming, or both. Controllers may support streaming only from an RTSP server or they may support HTTP progressive download.

If you are using a live-streaming URL, then it is not possible for the MMF framework to automatically detect the format of the video. In this case, you must specify the controller UID as shown in Section 4.4.

While streaming data from a server, the playback might pause occasionally because extra data needs to be downloaded or buffered from the server. You might want to consider using the video-loading progress methods to show when this buffering is taking place so that the user does not think that your application has crashed:

```
void GetVideoLoadingProgressL(TInt& aPercentageComplete);
```

It is also possible to request callbacks to provide information about buffering during video streaming using the `RegisterForVideoLoadingNotification()` method.

```
void RegisterForVideoLoadingNotification(  
                                         MVideoLoadingObserver&  
                                         aCallback);
```

You need to provide a class that is derived from the `MVideoLoadingObserver` base class and overrides the `MvloLoadingStarted()` and `MvloLoadingComplete()` methods. Whenever a loading or buffering operation starts, the `MvloLoadingStarted()` method is called. Once the loading or buffering completes, the `MvloLoadingComplete()` method is called. When streaming, it may not be possible for the whole video to be received in one go. In this case, multiple calls to the methods will occur.

`RegisterForVideoLoadingNotification()` can be called before the video open operation has been performed.

Certain methods don't really make any sense during live streaming, such as pausing or seeking to a new position using the `SetPosition()` method. If your application knows that live streaming is being used, then you might consider disabling these operations in your application so that the end user cannot access them.

4.11 Recording Video

Figure 4.10 shows the user interactions for the basic video record, pause, stop sequence for a file. They show the API method calls from the user

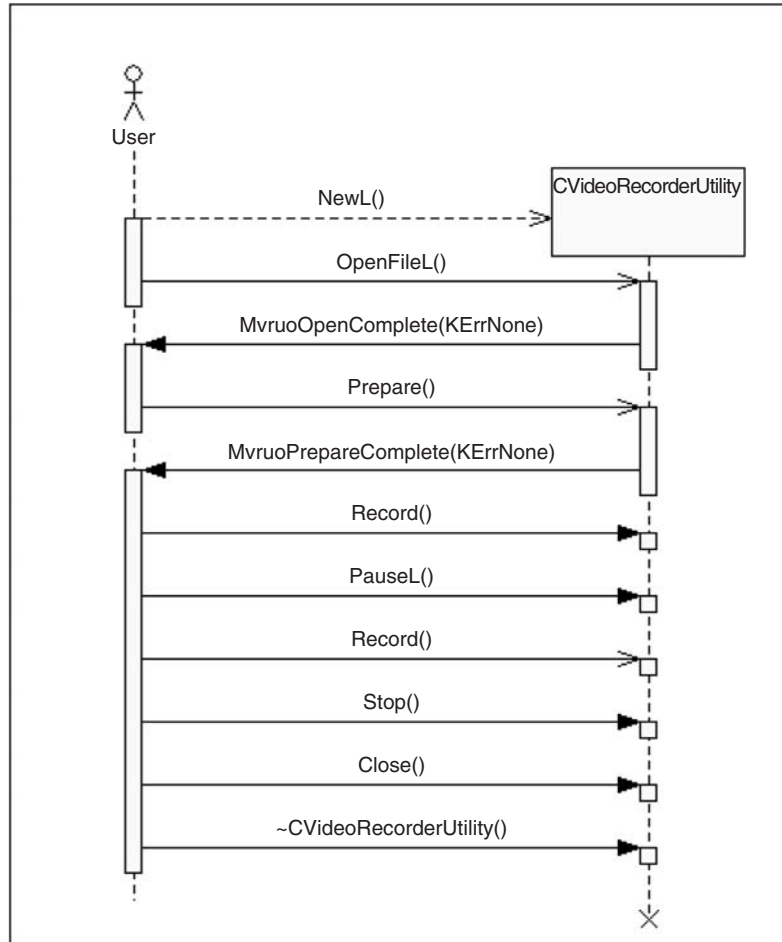


Figure 4.10 Recording sequence

and the callbacks from the CVideoRecorderUtility class. In the following sections, we look at these API method calls in more detail.

4.11.1 Creating the Class

The CVideoRecorderUtility is instantiated by calling its static NewL() method.

```

static CVideoRecorderUtility* NewL(
    MVideoRecorderUtilityObserver& aObserver,
    TInt aPriority,
    TMdaPriorityPreference aPref);

```

As with video playback, the video recorder class takes a parameter, `aObserver`, which is used to receive callbacks when asynchronous functions have completed. The `aPriority` and `aPref` parameters serve the same purpose as the parameters passed to `CVideoPlayerUtility::NewL()` method. See Section 4.9.1 for more details about them.

The observer parameter for the `NewL()` method is a class that you must derive from the base class `MVideoRecorderUtilityObserver`. Your class must override the methods defined in the `MVideoRecorderUtilityObserver` class.

```
class CMyVideoRecorderObserver : public CBase,
    public MVideoRecorderUtilityObserver
{
public:
    void MvruoOpenComplete(TInt aError);
    void MvruoPrepareComplete(TInt aError);
    void MvruoRecordComplete(TInt aError);
    void MvruoEvent(const TMMFEvent& aEvent);
};
```

These methods are very similar to the playback observer that is described in Section 4.5. Please refer to that section for a fuller description.

4.11.2 Opening the Video for Recording

You can open a file to record a video into using the `OpenFileL()` methods:

```
void OpenFileL(const TDesC& aFileName, TInt aCameraHandle,
    TUid aControllerUid, TUid aVideoFormat,
    const TDesC8& aVideoType, TFourCC aAudioType);
void OpenFileL(const RFile& aFile, TInt aCameraHandle,
    TUid aControllerUid, TUid aVideoFormat,
    const TDesC8& aVideoType, TFourCC aAudioType);
```

You have the option of specifying a file name into which the video will be recorded or a handle to a file that is already open for writing.

The `aCameraHandle` parameter is the handle of the camera that you want to record from. Your camera should be created and set up ready to start recording. The handle can be obtained using the `CCamera::Handle()` method. See Section 3.6 for more information on setting up the camera for recording.

When recording a video, you need to specify the UID of the controller and format that you want to use. You can use code that is very similar

to the example code in Section 4.4 to do this. Instead of requesting the `PlayFormats()` for a controller, you choose the record formats, using the `RecordFormats()` method:

```
RMMFFormatImplInfoArray& recordFormats = controllers[i]->RecordFormats();
```

You can also specify which video codec and which audio codec to use for the recording. The `aVideoType` parameter is used to specify the MIME type of the video codec and the `aAudioType` parameter is used to specify the FourCC code of the audio codec. On S60 phones you don't need to specify the codec types if you don't want to (the parameters have default NULL values); in this case, the controller chooses for you. On UIQ phones, you are required to choose the codec types. Failing to set the video type results in an error code being returned. Failing to set the audio type means the recording has no audio.

Where multiple codecs are available to choose from, you need to think about how to handle the choice. You could ask the user to choose or you could try to make the decision in your application. The choice of codec depends on a number of factors. You may want to maintain compatibility with other devices which, for instance, may not support more advanced formats such as the H.264 video format, or you may want the high level of compression that H.264 gives to save storage space on the device. For audio, you may choose the AMR codec, if you know you will be recording mostly speech, or you may choose a more compatible general-purpose format, such as AAC.

You can choose to record your video into a descriptor buffer:

```
void OpenDesL(TDes8& aDescriptor, TInt aCameraHandle,
             TUid aControllerUid, TUid aVideoFormat,
             const TDesC8& aVideoType, TFourCC aAudioType);
```

This has the same parameters as the `OpenFileL()` methods, except that it takes a descriptor instead of the file name or file handle.

You can also choose to stream your video to a specified URL; however, you should note that the majority of Symbian smartphones do not currently support this.

```
void OpenUrlL(const TDesC& aUrl, TInt aIapId, TInt aCameraHandle,
             TUid aControllerUid, TUid aVideoFormat,
             const TDesC8& aVideoType, TFourCC aAudioType);
```

You need to specify the URL to stream the recording to and the identifier for the access point that should be used to connect to the URL.

All the open commands are asynchronous and you must wait for the `MvruoOpenComplete()` method of your observer to be called before proceeding. If the open operation fails, then the initial API call can leave with an error or an error code can be passed to the observer method. You need to handle both these scenarios when using these APIs.

4.11.3 Preparing to Record

Once the video open operation has completed, you need to prepare to start the recording by calling the `Prepare()` method. This allows the video controller to allocate the buffers and resources it requires to perform the recording.

```
void Prepare();
```

This is an asynchronous method, and you should wait for the `MvruoPrepareComplete()` method of your observer to be called before proceeding.

4.11.4 Starting the Recording

To start recording the video, you call the `Record()` method. This must be done after the prepare operation has completed.

```
void Record();
```

This is an asynchronous method. When the recording finishes the `MvruoRecordComplete()` method of your observer is called. Note however that the method is not called if you specifically call `Stop()` to stop the recording.

4.11.5 Pausing the Recording

You can pause the current recording using `PauseL()`:

```
void PauseL();
```

4.11.6 Stopping the Recording

You can stop the current recording using `Stop()`:

```
TInt Stop();
```

4.11.7 Closing the Recorded File

After the recording has completed or been stopped, you can close the file that is storing the video:

```
void Close();
```

4.12 Controlling the Video that Is Recorded

Frame Rate

The frame rate for the recording can be specified using the `SetVideoFrameRateL()` method. You can read the current frame rate being used with the `VideoFrameRateL()` method.

```
void SetVideoFrameRateL(TReal32 aFrameRate);  
TReal32 VideoFrameRateL() const;
```

The frame rate is a floating-point number specifying the number of frames per second.

From Symbian OS v9.4 onwards, you can specify whether the frame rate is fixed or variable using the `SetVideoFrameRateFixedL()` method.

```
void SetVideoFrameRateFixedL(TBool aFixedFrameRate);
```

This method indicates to the controller that it should attempt to output a constant frame rate and it may result in a lower picture quality being output at lower bit rates. You should note that this is only an indication to the controller that you would like a constant frame rate. The controller may not be fully able to meet your request and may still skip some frames.

You can check whether the frame rate is fixed using the `VideoFrameRateFixedL()` method.

```
TBool VideoFrameRateFixedL() const;
```

By default, the frame rate is not fixed, although at higher bit rates the controller may be able to output a constant frame rate anyway.

Frame Size

You can specify the frame size for the recording using the `SetVideoFrameSizeL()` method. You can retrieve the current frame size using the `GetVideoFrameSizeL()` method.

```
void SetVideoFrameSizeL(const TSize& aSize);  
void GetVideoFrameSizeL(TSize& aSize) const;
```

Duration

You can get the duration in microseconds of the current video recording:

```
TTimeIntervalMicroSeconds DurationL() const;
```

Bit Rate

You can set the video bit rate (in bits per second) that should be used for the recording using the `SetVideoBitRateL()` method. You can read the current video bit rate using the `VideoBitRateL()` method.

```
void SetVideoBitRateL(TInt aBitRate);  
TInt VideoBitRateL();
```

Maximum Clip Size

You can specify a maximum size for a video recording. This limits the recording size to a maximum number of bytes.

```
void SetMaxClipSizeL(TInt aClipSizeInBytes);
```

If you don't want to specify a limit, then set the `aClipSizeInBytes` parameter to `KMMFNoMaxClipSize`.

Recording Time Available

You can ask the controller for an indication of how much longer the current video recording can continue using the `RecordTimeAvailable()` method. The controller examines the available resources, such as file space and buffer space, in order to return an estimate of the time remaining in microseconds.

```
TTimeIntervalMicroSeconds RecordTimeAvailable() const;
```

Video Type

You can get a list of the video codecs supported by the currently selected controller using the `GetSupportedVideoTypesL()` method.

```
void GetSupportedVideoTypesL(CDesC8Array& aVideoTypes) const;
```

This method populates your array of descriptors `aVideoTypes` with the names of supported video codecs. The format of the video types is not specified. Depending on the controller, the method might return a list of video MIME types or it might return a list of FourCC codes; you need to ensure that you can handle either.

Once you have the list of supported video types, you can select the one you want using the `SetVideoTypeL()` method:

```
void SetVideoTypeL(const TDesC8& aType);
```

Where a controller has found multiple codecs that could be used, it will already have chosen one that it thinks is suitable. There is no need for you to change this unless you have a requirement that a specific codec is used. For some information on how to choose a codec to use, see Section 4.11.2.

Getting MIME Type

You can get the MIME type of the video format that the controller is currently using with the `VideoFormatMimeType()` method:

```
const TDesC8& VideoFormatMimeType() const;
```

Pixel Aspect Ratio

The concept of pixel aspect ratio was first introduced in ‘Scaling’, Section 4.6.4. Pixel aspect ratio is important while recording a video. If you know the aspect ratio of the display that your recorded video will be shown on, then you can adjust the pixel aspect ratio in order that the video scales properly to the display. Typically, Symbian smartphone displays use a pixel aspect ratio of 1:1. For TV broadcast signals, high definition TV uses 1:1, NTSC uses 10:11 and PAL uses 59:54.

If you want to change the pixel aspect ratio used to encode video data, you can ask the controller for a list of pixel aspect ratios that it supports.

```
void GetSupportedPixelAspectRatiosL(  
    RArray<TVideoAspectRatio>& aAspectRatios) const;
```

This returns an array of `TVideoAspectRatio`; this class stores two public data members, the `iNumerator` and `iDenominator`, which give the X and Y pixel widths respectively.

Given the list of supported ratios, you can then choose which one you want. To set the pixel aspect ratio, you should call the `SetPixelAspectRatiosL()` method and pass the `TVideoAspectRatio` containing your choice.

```
void SetPixelAspectRatiosL(const TVideoAspectRatio& aAspectRatio);
```

You can check which pixel aspect ratio is currently being used by calling the `GetPixelAspectRatiosL()` method which will fill a `TVideoAspectRatio` passed to it with the correct values:

```
void GetPixelAspectRatiosL(TVideoAspectRatio& aAspectRatio) const;
```

Video Quality

From Symbian OS v9.4, it is possible to control the quality of the output video recording using the `SetVideoQualityL()` method:

```
void SetVideoQualityL(TInt aQuality);
```

The `aQuality` parameter is a value from 0 to 100 which defines the quality that you require. A quality value of 0 indicates the lowest quality possible, a value of 50 indicates a normal quality, and value of 100 indicates the highest quality lossless output possible. Whilst you can set any quality value between 0 and 100, it is preferable to use the values defined in the `TVideoQuality` enumeration which defines values for low, normal, high and lossless quality. If a controller does not support the exact quality value that you specify, it uses the nearest quality value that it does support.

You should note that setting the quality value may cause the controller to modify the bit-rate setting that it is using in order to meet the quality value that you specified. The controller may also modify the frame rate, unless you have previously specified that the frame rate is fixed by calling the `SetVideoFrameRateFixedL()` method. Note also that changing the bit-rate or frame-rate settings overrides the quality setting that you requested.

You can query the current quality value using the `VideoQualityL()` method:

```
TInt VideoQualityL() const;
```

The `VideoQualityL()` method always returns the value set by the most recent call to `SetVideoQualityL()`. If you have subsequently changed the bit rate or frame rate, the value it returns will not have been adjusted to take into account those changes.

Enabling Video Recording

From Symbian OS v9.4 onwards, it is possible to request that only the audio track is stored in the video by disabling the video using the `SetVideoEnabledL()` method.

```
void SetVideoEnabledL(TBool aEnabled);
```

You can check whether the video is enabled or not using the `VideoEnabledL()` method.

```
TBool VideoEnabledL() const;
```

4.13 Controlling the Audio that Is Recorded

Priority

Section 4.9.1 gave details about the priority value that is used to resolve resource conflicts for the audio devices. The priority can also be set and retrieved directly:

```
void SetPriorityL(TInt aPriority, TMdaPriorityPreference aPref);  
void GetPriorityL(TInt& aPriority, TMdaPriorityPreference& aPref) const;
```

These methods take the same parameters as the `CVideoRecorderUtility::NewL()` method.

Bit Rate

It is possible to set and get the audio bit rate (in bits per second) that should be used when encoding the video.

```
void SetAudioBitRateL(TInt aBitRate);  
TInt AudioBitRateL() const;
```

Audio Type

It is possible to get a list of all the audio types that a controller can handle.

```
void GetSupportedAudioTypesL(RArray<TFourCC>& aAudioTypes) const;
```

This returns an array of FourCC codes each of which identifies one supported audio codec type.

Given the list of supported audio types, you can then select which one you want to use when encoding the audio track for the video by calling `SetAudioTypeL()` and passing the chosen FourCC code to it.

```
void SetAudioTypeL(TFourCC aType);
```

Where a controller has found multiple audio types that could be used, it will already have chosen one that it thinks is suitable. There is no need for you to change this unless you have a requirement that a specific codec is used. For information on how to choose a codec to use, see Section 4.11.2.

You can retrieve which audio type is currently in use by using the `AudioTypeL()` method.

```
TFourCC AudioTypeL() const;
```

Audio Enabling

You can specify whether or not you want your recorded video to contain audio using the `SetAudioEnabledL()` method:

```
void SetAudioEnabledL(TBool aEnabled);
```

You can check whether you are storing the audio using the `AudioEnabledL()` method.

```
TBool AudioEnabledL() const;
```


Audio Gain

To find out the maximum audio recording gain that the controller can support, you can call the `MaxGainL()` method:

```
TInt MaxGainL() const;
```

Once you know the maximum gain value supported, you can use this to set the audio recording gain to be any value from 0 to the maximum gain:

```
void SetGainL(TInt aGain);
```

You can check the current gain value by calling the `GainL()` method:

```
TInt GainL() const;
```

Number of Audio Channels

You can find out the range of supported audio channels for the controller using the `GetSupportedAudioChannelsL()` method:

```
void GetSupportedAudioChannelsL(RArray<TUint>& aChannels) const;
```

This returns an array of integers, each entry being a supported number of audio channels. Using this information, you can then set the number of audio channels that you would like to use for the recording:

```
void SetAudioChannelsL(const TUint aNumChannels);
```

You can also find out how many audio channels are currently being used:

```
TUint AudioChannelsL() const;
```

Audio Sample Rates

You can obtain a list of audio sample rates that the controller supports using the `GetSupportedAudioSampleRatesL()` method:

```
void GetSupportedAudioSampleRatesL(RArray<TUint> &aSampleRates) const;
```

This returns an array of integers, each entry being a sample rate that is supported.

You can set the audio sample rate to use with the `SetAudioSampleRateL()` method:

```
void SetAudioSampleRateL(const TUInt aSampleRate);
```

You can retrieve the audio sample rate currently being used by calling `AudioSampleRateL()`:

```
TUInt AudioSampleRateL() const;
```

4.14 Storing Metadata

You can get the number of metadata entries and access individual metadata entries for your recorded video:

```
TInt NumberOfMetaDataEntriesL() const;  
CMMFMetaDataEntry* MetaDataEntryL(TInt aIndex) const;
```

These methods are the same as for video playback. See Section 4.8 for a full description.

If you have the index of a metadata item, then you can remove or replace it if you want to. To remove a metadata item use the `RemoveMetaDataEntryL()` method specifying the index of the entry:

```
void RemoveMetaDataEntryL(TInt aIndex);
```

To replace a metadata item, use the `ReplaceMetaDataEntryL()` method. You need to specify the index of the existing item and a `CMMFMetaDataEntry` containing the new information:

```
void ReplaceMetaDataEntryL(TInt aIndex,  
                           const CMMFMetaDataEntry &aNewEntry);
```

It is possible to add a new item of metadata using the `AddMetaDataEntryL()` method:

```
void AddMetaDataEntryL(const CMMFMetaDataEntry& aNewEntry);
```

This method takes a `CMMFMetaDataEntry` parameter containing the information about the new item of metadata. The controller can then use the name and value data members within the `CMMFMetaDataEntry` class to add the metadata to the recorded video.

4.15 Custom Commands

Custom commands are an extension mechanism that allows controller-specific functionality to be accessed by third-party developers. The commands can be sent directly to the controllers from the client APIs. As the functionality is controller-specific, the authors of the controller would have to choose to make information publicly available about what the functionality is and how to access and use it.

The areas of controller functionality are called interfaces, and there can be a number of methods per interface. Each interface has its own UID and the client can call a specific method on the interface by specifying the UID and a function number.

Both video playback and video recording classes provide the same set of methods to access controller-specific custom commands.

```
TInt CustomCommandSync(const TMMFMessageDestinationPckg& aDestination,
                      TInt aFunction, const TDesC8& aDataTo1,
                      const TDesC8& aDataTo2, TDes8& aDataFrom);
TInt CustomCommandSync(const TMMFMessageDestinationPckg& aDestination,
                      TInt aFunction, const TDesC8& aDataTo1, const TDesC8& aDataTo2);
void CustomCommandAsync(const TMMFMessageDestinationPckg& aDestination,
                      TInt aFunction, const TDesC8& aDataTo1,
                      const TDesC8& aDataTo2, TDes8& aDataFrom,
                      TRequestStatus& aStatus);
void CustomCommandAsync(const TMMFMessageDestinationPckg& aDestination,
                      TInt aFunction, const TDesC8& aDataTo1,
                      const TDesC8& aDataTo2, TRequestStatus& aStatus);
```

There are both synchronous and asynchronous versions of the custom command methods.

Each method has a parameter called `aDestination`, which is a `TMMFMessageDestinationPckg`. This is a `TPckgBuf` containing a `TMMFMessageDestination`. The `TMMFMessageDestination` contains the UID of the interface that the controller exposes. Each method also has a parameter called `aFunction`. This specifies which function to call on the exposed interface. The values for the other parameters are specific to the interface and function exposed by the controller.

4.16 Examples and Troubleshooting

The various developer forums for Symbian OS are a good source of example code. For instance, the Forum Nokia website has an audio and video resources page at www.forum.nokia.com/main/resources/technologies/audiovideo/audio_video_documents.html#symbian.

That page contains a number of useful example applications, including a video example application. While the application is specifically aimed at the Nokia S60 3rd Edition SDK, the code that uses the client API methods is generic across all platforms and could be used for applications written for other SDKs.

It can be very useful to download videos in a format that you can use for testing. There are a large number of websites, such as www.mobile9.com and www.mobango.com, that allow you to download video content for use on your phone. If you can't find the format you are looking for, then there are a large number of video conversion tools available which can convert from one format to another. For instance, eRightSoft SUPER (www.erightsoft.com/SUPER.html) supports a very extensive set of container and codec formats.

If you are planning to stream video from the Internet then you need to be aware that some network operators restrict access to video streaming services. They may use a 'walled garden' or 'whitelist' system, only allowing their own hosted content or that of specific third parties. If you are having trouble connecting to your server then this may be the issue. You can check this by using a WLAN-enabled phone or another network to connect to the same server.

Another major consideration for video-streaming applications is selecting the frame size and bit rate appropriately. If you try to stream video content with a larger frame size or bit rate than the device supports, usually only the audio component is played. There is a wide variation in maximum sizes supported by current smartphones from QCIF (176×144) up to VGA (640×480). If you are producing the content, you either need to provide different versions for different handsets or select a size small enough to be supported on all of your target devices. If your application uses third-party content then you should check the frame size and warn the user about unsupported content where possible. The platform video controllers generally just provide their best effort to play whatever is requested.

Similarly, for the selection of appropriate bit rates, you need to test your applications with all the bearers you expect to support. The maximum

bit rates supported on a bearer are rarely achieved in practice (see Table 4.2 in Section 4.7 for some realistic values for common bearers). If you attempt to use too high a bit rate, then either the streaming server automatically adapts, resulting in visible loss of quality, or the video may stall, re-buffer or stop.

5

Multimedia Framework: Audio

5.1 Introduction

This chapter is about adding features to your Symbian OS applications to play or record audio. As we stated in Chapter 2, there is a whole subsystem dedicated to playing audio files, clips and buffers; it can be quite daunting at first – especially deciding which of the various API classes is the most suitable.

The sample code quoted in this chapter is available in the Symbian Developer Library documentation (found in your SDK or online at developer.symbian.com). You'll find it in the Symbian OS Guide, in the Multimedia section (there are a set of useful 'How to...' pages which give sample code and sequence diagrams for each task). We also recommend that you take a look at the sample code that accompanies two of our most recent books. Check out the code for Chapter 4 of *Games on Symbian OS* by Jo Stichbury *et al.* (2008) (developer.symbian.com/gamesbook) and the code for *Symbian C++ for Mobile Phones Volume 3* by Richard Harrison & Mark Shackman (2007) (developer.symbian.com/main/documentation/books/books_files/scmp_v3/index.jsp). For each book, the source code can be downloaded from the link near the bottom right corner of the page. Feel free to re-use and extend it – and if you want to share your updates, you can post them on the wiki page for this book at developer.symbian.com/multimediabook-wiki.

In this chapter, we go through the classes in turn, looking at what the classes are used for or how they mainly are used, and also a bit on what not to try. Before we do this, it might be worth just noting the main APIs for reference. They are shown in Figure 5.1.

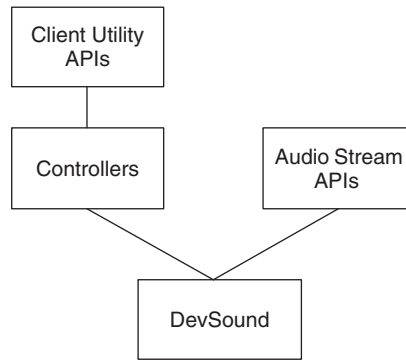


Figure 5.1 The Symbian OS audio subsystem

This is a very simplified view compared to what we show in the architectural pictures, but it shows how the various APIs are organized. Roughly speaking, there are two groups:

- Audio client utilities work on files and the like. Most importantly, they are implemented using ‘controllers’ – special plug-ins that run in their own threads. Examples are `CMdaAudioPlayerUtility` and `CMdaAudioRecorderUtility`.¹
- Audio stream utilities work primarily with buffers of data and rely on the lower-level adaptation (DevSound, and below) for support for different formats. Examples include `CMdaAudioInputStream` and `CMdaAudioOutputStream`.²

There is a third group that does not naturally fit into these categories. For example, `CMdaAudioToneUtility` plays files, but the implementation is somewhat different and internally it has more in common with `CMdaAudioOutputStream` than, say, `CMdaAudioPlayerUtility`. We usually group it with the stream utilities, but that is for internal reasons and from an API view it is perhaps best viewed as a special case.

DevSound is a lower-level API that is used, behind the scenes, by the others. It represents the ‘audio adaptation’, which is the level on the device hardware – for example, it encapsulates hardware-specific codecs and the various sound drivers themselves. An applications developer can largely ignore this – just take it as read that the code at this layer can vary

¹To use these utilities, include `MdaAudioSamplePlayer.h` or `MdaAudioSampleEditor.h` and link against `mediaclientaudio.lib`.

²To use these, include `MdaAudioInputStream.h` or `MdaAudioOutputStream.h` respectively and link against `mediaclientaudioinputstream.lib` or `mediaclientaudiostream.lib`.

tremendously from one smartphone model to the next. Symbian OS is designed to support hardware-accelerated audio encoding and decoding but you need not really consider it – it all depends on how the various plug-ins are implemented.

Technically some of the adaptation exists at the controller level – that is, different devices have different audio plug-ins. This should not concern you much, except to accept that the file formats supported from one model to the next vary. However, we look at how the code can be written in a portable manner that does not place too much reliance on the specifics of implementation³ – you can write code that is highly dependent on the specifics of a particular device but this should usually only be used by test code.

5.2 Audio Input and Output Streams

These simple APIs are about playing or storing descriptors full of audio data. They are very simple APIs and are effectively thin layers over DevSound (see Section 5.7). There is no separate thread and the streams are more limited in the formats supported than the client utilities described above.

As APIs, audio input and output streams are fairly simple, related to the recording and playing of audio data to or from descriptor buffers. In describing them, it is perhaps preferable to state what they are not: they are not an alternative to the client utilities and they do not directly deal with the same sort of audio data. Audio client utilities generally deal with formatted audio files (also known as ‘wrapped audio’), where the audio data is held in an audio file, such as a WAV file, along with headers and other extra information. The code that deals with these files has to know how to process the headers and to jump, on request, to the audio data itself. The audio input and output streams, on the other hand, deal just with the audio data itself, for example, PCM, or similar, data. In some ways, playing and recording PCM data is what they are best suited to; theoretically they can be used with any format supported by the DevSound level, but sometimes formats need additional setup calls using private or non-portable APIs. It is perhaps best to stick to PCM – be it 8- or 16-bit – as it is the most useful.

The other thing to note is that these calls are not primarily intended for ‘streaming data’, as in playing streamed data from the web. There is nothing to stop that use, specifically, except that for much of the time the format is not appropriate – if the downloaded data is PCM, it would work; if it is WAV, AAC or similar, then it won’t – at least, not without a lot of additional work in your application.

³For more advice on this subject, see Chapter 8.

In practice, these APIs mainly cover two key use cases and a third to a lesser extent:

- An application contains its own audio engine (typically portable code) that processes files, etc. and contains its own codecs, producing or accepting PCM data at the lower level. This is a typical scenario where you are porting code from other systems. Audio input and output streams are the recommended approach at the lowest level.
- Your program generates audio data or needs to process it. This is in many ways similar to the previous category but it typically relates to the function and not because you have existing code – that is, you’d write new code this way too.
- Your program plays ‘raw’ audio data from files or resources. This is more portably done using `CMdaAudioPlayerUtility` but using `CMdaAudioOutputStream` uses fewer run-time resources and starts more quickly.

To a large extent, `CMdaAudioInputStream` is the mirror image, and again primarily covers the first two use cases – it can also be used to fill your own buffers with ‘raw’ audio data, encoded but unwrapped.

5.2.1 Example

This example uses an audio input stream and an output stream to record some data and then play it back. The example does not assume it is being run from a GUI environment, but that would be the standard use case. It is structured as an ‘engine’ with an associated API and callback, and can thus be called from any C++ code running with an active scheduler and using the usual Symbian OS approach (TRAP–leave) for exceptions.

The code roughly breaks into two – input and output – but first let’s look at the ‘common’ code: the headers and constructors. The header declares the main class `CIOStreamAudio` and its observer `MAudioIOStreamObserver`.

```
class MAudioIOStreamObserver
{
public:
    // Signal recording has stopped - for info
    virtual void RecordingStopped() = 0;
    // Signal completion of test with associated error
    virtual void PlayOrRecordComplete(TInt aError) = 0;
};

class CIOStreamAudio : public CBase,
                      public MMdaAudioInputStreamCallback,
                      public MMdaAudioOutputStreamCallback
{

```

```

enum TState
{
    EStateIdle,
    EStateOpeningInput,
    EStateReading,
    EStateOpeningOutput,
    EStateWriting,
    EStateWritingPostBuffer
};

public:
    static CIOStreamAudio* NewL(MAudioIOStreamObserver* aObserver);
    ~CIOStreamAudio();

public:
    void RecordL(TInt aBufferSize);
    void StopAndPlay();

public:
    // from MMdaAudioInputStreamCallback
    void MaiscOpenComplete(TInt aError);
    void MaiscBufferCopied(TInt aError, const TDesC8& aBuffer);
    void MaiscRecordComplete(TInt aError);
    // from MMdaAudioOutputStreamCallback
    void MaoscOpenComplete(TInt aError);
    void MaoscBufferCopied(TInt aError, const TDesC8& aBuffer);
    void MaoscPlayComplete(TInt aError);

protected:
    CIOStreamAudio(MAudioIOStreamObserver* aObserver);
    void ConstructL();
    void OnOpenL();
    void ReadNextBlock(const TDesC8& aBuffer);
    void WriteNextBlockL(const TDesC8& aBuffer);
    void Complete(TInt aError);

private:
    MAudioIOStreamObserver*const iObserver;
    TState iState;
    RBuf8 iMainBuffer;
    CMdaAudioInputStream* iInputStream;
    CMdaAudioOutputStream* iOutputStream;
    TPtr8 iBufferPtr;
};

CIOStreamAudio* CIOStreamAudio::NewL(MAudioIOStreamObserver* aObserver)
{
    CIOStreamAudio* self = new(ELeave) CIOStreamAudio(aObserver);
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop(self);
    return self;
}

CIOStreamAudio::~~CIOStreamAudio()
{
    {
        if (iInputStream)
        {
            iInputStream->Stop();
            delete iInputStream;
        }
    }
    if (iOutputStream)
    {

```

```

        iOutputStream->Stop();
        delete iOutputStream;
    }
    iMainBuffer.Close();
}

void CIOStreamAudio::ConstructL()
{
    iInputStream = CMdaAudioInputStream::NewL(*this);
    iOutputStream = CMdaAudioOutputStream::NewL(*this);
}

CIOStreamAudio::CIOStreamAudio(MAudioIOStreamObserver* aObserver) :
    iObserver(aObserver),
    iBufferPtr(NULL, 0)
{
}

void CIOStreamAudio::Complete(TInt aError)
{
    iState = EStateIdle;
    iObserver->PlayOrRecordComplete(aError);
}

```

This code follows the usual Symbian coding idioms. Notice how in the destructor, both `iInputStream` and `iOutputStream` are stopped prior to deletion (if they are not playing this does no harm) and the code first checks the pointers are not `NULL`. It is very important to do this since the pointers may be `NULL` in some low-memory scenarios.

5.2.2 Stream Calls for Recording

Like most of the audio APIs, `CMdaAudioInputStream` operates in two phases: initialization and the action itself. The key operations are shown below: `RecordL()` is called by the application and the `MaioOpenComplete()` callback is called from `CMdaAudioInputStream` itself.

```

void CIOStreamAudio::RecordL(TInt aBufferSize)
{
    // if buffer is already allocated, throw away and re-create
    iMainBuffer.Close();
    User::LeaveIfError(iMainBuffer.Create(aBufferSize));
    iInputStream->Open(NULL); // use default settings
    iState = EStateOpeningInput;
}

```

The `RecordL()` operation initializes `iMainBuffer` (where we store the recorded data) and opens the input stream. Technically initialization

of the main buffer could be delayed until first use, which follows below, but doing it here is easier to drive from your client code – usually the leave happens when you are handling a key-click or screen event and it is simpler to relay it back to the user. The key thing, though, is the call to `CMdaAudioInputStream::Open()`. This is asynchronous – it generates a `MaiscOpenComplete()` callback asynchronously, after `Open()` has returned. We now almost always use `NULL` as the parameter; originally, if you wanted to override settings such as sample rate and format, this is where you did it, but the API has evolved and now we do this a bit later in the sequence. You'll probably notice the states in this scenario – these are purely to do with the example code and its asynchronous nature.

Following the `Open()` call, you get a `MaiscOpenComplete()` callback:

```
void CIOStreamAudio::MaiscOpenComplete(TInt aError)
{
    ASSERT(iState == EStateOpeningInput);
    if (aError != KErrNone)
    {
        Complete(aError);
    }
    else
    {
        iMainBuffer.Zero();
        iState = EStateReading;
        ReadNextBlock(KNullDesC8); // kick off a new read -
        // KNullDesC8 for first buffer
    }
}
```

If the error code is not `KErrNone`, opening failed for some reason and we send a failure callback to the application via `Complete()`. Otherwise, we reset the main buffer via `Zero()` and start to read via `ReadNextBlock()` – note the trick of supplying `KNullDesC8` to state that nothing was read in the ‘previous iteration’, which is definitely true since this is the first! It is perhaps worth noting the use of `KNullDesC8`, instead of `KNullDesC`. Essentially, where the audio APIs use descriptors to pass data back and forth, we almost always use eight-bit descriptors not (standard) 16-bit ones. This should make sense – you use eight-bit descriptors for binary data.

Next let's look at `ReadNextBlock()`:

```
void CIOStreamAudio::ReadNextBlock(const TDesC8& aBuffer)
{
    ASSERT(iState == EStateReading);
    // buffer is tail of iMainBuffer. Shift length of
    // iMainBuffer and get the next bit
    TInt lengthRecorded = iMainBuffer.Length() + aBuffer.Length();
}
```

```

iMainBuffer.SetLength(lengthRecorded);
iBufferPtr.Set(const_cast<TUInt8*> (iMainBuffer.Ptr()) + lengthRecorded,
               0, iMainBuffer.MaxLength() - lengthRecorded);
TRAPD(error, iInputStream->ReadL(iBufferPtr));
if (error != KErrNone)
{
    Complete(error);
}
}

```

We need to point out that there are actually two buffers mentioned by this class: `iMainBuffer` is the whole recorded length of data and `iBufferPtr` is used as a running ‘save here’ reference. In its standard use, this program is used to record two seconds of data via a timer in the application and the `aBuffer` size parameter is set to always be large enough – the default of 8000 samples per second, 16 bits per sample and mono works out to be 16 KB/s, so 40 KB should be ample. So we want to record into a large descriptor. However, by default, audio recording works by generating lots of short buffers, and if you ask for too much data it actually returns only so much each time. Typically this data is 4–16 KB, but it could be larger and could even be smaller. The thing to do is not to write your application assuming the size of the underlying low-level buffers.

`ReadNextBlock()` works by arranging for `iBufferPtr` to point to the rest of `iMainBuffer` – that is the part of `iMainBuffer` that has not yet been recorded to. In the next iteration, the length of `iMainBuffer` is updated to include newly recorded data and `iBufferPtr` is updated to point to what is now the unrecorded section.

This example ignores the problem of the buffer being too small, but it should really just stop recording if there is insufficient space in the buffer – you are free to update the code to handle that.

The call to request a new buffer of data is `CMdaAudioInputStream::ReadL()`, which generates a subsequent, asynchronous `MaiscBufferCopied()` callback. Basically if you get an error, apart from `KErrAbort`, you stop processing and tell the main client. Otherwise, you keep processing – in this program calling `ReadNextBlock()`.

```

void CIOStreamAudio::MaiscBufferCopied(TInt aError, const TDesC8& aBuffer)
{
    ASSERT(iState == EStateReading);
    if (aError != KErrNone)
    {

```

```

    if (aError != KErrAbort) // aborts happen on Stop as buffers are
                           // recovered; need to ignore
    {
        iInputStream->Stop();
        Complete(aError);
    }
}
else
{
    ReadNextBlock(aBuffer);
}
}

```

It is perhaps worth noting that the API itself supports double or quadruple buffering – you can fire off several `ReadL()` calls and they will be responded to sequentially with separate `MaiscBuffer-Copied()` calls. Thus if you have buffers `buffer1` and `buffer2`, you can do `ReadL(buffer1)` immediately followed by `ReadL(buffer2)`. The system adds data to `buffer1` and calls `MaiscBufferCopied()` for `buffer1`; it then adds data to `buffer2` and calls `MaiscBufferCopied()` for `buffer2`; the `aBuffer` parameter says what the associated buffer is, as well as giving the length of the data actually read. As you’re probably guessing, there is an internal FIFO queue to implement this.

You can use this, but do you have to? Our experience says no – even if you want to read data into alternate buffers, so you can process one buffer while reading the next, you can still follow the pattern of only having one outstanding call to `ReadL()` but supplying a different descriptor each time.

Having said all this, there is a requirement for this code to run in a ‘timely manner’ – if you delay calling `ReadL()` for too long, then the underlying system runs out of space to write its data. On some systems, this results in a `KErrOverflow` error on the `MaiscRecordComplete()` callback:

```

void CIOStreamAudio::MaiscRecordComplete(TInt aError)
{
    ASSERT(iState == EStateReading && aError != KErrNone);
    Complete(aError);
}

```

This has an implicit `Stop()` action; that is, if an error occurs, it is as if `Stop()` has been called on the audio input stream and you don’t have to call `Stop()` yourself. As shown here, all you can do is to notify the client application. As well as overflow, you will see other errors such as internal errors and also ‘throw-offs’ where the underlying layer decides to stop you to allow something else to run (see Section 5.8).

5.2.3 Stream Calls for Playing

In this example, the trigger to stop recording and to play the descriptor is the `StopAndPlay()` call from the parent application. This is fairly simple:

```
void CIOStreamAudio::StopAndPlay()
{
    if (iState == EStateReading)
    {
        // no effect unless we are actually recording
        iInputStream->Stop();
        iObserver->RecordingStopped();
        iOutputStream->Open(NULL); // needs to agree with what we gave
                                   // to the input stream
        iState = EStateOpeningOutput;
    }
}
```

It stops the recording, notifies the client – assumed to be useful here but that depends on your client – and then proceeds to play.

There is a potential defect here when recording: calling `CMdaAudioInputStream::Stop()` is simple to use but any outstanding data that has been captured from the microphone and not passed to the client is lost. This is usually not important – it will be a fraction of a second. If it is important, then versions of `CMdaAudioInputStream` from Symbian OS v9.3 have an additional call, `RequestStop()`, which immediately stops recording at the microphone but still passes data through. If you use this call, your application should keep going until `MaiscRecordComplete()` is received, at which point the audio input stream object is stopped. This is left as an exercise for you to do yourself...

Playing is often simpler than recording for a key reason: if you give `CMdaAudioOutputStream` a large buffer, it plays until completion, rather than you observing it as short buffer copies. However, opening and closing the audio output object is virtually the same as for the audio input object. The `CMdaAudioOutputStream::Open()` call is made, using default parameters, and there is a subsequent `MaoscOpenComplete()` callback generated.

```
void CIOStreamAudio::MaoscOpenComplete(TInt aError)
{
    ASSERT(iState == EStateOpeningOutput);
    TInt error = aError;
```



```

if (error == KErrNone)
{
    iState = EStateWriting;
    iOutputStream->SetVolume((iOutputStream->MaxVolume() + 1) / 2);
    TRAP(error, iOutputStream->WriteL(iMainBuffer));
}
if (error != KErrNone)
{
    Complete(aError);
}
}

```

This version sets the volume and plays the data. ($\text{MaxVolume}() + 1) / 2$ is a good default volume but, arguably, this should have been supplied by the application.) Note there is no ‘global volume’ concept – the volume for each instance is set independently and then the adaptation works things out.

Playing the data is very simple: `CMdaAudioOutputStream::WriteL()` is passed the data you have recorded in `iMainBuffer`. This asynchronously plays the data to completion, so that you get a single callback at the end, `MaoscBufferCopied()`:

```

void CIOStreamAudio::MaoscBufferCopied(TInt aError,
                                       const TDesC8& IFDEBUG(aBuffer))
{
    ASSERT(iState == EStateWriting);
    if (aError != KErrNone)
    {
        // ignore any KErrAbort returns - this would happen during a Stop()
        // call if we were playing
        if (aError != KErrAbort)
        {
            Stop();
            Complete(aError);
        }
    }
    else
    {
        ASSERT(aBuffer.Length() == iMainBuffer.Length());
        // output almost complete - have been asked for more data
        iState = EStateWritingPostBuffer;
    }
}

```

The normal behavior is for this to be called with `KErrNone` and the buffer name – in a similar way to `CMdaAudioInputStream`. `KErrAbort` occurs during `Stop()` scenarios and usually needs to be deliberately ignored. Otherwise, this callback implementation does not actually do anything in this instance.

Following this call, because there is no other data, you should get a `MaoscPlayComplete()` call with `KErrUnderflow` – for historical

reasons, successful plays are usually indicated with the `KErrUnderflow` result:

```
void CIOStreamAudio::MaoscPlayComplete(TInt aError)
{
    ASSERT(iState == EStateWriting || iState == EStateWritingPostBuffer);
    TInt error = aError;
    if (aError == KErrUnderflow && iState == EStateWritingPostBuffer)
    {
        error = KErrNone; // normal termination is underflow following a
                        // buffer request
    }
    Complete(error);
}
```

Changing `KErrUnderflow` to `KErrNone`, as shown here, is usually the best thing to do – so it looks like your engine completes with `KErrNone`. Note that other errors are possible and again indicate run-time errors or throw-off. You should pass them back to the client for handling (see Section 5.8 for a discussion of how the client should handle such errors).

5.2.4 Variations

Based on these examples, there are quite a few variations you can implement:

- Run the audio input and output streams concurrently. This is fairly easy to see how to do: just use different buffers.
- Process the input, instead of just saving it – run the input through some function. As mentioned, you can have a couple of input buffers that are used alternately and can process the data of the read buffer after you've called `ReadL()` on the other function – do it that way around to read the new data concurrently with processing the previous data.
- Use formats other than PCM16. Both `CMdaAudioInputStream` and `CMdaAudioOutputStream` have a call, `SetDataTypeL()`, which is called prior to `Open()`. These take a parameter, `TFourCc`, which is nominally a 32-bit value but which is constructed from four characters and is (semi-)readable. The standard ones are defined in `mmffourcc.h`.

For example, PCM16 is represented by the named literal `KMMFFourCCCodePCM16`, which has the associated value " P16", while eight-bit unsigned PCM is represented by the literal `KMMFFourCCCodePCMU8` with value " PU8". You are encouraged to use the named literals but, as usual, it is the value that matters. If you do override it, you should normally use the same the format for both input and output, unless you want some strange effects!

- Use non-default sample rate and mono–stereo settings. They are set using calls to `SetAudioPropertiesL()`, which is called in the callback after the `Open()` call – either `MaoscOpenComplete()` or `MaoscOpenComplete()`.⁴ In practice, only some sample rates are supported. We suggest you use the standard set of 8000, 16000, 11025 or 22050. Others are supported, including CD quality 44100, but they start to have more ‘real-time’ issues. Not all sample rates are supported by all format types.
- Use continuously. Rather than recording into a fixed-length buffer, extend the buffer as you progress – possibly using `RBuf8` and associated `ReAlloc()` and `Append()` calls.

There are many other calls, including those to set the bit rate and give ‘audio priorities’, which you do not need to use most of the time and possibly never.

5.3 Audio Player Utility

If `CMdaAudioOutputStream` is about playing blocks of encoded audio data, `CMdaAudioPlayerUtility` is primarily about playing files. In some ways, that is misleading: it can play descriptors and content given as a URL.⁵ Nevertheless, its primary use case is the playing of files and it essentially treats these other sources as if they were files.

The difference from streams is fairly straightforward – audio files are usually formatted in a structure that contains ‘headers’ and other data structures, in addition to the audio data itself. With `CMdaAudioInputStream`, parameters such as the format to use and the sample rate must be supplied by additional calls, but this information is usually obtained from an audio file and thus `CMdaAudioPlayerUtility` is able to play virtually all files without additional information – the exceptions are very special cases and can usually be ignored.

There are actually more subtle differences: when playing from a file, there is the concept of a ‘position’ at which it is currently playing. The player utility supports calls to find out the current position and the overall length and to set the current position. These enable the implementation of a progress dialog or a conceptual drag and go to a position interface. The API also supports other related concepts such as repeat play and ‘play window’.

In its basic form, `CMdaAudioPlayerUtility` is easier to use than `CMdaAudioOutputStream` – if you want to play a file, then it is quite

⁴In case you are wondering, the `Open()` operation selects the lower-level adaptation objects and `SetAudioPropertiesL()` is viewed as passing parameters to them.

⁵Assuming they are supported by the controller plug-in being used. Many audio controller plug-ins don’t support playback from a URL.

simple to use. Looking at the source or the documentation, you might not see it that way but a lot of the API calls are redundant – Symbian tends to keep on supporting old calls where possible, so as APIs get older there are often calls supported for existing software that you’d not use for new code. `CMdaAudioPlayerUtility` is an old API – it has been supported on almost all versions of Symbian OS – and now contains many such calls. If you strip it down, it is a fairly simple API.

5.3.1 Playing a Named File

This example plays a file given a filename – the simplest thing to do, although not always the most flexible approach (as we’ll see in Section 5.3.2). You can also reference a file via an `RFile` object or a `TMMSource`. The basic construction and destruction code follows the same pattern as that in the audio stream calls in Section 5.2. The primary difference is the call in `ConstructL()`:

```
void CPlayAudio::ConstructL()
{
    iUtility = CMdaAudioPlayerUtility::NewL(*this);
}
```

The basic operation is quite simple – the utility is opened, you wait for a callback to confirm opening, request play and are then told that play is complete. Going through that in more detail, the first command is the `PlayL()` call provided in this example for use by the client application:

```
void CPlayAudio::PlayL(const TDesC& aFileName)
{
    iUtility->OpenFileL(aFileName);
}
```

This opens the filename passed and (unless it leaves) leads to a `MapcInitComplete()` callback:

```
void CPlayAudio::MapcInitComplete(TInt aError,
                                   const TTimeIntervalMicroSeconds& /*aDuration*/)
{
    {
        if (aError != KErrNone)
        {
            iObserver->PlayComplete(aError);
        }
        else
        {
            iUtility->Play();
        }
    }
}
```

aDuration is the duration of the clip; it is often ignored, as here.

The observer mentioned here is a local observer interface back to the client application, but the basic point holds true – if an error is returned the cycle is stopped and you just notify the client or user appropriately. If the file is successfully opened, just call `Play()` and the clip is played, eventually leading to a `MapcPlayComplete()` callback:

```
void CPlayAudio::MapcPlayComplete(TInt aError)
{
    // play has completed
    iObserver->PlayComplete(aError);
}
```

You can normally just pass this on. Success is indicated via `KErrUnderflow`, which perhaps you'd want to change to `KErrNone`.

5.3.2 Variations

Based on these examples, there are quite a few variations you can implement:

- Display the duration of the clip and track progress. The best way to get the duration is by the following call:

```
TMMFDurationInfo Duration(TTimeIntervalMicroSeconds& aDuration);
```

This is better than the alternatives as it also returns a `TMMFDurationInfo` value that can indicate if the duration is 'valid' (so the returned `aDuration`⁶ is important), is unknown for some reason or is 'infinite' – the latter makes more sense for URL clips and relates to real-time streamed audio, or similar, where there is no real concept of duration.

Position is returned by:

```
TInt GetPosition(TTimeIntervalMicroSeconds& aPosition);
```

This returns the current playing position, at least to a good approximation. Call on a timer, say every 100 ms, to show progress – every 50 ms, if you want really smooth progression.

- Stop before getting to the end. Use `CMdaAudioPlayerUtility::Stop()` to stop playing and no further callbacks are generated until you request something else.

⁶If your file has a variable bit rate then the returned duration is calculated and may not be 100% accurate.

- Support end users in pause–play–pause scenarios. For pause, use `CMdaAudioPlayerUtility::Pause()` and, to resume, `CMdaAudioPlayerUtility::Play()`. We tend to track the state and not call `Pause()` when it is already paused or `Play()` when playing – it is not hard to do, just maintain a flag.
- Set the play position. Call `CMdaAudioPlayerUtility::SetPosition()`⁷ with the appropriate position given in microseconds. Basically, this is the same as seeking a file – new data is read from the stated position. This is a ‘goto’ operation and is commonly used for the equivalents of dragging the position marker on a control.

Note that `SetPosition()` should jump to the new position immediately, but if you listen very carefully you might hear a small delay in switching to the new position – it depends on the internal implementation. Also note that although positions are expressed in microseconds, the actual positioning will never be that accurate. How accurate it is depends on the encoding method – many compression schemes break the sound stream into frames of a specific length, each of which represents many samples of data, and you can often only ‘seek’ to a frame boundary.

- Play a specific window. The scenario is quite simple – your application wants to start playing a bit after the start and stop before the end is reached. For example, in a 10-second clip, you might play seconds 3 to 6. This typically relates to the user moving some form of slider to show where play should start and another showing where it finishes. You can do this yourself, using `SetPosition()` and polling `Position()`, but there is explicit support using:

```
TInt SetPlayWindow(const TTimeIntervalMicroSeconds& aStart,
                  const TTimeIntervalMicroSeconds& aEnd)
```

This basically works as expected – if the position is less than `aStart`, it is set to `aStart`; if it is greater than `aEnd`, it is set to `aEnd`. Otherwise it does not change position. The clip stops when you reach the end of the window. Be careful of the fact that the position is not always moved and then not always to the beginning. You may set the play window and then wonder why nothing happens when you play the clip; you may actually be at the end. Also note that `SetPosition()` calls are always relative to the start of the clip and not to the play window.

⁷For safety it is better to call `Pause()`, `SetPosition()` and then `Play()` as some controllers do not correctly support a `SetPosition()` call during playback.

You can change the play window or clear it using:

```
TInt CMdaAudioPlayerUtility::ClearPlayWindow();
```

- Repeat playing the clip. You can do this yourself but not as cleanly or efficiently as MMF.

```
void CMdaAudioPlayerUtility::SetRepeats(TInt aRepeatNumberOfTimes,
    const TTimeIntervalMicroSeconds& aTrailingSilence)
```

The parameter `aRepeatNumberOfTimes` requests a number of repeats – so 0 means play once, the default behavior. A special value, `KMdaRepeatForever`, can be used to request indefinite repeats – you have to use `Stop()` to end it. Otherwise use an explicit value. The gap between two iterations can be set with `aTrailingSilence` – as with most timings on Symbian OS, it is approximate, so don't over-rely on its accuracy. This is good for background music, but take care – some clips are more appropriate than others.

- Play an `RFile` object. It is common in Symbian OS now to be given an open `RFile` object and to have to use that. There is an alternative `CMdaAudioPlayerUtility` overload for this:

```
void OpenFileL(const RFile& aFile);
```

This takes `RFile` instead of the filename, but otherwise the sequence is the same.

It is also possible to use `TMMSource`,⁸ as in:

```
TMMFileSource filesource(aFileName, KDefaultContentObject, EPlay);
iUtility->OpenFileL(filesource);
```

This uses a slightly different overload. This code is intended for use with the Symbian OS DRM subsystem – if you have sufficient rights it decrypts the file and plays it. This is possibly outside your use cases, but there are specialist uses for the subsystem other than standard DRM ones.

⁸See Section 2.3 for more on `TMMSource` and DRM support.

- Set the balance. For stereo output (such as through headphones), you can set the balance anywhere between `KMMFBalanceMaxLeft` and `KMMFBalanceMaxRight` using `CMdaAudioPlayerUtility::SetBalanceL()`. Most applications will want to stick with the default value `KMMFBalanceCenter`. This setting is also available for all of the other client and stream utilities that produce audio output.
- Set the volume. You can set the volume anywhere between 0 and the value returned by `CMdaAudioPlayerUtility::MaxVolume()` using `CMdaAudioPlayerUtility::SetVolume()`. The actual maximum volume value varies between platforms and devices. Again, this functionality is available for all of the audio output APIs. See Section 8.13 for more on volume control.

5.4 Audio Recorder Utility

Since `CMdaAudioPlayerUtility` is used for playing, you'd assume that `CMdaAudioRecorderUtility` was used for recording, and in general terms you'd be right. However, things are a bit more complex than that. For historical reasons, the utility was deliberately designed to cover the 'voice recorder' use case – where an end user records something, possibly listens to it, records some more, etc. Hence, despite its name, this API both records and plays.

An additional complication is that `CMdaAudioRecorderUtility` forms a slightly different API pattern to `CMdaAudioPlayerUtility` – rather than specific callbacks to state that something has finished, a single callback function is called multiple times showing progress. Which you prefer is probably a matter of choice. `CMdaAudioRecorderUtility` potentially gives more progress info, but the callbacks have to be implemented with care.

The fact that `CMdaAudioRecorderUtility` also plays means that some programmers have used it almost exclusively – preferring it to `CMdaAudioPlayerUtility`. We'd advise against that – `CMdaAudioPlayerUtility` has more features and is generally more efficient.

`CMdaAudioRecorderUtility` requires an observer to be supplied on creation of type `MMdaObjectStateChangeObserver`. This is defined as follows:

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


The `MoscoStateChangeEvent()` call is key to the client – it effectively forms a central state machine. You are given both previous and current (latest) states – at least the states of the underlying recorder utility. You don't normally have to maintain your own states in the client, which simplifies things. The states returned are defined in `CMdaAudioClipUtility::TState`:

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

Remember that you normally have to prefix these with `CMdaAudioClipUtility::` in the source. You must say `CMdaAudioClipUtility::ENotReady` or `CMdaAudioRecorderUtility::ENotReady`, depending on the level you are dealing with.

5.4.1 Recording to a Named File

We are going to look at code that does something similar to the audio input-output stream example; it records something and then plays it back. For this API, we record into a file and the file remains afterwards. Skipping the declaration, the construction is fairly predictable. The key `ConstructL()` is:

```
void CRecordAudio::ConstructL()
{
    iUtility = CMdaAudioRecorderUtility::NewL(*this);
}
```

To start, the client calls `RecordL()`, which is implemented as:

```
void CRecordAudio::RecordL(const TDesC& aFileName)
{
    iUtility->OpenFileL(aFileName);
}
```

This requests the opening of a file. We'll return later to some of the subtleties of this call. The key is that it is asynchronous and leads to a subsequent `MoscoStateChangeEvent` callback. The implementation of this callback becomes the heart of the client code:

```

void CRecordAudio::MoscoStateChangeEvent(CBase* aObject,
                                         TInt aPreviousState, TInt aCurrentState, TInt aErrorCode)
{
    aObject = aObject; // to stop warning
    TInt error = aErrorCode;
    if (error == KErrNone)
    {
        // only interested in some state transitions ...
        if (aPreviousState == CMdaAudioRecorderUtility::ENotReady &&
            aCurrentState == CMdaAudioRecorderUtility::EOpen)
        {
            // have opened the file
            TRAP(error, OnOpenL());
        }
        else if (aPreviousState == CMdaAudioRecorderUtility::EPlaying &&
                 aCurrentState == CMdaAudioRecorderUtility::EOpen)
        {
            // normal termination of play
            iObserver->PlayOrRecordComplete(KErrNone);
        }
    }

    if (error != KErrNone)
    {
        // stop now
        iUtility->Stop();
        iObserver->PlayOrRecordComplete(aErrorCode);
    }
}

```

This handles two sets of scenarios: what to do on errors, including ones it generates and some key use cases. When an error occurs, we merely call `Stop()` on the utility and respond to the conceptual main application with a callback. The `Stop()` is usually not required – if the client told you it had an error, it should have stopped anyway, but it's safer always to call it to cover other failure cases.

Returning to positive situations, if you traced through the code with a debugger or added logging, you'd find that this call is made several times. It turns out you generally only need to trace some of the transitions – in this case we need to know when we've opened the file and when play completes. Other `KErrNone` transitions can be ignored. For example, we need do nothing on the transition from `EOpen` to `ERecording`, so we just ignore it. You may want to add code for other scenarios – or, perhaps, remove it.

The transition between `ENotReady` and `EOpen` follow on from an `OpenFileL()` call on the utility. The structure here is to call the `OnOpenL()` method:

```

void CRecordAudio::OnOpenL()
{
    // let's record to ADPCM
}

```

```
iUtility->SetDestinationDataTypeL(KMMFFourCCCodeIMAD);  
iUtility->RecordL();  
}
```

This is quite simple, it sets the data type and then starts recording – as you might guess `RecordL()` is the request to record itself. If you wanted to set other parameters, such as sample rate to use, then this is done before `RecordL()`.

This particular setting requests (IMA) ADPCM encoding. If you are very observant, you may be wondering how we control the file type – this API is typically used to write to file types such as WAV and we’ve not even mentioned that! We are exploiting a feature of the API, where the file type to use is taken from the suffix of the filename given to the `OpenFileL()` statement. It turns out that this is the easiest way of arranging for a particular file type and is usually the most convenient in a GUI environment. The statement here basically assumes it has a WAV file – other formats that also support this ADPCM setting would also work. More likely, the formats supported are highly specific to the underlying implementation and you need to be careful about what values to use. From a programming view point you have a number of choices here:

- Do not call `SetDestinationDataTypeL()` – every file type has a default.
- Call `SetDestinationDataTypeL()` explicitly – as here, but it could come through from the main GUI rather than being hardwired.
- Discover from the `CMdaAudioRecorderUtility` object, once it has been opened, a list of the formats supported, choose one and set that. Discovery is via the call `GetSupportedDestinationDataTypesL()`.

It is tempting to write your code to negotiate – checking to see if something is supported before setting it. In practice, most applications don’t need to do this – if you know what you want, set it and handle the leave appropriately. This is almost certainly quicker and easier. The exceptions are usually test code or middleware.

Variations on this scheme include something like:

```
TRAP_IGNORE(iUtility->SetDestinationDataTypeL(KMMFFourCCCodeIMAD));
```

This says ‘try to use IMA ADPCM, but if you can’t just use the default’. That is a fairly sensible approach – if you get your guess wrong, treat it as non-fatal.

A slightly more complex version might look at the filename suffix and use a version based on that. In reality, that is beyond the needs of many applications.

Returning to the example program, at some point the main program decides to stop recording and plays back the sound, using `StopAndPlayL()`:

```
void CRecordAudio::StopAndPlayL()
{
    iUtility->Stop();
    iObserver->RecordingStopped(); // just let the calling code know
    iUtility->PlayL();
}
```

This plays the file from the beginning and at the end you get a `MoscoStateChangeEvent()` callback to show the switch from `EPlaying` to `EOpen`. If you look back to the function above, you'll remember this is handled and just tells the client. The `Recording-Stopped()` and `PlayOrRecordComplete()` callbacks mentioned are arbitrary – they make sense in this particular test – and you are free to do what makes sense for your application.

One issue we've skipped over a bit is that of `Position()`. We're able to play the content just by calling `PlayL()` because the `Stop()` call does an implied rewind to the beginning. This is useful in some scenarios, such as this simple one, but it is not always quite what you want. There is no pause functionality in `CMdaAudioRecordUtility`, so if you need to support pause, use `Stop()` or an alternative technique – such as using `CMdaAudioPlayUtility` for the play functions. Forgetting the latter, at least for the moment, you'll find that you can get a reasonable approximation to play-pause-play but you have to handle position yourself. Thus a 'pause' sequence looks something like:

```
iCachedPosition = iUtility->Position();
iUtility->Stop();
```

A 'resume' sequence looks something like:

```
iUtility->SetPosition(iCachedPosition);
iUtility->PlayL();
```

5.4.2 Variations

Based on this example, there are variations you can implement:

- Record some more audio to an existing file. You can use this in two ways – record soon after the original or open an already open file and


```
iUtility->OpenL(aFromFileName, aToFileName);
}
```

Again, the suffix for the filename is the main way of selecting the format – if you state that the output is `c:\temp.wav`, then the file will be a WAV file.

Because it shares many API features with `CMdaAudioRecorderUtility`, the client code that drives this class is again based on `MoscoStateChangeEvent` callbacks, with very similar parameters. This example is typical:

```
void CConvertAudio::MoscoStateChangeEvent(CBase* aObject,
                                         TInt aPreviousState,
                                         TInt aCurrentState,
                                         TInt aErrorCode)
{
    __ASSERT_DEBUG(aObject == iUtility, User::Invariant());
    aObject = aObject; // to stop warning
    TInt error = aErrorCode;
    if (error == KErrNone)
    {
        if (aPreviousState == CMdaAudioConvertUtility::ENotReady &&
            aCurrentState == CMdaAudioConvertUtility::EOpen)
        {
            // have opened the file
            TRAP(error, OnOpenL());
        }
        else if (aPreviousState == CMdaAudioConvertUtility::EPlaying &&
            aCurrentState == CMdaAudioConvertUtility::EOpen)
        {
            // normal termination
            iObserver->ConvertComplete(KErrNone);
        }
    }
    if (error != KErrNone)
    {
        // stop now
        iObserver->ConvertComplete(aErrorCode);
    }
}
```

The structure is very similar to that of the recorder. We'll assume you have read Section 5.4 and look at the bit that differs – `OnOpenL()`. Again this is virtually the same as for recording, for example:

```
void CConvertAudio::OnOpenL()
{
    // let's convert to 16-bit PCM
    iUtility->SetDestinationDataTypeL(KMMFFourCCCodePCM16);
    iUtility->ConvertL();
}
```

In this example, we explicitly try to request 16-bit PCM. If this is not supported, we treat it as a fatal error. Again, as with recording, we could decide to ignore the error, say, and just use the default. However, the requirements for conversion are often different – this is often used when sending a file to another end user to ensure that a known, ‘common’ file format is used. In that case, we usually want to be fussy about the format and treat problems as an error. As with recording, if you want to override the sample rate, the mono–stereo setting, etc. from the default, do it before you call `ConvertL()`. The calls are very similar.

You can use conversion sequences to append to an existing file as well as copying. If you want to ensure you are making a copy, delete the file first:

```
iFs.Delete(iToFileName); // Ignore errors
```

If you want to use the append functionality, be careful about setting data types, sample rates, etc. As with recording, it is best not to do this when you are appending, as the values are read from the existing file.

You can use a play window to convert only a portion of the original. If you only want part of the file, a play window is preferred to using `CropL()` or `CropFromTheBeginningL()` to remove part of the output – generating a file and then removing some of it is less efficient.

Conversion potentially uses different sets of plug-ins to playing or recording – don’t assume that, just because you can play format A and record format B, you can convert from format A to format B.

5.6 Tone Utility

This section is about playing tones – simple tones, DTMF and tone sequences. This functionality is provided by `CMdaAudioToneUtility`.⁹ Although the same class is used, the calling sequences differ and can be dealt with separately.

`CMdaAudioToneUtility` shares many characteristics with `CMdaAudioOutputStream` – it acts as a relatively thin layer over `DevSound` and neither runs in a separate thread nor uses a controller object. Nevertheless there are differences: rather than supplying data in buffers, virtually everything is supplied as parameters to the associated play calls. These are, in turn, passed to the underlying `DevSound` layer, so that the traffic between the client application and the underlying audio-adaptation layer is much reduced – compared to the audio output stream scenarios, at least. One of the reasons for this is that you can supply the parameters fairly succinctly: for example, 100 Hz tone for two seconds can be stated

⁹To use `CMdaAudioToneUtility`, include `MdaAudioTonePlayer.h` and link against `mediaclientaudio.lib`.

in several words – similarly DTMF tone requests can be given as a string of digits. Even tone sequences are typically more compact than normal audio formats, and internally whole descriptors’ worth of data are passed down to the adaptation, instead of on a buffer-by-buffer case.

5.6.1 Playing Dual-Tone Multiple Frequency Signals

Dual-Tone Multiple Frequency (DTMF) is a standard set of tone signals traditionally used for ‘digital dialing’ on analog phone systems. You can obviously use it for other purposes, if you wish. Given its original usage, the DTMF function can handle the keys of a normal phone keypad (0–9, * and #). Additionally ‘A–F’ are defined: E and F are synonymous with * and #, respectively, while A–D have special uses and are not commonly used. If you really want to know about the interpretation of these, try typing ‘DTMF’ into a search engine – it will give you all you want to know and probably a lot more!

Early use cases for this code included playing phone numbers as DTMF down a phone line; for that reason, a few additional characters are supported: spaces are ignored and ‘,’ is treated as a pause. However, there is no support for the additional characters you often get in phone numbers, such as ‘() /-’. This use case is largely deprecated, but if you ever do want to render phone numbers as DTMF, do take care – you may have to strip some extra characters.

Playing DTMF strings using `CMdaAudioToneUtility` is very simple. You set some properties, such as volume and tone lengths to use, and tell it to play the strings. As with other multimedia APIs, there is slight complexity to do with the asynchronous nature of the calls, but this is not particularly major. Perhaps it is best just to look at an example. This is very similar to the other examples in this chapter and we won’t look at all the code. The `ConstructL()` is shown as:

```
void CDtmfSeq::ConstructL()
{
    iToneUtility = CMdaAudioToneUtility::NewL(*this);
}
```

There are some additional optional parameters to the `NewL()` call, but they are not usually required.

Starting play of DTMF looks like:

```
_LIT(KExampleDtmf, &123456789");
void CDtmfSeq::Start()
{
    ASSERT(iState == EStateIdle); // calling code must wait for callback
    iToneUtility->PrepareToPlayDTMFString(KExampleDtmf);
}
```



```
iState = EStatePreparing;
}
```

The sequence is a bit different to the other APIs, but in general there is a `Prepare...()` call and in return you get a callback to `MatoPrepareComplete()` to state that this 'preparation' phase is complete. This specific example handles the various callbacks as a central `Fsm()` call:

```
void CDtmfSeq::MatoPrepareComplete(TInt aError)
{
    Fsm(EEventPrepared, aError);
}

void CDtmfSeq::Fsm(TEvent aEvent, TInt aError)
{
    TInt error = aError;
    if (error == KErrNone)
    {
        switch (iState)
        {
            case EStatePreparing:
            {
                ASSERT(aEvent == EEventPrepared); // nothing else expected
                // set volume to 3/4 of the maximum volume
                iToneUtility->SetVolume(3*iToneUtility->MaxVolume()/4);
                iToneUtility->Play();
                iState = EStatePlaying;
            }
            break;
            case EStatePlaying:
            {
                ASSERT(aEvent==EEventPlayed); // nothing else expected
                // normal completion
                iState = EStateIdle;
                iObserver->SequenceComplete(KErrNone);
            }
            break;
        }
    }
    if (error != KErrNone)
    {
        Cancel();
        iState = EStateIdle;
        iObserver->SequenceComplete(error);
    }
}
```

On preparing, this sets the volume to three-quarters of `MaxVolume()` and then calls `Play()`. On any errors, the system 'cancels' before returning an error callback. This cancel operation is quite important:

```

void CDtmfSeq::Cancel()
{
    switch (iState)
    {
        case EStateIdle:
            // do nothing
            break;
        case EStatePreparing:
            {
                iToneUtility->CancelPrepare();
            }
            break;
        case EStatePlaying:
            {
                iToneUtility->CancelPlay();
            }
            break;
    }
}

```

There is no single ‘Stop’ call and it’s probably better to be careful about what you cancel.

On termination of play, you get a `MatoPlayComplete()` callback, which in this example is fed back the same way.

```

void CDtmfSeq::MatoPlayComplete(TInt aError)
{
    Fsm(EEventPlayed, aError);
}

```

For security reasons, it is worth being careful what you play. If it can be heard by anyone other than the end user, it is possible for someone to hear and transcribe the number or to record it automatically. If you are just using it for an effect, then perhaps ensuring the DTMF string is not the real string might be a good idea – in the same way, ATMs now tend to play the same tone for each key.

Note: DTMF data is given as 16-bit descriptors, reflecting that it probably originates as a phone number and is human-readable rather than binary.

5.6.2 Playing Tones

‘Tones’ in this context are pure sine waves – well, as pure as can be achieved. Dual tones are two sine waves mixed together. The calls are pretty simple to use – apart from ensuring the volume, etc. is correct, you just give the frequencies to generate and the time to play, then the system does the rest. The use is very similar to the DTMF example, apart from how you specify what to play. Key to the implementation are two calls:

```
void PrepareToPlayTone(TInt aFrequency,
                      const TTimeIntervalMicroSeconds& aDuration);
void PrepareToPlayDualTone(TInt aFrequencyOne, TInt aFrequencyTwo,
                          const TTimeIntervalMicroSeconds& aDuration);
```

As before, these don't actually make any sound, but they set up the tone utility so that a subsequent `Play()` call makes the correct sound. You call `Play()` once you receive the `MatoPrepareComplete()` callback.

You might want to abandon the tone play before it has got to the end, for example, because the application cancels an associated screen view. To do this, make a call from your API that implements something like the `Cancel()` call given in Section 5.6.1.

5.6.3 Playing Tone Sequences

Playing a tone sequence is akin to playing an audio file (see Section 5.3) but sequence files are inherently simpler. Not only is no controller required but the sequence is decoded in the adaptation layer. Although the use case is similar to that of playing audio files, the calling pattern is similar to that of playing DTMF or tones.

There are several ways of playing a tone sequence:

- provide the tone sequence in a descriptor
- provide a filename that contains the tone sequence data
- use `RFile::Open()` to open a file that contains the tone sequence
- play a fixed sequence by index.

Internally there is not that much difference between them – the `CMdaAudioToneUtility` implementation reads the file contents into a descriptor and plays that – but that is internal and not really observable to the client. If you have a file, then generally we recommend using the `RFile` approach as it is easier to change your code to support files passed from other applications. However, in most situations the differences are minor.

Code would be similar to the DTMF example, but using different `Prepare...` calls:

```
void PrepareToPlayDesSequence(const TDesC8& aSequence);
void PrepareToPlayFileSequence(const TDesC& aFileName);
void PrepareToPlayFileSequence(RFile& aFile);
```

The two `PrepareToPlayFileSequence()` calls play a file when given either a filename or an `RFile` object. `PrepareToPlayDes-`

`Sequence()` effectively expects the contents of a file. Playing a fixed sequence is a bit different. They are built into the audio adaptation and are accessed by index. The following call plays a sequence:

```
void PrepareToPlayFixedSequence(TInt aSequenceNumber);
```

The following method tells you how many fixed sequences are provided on a given device:

```
TInt FixedSequenceCount();
```

The following method tells you the name of the associated sequence:

```
const TDesC& FixedSequenceName(TInt aSequenceNumber);
```

A key scenario is to read the names into a `CDesCArray` object, using a `for` loop, which can then be used as part of a selection box.

Apart from that, this API is similar in form to playing from a file. However, it is not always implemented. If it is not implemented, `FixedSequenceCount()` returns 0. If you do use these calls, take care to ensure that the behavior is valid if `FixedSequenceCount()` returns 0 – be careful of using choice lists with no content. Remember that the following call does not reliably play a sound:

```
PrepareToPlayFixedSequence(0);
```

If you do just want to make a sound, then playing simple tones is probably more reliable.

5.7 DevSound

`DevSound` (properly `CMMFDevSound`) is the fundamental component at the bottom of the MMF API stack. It encapsulates the audio adaptations for devices. Under `DevSound` are found the lower-level audio codecs, the audio policy (what happens if more than one client requests to play or record concurrently) and the links to the various audio devices. These features are common to all the `DevSound` implementations on different Symbian smartphones, but the underlying architecture varies from one manufacturer to another, and even from one model to another.

Details of how to use `CMMFDevSound` are not described further here – the headers, libraries and documentation for the class is not

included in every SDK, and in many situations the API differs from that generally convenient for application programmers. Nevertheless it is worth knowing that DevSound is ‘there’ – even just to understand how the various APIs link together.

In general, `CMMFDevSound` comes into its own if you are writing middleware or adding new MMF plug-ins. That is somewhat beyond the scope of this book – you should consult the relevant documentation in the Symbian Developer Library and will probably need to seek further advice anyway.

5.8 Audio Policies

It is sometimes tempting to forget this fact when writing an application, but Symbian OS is a multi-tasking system and there are always some other programs running. At various times, your programs may be running in the foreground or in the background – let alone the fact that some services are hidden but running continuously. Having said that, resources on a smartphone are always limited – particularly if special hardware is used to support audio codecs. Additionally, however, there are requirements that some sounds should be played whatever else is running, for the device to be acceptable to consumers – for example, the telephony application should always be able to generate a ringtone.

The approach taken by Symbian OS is to allow many programs to play audio concurrently, without particular consideration as to whether they are in the foreground or the background – so switching to the background does not automatically stop your program’s sound being generated. At the same time, decisions must be made as to what happens when several clients simultaneously request to play or record. There are several possible options that the device can follow. The basic decision is whether to mix several clients together or to decide to play one of them.

Another perspective is to consider what happens when a ‘new’ client wishes to start to play or record:

- The new client can be prevented from starting.
- Existing clients can be stopped and the new client started.
- The new client can be run at the same time as existing clients.

If the request concerns playing and the end user is listening to the sound, the end user can usually distinguish this behavior. However, from a client perspective, you don’t see this – the client is told about when a sound is stopped and, by inference, when it plays.

So what actually happens? On one level, the options relate to the software and hardware components available. At the time of writing,

the standard Symbian OS emulator on Windows (shipped with many SDK environments) is not capable of mixing and thus can't support simultaneous play. However, a key feature of Symbian OS is that, apart from requesting play or record in the first place, neither users nor applications have direct control of what actually happens in these circumstances. The real decision is up to the *audio policy* of the device – a set of rules as to what should happen in these scenarios. This policy is encoded into a central unit – typically called the *audio policy server* or the *audio resource manager*. Applications request to play or record, the decision as to whether they can is made centrally, and that decision is enforced by the audio adaptation.

This shows itself to your client by different error values. For example, a pre-empted player utility gets a `MapcPlayComplete()` callback with `KErrInUse`, `KErrDied` or `KErrAccessDenied` – the specific error value depends on the adaptation and your code should treat all the same. As with other `MapcPlayComplete()` callbacks, the player utility is left in a stopped state.

Note that one effect of the API is that there is virtually no difference in calling pattern between two seemingly key scenarios: a new client requests to play but this is immediately rejected; and a new application requests to play, starts to play and then is thrown-off (soon) afterwards. Part of the reason for this concerns the asynchronous way that almost all audio APIs request play operations. When you fire off a play or record request, the adaptation returns quite quickly and plays in the background – the asynchronous call pattern. The `Play()` or `PlayL()`, function call does not wait to find out what happens – so avoiding the latencies that sometimes occur and keeping the calling client 'lively'.

The disadvantage to this approach is that if, say, you are showing a progress dialog, it might look to the end user as though play started when it did not. In reality, we believe that end users are probably not that fussy and that it helps to update playing–stopped buttons, for example, just to show the command has been accepted. If your application really needs to be fussy, then perhaps wait until some progress has been detected – for example, the value returned by `GetPosition()` has changed.

What your application should do on pre-emption depends on the use case. For sound effects and similar sounds, you'd just forget about it and treat it as if you'd reached the end anyway – this probably comes for free, since such client code often ignores failure to play, treating it as a non-fatal error that is not worth telling the client about. At the other end of the scale, on a music player, you will want to change the state of the view to show the player has been stopped and to allow the user to restart from the current position – in terms of code, this comes for free.

To recap, Symbian OS supports concurrent playing and recording for several clients, but what you get depends not only on the hardware's features but also on what the device manufacturer wants to happen – the

‘policy’. This has a number of implications for you as an application writer:

- You must always prepare for audio clients to be pre-empted.
- The number of clients you can simultaneously play is dependent on the particular smartphone model and on what other programs are running.

5.9 Priority Settings

To summarize from the previous section, if you fire off several play requests from your application, it is not really your decision as to what gets played together. This raises additional questions, including how the audio policy knows what to give priority to.

The audio policy is based on a combination of what the policy server can find out about a client process (such as the SecureID of the process and its capabilities) and extra parameters supplied by the client via calls such as:

```
TInt SetPriority(TInt aPriority, TMdaPriorityPreference aPref);
```

This method is provided by `CMdaAudioPlayerUtility`. These parameters are interpreted as follows:

- `aPriority` is an integer officially between -100 and 100; the internal property defaults to 0. There are literal values defined for the priority: `EMdaPriorityMin`, `EMdaPriorityMax` and `EMdaPriorityNormal`. However, any value in the range is usable and, in general, it is easier to define your own constants with associated values.
- `aPref` is slightly more complex. It is best considered as consisting of two fields: the lower 16 bits and the upper 16 bits.
 - The lower 16 bits are flags that state requests about ‘quality’ – whether the stream can be mixed or played at a lower sample rate with another stream. To be honest, these are merely suggestions – adaptations can and do ignore these flags. Unless you’ve been given additional information, it is probably safest to ignore this, supplying 0 as the value for flags.
 - The upper 16 bits are officially reserved by Symbian for use by manufacturers. In practice, this means that on the different UIs, there are special values defined with specific meanings.

Initially, try just leaving the default behavior and not actually setting the values – the equivalent of using:

```
SetPriority(EMdaPriorityNormal, EMdaPriorityPreferenceNone);
```

If this does not work, it is probably because you are dealing with one of the following scenarios:

- Another application is running, which is itself using audio clients, and that is stopping audio for your application.
- You want to run several audio clients together, but you know that not all can run on all devices and you give some priority so the important ones generally run.

At this point, it is probably worth stepping back a bit and thinking about the overall effect of your decision on the device. Sounds can perhaps be divided into three categories:

- ‘Must have’ sounds that must be played, usually on their own, or the device will not be acceptable to the mobile networks. The obvious example is the ringtone for incoming calls.
- ‘General’ sounds played as part of an application’s core functionality – if your application does not play the sound, the end user will see it as not working. Examples include a music player not playing tracks.
- ‘Additional’ sounds, which add to the richness of a particular application but are not essential and can be omitted without most end users really noticing. Obvious examples include key clicks, but additional things such as shutter-sound effects for a camera fall into this category.

Some scenarios are not quite so easily categorized. Is the background music of a game essential? Probably not, but it is worth a thought. Are the explosion effects of a game essential? Possibly, but again they are worth a thought.

So what values do you use? It is up to the built-in software to ensure that ‘must-have’ sounds are heard. For ‘general’ sounds, the default is probably sufficient. For ‘additional’ sounds, a negative number is more applicable. Try that and see how it interacts with other applications.

5.10 Miscellaneous

5.10.1 Platform Security and Capabilities

Symbian OS v9.1 introduced platform security features to stop untrusted or unsigned applications performing key operations. So the question often

asked is ‘will the platform security features stop me writing an application that plays or records audio?’

The simple answer is ‘not really’. Your application doesn’t need any capabilities to play audio. There is a capability termed `MultimediaADD` that is required by some middleware, the adaptation and some key applications, but it is very unlikely that your application will need it.

Recording is slightly different: the `UserEnvironment` capability is needed to record. If you don’t have it, the `RecordL()` or another operation fails. However, the `UserEnvironment` capability is generally set by phone manufacturers to be a user-grantable capability (see Section 2.2 for further information).

5.10.2 MIDI Player Utility

There is a special client API, `CMidiClientUtility`, specifically provided for the playing of MIDI clips from files, etc. This provides much more control over playing MIDI, allowing you to change many, if not most, of the MIDI parameters (choosing instrument banks and the gain of the individual channels).

The penalty of this API is that it is a lot more complex to use than `CMdaAudioPlayerUtility` – for example, there are a lot more callbacks that need addressing and even more of the operations are asynchronous. Most likely, if you really need to use this API you need knowledge of the underlying MIDI engine and how it operates. If you merely want to play a MIDI file, then usually using `CMdaAudioPlayerUtility` is fine – it will play the file with default parameters.

6

Image Conversion Library

6.1 Introduction

The Image Conversion Library (ICL) is very useful for any application that needs to do some image manipulation.¹ Many applications with a user interface fall into this category, for example, if they need to perform some application-specific animation effects or make use of well-known image formats such as JPEG, PNG, GIF, etc.

The main use cases for the ICL are:

- conversion of an image in a well-defined image format into a native Symbian OS bitmap²
- conversion of a native Symbian OS bitmap into a well-defined image format.
- image transformations (rotation, scaling, etc.) applied to native Symbian OS bitmaps or directly to an image in a well-defined format.

All the other use cases are extensions to these three, with one exception – there are some (fairly small) sets of classes which can only handle native bitmaps.

Despite being called a ‘library’, the ICL is more than that (see Figure 6.1). It comprises a diverse set of APIs and is not just a single library.

¹We assume that you have some knowledge of computer graphic principles. We use some computer graphics terms without explanation of their meaning. For example, we do not define words such as pixel, color depth, bitmap, JPEG, PNG, compressed image, palette or animated image. Those topics are well covered by other documentation.

²Native bitmaps are represented by the class `CFbsBitmap` on Symbian OS. See the Symbian Developer Library documentation in your chosen SDK for details.

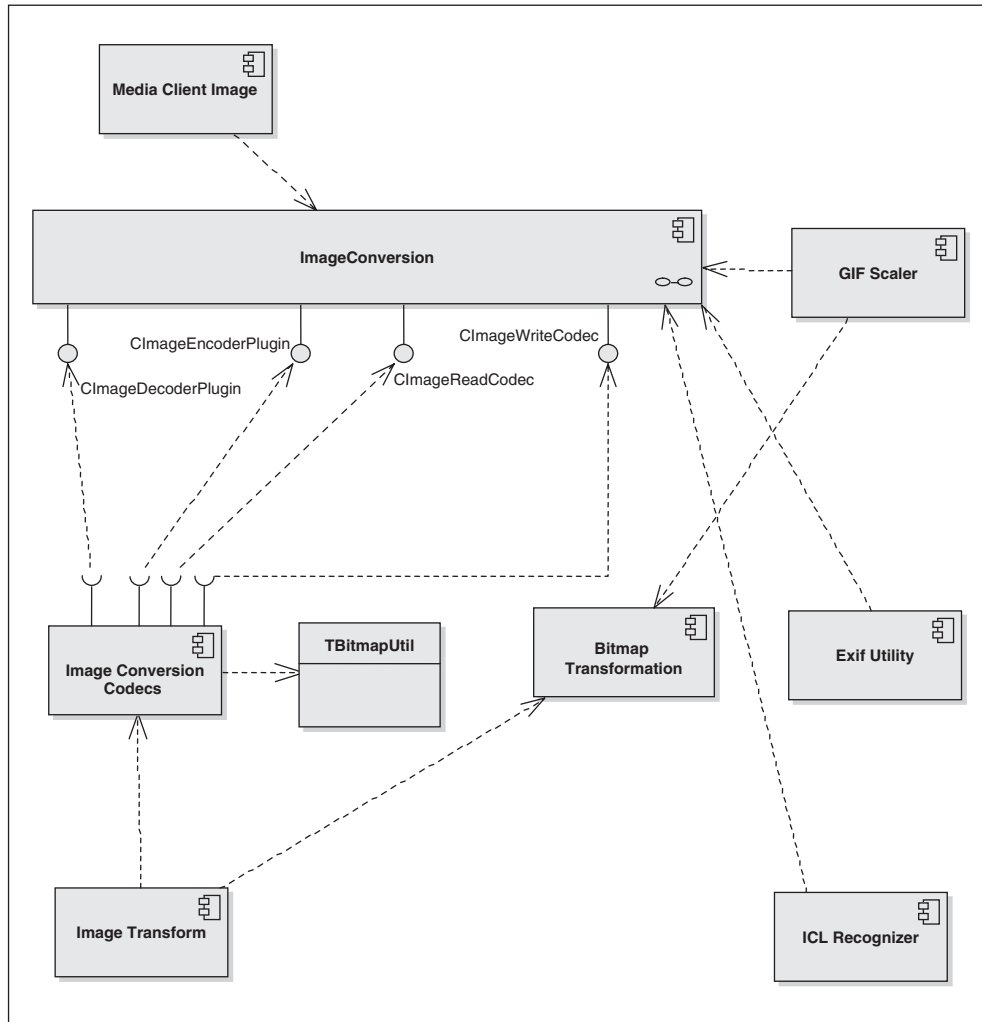


Figure 6.1 Structure of the Symbian ICL subsystem (licensee subsystems may have their own variations)

The biggest part of the subsystem is the Image Conversion component,³ which provides applications with the ability to decode an image from a well-defined format to a native Symbian OS bitmap and vice versa. Most of the other components have dependencies on the Image Conversion component, which means that they extend a possible set of use cases provided by that component.

³In this chapter, to avoid any confusion arising because part of the subsystem has the same name as the whole, we refer to the part of the ICL that actually deals with converting images from one format to another as the 'Image Conversion component'.

Other parts of the subsystem include:

- **Media Client Image:** this component contains old (before Symbian OS v7.0s) APIs which are included for source compatibility but which should not be used by new applications.
- **GIF Scaler:** this component performs GIF image scaling without the need to convert a GIF image into a native bitmap.
- **Image Conversion Codecs:** These are the workhorses (plug-ins) which perform tasks specific to an image format.
- **Image Processor:** This is a type of utility class to help with low-level image manipulation and color conversion; it owns a `TBitmapUtil` object.
- **TBitmapUtil:** This is a set of utility methods that allow pixel-wise native bitmap operations.
- **Bitmap Transformation:** This is a set of classes which can do simple bitmap transformations (scaling, rotation, etc.)
- **Exif Utility:** This is a set of helper classes which simplify Exif metadata processing.
- **Image Transform:** This API can be used to perform transformations (such as scaling and thumbnail addition or removal) of images in a well-defined format.
- **Image Display:** This can be used by applications for easy display of animated image formats. However, S60 doesn't include this component so it can't be used for code that is intended to be portable across UI platforms.
- **ICL Recognizer:** This is a set of services which implement MIME type and image type recognition for both the ICL and system-wide application services.

The main purpose of the Image Conversion component is to provide image conversion services between well-defined formats and native Symbian OS bitmaps (known as encoding and decoding). The API has been designed so that it provides a developer with a unified interface capable of handling virtually any image format. The high-level class diagram of this API is shown in Figure 6.2. Although a fair number of classes are shown, there are additional classes that have been omitted for clarity.⁴

⁴This chapter does not describe the plug-in APIs of the ICL. These lie outside the scope of this book, because they are specifically for use by those writing their own plug-ins for use by `CImageDecoder`, `CImageEncoder`, `CImageTransform` or `CImageDisplay`.

The key player in image decoding is the `CImageDecoder` class. This API is an interface to an extendable plug-in-based system. The plug-in management tasks are handled by `ECOM`.

The API can be used in just three simple steps:

1. Create an image decoder object by providing a factory method with image data.
2. Initiate the image-conversion process.
3. Wait for a request completion and handle the conversion result.

6.2.1 Simple Decoding Example

```
#include <imageconversion.h>
// these are class members of CSimpleDecodeHandler
// RFs iFs;
// CImageDecoder* iDecoder;
// CFbsBitmap* iBitmap; // Native Symbian OS bitmap
// TRequestStatus iStatus;
void CSimpleDecodeHandler::BeginDecodingL()
{
    // we assume that iFs is already initialized, i.e. connection to the
    // file server has been established, and initialize a decoder object -
    // normally a filename passed in from elsewhere rather than hard
    // coding a literal value.
    iDecoder = CImageDecoder::FileNewL(iFs, _LIT("jpeg_image.jpeg"));
    // create a destination bitmap object
    iBitmap = new (ELeave) CFbsBitmap();
    // initialize bitmap by making its size 20x20 and 24bpp color mode
    User::LeaveIfError(iBitmap->Create(TSize(20,20), EColor16M));
    // kick off decoding
    iDecoder->Convert(&iStatus, *iBitmap);
    SetActive();
}
// This will be called once decoding is complete
void CSimpleDecodeHandler::RunL()
{
    // handle decoding result
    if (iStatus.Int() == KErrNone)
    {
        // do something with converted bitmap
    }
}
```

The good news is that it could be as simple as this; the bad news is that this piece of code may not work on some devices. However, we will come back to this example later (see Section 6.2.4), as it has been made as clear as possible by using the simplest API methods.

We passed a `TRequestStatus` as one of the parameters to the `CImageDecoder::Convert()` method. `CImageDecoder` is not an active object, despite the fact that it has a `Cancel()` method; instead it is a façade for a set of objects that perform a number of tasks asynchronously.

We've just decoded a JPEG image into a native bitmap. It is rather a simple piece of code, but it is missing one important step – the proper way to create a native bitmap, as we just used a hard-coded size and color mode. This is the place where we should take a closer look at the Image Decoder API.

6.2.2 Setting up Initial Parameters

Usually image decoding is not as simple as we've shown, assuming we care about the end result and performance. In most cases, compressed images are encoded using a certain color depth. The destination device for our native bitmap has its own capabilities too, such as color depth and resolution.

So, in order to achieve the best possible results we have to set up some decoding options before starting the decoding process.

There are three options for loading the right plug-in into memory:

- Fully automatic plug-in resolution: Image conversion implicitly uses subcomponents (recognizer and resolver) to detect the given image type and loads the best matching plug-in for that image. This is what we used in the code above, just by providing the `RFs` instance and filename.
- Semi-automatic plug-in resolution: The API user requests a specific type of decoder by supplying either the MIME type or the image type UID and the best matching plug-in is used to handle this image. The plug-in system is not limited to just one plug-in for the given image type.
- Manual resolution: The API user explicitly specifies which plug-in to use by supplying its UID. The plug-in system is built on top of `ECOM`, so the usual plug-in identification mechanism applies.

There are several overloaded methods which can be used to create an image decoder object; they can be divided into two groups:

```
CImageDecoder::FileNewL()  
CImageDecoder::DataNewL()
```


The first group creates an image decoder object for an image which is to be read from the file system or DRM-enabled storage during decoding. The second group can be used to decode an image whose data resides in a memory buffer. Let's consider one of the `FileNewL()` methods:

```
FileNewL(RFs& aFs, const TDesC& aSourceFilename,
        const TOptions aOptions = EOptionNone,
        const TUid aImageType = KNullUid,
        const TUid aImageSubType = KNullUid,
        const TUid aDecoderUid = KNullUid);
```

- `aFs` is a reference to a file server session object, this saves some memory and time by sharing the same file server connection between several objects.
- `aSourceFilename` is a file name (including path). There is a version of the same method which accepts a handle (`RFile`) to an already opened file.
- `aOptions` is a very important parameter as it can be used to set certain decoding options which can be specified only during the creation of the decoder object. We will come back to this parameter description in Section 6.2.6).
- `aImageType` is the image type UID that can be used to trigger the semi-automatic plug-in resolution mode. This parameter narrows the list of plug-ins to be considered as candidates for the image decoding. For example, an API user may want to handle only JPG images; in such cases, `KImageTypeJPGUid` should be used. Other Symbian-defined image type UIDs can be found in the `icl\imagecodecddata.h` header file.
- `aImageSubType` can be used to narrow a list of decoder plug-ins to a certain image subtype. You can specify this parameter only if the image type has been supplied. Certain image types may have defined subtypes which may be handled differently. Symbian-defined image subtypes can be found in the `icl\imagecodecddata.h` header file. For example, `KImageTypeTIFFUid` has two subtypes, `KImageTypeTIFFSubTypeLittleEndianUid` and `KImageTypeTIFFSubTypeBigEndianUid` for little-endian and big-endian formats respectively.
- `aDecoderUid` has a dual meaning. It can either mean that the plug-in resolver has to load a specific plug-in (for example, in the case of the BMP plug-in, it is `KBMPDecoderImplementationUidValue`) or one of a class of decoder plug-ins (that is, a set of plug-ins which may support different image formats but share some common features,

such as, Exif metadata support). These decoder plug-ins may wish to expose extra features which are not a part of the standard `CImageDecoder` class by implementing a class derived from `CImageDecoder`. For example, there is a `CJpegEXIFDecoder` class which is derived from `CImageDecoder`; how can we get an instance of that class if `FileNewL()` returns `CImageDecoder`? It can be done by using a standard C++ typecast such as `static_cast<>`, but only if the decoder object has been created using the `KUIDICLJpegEXIFInterface` value for the `aDecoderUid` parameter.

Once a decoder object has been created using one of the factory methods, the plug-in has been selected and it's too late to specify a specific implementation. You can query the decoder's implementation UID after creation and typecast to the appropriate sub-class to use extended functionality, but this is likely to make your code structure more complex.

The following overload can be used to create an image decoder object by supplying the image MIME type, thus narrowing the list of potential plug-ins to be considered:

```
CImageDecoder* FileNewL(RFs& aFs, const TDesC& aSourceFilename,
                        const TDesC8& aMIMEType,
                        const TOptions aOptions = EOptionNone);
```

There is no list of predefined MIME types supported by the system, instead it can be discovered at run time by calling the static method `CImageDecoder::GetMimeTypeFileL()`.

6.2.3 Getting Information about the Opened Image

Our first attempt to open and decode the image used quite a naïve approach to the destination bitmap size – we just used a square bitmap of size 20 by 20 pixels. The following methods of the `CImageDecoder` class can be used to obtain the most vital information about the opened image:

```
TInt FrameCount() const
const TFrameInfo& FrameInfo(TInt aFrameNumber = 0) const
```

`FrameCount()` is used to obtain the number of frames from which this image is made (except for the special case of the MNG format, which is discussed in Section 6.2.12). This enables `CImageDecoder` to handle multiframe or animated image formats such as GIF. In most

cases, `FrameCount()` returns 1 for still image formats, such as PNG, however some still image formats (such as the Symbian OS MBM image format) allow the embedding of several images into a single file and `FrameCount()` should return the number of such images. Neither `CImageDecoder::FrameInfo()` nor `CImageDecoder::Convert()` can be used with frame numbers greater than or equal to `CImageDecoder::FrameCount()`.

`FrameInfo()` retrieves information about the specific frame of the image. It returns a `TFrameInfo` structure. At the moment, we are interested in just three public members of this structure:

```
TDisplayMode iFrameDisplayMode;
TRect iFrameCoordsInPixels;
TSize iOverallSizeInPixels;
```

iFrameDisplayMode

`TDisplayMode` provides information about the frame's native color mode. This color mode should be used for the frame decoding in order for the output bitmap to be as close to the original image as possible. An application may consider using the current system-wide display mode instead of the frame's original color mode in the case when a converted frame will be displayed within the application UI (the custom icon use case).

When you use the image's native color mode, the decoder may have to do less work. If the image's original color mode differs from the system-wide one, then (depending on the screen driver and underlying hardware), the final image presented to an end user may appear of lower quality in comparison to the decoder-converted image.

When you use the system-wide color mode, the decoder plug-in produces the best-looking (from its point of view) frame when converted to the destination color mode. However that may come at the price of reduced decoder performance as it involves perceptual color conversion.

There is no rule of thumb in this situation. In many cases, decoding a frame to the widest color mode (`EColor16M` or `EColor16MA`, if supported) produces the best performance–quality ratio, at some expense of memory consumption. Performance issues are discussed in Section 6.2.15.

iFrameCoordsInPixels

This `TRect` contains the coordinates and size of the given frame within the image. For animated image formats (such as GIF), a frame may be situated somewhere within the whole image, hence the use of a `TRect` rather than just a `TSize`. For still images, in most cases, the `TRect` would look like this: `(0, 0, image_width, image_height)`.

iOverallSizeInPixels

We can easily obtain the frame size by using the `Size()` method of the `TRect` class. Alternatively, where we know we are dealing with single-frame images we would typically use the third member, `iOverallSizeInPixels`, rather than the frame coordinates to get the image size. However, when we are dealing with an animation this member has a different interpretation (as shown in Figure 6.3) and we would typically use the frame coordinates for individual frame sizes.

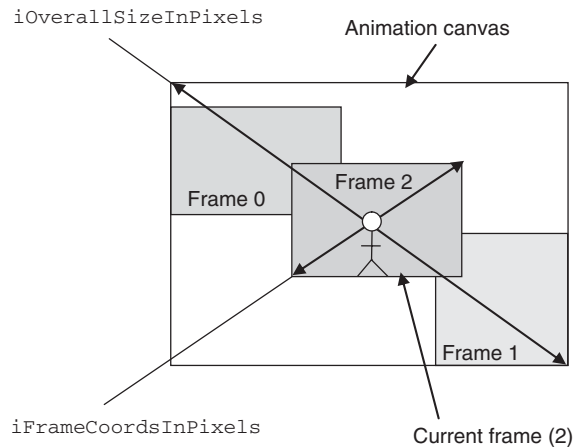


Figure 6.3 Animation composition and frame coordinates

Note how individual frames within an animation can change size and position on the animation canvas.

Using the TFrameInfo Structure

So, to be accurate we should refine our simple image decoding class by creating the destination bitmap as shown in this code:

```
// get the first frame information
const TFrameInfo frameInfo( iDecoder->FrameInfo(0) );
// create a destination bitmap of the frame size and native color mode
User::LeaveIfError( iBitmap->Create(frameInfo.iFrameCoordsInPixels.Size(),
                                frameInfo.iFrameDisplayMode) );
```

The `TFrameInfo` structure is the main source of information about successfully opened image frames. The `CImageDecoder` class doesn't have a notion of 'the whole image', instead an API user should think about an image as a sequence of frames.

Table 6.1 provides a summary of the information which can be obtained about the image frame using public data members or methods.

Table 6.1 Contents of `TFrameInfo`

| | |
|-----------------------------------|---|
| <code>TFrameInfoState</code> | The current frame processing state. Usually all the image frames are parsed once the <code>CImageDecoder</code> instance is created i.e. during object creation. However there is no strong guarantee that all the decoder plug-ins work in this way. Another case in which frame status has to be checked is progressive image decoding (see Section 6.2.13), which can be used when an image is streamed across a network. It is safe to rely on the rest of this structure data only if <code>TFrameInfo::CurrentFrameState() == EFrameInfoProcessingComplete</code> . |
| <code>CurrentDataOffset()</code> | Retrieves the data pointer offset being used by the plug-in while processing this frame. It is of little use to API clients but is used by the framework to interact with the plug-in. |
| <code>FrameDataOffset()</code> | Retrieves the frame data offset within the whole image file; again, it is used by the framework to manage data transfers to the plug-in during image decoding. |
| <code>iFrameCoordsInPixels</code> | The current frame coordinates in pixels within the animation sequence. Some animated image formats (e.g. GIF) support animation frames in which position and size may vary. Use <code>iOverallSizeInPixels</code> for single-frame images. |
| <code>iFrameSizeInTwips</code> | Current frame size translated into twips. ⁵ (You can ignore twips unless you are writing ‘what you see is what you get’ publishing software.) |
| <code>iBitsPerPixel</code> | The native color depth of the frame, which may change from frame to frame. |
| <code>iDelay</code> | The animation renderer should perform this delay (in microseconds) after presenting this frame to a viewer and before rendering the next frame. |
| <code>iFlags</code> | This member holds a number of flags (see Table 6.2) that define the current frame properties. |
| <code>iOverallSizeInPixels</code> | The size which is needed to compose this frame into an animation sequence taking into account the animation canvas size. It can be seen, roughly, as the overall size of the animated image. In most cases, it is equal to <code>iFrameCoordsInPixels.Size()</code> . The GIF image decoder supplied by Symbian always sets this member to the <code>iFrameCoordsInPixels.Size()</code> . |
| <code>iFrameDisplayMode</code> | The frame’s original color mode. In most cases, it is better to use the same color mode when creating the frame’s destination bitmap. |
| <code>iBackgroundColor</code> | The background color, which you are advised to use when the animation canvas is being drawn for the first time or is being cleared due to the <code>ERestoreToBackground</code> instruction. |
| <code>iFrameSizeInPixels</code> | The current frame size in pixels. It is valid only when <code>EUsesFrameSizeInPixels</code> is set. This member can be seen as redundant; however it was introduced in Symbian OS v9.2 to fix some ambiguity which arises when handling animated GIF images. This member, if valid, defines the original unadjusted frame size in pixels. |

⁵The term ‘twip’ is derived from ‘twentieth of an inch point’ – it is a typographical measurement defined as 1/20 of a typographical point, about 1/1440 inch.

Table 6.2 Possible Contents of `iFlags`

| | |
|-------------------------------------|---|
| <code>EColor</code> | An indicator that the frame has separate color channels; its absence means that the frame is grayscale. |
| <code>ETransparencyPossible</code> | An indicator that the frame has some transparent parts, i.e. they are masked out; its absence means that all the pixels of the frame are opaque and visible. |
| <code>EFullyScaleable</code> | An indicator that the frame can be scaled to virtually any destination size; absence of this flag means that scaling capabilities are restricted to a certain set of coefficients (see Section 6.2.4 for more information about scaling). |
| <code>EConstantAspectRatio</code> | An indicator that the frame can only be scaled to a destination size that is of the same aspect ratio (i.e. width to height ratio) as the original frame size. |
| <code>ECanDither</code> | An indicator that the frame can be decoded to virtually any destination display (color) mode; absence of this flag means that the frame is decoded to the display mode specified by <code>iFrameDisplayMode</code> . |
| <code>EAlphaChannel</code> | An indicator that the frame has an alpha channel; absence of this flag means that all the pixels are fully opaque. This flag appears only if the <code>ETransparencyPossible</code> flag is present. In most cases, the alpha channel is obtained using the optional mask parameter of the <code>Convert()</code> method. It is not possible to enquire the bit depth of the alpha channel, so the <code>EGray256</code> type of color mode should always be used to obtain alpha channel information. |
| <code>ELeaveInPlace</code> | An indicator that the animation canvas contents should be left as they are before displaying this frame. This flag also defines a disposal bookmark (see <code>ERestoreToPrevious</code>). This flag, <code>ERestoreToBackground</code> and <code>ERestoreToPrevious</code> can be used by the decoder plug-in to specify the frame composition instructions for animated (multiframe) images. These flags are mutually exclusive. If none of these flags are present, you should replace the existing contents of the animation canvas with this frame's pixels and define the frame disposal bookmark. |
| <code>ERestoreToBackground</code> | An indicator that the animation canvas contents should be filled using <code>iBackgroundColor</code> before displaying this frame. This flag also defines a disposal bookmark. |
| <code>ERestoreToPrevious</code> | An indicator that the animation canvas contents should be restored to the contents defined by the latest disposal bookmark before displaying this frame's contents. |
| <code>EPartialDecodeInvalid</code> | An indicator that the output bitmaps do not contain image data suitable for displaying. This flag may appear when a request to the <code>Convert()</code> method has been completed with a <code>KErrUnderflow</code> error code (see Section 6.2.8 for more information). |
| <code>EMngMoreFramesToDecode</code> | An indicator that is used only by the MNG plug-in to indicate that there are more frames to decode. See Section 6.2.12 for more details. |
| <code>EUsesFrameSizeInPixels</code> | An indicator that the <code>iFrameSizeInPixels</code> member is valid. Absence of this flag means that <code>iFrameSizeInPixels</code> should not be used by the client application. |

6.2.4 Decoding to a Different Size

Let us consider a rather common use case of decoding an image to a different (usually reduced) size from its original. That use case can often be seen in applications that do not control the way images are fed into an application, such as image-viewing applications, applications that can exchange images, MMS applications, etc.

We would like to be able to scale images to a certain size, either so it is possible to fit several image thumbnails into the application workspace or to show a small image preview. Another use case would be to obtain the next smallest size to fit the screen whilst preserving the aspect ratio of the image (i.e. not stretching).

There are several ways of scaling images to a certain size:

- use features specific to an image format, such as thumbnail support (see Section 6.2.5)
- use output bitmap scaling (see Section 6. 6.1)
- use `CImageDecoder` API support for decoding images to a different size.⁶

The latter option is probably the most difficult to use because its level of support may vary across the range of decoder plug-ins. It is usually the most efficient option in terms of performance and memory consumption, as a decoder plug-in may perform the scaling operation efficiently in the course of image decoding and even skip decoding of image regions and pixels which are not required. Using decoder-supplied features is not straightforward, as it requires discovery of decoder plug-in capabilities at run time.

A well-written application should follow an algorithm such as the one shown in Figure 6.4.

Calculating the Size

How can that algorithm be translated into code? Most pieces of the puzzle are already there – the algorithm has references to the `CImageDecoder::ReductionFactor()` and `CImageDecoder::ReducedSize()` methods. Let us take a closer look at them.

⁶In Symbian OS v9.5, the `CImageDecoder` API has explicit support only for scaling down images. Support for scaling up will probably be made available in a later version of Symbian OS. Symbian OS v9.5 has prototypes of the APIs, so curious readers may explore `imageconversion.h`.

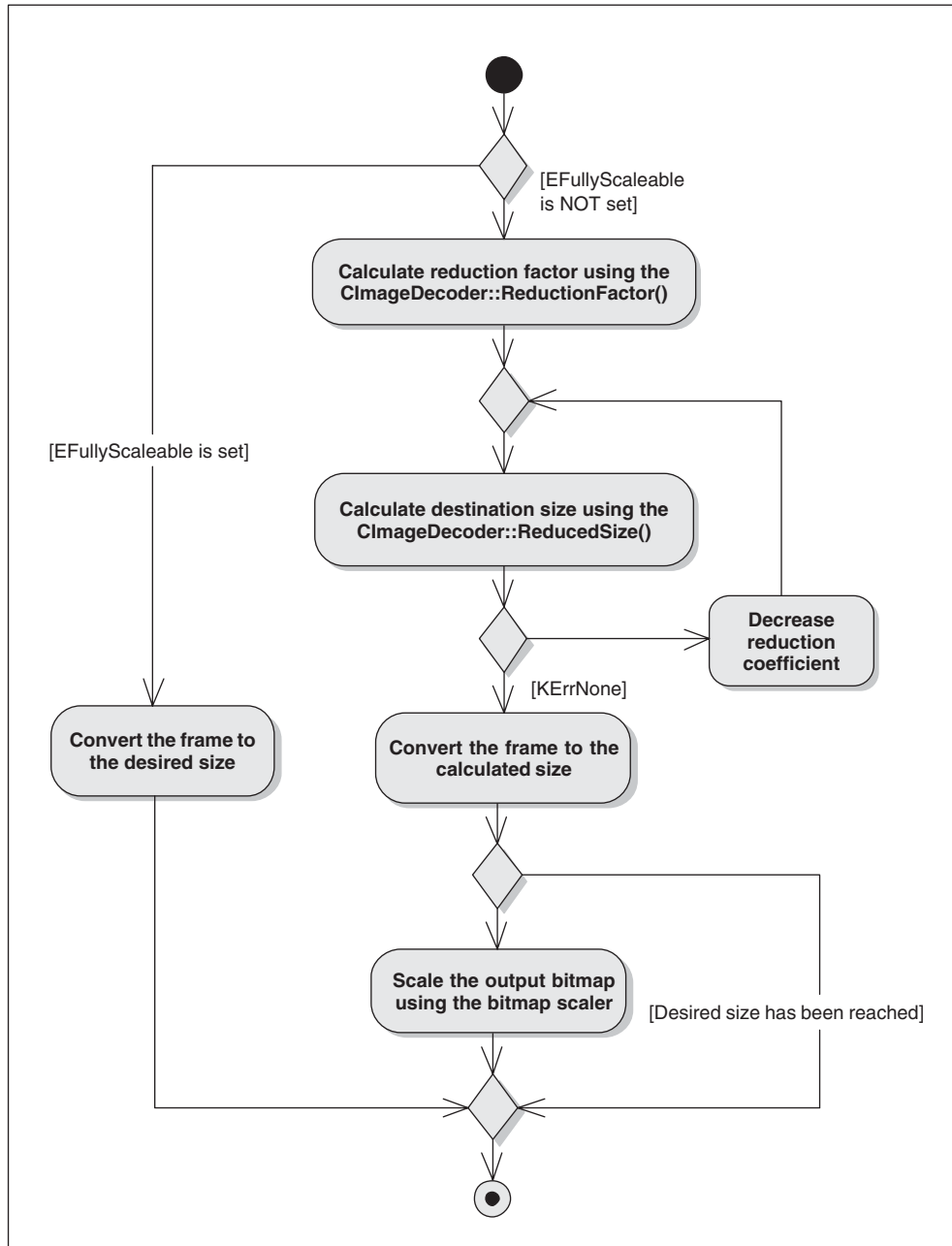


Figure 6.4 Decoding an image to a desired size


```
TInt ReductionFactor(const TSize& aOriginalSize,
                    const TSize& aReducedSize) const
```

- `aOriginalSize` is the frame's original size which can be obtained from the `TFrameInfo`.
- `aReducedSize` is the desired destination size.

The result of this method would be an integer which is the scaling coefficient calculated by the decoder plug-in based on the parameters provided. It is a value specific to the plug-in but, in most cases, it is equal to the binary logarithm of the divisor of the ratio of the image to the destination size.

```
TInt ReducedSize(const TSize& aOriginalSize,
                TInt aReductionFactor,
                TSize& aReducedSize) const
```

- `aOriginalSize` is the frame's original size which can be obtained from `TFrameInfo`.
- `aReductionFactor` is the scaling coefficient obtained by calling the `ReductionFactor` method.
- `aReducedSize` is an output value which represents a size for the given scaling coefficient.

The return value is a standard error code, where `KErrNone` represents successful operation (as usual) and also means that `aReducedSize` has been set to a valid value.

This two-step process is required due to the simple fact that different decoder plug-ins may apply different logic as to what scaling coefficients mean, the coefficient range availability and number rounding (which usually depends on the underlying image format).

To illustrate this, try deciding the output image size in a case when an end user asks for an image of size (35,33) to be decoded to size (15,16).

Converting the Frame

Now we have sorted out the destination size (well, the destination size adjusted to the closest possible match which the decoder plug-in can manage), it is time to convert the frame into a bitmap. This operation does not require any additional effort in comparison to what was done in Section 6.2.1. The only difference is that the calculated destination image size is used when creating the, output bitmap instead of the original image size:

```

const TSize frameSize( frameInfo.iFrameCoordsInPixels.Size() );
// calculate the reduction factor for the given desired size
TInt reductionFactor = iDecoder->ReductionFactor(frameSize,
                                                aDesiredSize);
// try to get the size object for the given reduction factor
User::LeaveIfError( iDecoder->ReducedSize(frameSize, reductionFactor,
                                                aEffectiveSize) );

// now create a bitmap
User::LeaveIfError( iBitmap->Create(aEffectiveSize,
                                frameInfo.iFrameDisplayMode) );

```

The image decoder plug-in tries to decode the frame so it fits into the destination bitmap as well as possible.

So why can't we simply create a bitmap of the desired destination size and assume that the decoder will adjust it automatically? The answer is simple – a decoder plug-in (at least those which are supplied by Symbian) never adjusts the destination bitmap size because it would destroy the contents; rather, it renders the image into the top-left corner of the bitmap. When the destination size can't be matched exactly, either the output contains a border or an error code is returned and no image decoding takes place. The latter may happen when the destination size cannot be created due to restrictions on the scaling coefficient range or if the given plug-in doesn't support downscaling.

Implementing the Conversion

Now let us translate the algorithm from Figure 6.4 into a piece of code. For the time being we are not going to use bitmap scaling to reach the desired destination image size.

The user of `CalculateEffectiveSize()` has to decide if the effective size satisfies their needs. In the case where the smallest achievable size is too big, you should use bitmap scaling to further reduce the scale of the output bitmap.

```

// Calculate the best possible match for the desired destination size
// Will return error if no downscaling is supported by the plug-in
TInt CSimpleDecodeHandler::CalculateEffectiveSize(
    const TSize& aDesiredSize,
    TInt aFrameNo, TSize& aEffectiveSize) const
{
    const TFrameInfo& frameInfo = iDecoder->FrameInfo(aFrameNo);
    // where the frame is fully scalable, we do not need to use the
    // reduction factor as the decoder scales to any destination size
    if ((frameInfo.iFlags & TFrameInfo::EFullyScaleable) ==
        TFrameInfo::EFullyScaleable)

```

```

{
    // if the aspect ratio has to be preserved, then we should calculate
    // the effective size taking into account the original size.
    // This is left as an exercise for the reader.
    aEffectiveSize = aDesiredSize;
    return KErrNone;
}
// get the original frame size
const TSize frameSize( frameInfo.iFrameCoordsInPixels.Size() );
TInt error;
// calculate the reduction factor for the given desired size
TInt reductionFactor = iDecoder->ReductionFactor(frameSize,
                                                  aDesiredSize);
do
{
    // try to get the size object for the given reduction factor
    error = iDecoder->ReducedSize(frameSize, reductionFactor,
                                  aEffectiveSize);

    // An error usually means that the downscaling coefficient required is
    // too big so we try reducing it until we get a size which is
    // supported
    // while (error != KErrNone && --reductionFactor > 0);
    // Downscaling is not available, as reduction factor of 0 means the
    // original size
    if (reductionFactor < 0)
    {
        return KErrNotSupported;
    }
} while (error != KErrNone);
return error;
}

```

6.2.5 Decoding Thumbnails

The `CImageDecoder` class also has support for decoding image thumbnails. This support is fairly generic and even decoders for image formats that don't have a thumbnail concept may implement it by generating thumbnails from a normal image on request.

However simple the API is, this feature is a bit tricky to use. `CImageDecoder::SetImageTypeL()` can be used to switch between image types. You should use `EImageTypeThumbnail` for the thumbnail and `EImageTypeMain` for the main image. It is possible to call this method many times thus switching back and forth between thumbnail and main image.

```

// By default, the image decoder uses the main image, so there is no
// need to call iDecoder->SetImageTypeL(CImageDecoder::EImageTypeMain);
const TFrameInfo mainImageInfo( iDecoder->FrameInfo(aFrameNo) );

```

```
// try to switch to the thumbnail image of the given frame
TRAPD(error, iDecoder->SetImageTypeL(CImageDecoder::EImageTypeThumbnail));
TFrameInfo thumbnailInfo;
// read the thumbnail information if there is one
if (error == KErrNone)
{ // using the first frame in this example
  thumbnailInfo = iDecoder->FrameInfo(0);
}
```

If `SetImageTypeL(EImageTypeThumbnail)` succeeds, the decoder object switches to the thumbnail data so the thumbnail ‘frame’ information is seen instead of the main image frame information.

If `SetImageTypeL(EImageTypeMain)` is called, the image frame information is swapped back.

We are not taking a reference to `TFrameInfo`, as was done previously, but a full copy of the class. This is needed in order to protect ourselves from the `FrameInfo()` implementation details as otherwise there is a risk of data mutability caused by the `SetImageTypeL()` call.

6.2.6 Decoder Creation Options

We have explored the basic (and the most common) use cases for image decoding, so now is the time to dive into some optional parameters which can be specified during the creation of the image decoder object. These parameters can not be changed once the object has been created. These options are defined in the `CImageDecoder::TOptions` enumeration in Table 6.3.

It is possible to request several options at a time by combining them using the bitwise or operator. The result must be of `CImageDecoder::TOptions` type so it can be type-cast or you can construct a new object as follows:

```
iDecoder = CImageDecoder::FileNewL(iFs, aFileName,
    CImageDecoder::TOptions(CImageDecoder::
        EAllowGeneratedMask |
        CImageDecoder::EOptionAlwaysThread));
```

6.2.7 DRM Content Support

The `CImageDecoder` API has full support for handling DRM content. There are several `CImageDecoder::FileNewL()` overloads which can be used to create a decoder for decoding a DRM-protected image.

Table 6.3 The `CImageDecoder::TOptions` Enumeration

| | |
|--|---|
| <code>EOptionNone</code> | This option means ‘nothing special has been requested’; it is used by default if no other options have been specified. |
| <code>EOptionNoDither</code> | This option instructs the decoder object not to apply color dithering when a frame is being decoded to a color mode other than native one i.e. the one which can be retrieved using the <code>iFrameDisplayMode</code> member of the frame information structure. Note: Color dithering is a rather expensive operation, so if you care more about performance than quality, then it is better to specify this flag. |
| <code>EOptionAlwaysThread</code> | This option allows an API user to ask the framework to create a decoder object in a separate operating system thread. This is often useful when the responsiveness of the application user interface is a top priority during the decoding operation. Creating an OS thread involves some overhead, in that it takes longer to create the decoder, consumes slightly more memory and involves inter-thread communication as the framework will have to marshal a vast majority of API calls into another thread. Note: Use of this option still requires the application to respect the active scheduler. An application must never call <code>User::WaitForRequest()</code> to wait for completion of the <code>CImageDecoder::Convert()</code> call, as the framework itself may use active objects during the decoding process to communicate between threads and this would cause a deadlock even if the decoder object operates in a separate thread. |
| <code>EOptionAllowZeroFrameOpen</code> | This flag instructs the framework to successfully complete decoder object creation even if there is not enough data to create even one frame information structure. This flag is used during streamed data decoding. Note: If this flag has been specified, then an application checks <code>IsImageHeaderProcessingComplete()</code> when decoder creation fails, in order to determine if the frame information has been fully populated. |

(continued overleaf)

Table 6.3 (continued)

| | |
|---------------------|--|
| EAllowGeneratedMask | This option instructs the decoder to automatically generate a mask bitmap when no transparency information is present in the image. This flag still doesn't guarantee the ability to retrieve some meaningful transparency information by passing the aMask bitmap parameter to the Convert() method. At the time of writing, the only image decoder supplied by Symbian that supports this flag is the WMF image decoder. |
| EPreferFastDecode | This option instructs the image decoder to perform decoding as fast as possible. A decoder, at its own discretion, may sacrifice some quality and use less precise algorithms when decoding the image. Many decoders silently ignore this flag, so it should be safe to specify whenever you favor performance at the expense of quality. |

Decoding of the DRM-protected content is similar to decoding a regular image, apart from setting up some DRM-specific parameters when creating a decoder object:

```
CImageDecoder* decoder = CImageDecoder::FileNewL(fs,
    TMMFileSource(aFileName,
        ContentAccess::KDefaultContentObject,
        ContentAccess::EView,
        EFalse));
```

The TMMSource class defined in mm\mmcaf.h and its derivatives TMMFileSource and TMMFileHandleSource provide containers for DRM-specific parameters for the DRM content referred to by the file name and file handle respectively. Decoding DRM content stored in a memory buffer is not supported for security reasons.

DRM support in Symbian OS is covered in Chapter 2 and in the Symbian Developer Library documentation found in your chosen SDK.

6.2.8 Interpreting Error Codes

Unfortunately real-world applications have to deal with errors. Errors may arise due to input data being invalid, inconsistent application requests to the API, implementation restrictions, and so on.

The image decoder API uses normal system-wide error codes to let the user know if something goes wrong. There are no CImageDecoder-specific error codes defined.

The most common source of error codes is the set of `CImageDecoder` factory functions such as `FileNewL()`. These methods may leave with various error codes, such as `KErrNoMemory` (which can be easily interpreted) or any other file-system-related error codes because the API implementation uses file system functionality and does not translate error codes but simply passes them on.

Some error codes may mean different things depending on when the errors occur. Table 6.4 lists the most common error codes which may be produced by the `CImageDecoder` API.

Table 6.4 Common `CImageDecoder` Error Codes

| | |
|----------------------------|--|
| <code>KErrNotFound</code> | This error code may mean either ‘source file not found’ or ‘no decoder capable of decoding the given image format found’. One way of avoiding this ambiguity is to use the version of <code>FileNewL()</code> which accepts a file handle (<code>RFile</code>); if the file has been successfully opened, then <code>KErrNotFound</code> means that no suitable decoder plug-in has been found. |
| <code>KErrUnderflow</code> | <p>This error code should usually be read as ‘system has insufficient data in the image file/buffer to complete an operation’. This usually occurs when an attempt is made to decode a partially downloaded (truncated) image file. This error code may be encountered during the creation of the decoder object or during decoding. In the latter case, the error is not fatal, as the API client has the following options to deal with the error code:</p> <ul style="list-style-type: none"> • When more data can be obtained for the image (e.g. if the file being streamed from a network connection), it can wait for more data to arrive, append the data to the original file or memory buffer and retry the decoding attempt by calling <code>CImageDecoder::Convert()</code> with the same parameters. • When no more data can be obtained, it can check the presence of the <code>EPartialDecodeInvalid</code> flag for the given frame. If this flag is present, then the output bitmap doesn’t contain any valid image data and the error is fatal. If the flag is absent, it means that the output bitmap has part of an image decoded into it and the application may use this bitmap. Unfortunately, due to API limitations, it is not possible to determine which regions of the image have been successfully decoded. |
| <code>KErrArgument</code> | This error code usually means that the destination bitmap size cannot be used for decoding this frame; it usually happens when the API client fails to use <code>ReducedSize()</code> and <code>ReductionFactor()</code> to calculate an achievable destination size (see Section 6.2.4). |

(continued overleaf)

Table 6.4 (continued)

| | |
|------------------|--|
| KErrNotSupported | This error code usually means that the decoder failed to cope either with the destination bitmap color mode (see Section 6.2.3) or with the image itself. For example, during the creation phase the image headers may seem to be fine for the decoder loaded, however during the decoding process the decoder may run into some image file format feature which prevents the image from being decoded correctly (since it is not supported by the given decoder). |
| KErrCorrupt | This error may come from one of two sources: <ul style="list-style-type: none"> the file system, in which case something serious has happened to the underlying file system the decoder plug-in, in which case the decoder has found some major image file format discrepancies and cannot proceed further with this decoding. Unfortunately it is not possible to figure out if there was any pixel data decoded at all. |

6.2.9 Additional Frame Information

The `TFrameInfo` structure is a good source of information about opened image frames. It has been designed to be quite generic and provide the most common (and most useful) information.

Some image formats may provide much more information about their frames than `TFrameInfo` may accommodate. There are two more mechanisms for retrieving information about image frames:

- frame comment retrieval
- frame metadata retrieval.

Frame comment retrieval methods can be used to get hold of textual information which may exist in the image. This is usually some sort of author commentary about the image, copyright notices, and so on.

The `CImageDecoder` API defines two groups of methods: one for retrieving image-wide data and one for frame-specific data:

```
TInt NumberOfImageComments() const
HBufC* ImageCommentL(TInt aCommentNumber) const
```

It is quite easy to get all the image comments once an instance of the decoder object is created:

```
const TInt KNumOfComments = aDecoder.NumberOfImageComments();
```



```
for (TInt i=0; i < KNumOfComments; i++)
{
    HBufC* comment = aDecoder.ImageCommentL(i);
    // do something with the comment
    // de-allocate memory
    delete comment;
}
```

The thing to be aware of is that `ImageCommentL()` returns a Unicode descriptor, while many image formats have been designed to use 8-bit ASCII comments. Most of the decoders pad the 8-bit character set to the 16-bit one using the standard `TDes::Copy()` method (the most significant bytes are filled with zeros).

Dealing with frame-specific comments is very similar to dealing with image-wide ones but we have to use frame-specific methods and provide a frame number:

```
const TInt KNumOfComments = aDecoder.NumberOfFrameComments(aFrameNumber);
for (TInt i = 0; i < KNumOfComments; i++)
{
    HBufC* comment = aDecoder.FrameCommentL(aFrameNumber, i);
    // do something with the comment
    // de-allocate memory
    delete comment;
}
```

The following two methods can be used to retrieve human-readable information about the given frame of the image:

```
CFrameInfoStrings* FrameInfoStringsLC(TInt aFrameNumber = 0)
CFrameInfoStrings* FrameInfoStringsL(TInt aFrameNumber = 0)
```

`FrameInfoStringsL()` should provide localized textual information i.e. using the system's current locale settings. You must delete the instance of `CFrameInfoStrings` once you've finished with it.

Apart from well-defined entities such as textual comments, many decoder plug-ins may expose additional (at times quite low-level) information about image frames. This kind of information is specific to a format and quite often applications don't need to know about it.

`CImageDecoder` defines the following method, which returns a collection of abstract pieces of frame information for the given frame number:

```
const CFrameImageData& FrameData(TInt aFrameNumber = 0) const
```

`CFrameImageData` exposes a collection of abstract frame and image properties (see Figure 6.5), which an application identifies using the

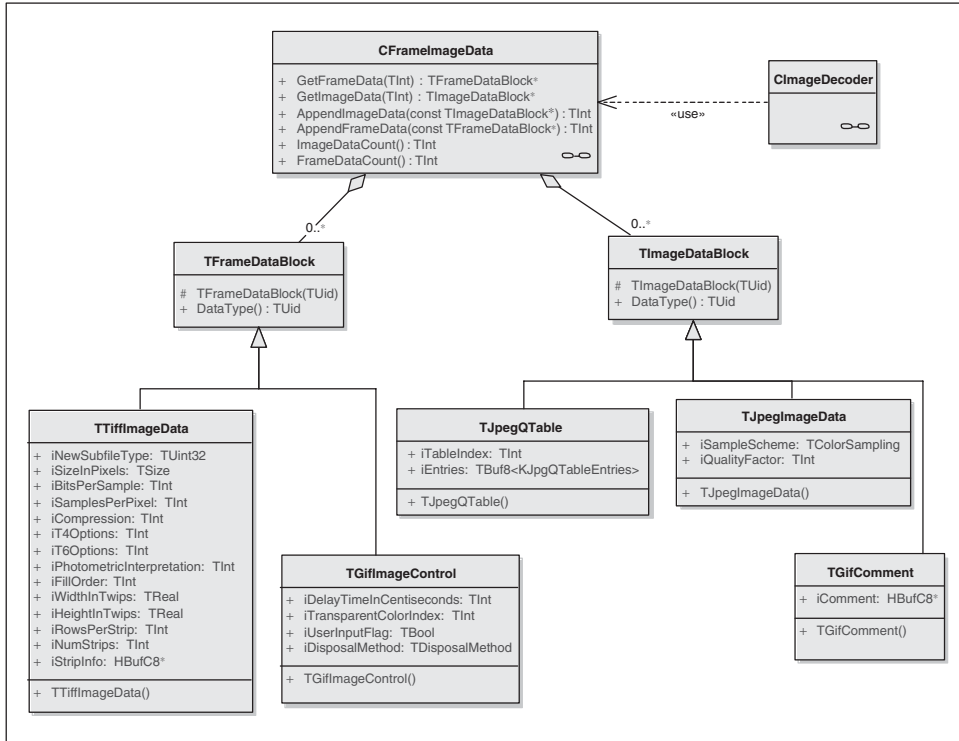


Figure 6.5 CFrameImageData class diagram

`DataType()` method. All the Symbian-defined data type `UId` definitions can be found in the `icl\ImageCodecData.h` header file along with specialized definitions of the data structures.

The API client should implement the following algorithm in order to collect information from the format-specific frame properties:

1. Retrieve an instance of `CFrameImageData` for the given frame index.
2. Iterate over all the properties using the `CFrameImageData::FrameDataCount()` and `CFrameImageData::GetFrameData()` methods.
3. Using `TFrameDataBlock::DataType()`, make a decision on whether the current property is of interest and cast it to the appropriate type.

It should be noted that `CFrameImageData` exposes two collections of properties; the second one can be retrieved by calling `GetImageData()`.

This collection is meant to contain the image-wide format-specific properties, despite the fact an API client has to retrieve it by specifying the frame number. As can be seen from Figure 6.5, there are a fair amount of image-wide properties defined.

The application should iterate over all the image frames in order to collect the image-wide format properties as they can be spread across all the frames. Some implementations may use frame zero to store image-wide properties; that is implementation-specific and a well-written application should not make this assumption.

Here is a fairly simple implementation of the algorithm described above:

```
// get number of frames in the image
const TInt KFrameCount = aDecoder.FrameCount();
// iterate all the frames
for (TInt frameIndex = 0; frameIndex < KFrameCount; frameIndex++)
{
    // retrieve properties collection
    const CFrameImageData& data = aDecoder.FrameData(frameIndex);
    // get number of the elements in the collection
    const TInt KNumDataElements = data.ImageDataCount();
    // iterate all the properties
    for (TInt i = 0; i < KNumDataElements; i++)
    {
        // retrieve the current abstract block
        const TImageDataBlock* block = data.GetImageData(i);
        // depending on the block type, do something with it
        switch ( block->DataType().iUid )
        {
            case KJPGQTableUidValue:
            {
                const TJpegQTable* jpegQTable = static_cast<const
                    TJpegQTable*>(block);
                // use this information somehow
            }
            break;
            case KGIFBackgroundColorUidValue:
            {
                const TGifBackgroundColor* gifBgColor =
                    static_cast<const TGifBackgroundColor*>(block);
                // use this information somehow
            }
            break;
        }
    }
}
```

6.2.10 Plug-in Discovery

The CImageDecoder class has some functionality which allows retrieval of information about the decoder plug-in being used. These informational

methods can be useful for debugging or for implementing plug-in specific behavior (which is, in general, not very good practice).

The following method allows you to get the UID that uniquely identifies the plug-in:

```
TUID ImplementationUId() const
```

In fact there could be several plug-ins (each one with its own unique UID) able to handle the same image type UID. It is the framework's job to decide which plug-in to load for the given image, unless the API client explicitly specifies a plug-in UID to be loaded.

There are a number of static methods of `CImageDecoder` that allow discovery of the available plug-ins installed on the system the image formats they support prior to creation of the decoder object:

```
static void GetImageTypesL(RImageTypeDescriptionArray& aImageTypeArray);
static void GetFileTypesL(RFileExtensionMIMETypeArray&
                          aFileExtensionArray);
static void GetImageSubTypesL(const TUID aImageType,
                              RImageTypeDescriptionArray& aSubTypeArray);
static void GetMimeTypeFileL(RFs& aFs, const TDesC& aFileName,
                              TDes8& aMimeType);
static void GetMimeTypeDataL(const TDesC8& aImageData, TDes8& aMimeType);
static CImplementationInformationType* GetImplementationInformationL(
                                      TUID aImplementationUId);
```

Figure 6.6 describes the relationships between different entities which can be retrieved using these methods.

`GetImageTypesL()` and `GetFileTypesL()` can be used to retrieve `CFileExtensionMIMEType` and `CImageTypeDescription` structures, respectively, for all the installed plug-ins. These structures can be used to find out which image types can be handled by the system along with their attributes (the file extension by which a file can be recognized, MIME type, image type UID, etc.).

The `GetImageSubTypesL()` method can be seen as a helper that can be used to get all the image sub-types for the given image type.

The `GetMimeTypeFileL()` and the `GetMimeTypeDataL()` methods can be used to detect the underlying file and data buffer MIME types respectively. These methods do not use system-wide MIME type recognition routines; instead they match against all of the installed decoder plug-ins.

The `GetImplementationInformationL()` method can be used to retrieve the ECOM-like implementation information for a given decoder plug-in UID.

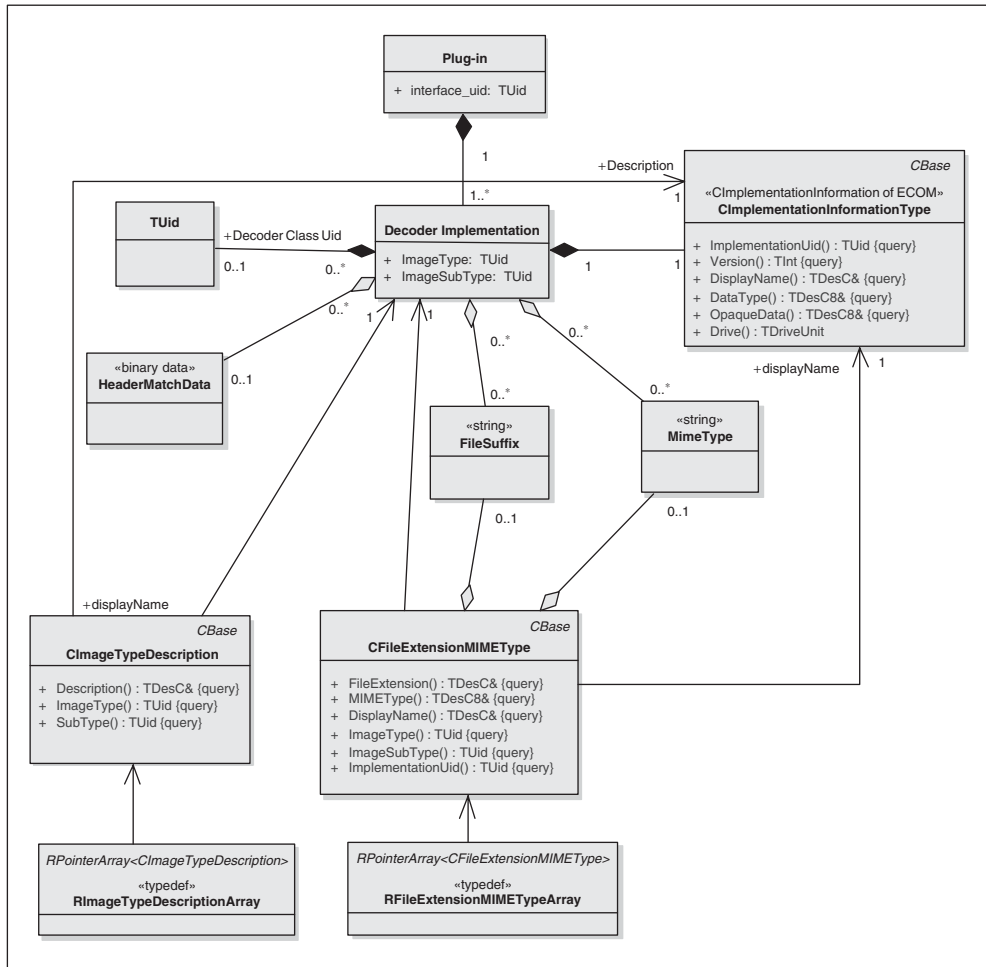


Figure 6.6 Relationships between plug-in information entities

6.2.11 Exif Metadata Processing

Exif metadata processing is closely coupled with the `CImageDecoder` API,⁷ which means that an image decoder object has to be instantiated in order to gain access to the metadata information.

⁷Nokia had created an API for Exif metadata processing before Exif support was introduced in Symbian OS v9.1. Nokia-owned APIs are not part of the core Symbian OS SDK. All the Symbian-provided Exif APIs are excluded from the S60 3rd Edition SDKs. All the UIQ 3 SDKs use core Symbian Exif APIs.

Exif metadata support is not a part of the core CImageDecoder API, but rather an optional extension to it implemented by the CJPEGExifDecoder class which is derived from CImageDecoder.

It is not possible to explicitly create an instance of the CJPEGExifDecoder, so an API client has to use a special technique to create an instance of the decoder object and then use standard C++ type casting. It is advisable to use the aDecoderUid parameter during the decoder instantiation in order to ensure type-safety when casting to CJPEGExifDecoder later on.

In order to get a decoder that supports the Symbian-defined Exif interface, an application should do something like this:

```
// We ask for the decoder object which supports the "Exif Jpeg Interface"
// by specifying the "class Uid" parameter
CImageDecoder* decoder = CImageDecoder::FileNewL(fs, aFileName,
                                                CImageDecoder::EOptionNone,
                                                KNullUid, KNullUid,
                                                TUid::Uid(KUidICLJpegEXIFInterface));

// now it is safe to cast that decoder object to the CJPEGExifDecoder
CJPEGExifDecoder* jpegExifDecoder =
    static_cast<CJPEGExifDecoder*>(decoder);
// now we can obtain MExifMetadata interface and use it to read raw Exif
// metadata or create an instance of the TExifReaderUtility
MExifMetadata* exifMetadata = jpegExifDecoder->ExifMetadata();
TExifReaderUtility exifReader(exifMetadata);
// We can get the "orientation" tag value, for example, like this
TUint16 exifOrientation;
if (KErrNone == exifReader.GetOrientation(exifOrientation))
{
    // make use of the tag value
}
```

The Exif API is quite straightforward and easy to use (apart from the initial step of Exif-compliant decoder creation). One thing missing from this API is that it is not possible to enquire which Exif metadata is present in a given image. The client has to try to get every Exif tag value it is interested in and handle the KErrNotFound error code if this tag is not defined in the image.

6.2.12 Decoder Plug-ins Provided by Symbian

Symbian supplies a number of CImageDecoder plug-ins in its reference SDK (see Table 6.5). Device manufacturers are free to ship these plug-ins with their phones, replace them with their own specific versions or remove them (if they do not need them).

Table 6.5 Decoder Plug-ins

| Image Format Decoder UID | Notes |
|--|--|
| Windows Bitmap (.bmp) KBMPDecoderImplementation- UIdValue | The 1, 4, 8, 16, 24, and 32 bpp and RLE-encoded formats are supported. Device manufacturers usually use this plug-in without changes. |
| Graphics Interchange Format (.gif) KGIFDecoderImplementation- UIdValue | GIF87a and GIF89a formats are supported but the 'Netscape Application Loop Block' is not supported. Device manufacturers usually use this plug-in without changes. |
| JPEG (.jpg) KJPEGDecoderImplementation- UIdValue | Baseline, extended sequential (single scan) and progressive with non-differential Huffman coding JFIF/Exif images are supported. Thumbnails are supported. JPEG2000 images are not supported. The maximum downscaling coefficient supported is 1/8. Many device manufacturers replace this plug-in with a hardware-accelerated version (the plug-in may be left but assigned a low priority). |
| Symbian OS MBM (.mbm) KMBMDecoderImplementation- UIdValue | It is possible to handle native Symbian OS bitmaps (MBM files) in the same way as other images and load them using the CImageDecoder interface. However, using CFbsBitmap::Load() is the recommended way to do it. Device manufacturers usually use this plug-in without changes. |
| Portable Network Graphics (.png) KPNGDecoderImplementation- UIdValue | All valid images are supported. Downscaling is not supported when decoding to EColor16MA or EColor16MAP. Device manufacturers usually use this plug-in without changes. |
| Windows Icon (.ico) KICODEDecoderImplementation- UIdValue | All valid 1, 4 and 8 bpp images, including multiframe icons are supported. Device manufacturers usually use this plug-in without changes. |
| TIFF Fax (.tiff) KTIFFDecoderImplementation- UIdValue | Group3, 1d and 2d coding, and Group4 fax compression, including multiframe faxes, are supported. Device manufacturers usually use this plug-in without changes. |
| Windows metafile (.wmf) KWMFDecoderImplementation- UIdValue | All valid Windows metafiles are supported. Device manufacturers usually use this plug-in without changes. |
| Wireless Bitmap (.wbmp) KWBMPPDecoderImplementation- UIdValue | All valid bitmaps are supported. Device manufacturers usually use this plug-in without changes. |
| Over The Air Bitmap (.ota) KOTADecoderImplementation- UIdValue | All valid bitmaps are supported. Device manufacturers usually use this plug-in without changes. |
| Multiple-image Network Graphics (.mng) KUIdIcIMngPluginImplUId | The 'low complexity' (LC) profile is supported. Embedded JNG images are not supported. CImageDecoder::FrameCount() is not supported. CImageDecoder::FrameInfo() is populated only for the current frame. This plug-in is usually included with UIQ smartphones but is excluded from S60 smartphones. |

The MNG decoder does not support frame count information. So how can a client iterate over all the frames? There is a special flag defined within the `TFrameInfo`, which can be used to check if there are more frames in the animation to decode: `TFrameInfo::EMngMoreFramesToDecode`. An application should use a piece of code such as the following to access all the MNG animation frames:

```
while ((decoder->FrameInfo(0).iFlags &
      TFrameInfo::EMngMoreFramesToDecode) ==
      TFrameInfo::EMngMoreFramesToDecode)
{
    //decode current frame
}
```

MNG animation may have several (sometimes even infinite) animation loops. This is one of the reasons why `FrameCount()` is not supported. You should not cache MNG animation frames as your application may easily run out of memory.

6.2.13 Streamed Decoding

The image decoder component has basic support for streamed decoding. It is possible to decode an image in which physical content is not available immediately in full. This kind of use case is mostly for web-browser-like image downloading. The image decoder API does not support ‘true’ streamed decoding, which means that an application has to accumulate incoming image data on its own.

Figure 6.7 shows an activity diagram for the following streamed decoding scenario. If `CImageDecoder::NewL()` leaves with `KErrUnderflow`, then it means that the ICL has insufficient data to find an appropriate decoder plug-in or a plug-in might have failed to initialize itself given the amount of available data. In this case, the API client has to add more data into the original source and this why an image decoder API does not support streaming in full. The client has to maintain all the input data during the decoding process and keep adding new data to the existing data source, whether a memory buffer or a file.

`CImageDecoder::DataNewL()` accepts a reference to a descriptor as the image data input buffer. In the case of streamed decoding, the descriptor is used during the decoding process, so you must ensure that the descriptor object reflects any changes due to data added to the buffer. `HBufC::ReAlloc()` is not suitable in this case as it creates a new descriptor object. The best way of doing this is to create an instance of `TPtrC` and keep it pointing to the currently active buffer using the `TPtrC::Set()` method. You must also keep the `TPtrC` instance in

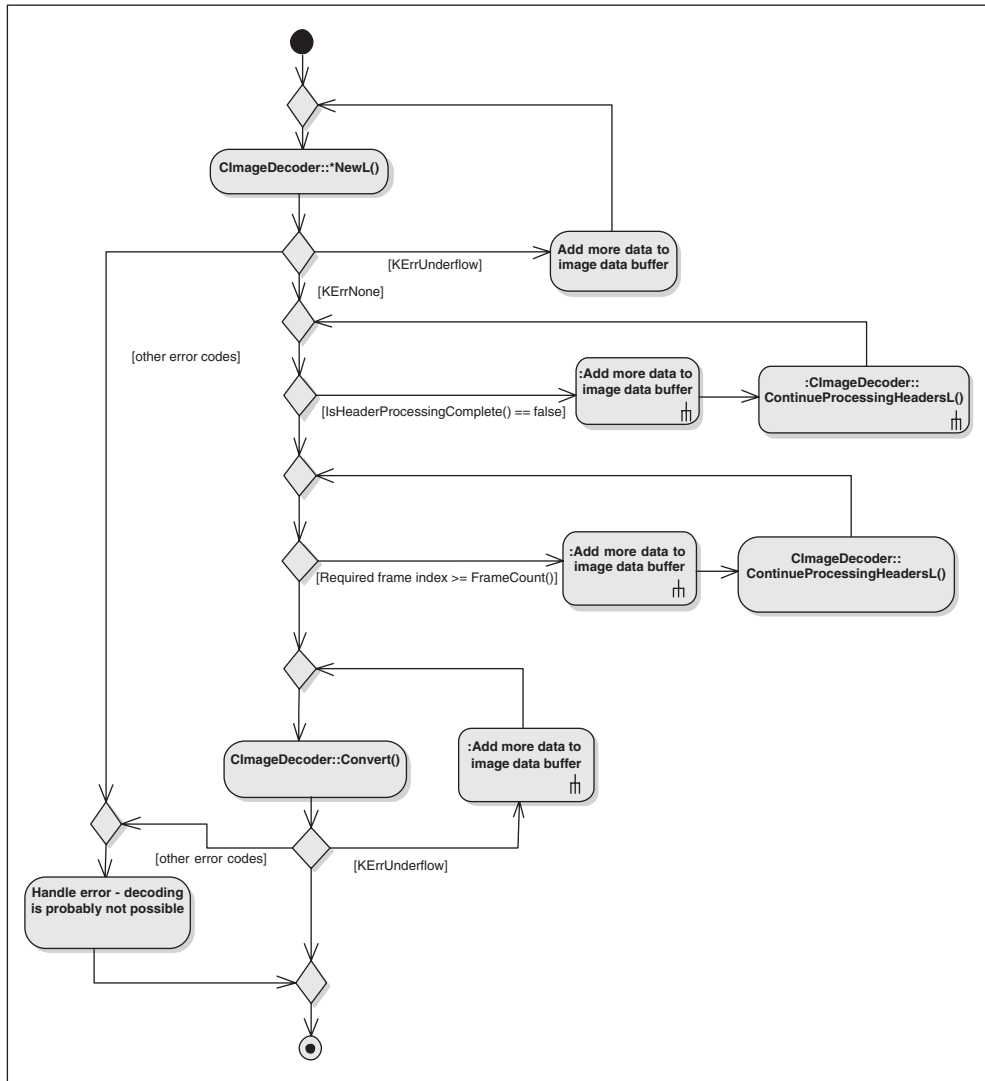


Figure 6.7 Activity diagram for streamed decoding

scope – it can't be a stack-allocated object. Addition of new data into the image data buffer may look like the code snippet below:

```
// declared as members
TUInt8* iBuffer;
TPtrC8 iBufPtr;
// buffer creation
iBuffer = User::AllocL(initialSize);
```

```
iBufPtr.Set(iBuffer, initialSize);
// buffer data addition to the existing buffer
TInt currentSize = iBufPtr.Size();
iBuffer = User::ReAllocL(iBuffer, newSize);
// append new data at the "currentSize" offset and set iBufPtr object
// to point to the new buffer location
iBufPtr.Set(iBuffer, newSize);
```

If the framework has enough data to find a plug-in, the loaded plug-in tries to parse input data and construct image information structures, such as `TFrameInfo`. The plug-in may encounter an unexpected end of data during this process:

- If there is not enough data to populate all the frame information structures, the plug-in returns 'false' from `CImageDecoder::IsHeaderProcessingComplete()` and the result of `CImageDecoder::FrameCount()` may be inaccurate. A client may assume that the application should keep adding more data to the buffer until `CImageDecoder::IsHeaderProcessingComplete()` returns 'true'. This approach may work in many cases but it may fail with certain image formats where, instead of frame information data for all the frames being placed at the beginning of the file, it is spread across the entire file. In such cases, 'streamed' decoding won't be seen by an end user as a progressive process, but more as 'entire file pre-buffering'.
- If there is sufficient data to load and initialize a plug-in but not enough data to populate at least one `TFrameInfo` structure then the default behavior for the framework is to leave with `KErrUnderflow`. This ensures that `CImageDecoder::FrameCount()` always returns at least 1. This approach however may incur a severe performance penalty if many attempts to create a decoder fail due to insufficient data, as every decoder creation attempt involves plug-in resolution and loading. A more advanced application may use the `EOption-AllowZeroFrameOpen` option of `CImageDecoder` to instruct the framework to return a valid decoder object even if no `TFrameInfo` structures have been created.
- If `CImageDecoder::IsHeaderProcessingComplete()` is `EFalse`, then the application keeps adding new data to the buffer and calling `CImageDecoder::ContinueProcessingHeadersL()` to let the decoder create the rest of the `TFrameInfo` structures or reach the required number of frame headers being parsed (which can be obtained by calling `CImageDecoder::FrameCount()`).

Once the desired frame number is less than the `CImageDecoder::FrameCount()` or `CImageDecoder::IsHeaderProcessingCom-`

plete() is 'true', then the application may call CImageDecoder::Convert() for this frame. The plug-in will decode as much of the frame pixel data as possible and may leave with KErrUnderflow when not all of the frame pixels are available. In the case of KErrUnderflow an application may (at its discretion) display partially decoded frame pixels if the EPartialDecodeInvalid flag of the relevant frame's TFrameInfo is not set. The application should keep adding data to the buffer and calling CImageDecoder::ContinueConvert() until it completes with KErrNone.

Implementing flexible streamed decoding can be quite tricky since it requires programming a sophisticated state machine which can take input from several active objects.

6.2.14 CBufferedImageDecoder

The CBufferedImageDecoder API can be used to simplify the streamed decoding process somewhat. It is a wrapper that encapsulates both the CImageDecoder and its input buffer (thus it can be used only when image data resides in memory).

In contrast to the CImageDecoder, the CBufferedImageDecoder can always be created even though there is not enough data to find an appropriate decoder plug-in. An application does not have to manage the entire image buffer on its own, but instead can pass only **new** data in and all the buffer management is done inside the wrapper. However, it means that an application has to check the CBufferedImageDecoder::ValidDecoder() method before calling any methods which require existence of a decoder plug-in, such as CBufferedImageDecoder::FrameCount(), otherwise it will panic.

The algorithm for using the CBufferedImageDecoder class should look like this:

1. An instance of the wrapper is created using the CBufferedImageDecoder::NewL() factory method.
2. A one-off operation, CBufferedImageDecoder::OpenL(), is required to open the buffered image decoder.
3. The application checks CBufferedImageDecoder::ValidDecoder(). If there was not enough data to create an underlying image decoder object (e.g. OpenL() has left with KErrUnderflow) then more data has to be added by calling CBufferedImageDecoder::AppendDataL() and CBufferedImageDecoder::ContinueOpenL() should be called to re-try the decoder creation attempt. Note that there is no need to keep the input buffer data 'alive', as a copy is taken by CBufferedImageDecoder.

4. The `CBufferedImageDecoder` methods `IsImageHeaderProcessingComplete()`, `ContinueProcesessingHeaderL()` and `ContinueConvert()` are used in the same way as for the `CImageDecoder` class.
5. Once image decoding has been completed, it is possible to re-use the same instance of the `CBufferedImageDecoder` for decoding other images by resetting its state (`CBufferedImageDecoder::Reset()`).

Overall, the `CBufferedImageDecoder` can be seen as a fairly simple helper class which can simplify buffered decoding a little at the cost of hiding all the advanced features of `CImageDecoder` from the client.

6.2.15 Performance Hints

There are several measures which you can take to increase performance of image decoding under some circumstances.

- Use `CImageDecoder::DataNewL()` instead of `FileNewL()` if the image is going to be decoded several times, so the file is read from the file system just once.
- Use the `CImageDecoder::EOptionNoDither` | `CImageDecoder::EPreferFastDecode` options during decoder creation if you don't need print-like quality and prefer faster decoding.
- Add the `CImageDecoder::EOptionIgnoreExifMetaData` option if you don't need Exif metadata to be parsed (and hence to be available).
- Either decode to the frame's 'native' color mode (`TFrameInfo::iFrameDisplayMode`) or `EColor16M` bitmap. The latter may in many cases be even faster as most of the Symbian-supplied decoders (and the framework itself) are optimized for this color mode. In fact many ICL components internally use a `TRgb`-like pixel data representation with subsequent conversion of this data into the destination color mode. Other parts of ICL are optimized for this color mode as well (see Section 6.6).

6.3 Encoding Images

Image encoding can be seen as a reverse operation for image decoding as it is all about taking a native bitmap and translating it (encoding it) into another image format. The encoding API is mostly represented by the

`CImageEncoder` class. It is a far simpler API than `CImageDecoder`, although it mirrors most of the decoding functionality.

There are two main use cases for the encoding API:

- The image editor use case. This use case boils down to the necessity of encoding a native bitmap in a well-defined image format, so that the image can be passed to other devices which do not necessarily share the native bitmap format.
- Saving a photo obtained using the onboard camera API.

Another reason for using some of the common image formats instead of native bitmaps is the limited storage size – while the native bitmap format is the most efficient way of storing images as regards performance, it is far from being the most optimal in terms of storage size or output file size.

6.3.1 Using the `CImageEncoder` API

The `CImageEncoder` API shares many usage patterns and data structures with the `CImageDecoder` API. Its basic usage scenario is essentially the same:

1. An encoder object is to be created by using one of the synchronous factory methods `CImageEncoder::DataNewL()` or `CImageEncoder::FileNewL()`.
2. Some optional encoding parameters may be adjusted synchronously.
3. An asynchronous encoding process is initiated by calling `CImageEncoder::Convert()`.
4. The application waits for the request to be accomplished and makes use of the output data in cases when no error has been reported.

So why is this API simpler than the decoding one? Let us examine the differences:

- The encoding client application has to explicitly specify the output file format (either by MIME type or image type UID) and the only input format supported is the native bitmap. There is no support for streamed or partial encoding: the input data must be presented in one complete native bitmap object.
- The encoding parameters may be seen as a bit more complicated as they may require the client application to have some knowledge about the destination file format in order to perform format-specific setup. However, a client application always knows the output file format as it has to specify it while creating an encoder object.

- The encoding process is much simpler as CImageEncoder supports only encoding of single-frame images without transparency information. No scaling is supported either, so the input bitmap is encoded in 1:1 scale where the format permits.
- The CImageEncoder class does not support streamed encoding so any error code apart from KErrNone is fatal and means that encoding has failed. The application may endeavor to fix the problem.

Symbian plans to significantly extend CImageEncoder to address most of these limitations in the near future. Curious readers may explore imageconversion.h to see the new prototype APIs in Symbian OS v9.5.

This is a fairly basic, but fully functional CImageEncoder example:

```
CSimpleEncodeHandler::CSimpleEncodeHandler() : CActive(EPriorityStandard)
{
}

void CSimpleEncodeHandler::ConstructL(const TDesC& aDestinationFileName,
                                     TUid aImageFormatId)
{
    // create a connection to the file server
    User::LeaveIfError( iFs.Connect() );
    // create an encoder object for encoding into a file
    iEncoder = CImageEncoder::FileNewL(iFs, aDestinationFileName,
                                     CImageEncoder::EOptionAlwaysThread,
                                     aImageFormatId);

    // Add this object to the scheduler
    CActiveScheduler::Add(this);
}

CSimpleEncodeHandler::~~CSimpleEncodeHandler()
{
    // Cancel any request, if outstanding
    Cancel();
    // Delete instance variables, if any, and close resources
    delete iEncoder;
    iFs.Close();
}

void CSimpleEncodeHandler::BeginEncode(CFbsBitmap& aSourceBitmap)
{
    // start encoding process
    iEncoder->Convert(&iStatus, aSourceBitmap);
    // Tell scheduler a request is active
    SetActive();
}

void CSimpleEncodeHandler::DoCancel()
{
    // cancel encoding process
    iEncoder->Cancel();
}
```

```

void CSimpleEncodeHandler::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        // bitmap has been encoded successfully
    }
    else
    {
        // something went wrong, deal with error code
    }
}

```

In contrast to the image decoder, the list of encoder object creation options is quite short (see Table 6.6).

Table 6.6 The `CImageEncoder::TOptions` Enumeration

| | |
|--|--|
| <code>EOptionNone</code> | This option means ‘no special options have been requested’; it is used by default if no other options have been specified. |
| <code>EOptionAlwaysThread</code> | This option allows an API user to instruct the framework to create an encoder object in a separate operating system thread and marshal all the calls when required. This option should be used when the responsiveness of the application user interface is the top priority as it involves some overhead in thread creation. Note: Even if this option has been used, the client must still properly manage active objects and <code>User::WaitForRequest()</code> cannot be used to wait for encoding process completion. |
| <code>EOptionGenerate-AdaptivePalette</code> | This option instructs the underlying encoder plug-in to generate an adaptive (i.e. ‘best match’) palette. This option is silently ignored by encoders that do not support adaptive palette generation. You should use this option only when performance is non-critical as it may make the encoder use complex and computationally expensive algorithms for palette calculation. When this option is not used, the Symbian-supplied encoders use the default system palette when required. In some extreme use cases (e.g. encoding a gradient-like picture into a GIF image format), use of the system-wide default palette would make the output file virtually useless, so an application should apply some intelligence as to when to use this option. |

6.3.2 Setting up Encoding Parameters

Apart from the instantiation-time parameters, there are parameters that can be set up before the encoding process takes place. This kind of parameter setup split is used to keep the `CImageEncoder` interface as generic as possible while allowing format-specific settings to be supplied by an API client if necessary.

The setup mechanism for `CImageEncoder` that is specific to each plug-in is driven by parameter data rather than custom interface functions (see Figure 6.8 for a UML diagram):

- There are two base classes, the image data abstract parameter container (`TImageDataBlock`) and the frame data abstract parameter container (`TFrameDataBlock`). These containers are T classes with just one data member of type `Uuid` and a protected constructor.
- There is a data container collection (`CFrameImageData`) class which can (despite its name) hold both of the base classes.
- The plug-ins which support their own encoding parameters have to create a class derived from one of these abstract data containers. The plug-in adds some data members and provides a public constructor so that it is possible to create an instance of that class.

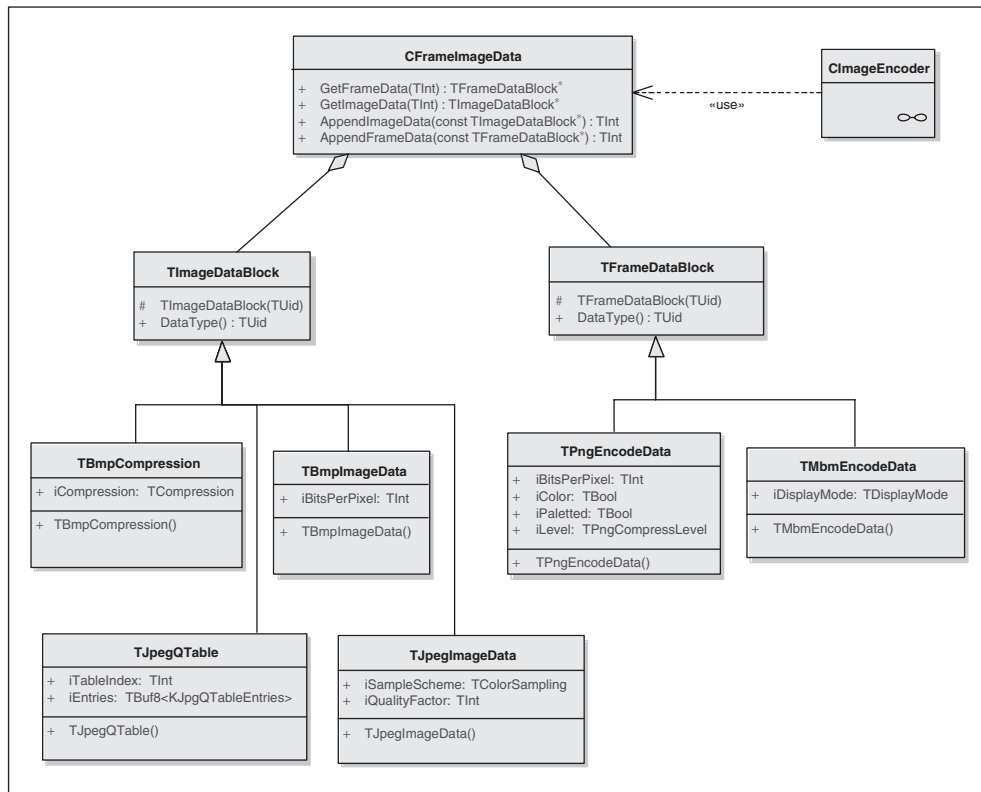


Figure 6.8 Setup parameters specific to a plug-in

Despite the fact that no multiframe encoding is currently supported, some plug-ins derive their parameters from the `TFrameDataBlock`; that is probably for future extensibility.

An API client wanting to customize encoding parameters should do the following:

1. Create an instance of the parameter collection holder (`CFrameImageData`).
2. Create and initialize instances of the desired parameters for the image type being encoded. For example, in the case of encoding to JPEG, these parameter classes could be `TJpegQTable` or `TJpegImageData`. These parameters can be created and added in any order and are optional unless their documentation says otherwise.
3. Pass an instance of the `CFrameImageData` as the third argument to `CImageEncoder::Convert()`.

If no encoding parameters are provided by the API client, an encoder plug-in applies defaults. These defaults can sometimes lead to undesired side effects. For example, the Symbian-provided BMP encoder always encodes images using the 24-bit per pixel mode unless it is told to use another color mode.

Now we can create a small piece of code that sets up a JPEG encoder to use an application-defined quality factor and sampling scheme, but lets it use default quantization tables:

```
void CSimpleEncodeHandler::BeginEncode(CFbsBitmap& aSourceBitmap)
{
    // create the encoding parameter collection
    iImageData = CFrameImageData::NewL();
    // create the Jpeg image encoding parameter on the heap
    TJpegImageData* jpegData = new (ELeave) TJpegImageData;
    CleanupDeletePushL(jpegData);
    jpegData->iSampleScheme = TJpegImageData::EColor444;
    jpegData->iQualityFactor = 90;
    User::LeaveIfError(iImageData->AppendImageData(jpegData));
    // remove the parameter pointer from the Cleanup stack
    // as the parameter collection owns it now
    CleanupStack::Pop( jpegData );
}
```

```
// start encoding process
iEncoder->Convert(&iStatus, aSourceBitmap, iImageData);
// Tell scheduler a request is active
SetActive();
}
```

There are some things to note about this code:

- Ownership of the `CFrameImageData` is not passed to the image encoder, so it is the API client's responsibility to delete it.
- Despite being T classes, all the parameters are allocated on the heap. That is because the `CFrameImageData` implementation tries to de-allocate memory pointed to by parameters as part of its destruction code.
- This code is a bit paranoid about encoder plug-in implementation and keeps the instance of `CFrameImageData` alive during the entire encoding process (we assume `iImageData` is a class member). Most plug-ins just read parameters during the synchronous part of the `Convert()` call, but it is better to be on the safe side.

Setting up encoding parameters is quite easy in cases where the application is always encoding to the same destination image format. It becomes a little bit tricky when the destination image format may be changed depending on some conditions as it would involve several `if` statements to create the encoding parameters.

Unfortunately, the encoder parameter setup implementation does not allow the application to discover the set of supported parameters for a given plug-in. One way of overcoming this limitation is to create as many parameters as needed and pass them in. At the moment, Symbian-supplied plug-ins ignore any unknown parameters and encode the image; however this behavior is not documented and may change. It also means that non-Symbian plug-ins may behave differently.

It is possible to ask an encoder plug-in to automatically generate a thumbnail for the image being encoded:

```
void CImageEncoder::SetThumbnail(TBool aDoGenerateThumbnail);
```

Not all image formats support thumbnails. What would happen if this method is used to enable thumbnail generation while a GIF image is being encoded, for example? Nothing, if the Symbian-supplied encoder plug-in is used. All Symbian-supplied encoder plug-ins ignore this call if thumbnail generation is not supported.

6.3.3 Encoder Plug-ins Provided by Symbian

Symbian supplies a number of `CImageEncoder` plug-ins in its reference SDK (see Table 6.7). Device manufacturers are free to ship these plug-ins with their smartphones, replace them with their own specific versions or remove them (if they do not need them). There is no 1:1 correspondence between decoder and encoder plug-ins, as it is not always feasible to create certain image types on a smartphone.

Table 6.7 Encoder Plug-ins

| Image Format Encoder UID | Notes |
|---|---|
| Windows bitmap (.bmp) <code>KBMPEncoderImplementation-UidValue</code> | The 1, 4, 8, and 24 bpp formats are supported. RLE-encoded formats are not supported. Device manufacturers usually use this plug-in without changes. |
| Graphics Interchange Format (.gif) <code>KGIFEncoderImplementation-UidValue</code> | Only the GIF87A format is supported. Neither mask encoding nor multiframe encoding is supported. Device manufacturers usually use this plug-in without changes. |
| JFIF/EXIF JPEG (.jpg) <code>KJPGEncoderImplementation-UidValue</code> | Color sampling can be monochrome, 420, 422 and 444. Valid quality settings are between 1 and 100. Automatic generation of thumbnails and Exif metadata is supported. Progressive and JPEG2000 formats are not supported. Only interleaved color components are supported. Many device manufacturers replace this plug-in with a hardware-accelerated version (the plug-in may be left but assigned a low priority). |
| Symbian OS MBM (.mbm) <code>KMBMEncodeData-UidValue</code> | Encoding to all valid Symbian display modes is supported. Device manufacturers usually use this plug-in without changes. |
| Portable Network Graphics (.png) <code>KPNGEncoderImplementation-UidValue</code> | The 1, 4, 8, and 24 bpp formats are supported. Three compression levels are available: no compression, best speed, best compression. Neither mask nor alpha channel encoding are supported. Device manufacturers usually use this plug-in without changes. |

6.4 Displaying Images

The `CImageDisplay` API can sometimes be seen as an analog of the `CImageDecoder` API; however it was designed to allow an application to easily display images, especially animated (multiframe) ones.

The `CImageDisplay` class⁸ is not a replacement for `CImageDecoder`, as it does not allow much access to the underlying image. It abstracts an application from the image details and, where it is possible to do so, from the plug-in capabilities as well.

This API shares many design elements with `CImageTransform` (see Section 6.5). However, it does not force an API client to use active objects when performing image decoding. This does not mean that the framework itself uses additional operating system threads to decode an image in the background. Instead, the framework makes use of active objects which operate in the client thread. Rather than passing a `TRequestStatus` into the framework, a client has to implement an observer interface through which it is notified when the framework has completed an operation.

The main advantages of using `CImageDisplay` for displaying images are:

- Scaling and rotation are provided as a part of core `CImageDisplay` functionality; these features are limited only by the capabilities of the bitmap manipulation APIs (see Section 6.6).
- All frame composition is done by the framework, so a client is provided with a ready-to-display bitmap for each animation frame.
- All the animation inter-frame delay processing is done by the framework, so a client is notified via the observer interface when a frame has to be displayed.
- All the animation frames are owned by the framework, so a client does not have to do resource management.

Using a `CImageDisplay` object to display animated images should save hundreds of lines of code, at the price of slightly reduced API flexibility (i.e. lack of access to low-level information about the image being decoded) and support for transparent use of threads. Unlike in the `CImageDecoder` API, there is no threading support and the plug-ins are always created in the client thread.

The framework implementation requires its users to adhere to a particular pattern of calls. Figure 6.9 shows a typical sequence of calls which an application should carry out in order to ‘play’ or display an image.

The framework can be used in the same way to display a single-frame image, which is treated as an animation with just one frame.

Unlike `CImageDecoder`, `CImageDisplay` does not combine the creation and plug-in loading phases. Instead, a plug-in is not attempted to be loaded until `CImageDisplay::SetupL()` has been called. A client has to issue several initialization calls and then call `SetupL()` to tell the

⁸This framework is not included in S60 3rd Edition SDKs.

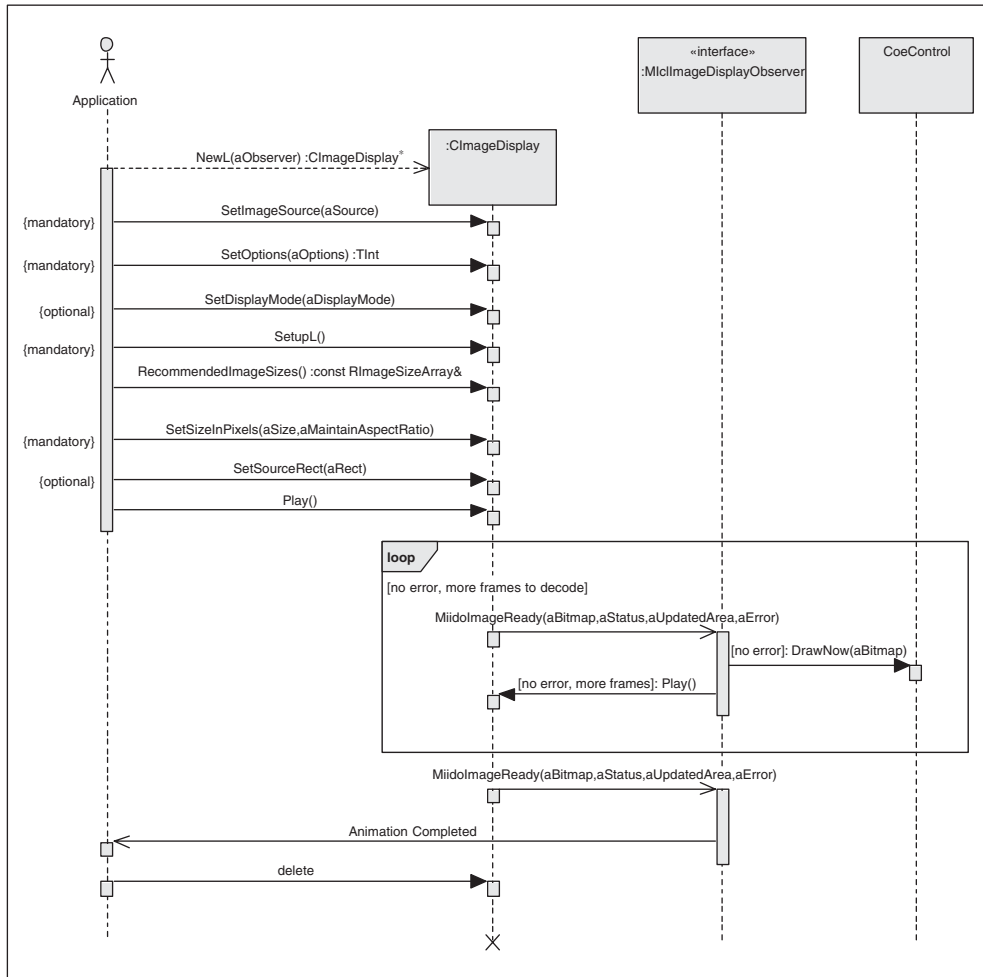


Figure 6.9 Typical sequence diagram for using CImageDisplay

framework that a plug-in which matches the given parameters should be loaded.

After successful completion of `SetupL()`, a client should perform some more animation setup based on the information about the image exposed by the plug-in. Once that setup is complete, the client has to call `CImageDisplay::Play()` to start animation decoding. Then the client has to yield back to the active scheduler (in the case of a UI application, that happens naturally once the current menu command has been handled).

The framework then begins calling the observer method (`MiidoImageReady`) for each animation frame when it is the right time to do so.


```

if (aError != KErrNone)
{
    // the framework encountered an error during playback, so handle it.
    // NOTE: you cannot "delete" the object from within this callback
}
if ((aStatus & CImageDisplayPlugin::EStatusFrameReady) ==
    CImageDisplayPlugin::EStatusFrameReady)
{
    const CFbsBitmap* frame = NULL;
    const CFbsBitmap* mask = NULL;
    iImgDisplay->GetBitmap(frame, mask);
    // display these bitmaps
}
if ((aStatus & CImageDisplayPlugin::EStatusNoMoreToDecode) ==
    CImageDisplayPlugin::EStatusNoMoreToDecode)
{
    // we reached the end of the animation
}
else
{
    // otherwise ask the framework to continue animation
    iImgDisplay->Play();
}
}

void CImageDisplayHandler::PlayImageL(const TDesC& aFileName)
{
    // re-use the framework object
    iImgDisplay->Reset();
    // Set the image source (to a file name, a handle or a memory buffer).
    // This is a mandatory call
    iImgDisplay->SetImageSource(TMMFileSource(aFileName));
    // Tell the framework to decode the main image (not a thumbnail)
    // and perform automatic rotation. Note that either EOptionMainImage or
    // EOptionThumbnail has to be defined otherwise the framework will panic
    iImgDisplay->SetOptions(CImageDisplay::EOptionMainImage |
        CImageDisplay::EOptionAutoRotate);
    // Tell the framework to load a plug-in
    iImgDisplay->SetupL();
    // Retrieve the original image size
    TSize originalSize(iImgDisplay->RecommendedImageSizes()[0]);
    // Set the size to which the image is to be decoded. It can be
    // virtually any size. This call must be made before Play()
    iImgDisplay->SetSizeInPixels(TSize(originalSize.iWidth*2,
        originalSize.iHeight*2));

    // start the animation
    iImgDisplay->Play();
}

```

6.5 Transforming Images

The image transformation API⁹ is designed to aid applications to transform images which are encoded in a format other than native bitmap format. Native bitmaps can be efficiently transformed using the bitmap

⁹This API is not included in the S60 3rd Edition SDKs.

transformation API (see Section 6.6). The image transformation API is an extendable, plug-in-based framework, which shares many design principles with `CImageDecoder`.

At the time of writing, `CImageTransform` was able to perform the following operations:

- image scaling
- thumbnail addition/removal
- Exif metadata editing.

`CImageTransform` can be thought of as a processing unit that accepts an image in a well-defined format and some setup parameters and produces an image in a well-defined image format as an output.

6.5.1 Basic Use of `CImageTransform`

The following code example illustrates use of the `CImageTransform` API.

```
void CImageTransformHandler::BeginTranformL(const TDesC& aFileName,
                                           CImageTransform::TOptions aTransformOpt,
                                           const TSize* aNewMainImageSize)
{
    // the framework allows object re-use, but we have to reset it first
    iImgTransform->Reset();
    // set the source file name
    iImgTransform->SetSourceFilenameL(aFileName);
    // Create a file name for the destination file. Using the input file
    // name may lead to undesired side effects and file corruption,
    // so generate another.
    TFileName destFileName(aFileName);
    destFileName.Append(_L(".transformed"));
    iImgTransform->SetDestFilenameL(destFileName);
    if (aNewMainImageSize == NULL) // we do not need to re-size
                                   // the main image
    {
        // tell the framework not to resize the main image
        iImgTransform->SetPreserveImageData(ETrue);
        // but we must still call this method with any size parameter
        iImgTransform->SetDestSizeInPixelsL(TSize(1,1));
    }
    else
    {
        // Allow the framework to resize the main image
        iImgTransform->SetPreserveImageData(EFalse);
        // set the new main image size
        iImgTransform->SetDestSizeInPixelsL(*aNewMainImageSize);
    }
}
```



```

// Pass the defined options. Note that it is not possible to specify the
// thumbnail size, if a thumbnail is to be added.
iImgTransform->SetOptionsL(aTransformOpt);
// Tell the framework that all the options are set now
iImgTransform->SetupL();
// Obtain extension
CImageTransformPluginExtension* ext = iImgTransform->Extension();
// If extension is present and it is of desired type
if (ext != NULL && ext->UId().iUId == KUidTransformJpegExtension)
{
    // It can be cast to the CJPEGExifTransformExtension, so
    // we can gain access to the MExifMetadata interface
    MExifMetadata* exifAccessor =
        static_cast<CJPEGExifTransformExtension*>(ext)->ExifMetadata();
    // Create an instance of the EXIF reader helper
    TExifReaderUtility exifReader(exifAccessor);
    // Create an instance of the EXIF writer helper
    TExifWriterUtility exifWriter(exifAccessor);
    HBufC8* imageDescription = NULL;
    // Check if the image has a description tag
    if (KErrNotFound == exifReader.GetImageDescription(imageDescription))
    {
        // Create and set our own image description
        _LIT8(KImageDescription, "Image Transformed!");
        imageDescription = KImageDescription().AllocLC();
        User::LeaveIfError(exifWriter.SetImageDescription(
            imageDescription));
        CleanupStack::PopAndDestroy(imageDescription);
    }
    else
    {
        // description is already there, so deallocate it.
        delete imageDescription;
    }
}
// tell the framework to start the image transformation
iImgTransform->Transform(iStatus);
// tell the active scheduler there is an outstanding request
SetActive();
}

```

The parameters passed to the `BeginTransformL()` method are:

- `const TDesC& aFileName` is an input file name, no more than that.
- `CImageTransform::TOptions aTransformOpt` can be any combination of supported transformation options, used in the same way as `CImageTransform::SetOptions()`.
- `const TSize* aNewMainImageSize` is the (optional) destination image size. A caller may specify `NULL`, in which case no image re-sizing occurs.

The `BeginTransformL()` method can do almost anything that the API defines. The basic scenario of `CImageTransform` usage is as follows:

1. Create an instance of `CImageTransform`.
2. Perform mandatory setup: image source, destination and the destination image size. Note that the destination size setup call is mandatory and must always be made even if the image is not going to be re-sized, in which case any size can be specified.
3. Call `CImageTransform::SetupL()`. At this point the framework tries to load a plug-in. If no suitable plug-in is found, a leave occurs.
4. Perform additional optional setup via non-standard extensions. Unlike with the `CImageDecoder` API, these extensions are interface-based rather than data-based (see Figure 6.11 for an example of an extension).
5. Once all the setup is done, start the asynchronous transformation process by calling `CImageTransform::Transform()`.
6. Handle the transformation result.
7. If more images need to be transformed, re-use the framework object by calling `CImageTransform::Reset()`.

The framework enforces the call sequence by panicking applications which do not adhere to the required sequence of calls.

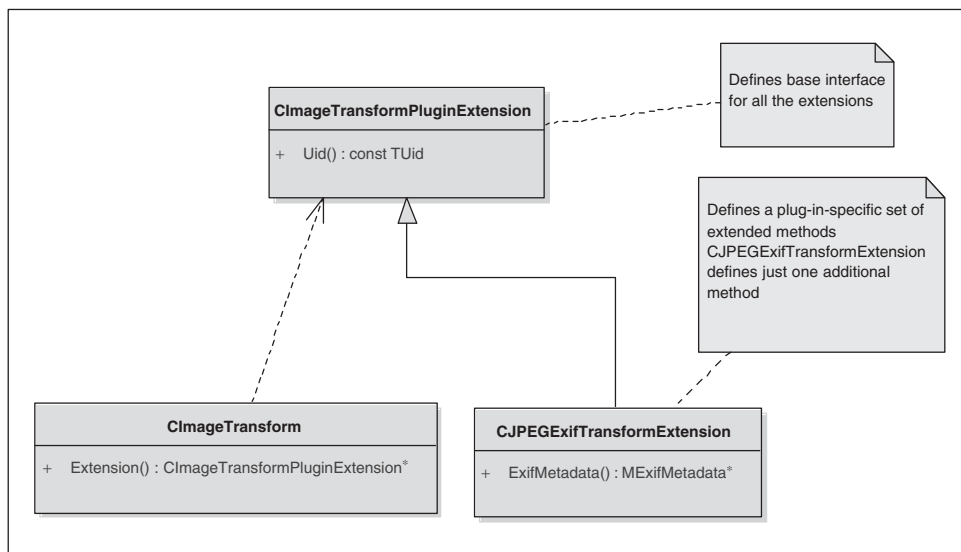


Figure 6.11 `CImageTransform` extension class diagram

`CImageTransform::TOptions` is a mix of framework operation instructions and output image options. If the input image has a thumbnail but `CImageTransform::EThumbnail` is not given to `CImageTransform::SetOptionsL()`, then the thumbnail is removed and the output image does not have a thumbnail.

6.5.2 Limitations of `CImageTransform`

At the time of writing, the only plug-in supplied by Symbian is a JPEG Exif-enabled plug-in. No other plug-ins are available from phone manufacturers, which means that use of `CImageTransform` is limited to JPEG format image re-sizing, thumbnail addition or removal and Exif metadata addition or editing.

A limitation of the existing JPEG plug-in implementation is that it is not possible to fine-tune JPEG compression parameters (quality factor, quantization tables, etc.) in the same way as is possible when encoding a bitmap using the `CImageEncoder` API.

6.6 Native Bitmap Operations

The main purpose of this API is to provide support for the two most common bitmap transformations: scaling and rotation. It is fairly straightforward to use this API, which is based on plug-ins. Symbian supplies default software-based plug-in implementations, which can be replaced by a device manufacturer with hardware-accelerated implementations.

These APIs can be very helpful for applications that view images, in cases where an image decoder cannot satisfy an application's requested output image size (see Section 6.2.4) or bitmap rotation is needed.

The API consists of two classes: `CBitmapScaler` and `CBitmapRotator`, which can perform scaling and rotation (in multiples of 90 degrees), respectively. These classes share the same basic usage pattern:

1. An instance of the class is created using the synchronous factory method, `NewL()`.
2. Optional synchronous setup methods (only for `CBitmapScaler` at the moment) are called.
3. The asynchronous transformation operation is invoked.

The Symbian-supplied implementations allow object instance re-use once the asynchronous operation has been completed: it is possible to use the same instance of a scaler to perform scaling of several bitmaps (but not simultaneously).

Both classes support two modes of operation – a bitmap can be transformed either ‘in-place’ or using a separate bitmap as the destination. The latter case is interesting as it allows the bitmap color mode to be converted as well. For example, it is possible to ask a bitmap of color mode EColor64K to be scaled or rotated and the result converted into the destination bitmap color mode.

6.6.1 Scaling Bitmaps

The following code demonstrates the basic use of the scaling API. It makes use of the single-bitmap variant of the API, specifies the destination size and asks the scaler to preserve the aspect ratio of the image:

```
void CSimpleBitmapScalerHandler::ConstructL()
{
    iScaler = CBitmapScaler::NewL();
    CActiveScheduler::Add(this); // Add to scheduler
}

CSimpleBitmapScalerHandler::~CSimpleBitmapScalerHandler()
{
    // Cancel any request, if outstanding
    Cancel();
    // Delete instance variables, if any
    delete iScaler;
}

void CSimpleBitmapScalerHandler::BeginScalingL(CFbsBitmap& aSource,
                                               const TSize& aDestSize,
                                               CBitmapScaler::TQualityAlgorithm aQuality)
{
    // We are not checking the error code of this operation as
    // there is nothing we can do about it
    iScaler->DisablePostProcessing(ETrue);
    User::LeaveIfError( iScaler->SetQualityAlgorithm(aQuality) );
    // start scaling with aspect ratio preservation
    iScaler->Scale(&iStatus, aSource, aDestSize, ETrue);
    // Tell scheduler a request is active
    SetActive();
}

void CSimpleBitmapScalerHandler::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        // do something with scaled bitmap
    }
    else
    {
        //handle error
    }
}
```

The benefits of the single-bitmap scaling method are as follows:

- The application does not have to create a destination bitmap and discard the source (if it is no longer needed). You should use

the two-bitmap API variant if you need to preserve the original bitmap.

- The application does not have to bother about the exact destination size. Scaling with aspect ratio preservation is probably the most common use case, but it is quite tricky. Let us imagine that an application needs to enlarge an image of size (20, 30) so it fits into a rectangle of size (43, 62). What is the output size in this case? It depends on the scaling algorithm implementation. We are dealing with a plug-in-based API which can have slightly different implementation-specific behavior. Where we're dealing with a single-bitmap API method, we let the plug-in do the hard work.
- This method should consume less memory than the two-bitmap one. Depending on the use case (either enlarging or downscaling the image), however, memory consumption can be the same as the two-bitmap API.

The benefits of the two-bitmap variant are as follows:

- An API client can potentially combine scaling with color depth transformation¹⁰ as the input and output bitmaps may be of different color modes.
- The original image is preserved.

If the two-bitmap scaling method is used with the aspect ratio preservation parameter set to 'true', then the default scaler implementation does its best to fit the result into the destination bitmap provided. In some cases, this may cause 'borders' to appear on the final image, if the application uses a destination size calculation logic which is different to that employed by the scaling plug-in. The best way of avoiding this problem is to use a single-bitmap scaling method.

The `CBitmapScaler` class has some optional setup methods which can be used to tune scaling behavior (see Table 6.8).

6.6.2 Performance Hints for Scaling

Performance of the default software implementation of the scaling plug-in can be quite sensitive to the input parameters. The two main factors which affect scaling performance are:

- source and destination bitmap color modes
- scaling algorithm quality.

¹⁰Color depth transformation can be quite costly in terms of both memory and CPU power consumption (especially when color dithering is enabled).

Table 6.8 Tuning Options for Scaling

| Method | Description |
|--------------------------------------|--|
| <code>UseLowMemoryAlgorithm()</code> | Instructs the scaler to use as little memory as possible. With the Symbian-provided reference implementation, you can gain some performance. It may be worth always using this option. |
| <code>DisablePostProcessing()</code> | Disables color dithering during color depth adjustments. Disabling color dithering, in most cases, provides a significant performance increase. |
| <code>SetQualityAlgorithm()</code> | <p>Allows changing of the scaling quality. As usual, the quality of image processing comes at the price of performance and memory consumption. The default scaler implementation supports the following quality levels:</p> <ul style="list-style-type: none"> • <code>EMinimumQuality</code> causes the scaler to use the ‘nearest neighbor’ algorithm. • <code>EMediumQuality</code> causes a simple weight-based interpolation to be applied in the horizontal dimension. • <code>EMaximumQuality</code> causes interpolation to be used for both vertical and horizontal dimensions. <p>If no quality setting method is called, then <code>EMaximumQuality</code> is used by default. Note: Plug-ins provided by Symbian OS interpret the parameters in this way, but other plug-ins may behave differently.</p> |

The first thing to consider while looking at performance is to make the application use the same color mode for both the source and destination bitmaps, thus avoiding pixel translations.

The scaler provided by Symbian OS is optimized to perform fairly fast pixel operations for bitmaps in which the color mode has eight bits per color channel. At the time of writing, these are: `EGray256`, `EColor16M`, `EColor16MU` and `EColor16MA`. Scaling bitmaps of other color modes involves color re-translation.

6.6.3 Rotating Bitmaps

The bitmap rotation API is quite similar to the scaling one as it has the same design and usage patterns.

It is a bit easier to use as it hasn’t got any additional setup parameters. It can do bitmap rotation in multiples of 90 degrees:

```

void CSimpleBitmapRotationHandler::ConstructL()
{
    iRotator = CBitmapRotator::NewL();
    CActiveScheduler::Add(this); // Add to scheduler
}

CSimpleBitmapRotationHandler::~CSimpleBitmapRotationHandler()
{
    // Cancel any request, if outstanding
    Cancel();
    // Delete instance variables, if any
    delete iRotator;
}

void CSimpleBitmapRotationHandler::BeginRotation(CFbsBitmap& aSource,
                                                CBitmapRotator::TRotationAngle aAngle)
{
    // begin rotation
    iRotator->Rotate(&iStatus, aSource, aAngle);
    // Tell scheduler a request is active
    SetActive();
}

void CSimpleBitmapRotationHandler::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        // Do something with rotated bitmap
    }
    else
    {
        // Handle error
    }
}

```

The main input parameters to the bitmap rotation API are:

- Rotation angle in multiples of 90 degrees. `CBitmapRotator::TRotationAngle` defines the set of rotation options supported.
- `CBitmapRotator::Rotate()` has two overloads: one using a single bitmap as both destination and source and one using two bitmaps, which allows the application to keep the source bitmap intact. The single-bitmap option does not provide any memory saving, as the default implementation creates a destination bitmap internally and then swaps bitmap handles.

6.7 Miscellaneous APIs

There are several helper classes which can be used by an application to perform pixel-level operations on native bitmaps (see Figure 6.12). These classes were designed as re-usable components for decoder plug-ins (as they do a fair amount of pixel operations), however since they are published APIs, they can be used by third-party applications as well.

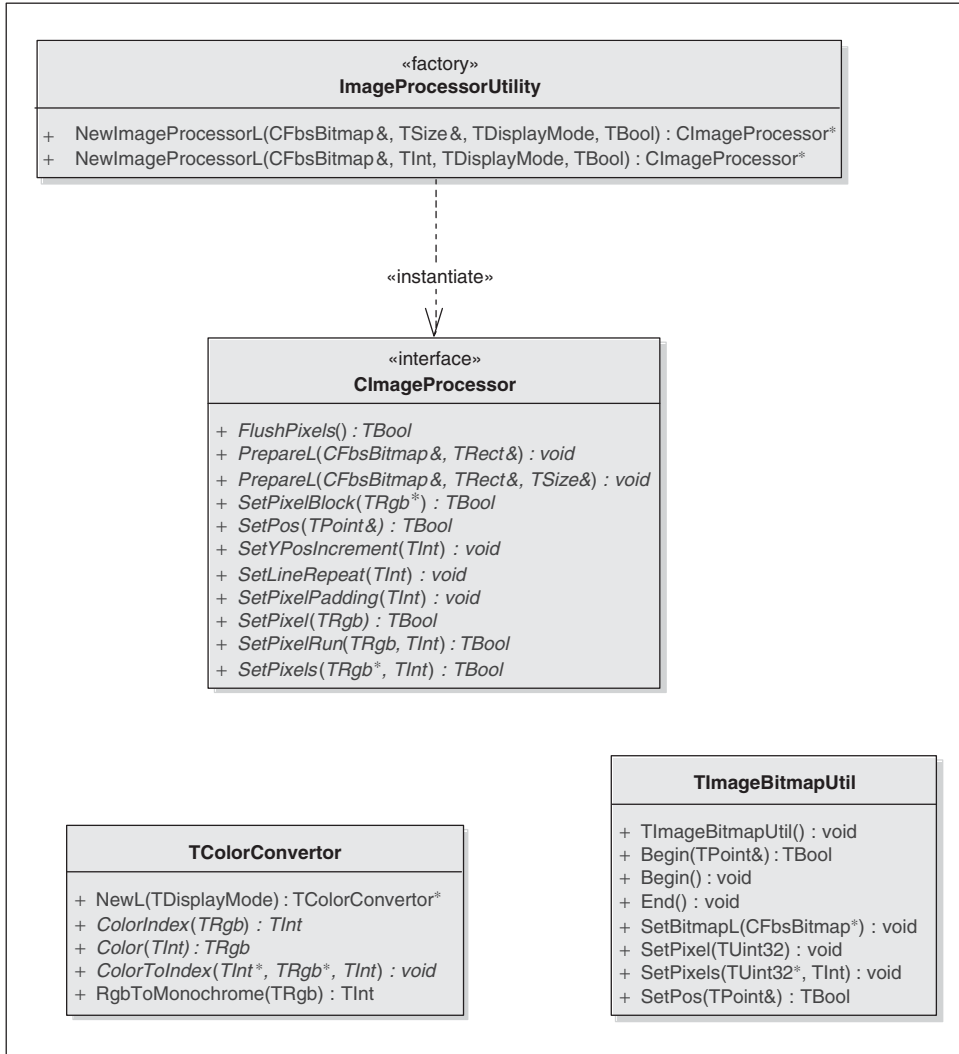


Figure 6.12 ICL pixel-processing APIs

The CImageProcessor API is designed to serve as a universal pixel data sink for an application which writes individual pixels directly, with added support for on-the-fly scaling and color conversion. The ImageProcessorUtility is a factory-like static class which can create implementations of CImageProcessor which correspond to the destination bitmap type and input pixel data display mode. The CImageProcessor class supports on-the-fly pixel format conversion to the destination bitmap color mode along with on-the-fly downscaling of incoming pixel data.

TImageBitmapUtil is a simple and lightweight utility class that allows applications to write pixels and pixel blocks into a Symbian OS bitmap.

The TColorConvertor utility class is an interface-based utility class which can be used as a universal color converter routine between a TRgb-based color representation and a color index value within a given color mode.

All these classes may be used by an application wanting to apply special effects to individual pixels of a native Symbian OS bitmap. In fact, the CImageProcessor implementations use TImageBitmapUtil and TColorConvertor to perform pixel processing.

The following code uses CImageProcessor to implement a simple effect on a native bitmap. It reduces the size by half and reduces the brightness to a quarter:

```
CFbsBitmap* ImageProcessorUseL(CFbsBitmap& aSrcBitmap)
{
    const TInt KReductionFactor = 1; // Means 1/2 of the original size
    // Create the destination bitmap object
    CFbsBitmap* destBitmap = new (ELeave) CFbsBitmap();
    CleanupStack::PushL(destBitmap);
    // Calculate the destination size using the defined reduction factor
    const TSize srcSize( aSrcBitmap.SizeInPixels() );
    const TSize destSize(srcSize.iWidth >> KReductionFactor,
                        srcSize.iHeight >> KReductionFactor);
    // Create the destination bitmap with the same color mode as source
    User::LeaveIfError(destBitmap->Create(destSize,
                                        aSrcBitmap.DisplayMode()));
    // Create an instance of CImageProcessor with the given reduction
    // We are going to use EColor16M and do not want to use color dithering.
    CImageProcessor* processor;
    processor = ImageProcessorUtility::NewImageProcessorL(aSrcBitmap,
                                                         KReductionFactor, ERgb, ETrue);
    CleanupStack::PushL(processor);
    // Create an image processor for the destination bitmap and use
    // the entire destination bitmap for placing the incoming pixel data.
    // NOTE: it is not a typo - PrepareL() expects unscaled coordinates
    processor->PrepareL(*destBitmap, TRect(srcSize) );
    // Brightness reduction factor
    const TInt KDimOutFactor = 2;
    // Iterate through all the source pixels
    for (TInt yPos = 0; yPos < srcSize.iHeight; yPos++)
    {
        for (TInt xPos = 0; xPos < srcSize.iWidth; xPos++)
        {
            TRgb pixel;
            aSrcBitmap.GetPixel(pixel, TPoint(xPos, yPos));
            TRgb newPixel;
            newPixel.SetRed( pixel.Red() >> KDimOutFactor );
            newPixel.SetGreen( pixel.Green() >> KDimOutFactor );
            newPixel.SetBlue( pixel.Blue() >> KDimOutFactor );
            processor->SetPixel( newPixel );
        }
    }
}
```

```
    }  
    // Write all the buffered pixels into the destination bitmap  
    processor->FlushPixels();  
    CleanupStack::PopAndDestroy(processor);  
    CleanupStack::Pop(destBitmap);  
    return destBitmap;  
}
```

One thing to note is that this `CImageProcessor` example is not an example of high-performance code. Rather, we have made it as readable as possible. Well-designed code should never call a virtual function per pixel image, but instead should deal with pixel buffer methods. Also, the multiple calls to `TRgb::SetRed()` and others should be replaced with one single method. However, high-performance code is usually not very readable.

7

The Tuner API

This chapter introduces the Tuner API which provides access to an FM/AM radio if radio hardware is present on the phone. It shows how to get started with the API and covers some common use cases with examples. After reading this chapter, you should be able to write a simple radio application and know where to find the SDK documentation for more advanced use cases.

It also discusses related technologies such as digital radio (DAB) and mobile television and how they may be supported by Symbian OS in the future.

7.1 Introduction

Since Symbian OS v9.1, there has been an API available for controlling a radio tuner on Symbian smartphones. However, it is important to note, that the Tuner API is currently only publicly available in the UIQ 3 SDKs. To date, the S60 3rd Edition SDK releases have excluded this component and do not provide an alternative. This means that you cannot write third-party radio tuner applications for S60 phones.

The first smartphone to use the Tuner API was the Sony Ericsson P990 which featured an FM radio with RDS¹ support. All smartphones based on UIQ 3.0 (which is built on Symbian OS v9.1) or later run applications built against this API. If the device does not have radio hardware (for example, the Sony Ericsson M600i), the API reports an error at initialization time, which the application should handle gracefully.

The Tuner API is separate from the other Symbian OS multimedia components. The underlying licensee implementations may make use of DevSound and MMF for audio output and recording but this is transparent

¹Radio Data System (RDS) is a standard for embedding information, such as the station name, in FM broadcasts.

to the API user. No prior knowledge of the other Symbian OS multimedia components is required in order to use the Tuner API.

Let's take a look at what the Tuner API enables you to do at a high level. Section 7.3 gives further detail and covers basic use cases. For more specific details, you may also find it helpful to refer to the API documentation in the Symbian Developer Library documentation in your SDK or online at ***developer.symbian.com***. The Symbian OS Reference has a section on the tuner which should be useful when reading the rest of this chapter. A shortcut to finding the documentation is simply to search the index for 'CMMTunerUtility'.

The Tuner API allows you to:

- query the capabilities of available tuners on the device (what frequency bands they support, whether they support RDS, or can record, etc.)
- tune into a given frequency
- scan automatically for the next frequency with a strong signal
- play the received audio
- record the received audio
- use RDS features (only available on compatible FM tuners):
 - get metadata such as the name of the radio station and current program
 - find alternative frequencies for the current radio station
 - automatically tune into news and traffic updates on other frequencies
 - get the current time.

7.2 Getting Started

The first thing you need to do when using the Tuner API is link to `tuner.lib` by adding it to your application's MMP file:

```
LIBRARY tuner.lib
```

You also need to include `tuner.h` in any source or header files that use the Tuner API:²

```
#include <tuner/tuner.h>
```

²This assumes that you already have `\epoc32\include` as a `SYSTEMINCLUDE` path in your MMP file.

7.2.1 Initialization

The Tuner API is structured around a central class, `CMMTunerUtility`, that takes care of initialization, de-initialization, basic tuner control, and access to a number of utility classes for more specialized functions:

- `CMMTunerAudioPlayerUtility` allows you to play the audio being received.
- `CMMTunerAudioRecorderUtility` allows you to record the audio being received.
- `CMMTunerScannerUtility` can automatically scan through all available frequencies pausing at each station it finds.
- `CMMRdsTunerUtility` allows you to access the RDS functionality in FM broadcasts.

These utility classes are covered in more detail in Section 7.3.

In order to get started, you need to check how many tuners (if any) are available using the static method `CMMTunerUtility::TunersAvailable()`. Then you probably want to query the capabilities of each using the `CMMTunerUtility::GetCapabilities()` method and choose which one to use based on the results returned. The simple example below illustrates this approach; it picks the first available FM tuner with RDS support:

```
TInt availableTuners = CMMTunerUtility::TunersAvailable();
// This will hold the index of the desired tuner
TInt desiredTuner = KErrNotFound;
// This is populated by GetCapabilities()
TTunerCapabilities tunerCaps;
for( TInt i = 0; i < availableTuners; ++i )
{
    // skip the tuner if this fails for some reason
    if (CMMTunerUtility::GetCapabilities(i, tunerCaps) != KErrNone)
    {
        continue;
    }

    if ((tunerCaps.iTunerBands & ETunerBandFm) &&
        (tunerCaps.iAdditionalFunctions & ETunerFunctionRds) )
    {
        // found a tuner with FM and RDS
        desiredTuner = i;
        break;
    }
}
// End of for loop

if (desiredTuner == KErrNotFound )
{ // No suitable tuners on this device
    User::Leave( KErrNotFound );
}
```

To create a `CMMTunerUtility`, you need to supply an observer class that implements the `MMMTunerObserver` interface. You should implement the observer code, which requires two methods to be defined: one to deal with the completion of asynchronous tuning operations and another to handle tuner control events and errors.

```
iMyTunerUtility = CMMTunerUtility::NewL(*iMyTunerObserver, desiredTuner);
```

Once the call to `NewL()` returns, the tuner utility has been created and initialized and is ready to be used.

7.2.2 Control and Priorities

It is possible for more than one instance of the Tuner API to be present at the same time. For instance there could be multiple applications running simultaneously, each of which use the API. This could potentially lead to problems such as two separate applications trying to tune to different frequencies at the same time. To prevent this, the API has been designed so that only one instance can be in control of the tuning functions at any given time. A tuner instance automatically attempts to gain control the first time one of the tuning functions is used (e.g. `Tune()`). It may also request control explicitly by calling `RequestTunerControl()`.

When an instance of `CMMTunerUtility` attempts to use the tuning functions while another instance has control, it is notified that the attempt failed via a callback to the `MMMTunerObserver::MToTunerEvent()` with a `KErrPermissionDenied` error. Use of the tuning functions will not succeed until `Close()` is called on the other tuner instance or it releases its control by calling `ReleaseTunerControl()`.

Each tuner instance is assigned an access priority when it is initialized (there is an optional access priority parameter for `NewL()` that can be used to request a higher or lower access priority than the default).³ Tuner instances with higher priorities can pre-empt those with a lower priority and take over control. In this case, the instance with the lower priority is notified that it has been displaced via the tuner observer method. For this reason, applications that use the tuner should be written to handle pre-emption by a higher-priority client at any time. Depending on the use case, it may be desirable to notify the user that this has occurred, to wait and try to regain control of the tuner automatically, or to do nothing until the user interacts with the application again.

³Priorities higher than `ETunerAccessPriorityNormal` require the `MultimediaAdd` platform security capability, which is restricted and requires that your application is Symbian Signed using the Certified Signed route. These priorities are intended for things such as a built-in radio-alarm function on a phone, which must be able to interrupt any other tuner applications in order to sound the alarm.

If your tuner instance fails to get control or is pre-empted by another, you can use `NotifyTunerControl()` to request a notification callback via the tuner observer when control becomes available again.

7.2.3 Cleanup

When your application has finished using the tuner API, it must delete its instance of `CMMTunerUtility`. This does all the necessary cleanup behind the scenes and, if no other applications are using the tuner, powers down the tuner hardware. However, if you use any of the other utility classes (e.g. `CMMTunerScannerUtility`) then these must be deleted before deleting the main `CMMTunerUtility`.

There is also a `Close()` method available on `CMMTunerUtility` which does the same de-initialization except that your instance of `CMMTunerUtility` is not deleted. This may be useful if your tuner utility is a member variable which is deleted by the owning class's destructor but is known not to be needed earlier. It is not necessary to delete associated utility classes before calling `Close()`; any ongoing activity is stopped automatically.

7.3 Basic Use Cases

7.3.1 Tuning

Tuning to a particular frequency is done via the `Tune()` method, which is an asynchronous command. Success or failure of a tuning operation is reported via the `MToTuneComplete()` method on the tuner observer. As mentioned in Section 7.2.2, calling `Tune()` automatically requests control of the tuner, if necessary. The method takes two parameters:

- the frequency (given in Hertz, so 95.8 MHz would be 95,800,000 Hz for example)
- the frequency band (FM, AM, LW, etc.).⁴

You can also query which frequency the tuner is currently set to at any time by using the `GetFrequency()` method.

7.3.2 Playing the Radio

Once you have tuned to a frequency, you probably want to play the sound. This is handled by the `CMMTunerAudioPlayerUtility` class.

⁴There is a version of `Tune()` that takes digital channel numbers instead of frequencies. It was put in by Symbian for future-proofing and is intended for systems such as DAB. However, at the time of going to press, there are currently no UIQ3 handsets that support this (see Section 7.4.1).

There can only ever be one instance of this class per instance of the `CMMTunerUtility` and it is retrieved using the `GetTunerAudioPlayerUtilityL()` method. This method expects an observer that implements the `MMMTunerAudioPlayerObserver` interface, which handles callbacks from the tuner audio player (more on these in a moment).

Before playback can commence, you must initialize the tuner audio player by calling its `InitializeL()` method. Success or failure of the initialization is signaled via the observer's `MTapoInitializeComplete()` method.

There are two optional parameters that set the sound device priority and priority preference. These are distinct from the tuner utility's priority setting. They behave just like the sound device priority settings for the MMF's audio player utilities, i.e. they are device-specific and may be ignored unless your application has the `MultimediaADD` capability. In most cases, omitting these values and using the defaults should be just fine! However, if you wish to learn more about sound device priorities then please refer to Chapter 5.

Once your `CMMTunerAudioPlayerUtility` has been initialized, you can start playback by calling `Play()`. To temporarily stop playback, use the `Mute()` method because the `Stop()` method releases the audio device and playback cannot be restarted afterwards. Only use `Stop()` when you don't need the tuner audio player utility any more, such as when shutting down your application.

The following example demonstrates these steps:

```
// get tuner audio player utility
iMyTunerAudioPlayer = iMyTunerUtility->
    GetTunerAudioPlayerUtilityL(*iMyTunerAudioPlayerObserver);
// initialize it
iMyTunerAudioPlayer->InitializeL();
// now we can start playing
iMyTunerAudioPlayer->Play();
// ...
// when we are done...
// stop
iMyTunerAudioPlayer->Stop();
// clean up
delete iMyTunerAudioPlayer;
iMyTunerAudioPlayer = NULL;
```

7.3.3 Recording from the Radio

The `CMMTunerAudioRecorderUtility` allows you to record audio from the radio to a file or descriptor. It behaves in much the same way as the tuner audio player utility. One of the differences is that `InitializeL()` has additional parameters to specify the recording destination

and format. Please check the Symbian Developer Library documentation reference guide for `CMMTunerUtility::GetTunerAudioRecorderUtility()` and class `CMMTunerAudioRecorderUtility` for more information.

7.3.4 Scanning for Frequencies

A useful feature is to scan frequencies automatically for strong signals. This can be achieved by using the tuner utility's `StationSeek()` method. It takes a start frequency, a tuner band, and a search direction (up or down); it searches until it finds the next strong signal and tunes into it. At this point, you receive a callback on the tuner observer's `MToTunerEvent()` method with an event type of `ETunerEvent`. If you want to know what frequency was found, you can retrieve it via the tuner utility's `GetFrequency()` method.

If you are listening to the radio while it is seeking stations, you may hear noise in between the radio stations. To avoid this you can enable 'squelching' (muting in frequencies without reception) using the `SetSquelch()` method.

7.3.5 Automated Scanning using the Scanner Utility

A common use case is to find all available radio stations in the area. To simplify this you can use the scanner utility class, `CMMTunerScannerUtility`, which is a convenience wrapper that scans through all available frequencies pausing at each discovered station for a specified amount of time.

To get the scanner utility you simply use the tuner utility's `GetTunerScannerUtilityL()` method. It requires an observer instance that implements the `MMM tunerObserver` interface. You can re-use the same instance that you have for the main `CMMTunerUtility` class or you can create a separate one. The scanner utility is then controlled via two main methods: `StationScan()` starts the station scan and `StopScan()` stops scanning and uses the currently tuned station.

7.3.6 Radio Data System

Radio Data System (RDS) allows broadcasters to embed some metadata and text information into FM radio stations. This allows radios to display the station name, music genre and other useful information. Where the hardware supports RDS (which can be queried when you probe a tuner's capabilities, see Section 7.2.1), the Tuner API lets you access the RDS data via the `CMMRdsTunerUtility` class.

Fetching into TRdsData

As with the other utilities, there is a method in the main tuner utility class, `GetRdsTunerUtilityL()`, that creates an instance of `CMM-RdsTunerUtility` for you. Similarly to the scanner utility, it expects a tuner observer as one of its parameters. You can re-use your existing observer if you wish.

Not every radio station broadcasts RDS data, so before attempting to use any RDS features it is recommended that you call the `IsRdsSignal()` method to query whether RDS data is present. If it is, you can retrieve information about the currently tuned station by using the `GetRdsData()` method, which reads the station's data into a `TRdsData` object. By default, all available data is fetched but you can limit it to a specific subset by using bitwise OR with values from `TRdsData::TField` that represent the information you need and passing this into `GetRdsData()` as the `aWhichData` parameter. Alternatively, you can request all of the data and extract the items you want separately afterwards. For instance, to retrieve the name of the currently tuned station, you could do the following:

```
// get the RDS utility
iMyRdsUtil = iMyTunerUtility->GetRdsTunerUtilityL( *iMyTunerObserver );
// make sure current station is broadcasting RDS data
if (iMyRdsUtil->IsRdsSignal() )
{
    // create RDS data object to hold info
    TRdsData stationData();

    // fetch station data
    TUInt32 whatInfoIsValid = 0;
    iMyRdsUtil->GetRdsData( stationData, whatInfoIsValid );
    // check station name is present
    if (whatInfoIsValid & TRdsData::EStationName )
    {
        // extract station name
        TRdsStationName stationName = stationData.iPs;
        // ... do something with the name
    }
}
```

Other RDS Data

There are a number of additional RDS features besides what is covered in `TRdsData`. The RDS utility provides a number of `Get()` methods to access this information. Since the data may change throughout the broadcast there are also a number of `Notify()` methods that enable you to register for notifications when the data changes. Notifications can be canceled using the corresponding `Cancel()` methods in the RDS utility. Although all of these follow the same general pattern each has a

specialized observer interface to handle its notifications. Please refer to the developer library's API documentation on `CMMRdsTunerUtility` for the details.

Different phones may or may not support all of these advanced RDS features so it makes sense to check which are available on your device using the RDS utility's `GetRdsCapabilities()` method. For example, to see if the `RadioText` feature is supported and register for notifications, you could do something like the following:

```
// get the RDS utility
iMyRdsUtil =
    iMyTunerUtility->GetRdsTunerUtility( *iMyTunerObserver );
// this will contain the capabilities
TRdsCapabilities rdsCaps;
iMyTunerUtility->GetRdsCapabilities( rdsCaps );
if ( rdsCaps.iRdsFunctions & ERdsFunctionRt )
{
    // Radio Text is supported!
    // register for notifications ...
    iMyTunerUtility->NotifyRadioText( *iRdsDataObserver );
}
```

Advanced Features

Beyond fetching information on the currently tuned radio station there are a few extra features in the RDS utility. They are explained in the API documentation, so we just outline them here:

- **Searching for stations:** You can scan for the next station which has a specific RDS property (such as program type). This behaves similarly to `StationSeek()`, discussed in Section 7.3.4. Look up the RDS utility's `StationSearchByXxx()` methods for more information.
- **Traffic and news announcements:** If supported by the broadcaster, the RDS utility can automatically re-tune to another station that has traffic or news announcements when they come in and then tune back once the announcement has ended. Look up the RDS utility's `SetNewsAnnouncement()` and `SetTrafficAnnouncement()` methods for more information.
- **Regional link:** Some national radio stations broadcast on different frequencies in different regions. If they support the regional link feature, the RDS utility can automatically switch to the station's alternative frequency if the signal on the currently tuned one is getting weak. This allows the end user to continue listening to the same station with no interruption when traveling between regions. Look up the RDS utility's `SetRegionalLink()` method for more information on this.

7.4 Future Technologies

7.4.1 Digital Audio Broadcasting

Digital Audio Broadcasting (DAB) is a standard for broadcasting digital radio. It is currently in use in a number of countries. Looking at the API documentation for the tuner utility, you may notice a few references to DAB; for example, the tuning functions have variants that take channel numbers instead of frequencies. However, there is no support in the current API for DAB's more advanced features, such as DLS radio text, so extensions would be necessary if full support were required.

So far there have been no Symbian smartphones with DAB support. However, other technologies such as streaming Internet radio and mobile TV can offer a similar experience to DAB and are more fully supported in Symbian OS. Mobile TV is briefly covered in Section 7.4.2 and Internet streaming is handled by the MMF APIs.

7.4.2 Mobile TV

There are now a number of standards for broadcasting digital television services to mobile devices, such as DVB-H, DMB and ISDB-T 1seg. Note that these are completely separate from streaming audio or video over an Internet connection. Streaming is usually a one-to-one connection between a server and a client application,⁵ so each new connection takes up a piece of the available bandwidth. Therefore there is a finite limit to how many streaming users there can be in a cellular network at any given time. Broadcasting, on the other hand, is a one-to-many connection where a tower broadcasts a signal that can be picked up by any number of receivers.

Although the various mobile TV standards are technically quite different from one another, they generally deliver the same kind of services to the user: a number of digital TV and audio channels, an electronic service guide (ESG) that lists channel and program details, support for subscription and pay-per-view channels and data services (teletext, filecasting, etc.). Since the functionality they offer is much richer and more complex than analog radio, the tuner API is not suitable for controlling them.

There is some support for mobile TV in Symbian OS, but it is in the form of a low-level hardware abstraction layer (HAL) that standardizes the interface between drivers for DVB-H receiver hardware and middleware products that take care of processing the received data in order to extract ESG data and TV services. The user-facing applications that allow you to watch TV are generally developed by the licensees and their partners

⁵Strictly speaking this is *unicast* streaming. Multicast streaming allows a single stream of data to go to any number of recipients on the network. Some mobile TV standards, such as MBMS, use multicasting over cellular networks to deliver TV services.

and talk directly to whatever middleware is present in the device. There is no standard Symbian API at this level for third-party application developers.

7.5 Sample Code

Symbian OS C++ for Mobile Phones, Volume 3 by Richard Harrison and Mark Shackman (2007) provides sample code for the tuner API. The code is available from developer.symbian.com/main/documentation/books/books_files/scmp_v3/index.jsp (use the source code link near the bottom right corner of the page). The example is in the \multimedia\tuner directory after you have installed the code package. It has been tested on the Sony Ericsson P1. The tuner implementation is identical on other Sony Ericsson smartphones based on Symbian OS v9.1.

8

Best Practice

This chapter is a collection of guidelines that apply to using the various multimedia APIs. Some may be more obvious than others, but all should save you some development time and frustration. All the usual Symbian C++ good practice applies too,¹ but we'll skip that for this discussion.

8.1 Always Use an Active Scheduler

This can't be stated enough: the multimedia subsystem cannot work without proper use of an active scheduler and associated active objects. Some APIs (for example, within ICL) explicitly use `TRequestStatus` to show asynchronous behavior and expect the client to drive them via an active object. With others, the behavior is callback-based and its asynchronous nature is hidden. Virtually all the APIs make extensive use of `CActive`-based active objects behind the scenes and need an active scheduler to operate correctly.

When making function calls from your main application, this should seem self-evident, because the same rules apply to the application framework. On Symbian OS, all standard applications run within an active scheduler, with calls made from the `RunL()` calls of individual active objects. You have to make an asynchronous call from one `RunL()` instance and exit without waiting for the response. For example, when handling menu events, you might write the following code to open a clip in a descriptor, pending play:

```
iPlayer->OpenDesL(*iMyBuffer);
```

¹Advice for C++ developers new to Symbian OS can be found in a range of Symbian Press books and booklets, available from developer.symbian.com/books, and in articles published online at developer.symbian.com.

You can fire off several such calls, but should then return through the `AppUi::HandleCommandL()` call or equivalent. At some later stage, the callbacks are made by the associated API implementation. At the root of these callbacks, there is always a `RunL()` call; if you have access to the Symbian OS source code then you can see this in the stack trace when you're debugging on the Windows emulator.

If you are running within the main application thread, this all comes for free – the standard application environment is itself based on active objects and needs an active scheduler. If you create your own threads, you need to create and install an active scheduler to get this to work. If you forget, the result is unpredictable – depending on the API – but, in general, not much will happen. The active objects are used not just for callback support but also for internal asynchronous processing.

Sometimes, you find yourself calling an asynchronous call from what was previously a synchronous function, for example when you are modifying existing applications to use the multimedia calls. It is preferable to redesign your main program: for instance to re-classify the calling function as asynchronous, so the underlying action is not considered 'finished' on return and to allow for actions in the background. Sometimes this is not possible or it looks undesirable. One option in this scenario is to use the `CActiveSchedulerWait` class, for which documentation is available in the Symbian Developer Library (in the Base section of the Symbian OS Reference).

Our experience is that it is less of a problem to use asynchronous styles of programming rather than forming synchronous calls from asynchronous ones. However, this is sometimes all you can do – sometimes you have to fit into an existing API structure that is synchronous. This is often the case if you are using existing code modules, perhaps ported from another operating system. If you do use `CActiveSchedulerWait`, then take extra care not to block program input while a relatively long action is occurring. Remember, the `RunL()` method in which you start the `CActiveSchedulerWait` is blocked until it returns.

8.2 Use APPARC to Recognize Audio and Video

As we noted in Chapters 4 and 5, the multimedia framework provides separate APIs for audio and video clip usage. Depending on the applications you are developing, you may want to handle both – i.e. given a file, you may want to use the video APIs for video files and the audio APIs for audio files. The question is how do you do this?

One approach would be to guess: to try to open a file as video and then open it as audio if this fails. This should be avoided for several reasons:

- MMF might manage to partially open the clip – only for it to fail towards the end of the sequence. It is comparatively expensive to open a multimedia clip – involving loading several plug-ins – so this may slow down your application.
- MMF might actually succeed in opening the clip in the wrong mode! You may open an audio file using the video API, with an associated blank screen – it depends on how the plug-ins behave. More likely, a video file opened as audio may legitimately support playing the audio channel.

The recommended approach is to use the application architecture server² to find the MIME type:

```
// Link against apgrfx.lib
#include <APGCLI.H>
RApaLsSession lsSession; // assume already connected
RFile fileToTest; // assume already opened
TDataRecognitionResult result;
TInt error = lsSession.RecognizeData(fileToTest, result);
TPtrC8 mimeType = result.iDataType.Des8();
```

Once we have this information, we can decide whether it is an audio or video clip by checking the MIME type:

```
const TInt KLength = 6; // either "audio/" or "video/"
_LIT(KAudio, "audio/");
_LIT(KVideo, "video/");
if (mimeType.Left(KLength).Compare(KAudio)==0)
{
    // Treat as audio
}
else if (mimeType.Left(KLength).Compare(KVideo)==0)
{
    // Treat as video
}
else
{
    // special case
}
```

A WAV file would show as ‘audio/wav’ and a 3GPP video file as ‘video/3gpp’. For this exercise, we’re not so much interested in the whole string but in the first part (officially, the ‘type’). In most cases, a clip whose MIME type begins with ‘audio/’ can be opened using the standard audio client APIs, e.g. `CMdaAudioPlayerUtility`, and one beginning with ‘video/’ can be opened with `CVideoPlayerUtility` – the same

²Information about APPARC can be found in the Symbian Developer Library documentation in your SDK or online at developer.symbian.com.

technique shows up other classes of file and can be easily extended. If there are specific formats that the application wishes to handle in a particular way, then the whole MIME type can be tested.

A variation on this is where the application already has a MIME type for the clip but may still prefer to use APPARC. For example, for browser-type applications, the MIME type is often supplied by the server as part of the download protocol. Using this has some advantages: you get to know the MIME type before downloading the file and it ought to be faster. However, there are a couple of disadvantages that might not be immediately apparent:

- For many types of file, there are actually several legal MIME types. For example, WAV files can be 'audio/x-wav' or 'audio/wav' – 'audio/x-wav' is perhaps best thought of as deprecated although is still legal. The Symbian recognizer should consistently indicate 'audio/wav' as the MIME type. Another system may vary in this regard. If you rely on the supplied MIME type, take extra care.
- For some file types, there are very similar 'audio' and 'video' versions. For example, an MP4 file may contain only audio data or may also contain video. As a first degree of approximation, the recognition system should look through the file for references to video – if found, it is treated as a video file; if not, as an audio file. Other systems may not do this.

In general, it is probably safer to ignore the supplied MIME type and use APPARC, as described. However, if you have yet to download the files then go for the supplied version – just take care!

8.3 Don't Use the Video Player to Open Audio Files

In some formats, it is possible to open an audio file using `CVideo-PlayerUtility`. However, this should not be relied upon; neither is it efficient. It may work on some variations of a device but not others, depending on what plug-ins are installed. It is better to use the video APIs to open video clips and the audio APIs to open audio clips. See Chapters 4 and 5, respectively.

8.4 Know that MMF and ICL Cannot Detect Some Formats

This can be thought of as the inverse of the guideline in Section 8.2! Most of the time, if you give MMF or ICL a file, it can open that file

and subsequently play or decode it, without you having to supply any more information. The mechanism behind this relies chiefly on there being a number or string at the beginning of a file that can reliably identify the format. This is true for most multimedia files, but not all and some relatively common file formats don't provide it, including MP3 and WBMP (Wireless Bitmap) files. In such scenarios, it is normal to use the filename extension.

So far so good, but what about when there is no suffix – such as when reading data from a descriptor? This can present a problem, and there are a number of different approaches:

- Don't use descriptors when you don't know the file format – if you know the suffix, when downloading a file for example, use a temporary file.
- If you know the MIME type (sometimes this is supplied by the communications protocol), then use it.
- If anonymous opening does not work, then use the APPARC server to recognize and supply the MIME type. If you've already called the APPARC server, for example to distinguish audio from video, then it does little harm to use the MIME type.

The final approach may sound counter-intuitive. If MMF or ICL can't work out the format, how can the application architecture server? APPARC recognition can use different algorithms – rather than just looking for magic numbers, it can look for internal information about the files. It can be worth a go.

If none of these work, it is probably safest to flag a clip as unsupported, using whatever manner your application would normally use. However, there are times when this is not sufficient and, at this point, you are most likely going to have to guess at the MIME type.

Should you really bother about this problem? We'd suggest not, unless it comes up as a usability problem in tests or you know you are going to play clips from descriptors that might fall under the 'can't detect' category.

8.5 Don't Use `CMdaAudioOutputStream` for Network Streaming

It is a common misconception, which is understandable given the name of the class, however, `CMdaAudioOutputStream` is not suitable for streamed data, in the sense of streaming audio over a network.

The name of the class goes back to the days of the Media Server on Symbian OS v7.0 (see Section 1.5.1). Its reasoning is largely lost now, but its intended use case was for playing unwrapped, raw-encoded data, to be fed into a DevSound codec. If you receive this sort of data over the network, this may be a valid approach. However, it is more likely that the streamed data is actually formatted data, requiring use of the MMF client utility classes and you should use the `OpenURL()`³ method of the appropriate API class.

How to tell? Well if the data stream just contains continuous audio data, then `CMdaAudioOutputStream` might be valid. Otherwise, look at the kind of data being transferred: ‘wrapped’ data, such as AAC or WAV files, requires a controller. You could write code that talks to the appropriate network source, strips out the extra data such as headers and network info, and uses `CMdaAudioOutputStream` to supply the encoded data to the codec. What you’d effectively be doing is writing a networking engine in parallel with the MMF client layer itself. The feasibility of doing this depends on the formats. Formats such as MP3 can be supported this way: on many devices, the codecs skip the extra data, but not on all platforms. Using `OpenURL()` on the client utilities is arguably more portable. Continuous PCM data can definitely be supported using `CMdaAudioOutputStream`.

One area where `CMdaAudioOutputStream` is useful is when your application generates the audio data. Perhaps it decodes audio data or generates the audio samples directly (for example, it may generate harmonic tones). You could use `CMMFDevSound` instead, but `CMdaAudioOutputStream` is generally considered easier to use. With `CMdaAudioOutputStream`, your client code pushes the generated audio using `WriteL()` and you can write several chunks of data without waiting for each buffer to be played by using additional chunks. `CMMFDevSound` works differently: it pulls data from the client code using the `BufferToBeFilled()` callback, which the calling code must handle in ‘good time’ or the audio may break up. Most people find `CMdaAudioOutputStream` easier to use, although in theory `CMMFDevSound` should be more memory efficient.

As an extra word of warning, the adaptations on particular devices may have more requirements over the calling sequence, particularly for the more complex formats. They may expect additional setup calls and constraints, such as only supplying whole frames in each `WriteL()`. In general, we advise that you try it to see what happens, but if you do hit problems that might be the reason. You’d have to seek advice on the particular smartphone concerned.

³Note that, on S60, `CVideoPlayerUtility` is more likely to support your audio stream than `CMdaAudioPlayerUtility`, contradicting some of the advice above! Of course the exact behavior depends on the controller plug-ins present on the device in question.

8.6 Use `CMdaAudioPlayerUtility` to Play Tone Sequence Files

At face value, the way to play a tone sequence file is to use `CMdaAudioToneUtility`. However, it is also possible to play such files using `CMdaAudioPlayerUtility` – device configurations include controllers to play these files, so you can play them like any other audio file.

In general, using `CMdaAudioPlayerUtility` is preferred to `CMdaAudioToneUtility`. Although it uses more runtime resources – an extra thread and several plug-ins to support controllers – the same `CMdaAudioPlayerUtility` code can play not only simple tone sequence files, but also music files, recorded data, MIDI files and so on. Most of what end users think of as ‘ringtones’ are normal audio files.

There is a more subtle reason for preferring `CMdaAudioPlayerUtility`: the format of the files supported by `CMdaAudioToneUtility` varies from one smartphone to another. Although it is usually the same for devices produced by any particular manufacturer, it is probably safer to use `CMdaAudioPlayerUtility`. Formats supported entirely via a controller plug-in are not accessible by `CMdaAudioToneUtility`. What is supported by the DevSound of one device might be implemented using a controller on another. Use `CMdaAudioPlayerUtility` and this need not concern you.

So why use `CMdaAudioToneUtility`? If you want to play simple tones, DTMF tones or sequences, it is still probably the simplest API to use. If you already require a `CMdaAudioToneUtility` object for this purpose, then maybe it can be doubled up to cover files too. However, this is a stretch and consistently using `CMdaAudioPlayerUtility` will almost certainly require you to write less code, and it will be more flexible.

8.7 Use `CMdaAudioPlayerUtility` to Play Clips

This is a generalization of the previous guideline. You may notice that there are several client-facing audio APIs capable of playing different sorts of audio file: as well as the general `CMdaAudioPlayerUtility`, you can use `CMdaAudioRecorderUtility`, there is a dedicated API for MIDI files and a tone utility capable of playing sequence files. It may be assumed that if you have a MIDI file you have to use the MIDI utility, and if you have a tone sequence file you have to use the tone utility. As stated in Section 8.6, this is not true for tone files and it is not true for any type of file. Even MIDI files can be played via the standard `CMdaAudioPlayerUtility` interface – you only need to use the special MIDI client utility if you want to set special, MIDI-specific parameters.

That does lead to the question of when to use the play facilities from `CMdaAudioRecorderUtility`. The key use case for using the recorder utility to play audio is when playing back a recording that has just been made.

If you just want to play a file, then it is best to use `CMdaAudio-PlayerUtility`. The plug-in scheme associated with this API plays sequences or MIDI files if required, with reasonable default settings where applicable. You can use virtually any file with the same code.

Having said that, the recorder utility does give more progress information – the callback should tell you when playing or recording starts, while the play utility callback just tells you when playing finishes. Some application writers think this feature is worth the extra overheads – it allows you to delay updating your client API until it is confirmed that play has started.

8.8 Don't Hardwire Controller UUIDs in Portable Code

Some of the `MMFOpen()` methods have the ability to select a controller by its UUID. It is tempting to use this, but it should be avoided at all costs⁴ since it leads to non-portable code. The controllers vary from device to device and the same format may be supported by different controllers on different devices, even those that are closely related.

Rather than stating explicitly which controller to use for a given scenario, use the default `KNullUuid` value for the controller – it tells MMF to go and find a suitable controller.

Just because you know the correct controller for device A does not mean it is valid for device B. Far better to be flexible and let MMF work it out – MMF looks at the data and works out what to open.

There is an occasional case when you need to tell MMF the format because it can't work it out. Again resist the temptation to state the controller UUID. The most portable solution is to state the MIME type, for example:

```
_LIT(KVideo3gpp, "video/3gpp");
iVideoPlayer->OpenUrlL(aUrl, KUseDefaultIap, KVideo3gpp);
```

This example is for video streaming, but the same goes for other APIs too. By not being specific, the MMF can find the most suitable controller for the data.

So why, you may be wondering, does this API feature exist? The straight answer is for testing purposes. If you are testing plug-ins, it is useful to

⁴For video recording, the controller UUID is a required parameter but a dynamic plug-in query and selection mechanism is provided. See Section 4.4 for details.

know which controller you are using. There are additional cases where engineers writing code for a particular device wish to be very specific about the controller to use, or perhaps engineers working for the company who wrote a specific controller wish to ensure their applications use the company's controller. Perhaps the controller has extra features they wish to exploit. If you want to write portable code that works independently of the specific controllers, don't do this.

There is a trick for audio recording to a file that allows you to avoid UUIDs altogether; the format can be selected by choosing the appropriate file suffix. For example, if you save to a filename `'C:\temp.wav'`, you get a WAV format file without having to give any further parameters. That saves you having to try to select the format via the controller and works in most situations.

8.9 Set Controller Thread Priorities Before Playing or Recording

Each controller runs in its own thread. For largely historical reasons, the thread priority is that of a normal thread. Experience suggests that in many, if not most, scenarios this is too low. Even though most of the processing time is spent at the lower codec level, there are time constraints at the controller level where, once asked to handle data, it must do so in a reasonably short time.

If the player controller layer runs too slowly, in the best-case scenario, the output device may be 'starved' of data and the end user will perceive unpleasant gaps in the audio; in the worst case, a `KErrUnderflow` error will occur and play will stop. The effect will depend on the particular device.

The opposite problem can occur for recording – too much data is read at the device and is not pushed into the file fast enough. Again the effect depends on the device, but recorded sound may be lost or a `KErrOverflow` error may be generated. Some controllers carry out some of the codec operation too; potentially this just makes things worse.

The solution is to raise the thread priority of the controllers. To a certain extent this requires balance and depends on what else is going on both inside the application at hand and in the wider system. Set the controller priority too high and something else will be 'starved'. Set it too low and the controller won't run fast enough.

There are basically two approaches:

- permanently set the controller priority
- set the priority just before playing or recording and reset it after.

Most of the time this does not make any difference, although the second option is perhaps better if your application tends to open in one stage and play in another (an application where the file is opened and then the user has to call play, for example) and supports querying of metadata and so on.

Note: this is partly about being a good smartphone citizen, but also about ensuring that end users see your application as well-behaved.

8.10 Recognize that Behavior Varies when Several Clients Are Active

This is one of those things to watch if you only have one test device: if your application tries to use several multimedia client objects at the same time, the effect varies from one device to another. To a certain extent, this is no different from memory usage. Multimedia implementations often have special memory and other resources, and your program can run out of them on one device and work on another, just like with normal memory. However, for audio in particular, things are more complex.

Essentially, a request to play is merely that – a request. Even if your program is running in the foreground, it is not its choice as to what runs. Whether several audio components are played is up to the ‘audio policy’. The rule sets that constitute this policy vary from one smartphone model to another. If you request things to happen, don’t be totally surprised if they are stopped by the audio system – your application needs to handle this. It may decide to ignore the rejection, because the sounds are not that important or it may decide to reflect this to the end user. This usually depends on the type of application: for example, key clicks can usually be skipped but a music player should show ‘stopped’ if the clip is stopped by the underlying system.

An additional issue is the underlying hardware – this is reflected partly in the policy, but not totally. If more than one client plays simultaneously through the same device, then this requires use of an audio mixer. Audio mixing is effectively a trade-off between quality and performance and power usage – on some devices, the resultant mixed audio will sound better than on others.

If you want to play several things at once – say background music coupled with key clicks – consider making this optional for the user. The quality of the mixed effect on some devices may be good, in others less so. End users may prefer to run the application without the background music but with other sound effects – give them the option.

8.11 Understand that the System is Based on Plug-ins

A commonly asked question is ‘what codecs and formats are supported by Symbian OS?’ There is no easy answer to this: at heart, all the key multimedia libraries are based on plug-ins – there are frameworks that load plug-ins as appropriate. The answer to what is supported depends almost totally on what plug-ins are provided, and thus is a changing story since they can be added to by downloads, etc.

In fact, there is no single story on an individual device either because of platform security rules (see Section 2.2). The usable plug-ins may vary from one application to another – an application cannot directly exploit plug-ins with fewer capabilities than it has.

The conclusion from this is that you should not write software that has a fixed notion of what features are available. A more subtle conclusion is that you should be very careful about caching the list of available plug-ins (it might change) or creating it globally – what is valid for one application might not be the correct list for another.

So what should you do? In general it is better to avoid code that tries to second guess what ICL, MMF etc. does – just try it instead and deal with the failure! For example, assume you have a collection of images and want to display them. Rather than trying to write code with this algorithm:

```
if (we support the file)
    open and display using ICL
else
    display default image
```

You should try something along the lines of this pseudo-code:

```
error = Open file in ICL
if (!error)
    error = convert file in ICL
if (!error)
    display ICL output
else
    display default image
```

OK, the real code will be more complex than this, as the conversion stage, at least, is asynchronous. However, it does show the underlying approach. It is quite normal in applications to show a default image, e.g. a question mark, should there be an error. This pseudo-code suggests showing the same default image in any case of error. This is probably the best thing to do – most users won’t be interested in the distinction

between errors. You could possibly distinguish between ‘transient’ errors, such as out-of-memory, where you might sometimes be able to decode an image (especially a big image) depending on which other applications are running. However, even that is probably best just treated as yet another error.

So much for images. For audio and video, you will probably want to distinguish between a failure to open and a failure to play, giving appropriate error messages. What you do depends on your application domain, what the sound is and the interface. For example, if you’re using ‘beeps’ to signal that something has finished, you can probably just ignore problems – an end user would generally be confused by an error report. However, if your application is a music player that is supposed to be playing a music clip, then you need to say that something went wrong. Quite what you say should depend on how the error is passed to you – `KErrNotSupported` or `KErrCorrupt` during the opening and initialization step usually says there is no suitable plug-in or that the file is not really an audio clip; during play, they suggest the file is corrupt and the plug-ins could not recover.

For recording, there is a temptation to want to list as options only those formats available. With some inside knowledge this is possible:

- You may be able to use ECOM to search the actual formats and this is the neatest solution. However, even then it may not be portable – just because the frameworks currently use controllers and formats does not mean they always will – a future version could easily start to exploit a different type of plug-in. This has changed in the past and will almost certainly change at some time in the future.
- Let the user select from a known list and handle the failure. This is guaranteed to be portable but at the cost of reduced usability, and it is not recommended.
- Before options are presented, quickly try to see what can be set up. There is no need to actually record – the requirement is to see what suffixes can be opened and then, if required, the supported raw-formats can be listed. This is both portable and usable but it incurs a performance penalty over querying the supported formats via ECOM, since the relevant controllers each have to be found and loaded by the framework and several files created and deleted.

If none of the above options are suitable, any implementation should support PCM16 WAV and IMA ADPCM WAV for both play and record. However, they do not give the best tradeoff between compression and quality – PCM gives high quality and no compression, whereas ADPCM gives average compression and low quality. Having said that, they are supported by virtually any audio platform, one way or another, and are

thus also safe for sending emails and similar communication. Otherwise AMR is commonly, if not universally, supported and is a good format to use for other audio.

8.12 Use **RFile** and **TMMSource** Instead of Passing a File Name

When using `CMdaAudioPlayerUtility`, there are a number of ways of stating which file to play. These include:

```
static CMdaAudioPlayerUtility* NewFilePlayerL(const TDesC& aFileName,
                                              MMdaAudioPlayerCallback& aCallback,
                                              TInt aPriority = EMdaPriorityNormal,
                                              TMdaPriorityPreference aPref = EMdaPriorityPreferenceTimeAndQuality,
                                              CMdaServer* aServer = NULL);
```

or the following:

```
static CMdaAudioPlayerUtility* NewL(MMdaAudioPlayerCallback& aCallback,
                                     TInt aPriority = EMdaPriorityNormal,
                                     TMdaPriorityPreference aPref = EMdaPriorityPreferenceTimeAndQuality);
```

Followed by one of:

```
IMPORT_C void OpenFileL(const TDesC& aFileName);
IMPORT_C void OpenFileL(const RFile& aFile);
IMPORT_C void OpenFileL(const TMMSource& aSource);
```

The question is: which sequence to use?

The reason for the large number of similar calls is historical. When adding new features that require additional API methods, Symbian preserves the previous calls for use by existing applications. There is little reason to refactor existing applications just to use the latest API, but for new applications the general advice is clear: instead of passing the file-name to open, pass an open `RFile` handle. Especially if the `RFile` object is passed down by the application, this makes it easier for the application to deal with files passed by other subsystems – passing `RFile` handles around has replaced filenames as the standard Symbian OS method of passing a file from one application to another.

In addition to this, it is worth considering using `TMMSource` objects to pass the `RFile` handle to the multimedia subsystem. The main reason for using `TMMSource` is because it is useful for supporting DRM scenarios – it allows DRM-specific information to be supplied. Again,

even if your application does not currently require this, using `TMMSource` consistently makes things easier for future adaptation (see Section 2.3 for more on DRM and `TMMSource`).

There are equivalent calls in each of the multimedia player and decoder APIs, and they should be used for the same reasons.

8.13 Consider the Volume Control

The settings for `MaxVolume()` and `MaxGain()` vary from one device to another, although they are constant for a given device. Applications must not assume specific `MaxVolume()` or `MaxGain()` values across a range of devices.

So how does this affect your applications? Be very careful about volume sliders and other choice UI features. You have a choice: enquire about the `MaxVolume()` (or `MaxGain()`, as appropriate) setting and change the slider values; or hardwire a range (say 0 to 10) and then scale to the real MMF range. In general, people seem to prefer the former, but it is an application choice.

A second point concerns the default volume or gain. It is recommended that you use $(\text{MaxVolume}() + 1) / 2$ for the default volume, and the equivalent for gain.

As a side point, it is probably always a good idea to set the volume or gain. For some versions of Symbian OS, you need to or nothing can be heard.

Another feature you may find useful is volume ramping (using one of the `SetVolumeRamp()` methods), which provides a fade-in feature. The volume increases from zero to the current setting at the start of playback over a time period you specify. It's a good idea to use this for a short fade-in (say, half a second), even if you don't need a fade-in effect. With the audio stream APIs, or `DevSound`, it can help avoid any popping or clicking noises when starting playback. In addition, it can prevent a phenomenon known as audio shock, where a very loud sound is started suddenly and the end user's ear doesn't have time to adjust – potentially damaging hearing. There is no fade-out API in standard Symbian OS, although it may be supported as an extension on some devices – look at the appropriate SDK.

8.14 Know When to Use Threaded Requests on ICL

By default, ICL decoder and encoder plug-ins run in the same thread as the calling code, but the client can request that they run in a separate thread (see Chapter 6 for examples). It is perhaps tempting to do

this for performance reasons but, because of the extra overheads of thread support, using threads is not always much faster and is occasionally slower.

You should, almost certainly *not* do this – just accept the default setting. The only time you may want to use the option is when your program is doing something else that is time critical.

Normal ICL processing takes place in the same thread using `CActive`-based objects – a standard way of doing background-processing on Symbian OS. The disadvantage of this approach is that it takes a little longer for the program to respond to other events, such as key presses, pen taps and redraw requests. In such a case, the `CActive::RunL()` call doing the background processing must first finish before the higher priority event is handled. This is not as bad as it might seem – image processing is split into many reasonably short `RunL()` calls, instead of a single large one, so in normal processing this delay should be no more than 100 ms and not generally observable. However, in some scenarios it is important, e.g. if your application is streaming data and, more importantly, is streaming it in the main thread. In such situations, and only in such situations, you should perhaps use this option.

Why not always use it? Well apart from higher resource utilization, it turns out that creating, deleting and stopping the encoder and decoder objects takes a noticeable amount of time – so by using this option, your application might go slightly slower. OK, your end users probably won't see it but this option does not come without disadvantages.

There are other scenarios where the ICL codec runs in its own thread – where the codec itself requests to do so. Some codecs can't easily be written using the multiple `RunL()` approach and using a thread allows the codec to be supported without having to be completely rewritten.

To summarize: don't do this by default, but it can be useful in some circumstances if used with care.

8.15 Don't Have too many Clients

At one level, Symbian OS is a sophisticated, multi-tasking system that can have many applications open and executing concurrently; at another level, the target devices are still relatively small and have limited memory and processing power – especially compared with desktop computers. It is comparatively easy to ask the device to perform operations that are beyond it, trying to have too many active clients, decoding images, playing sounds, and playing video at the same time.

On a very simplistic level, the more clients you create, the more memory and other resources are used. However, the reality is more

complex: an ‘active client’ consumes significantly more resources than an idle one. By ‘active’ we mean, for example, an MMF client that is playing or recording, or an ICL client that is decoding or encoding.

The effects vary, but the basic rules are fairly straightforward: within reason, you can have several clips or images open, but if they are active, it is more likely there will be a problem.

The lowest memory usage comes from doing only one thing at a time. For example, if you have several images to decode to bitmaps, it is generally better to do one at a time. For standard decoding you only need to have a decoder object for the conversion operation and can then delete it, keeping the generated bitmap output to display. Similarly if you are transcoding several audio files, you should convert each file serially without trying to overlap.

Things get more tricky where the use case actually calls for you to do more than one thing. For example, where you are downloading several images concurrently and wish to display partially downloaded results, or wish to play more than one audio stream. For the latter case you should perhaps consider the reasonableness of the use case – playing four separate channels may work on some devices but it won’t on others – and perhaps have some option for ‘simple’ as opposed to ‘rich’ audio. For the partial download case, having a limit on the number of active decoders is probably a good idea, for example, in the region of two to four is a good rule of thumb.

We suggest you avoid the approach of hardwiring what works on a particular device since variations between firmware versions, other software being installed, and so on, can all make a difference. Two devices that end users believe to be identical may in fact have different performance characteristics. Don’t over-complicate things but do allow some options.

As a final point, too much going on may be a sign of a confusing application (apart from, perhaps, in games) – so take care!

8.16 Understand the Multi-heap Problem

This is a specific issue of trying to do too much at the same time, and with a bit of luck you will never run into it! However, it is worth knowing about, just in case.

The issue is to do with *chunks* – see `RChunk` in the Symbian Developer Library documentation. These chunks are used to support sections of memory. From an application perspective, they usually represent sections of real memory accessible by the process – they are used for various things including heaps, some DLLs, shared memory between

processes, device data buffers, etc. At the kernel level, these relate to memory management – how the virtual addresses are mapped to physical addresses, etc.

Although chunks do not normally concern application writers, they are quite a useful tool that many of the Symbian OS user-side libraries depend on. Each heap is a separate chunk, and that is a fundamental feature of C++. However, depending on the underlying hardware, the number of chunks that can be mapped into a process can be limited. At Symbian, we usually work to a rule-of-thumb where there is a limit of 16 chunks per user process – regardless of whether it is an application, a server or something else. This restriction only applies on some platforms, but it is a good idea to assume it if you want the code to be portable. What this means is that having too many heaps is an issue.

So how does this affect us? Well from Symbian's perspective, we've had to try to design the subsystems to not create too many heaps. From an application programmer's perspective, you have to be careful in some scenarios. In general, it should not affect you – unless you are trying to do too much at the same time. ICL instances normally share the user heap, apart from a few plug-ins that use temporary chunks or heaps to avoid fragmentation. If you stick to two or three active `Convert()` operations, lack of chunks should not give you problems.

MMF is, potentially, another matter. The problem is that each open client utility normally uses a separate heap with MMF – not just when they are playing or recording. If you open too many clients, you might hit the chunk limit – even if you just open several clips with different clients and don't actually play them! The best solution is to limit the number of clients that can be opened concurrently (as suggested above, to two or three) but there are some scenarios where this is difficult. As an alternative, you can give the `UseSharedHeap()` call to the various client utilities – note that the `CMidiClientUtility` is slightly different, as there is an additional `NewL()` overload that supplies `aUseSharedHeap` as a parameter. An example of usage is:

```
player = CMdaAudioPlayerUtility::NewL(*this);  
player->UseSharedHeap();  
player->OpenFileL(iFileName);
```

The key thing to watch is that `UseSharedHeap()` is called *before* you call `OpenFileL()` as that is where a new heap is normally created. All the clients concurrently invoked from a given client thread with this setting use the *same* heap. By default, if `Player1` and `Player2` are invoked at the same time, they each have their own heap; assuming a single client heap, there are three heaps overall. However, if `UseSharedHeap()` is

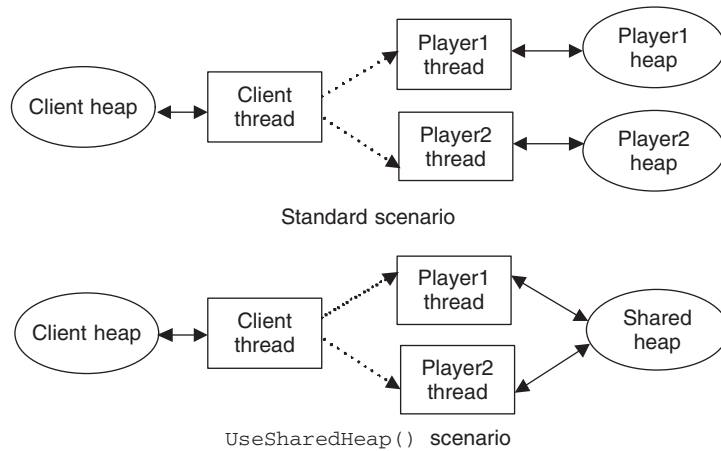


Figure 8.1 Using shared heaps

called in both Player1 and Player2, a single ‘MMF controller heap’ is used for both MMF client utilities – so there are only two heaps overall. This is illustrated in Figure 8.1.

In the normal scenario, each player utility has an associated controller thread and a controller heap, in addition to the thread and heaps of the client. Using `UseSharedHeap()`, each player still has its own thread, but they share a heap – although the heap is still separate from the client’s heap. This is more useful when many player, recorder or converter utilities are created and `UseSharedHeap()` is called on all of them! Nevertheless, it can be useful in some scenarios. As a rule of thumb, make this call if there are four or more MMF utility clients called from the same program simultaneously.

You may be asking, ‘Why not always do this?’ Well there are some disadvantages to using shared heaps for the controllers. Some latencies may increase, as, although the heaps are *thread-safe*, it does require semaphore waits. Probably the biggest issue is what happens if the controller should panic and die abruptly. In the normal scenario, the heap is deleted and the memory is recovered – ‘resource’ links, such as handles to other servers, to drivers, etc. are recovered by a kernel walker called the ‘undertaker’ that looks for and cleans up dead links.

In the shared heap scenario, the heap is only deleted when there are no clients left that use it – it is reference-counted by the client code. By implication, this relies on the normal destruction of the controller to recover memory; where this fails, the associated memory is not immediately recovered but appears as a memory leak. A controller panicking will fall into this scenario, as would memory leaks in the controller code. In general, it is probably only worth using this feature in problem scenarios.

Note the problem manifests itself by `KErrNoMemory` leaves or returns, even when free memory is available. In 99% of cases, `KErrNoMemory` means exactly that, but occasionally it implies lack of virtual memory or other such things, as in this case. If you appear to have memory issues but seem to have free memory, this scenario is worth considering.

References and Resources

Online Resources

Symbian Developer Network

Symbian Developer Network ([**developer.symbian.com**](http://developer.symbian.com)) is a good resource for accessing the technical and commercial information and resources you need to succeed in the wireless space. Symbian Developer Network provides technical information and news, and gives you access to the tools and documentation you need for developing on Symbian OS platforms. You can also find discussion and feedback forums, example code and tutorials, a searchable list of FAQs and a developer wiki.

The wiki page for this book can be found at [**developer.symbian.com/multimediabook.wikipage**](http://developer.symbian.com/multimediabook.wikipage). We'll post additional links that are of interest to multimedia developers working on Symbian OS on the wiki site – do stop by and add your own favorites to the wiki page. We'll also use the page to keep an errata list for this book, and to provide additional sample code, news and useful information.

The main Symbian Press site for this book, from which you can download a full set of sample code, is [**developer.symbian.com/multimediabook**](http://developer.symbian.com/multimediabook).

There is also a good set of multimedia example projects that were delivered with *Symbian OS C++ for Mobile Phones, Volume 3* (see the list of Symbian Press Books). The sample code is available from that book's Symbian Press site at [**developer.symbian.com/main/documentation/books/books_files/scmp_v3**](http://developer.symbian.com/main/documentation/books/books_files/scmp_v3).

Symbian OS Developer Sites

These websites are also of interest for those developing mobile multimedia applications on Symbian OS.

Forum Nokia: www.forum.nokia.com

The SDK for working on the S60 platform can be downloaded from Forum Nokia, as can the Carbide.c++ IDE, and other tools and utilities. Forum Nokia provides a number of documents on general S60 development, as well as a multimedia documentation section which can be found at **www.forum.nokia.com/main/resources/technologies/multimedia**. A discussion forum about creating multimedia on S60 can be found at **discussion.forum.nokia.com/forum**.

There is also an extensive developer wiki that contains a number of articles about the multimedia support on S60, as well as a range of code examples. The wiki can be found at **wiki.forum.nokia.com**.

UIQ Developer Community: developer.uiq.com

The UIQ Developer Community site has many resources invaluable to developers creating applications, content and services for UIQ 3 smart-phones. There is a wiki version of *UIQ 3: The Complete Guide* available at **books.uiq.com**. (The book is also available in conventional form from the publisher, at **eu.wiley.com/WileyCDA/WileyTitle/productCd-047069436X.html**.)

About Symbian, Symbian OS and UI Platforms

Company overview: **www.symbian.com/about**

Fast facts: **www.symbian.com/about/fastfacts.html**

MOAP user interface for the FOMA 3G network: **www.nttdocomo.com**

S60: **www.s60.com**

UIQ: **www.uiq.com**

Standards

The Khronos Group: **www.khronos.org**

MPEG: **www.chiariglione.org/mpeg**

JPEG: **www.jpeg.org**

GIF: **www.w3.org/Graphics/GIF/spec-gif89a.txt**

PNG and MNG: **www.libpng.org**

RTP: **tools.ietf.org/html/rfc3550**

RTSP: **tools.ietf.org/html/rfc2326**

SIP: **tools.ietf.org/html/rfc3261**

Symbian Press Books

Further information about Symbian Press books can be found at ***developer.symbian.com/books***.

If you are new to C++ development on Symbian OS and wish to find out more about the idioms, toolchain and application framework, we recommend these books to get you started:

- *Developing Software for Symbian OS 2nd Edition*, Babin. John Wiley & Sons.
- *Quick Recipes on Symbian OS: Mastering C++ Smartphone Development*, Aubert et al. John Wiley & Sons.

The following book is a more detailed reference book for creating C++ applications on S60 3rd Edition and UIQ 3:

- *Symbian OS C++ for Mobile Phones, Volume 3*, Harrison, Shackman et al. John Wiley & Sons.

These advanced-level books are useful for learning about Symbian OS technologies and idioms:

- *The Accredited Symbian Developer Primer*, Stichbury and Jacobs. John Wiley & Sons.
- *Smartphone Operating System Concepts with Symbian OS*, Jipping. John Wiley & Sons.
- *Symbian OS C++ Architecture*, Morris. John Wiley & Sons.
- *Symbian OS Explained*, Stichbury. John Wiley & Sons.
- *Symbian OS Internals*, Sales et al. John Wiley & Sons.
- *Symbian OS Platform Security*, Heath et al. John Wiley & Sons.

Index

- 2.5G networks 2
- 3D... 33
- 3G networks 7, 13, 34, 83
- 3GP 56, 211–12
- μ-Law 11
- A3F 33
- A920, Motorola 7
- AAC 11, 56, 107–17, 214
- aCameraHandle 90–3
- active objects 9–10, 108–10, 130–5, 200–1, 209–10, 218, 223
- active scheduler 9–10, 108–10, 209–10, 218, 223
- Adaptive Multi-Rate (AMR) 26–7, 56, 91–3, 221
- AddDisplayWindowL() 65–6, 71–6
- ‘additional’ sounds 138
- AddMetaDataEntryL() 100–1
- aDecoderUid 147–8, 168
- ADPCM 124–7
- aDuration 118–20
- aFileName 186–9
- aFrameNumber 148–9
- aFs& 147–8
- aImageSubType 147–8
- aImageType 147–8
- A-Law 11
- AllFiles 19–20
- AllocL() 171–4
- AMR *see* Adaptive Multi-Rate
- aNewMainImageSize 186–9
- animation 182–5
- aOptions 147–8
- aOriginalSize 153–7
- APPARC 210–12
- AppendDataL() 173–4
- appending operations, audio 126–7, 129
- Apple
 - iPhone 2
 - iPod 7
- AppUi::Handle-CommandL() 210
- aPref 84–7, 90, 97, 137–8
- aPriority 84–7, 90, 97, 137–8
- aReducedSize 153–7
- ASCII 163
- aSourceFilename 147–8
- ASSERT 111–17, 128–35
- asynchronous operations
 - 48–9, 61, 66, 110–17, 130–7, 175–81, 189–93, 200, 209–10
- ATMs 132
- aTransformOpt 186–9
- AU 11
- audio 2–5, 9–12, 15–34, 55–61, 62–5, 66–88, 97–100, 102–3, 105–39, 199–205, 210–27
 - see also* DevSound; Multimedia Framework
- APPARC 210–12
- appending operations 126–7, 129
- balance 86–7, 122
- best practice 210–27
- bit rates 87, 97–8, 102–3
- channels range 99
- classes 105–39
- client APIs 105–39, 223–4
- concurrency policy 135–9, 223–4

- audio (*continued*)
 - controllers 97–100, 106–7, 216–17
 - duration 118–22, 127
 - enabling audio playback 87, 98
 - enabling audio recording 98
 - examples 102–3, 108–22
 - file conversions 127–9
 - files 26–7, 106–7, 117–29, 133–5, 210–11, 221–2
 - gain support 99
 - input/output streams 107–17
 - Media Server 9–10, 214
 - MIDI 11–12, 139, 215–16, 225–6
 - mixing considerations 218
 - playback of audio 9, 11–12, 15–16, 24–7, 55–61, 62–5, 67–88, 106–10, 114–22, 129–39, 199–205, 210–12, 214–16, 220–1
 - player utility 26–7, 106–7, 117–22
 - playing named files 118–22, 221–2
 - policy server/resource manager 136–7
 - position-setting operations 119–22, 126–7
 - pre-emption
 - considerations 135–9
 - priority settings 85–7, 97, 137–9, 217–18
 - recorder utility 26–7, 106–7, 122–7
 - recording of audio 9, 11–12, 15–16, 23–31, 97–100, 106–7, 108–17, 122–7, 136–9, 199–205, 215–16, 220–1
 - recording to named files 122–6
 - repeated clips 121–2, 215–16
 - sample rates 99–100
 - sequence files 133–5
 - sound types 138
 - statistics 4–5
 - stop/pause sequences 119–22, 126
 - storage formats 56, 87, 107–8, 116–17, 124–7, 211–13, 219–20, 230
 - streaming 26–7, 106–39, 206, 213–14
 - synchronization issues 59
 - tone utility 26–7, 129–35
 - Tuner API 24, 30, 197–207
 - types 87, 98
 - variations of input/output stream
 - implementations 116–17
 - variations of player utility 119–22
 - variations of recorder utility 126–7
 - video 55, 59, 84–7, 97–100, 210–12
 - volume 86, 122, 131–5, 222
- Audio Stream APIs 106–7
- AudioBitRateL() 87, 98
- AudioChannelsL() 99
- AudioEnabledL() 87, 98
- AudioSampleRateL() 100
- aUseSharedHeap 225–6
- AVI 56, 59–60
- balance 86–7, 122
- bar-code readers 54
- BeginDecodingL() 145–6
- BeginEncode() 176–81
- BeginScalingL() 190
- BeginTransformL() 186–9
- best practice 209–27
- bit rates 83, 87, 94, 97–8, 102–3
- bitmap-based viewfinder 45–6
- bitmaps 28–9, 45–9, 70, 141–96, 213, 224
 - see also* decoding...; encoding...; Image Conversion Library
- helper classes 193–5
- in-place transformations 190
- native bitmap operations 185–6, 189–94
- rotation 192–3
- scaling 189–93
- BMP 28–9, 141–96, 213
- books, Symbian Press 105, 229, 231
- brightness settings, onboard camera 41–3
- C++ 225
- CActiveScheduler 176–81, 193, 209–10, 223
- CActiveSchedulerWait 210
- CAF *see* Content Access Framework
- CalculateEffectiveSize 156–7
- calendar 84–5
- callback classes 85–6, 107–17, 118–22, 128–9, 209–10
- camcorders 1–2, 10–12
- camera controls
 - see also* onboard camera concepts 39–43
- CameraInfo 41–3
- cameras 1–2, 6–7, 10–12, 15, 29–30, 35–54
- Canalys 5

- Cancel() 131–3, 145–6, 176–81, 190–3, 204
- CancelRegisterAudioResourceNotification() 86
- capabilities
 - capturing still images 46–8
 - capturing video 49–51, 84–5
 - concepts 19–21, 36–8, 46–8, 49–51, 84–5, 137, 138–9, 199–200, 205
 - onboard camera 46–8, 49–51
 - RDS 205
 - Tuner API 199–200, 205
 - types 19–20
 - video 49–51, 84–5
- CaptureImage() 48–9, 52–3
- capturing still images, concepts 46–9
- capturing video 49–52
- CBitmapRotator 189–93
- CBitmapScale 189–93
- CBitmapScaler... 190–3
- CBufferedImageDecoder 28–9, 144–96
- CCamera... 29–30, 36–54, 90–3
- CConvertAudio 127–9
- CDesCArray 134
- CDMA network 83
- CFbsBitmap 28–9, 45–9, 70, 141, 145–96
- CFileExtensionMimeType 144, 166–7
- CFrameImageData 163–5, 178–81
- CFrameInfoStrings 163–5
- channels range, audio 99
- chunks 49, 224–7
- CICLRecognizerUtil 144–96
- CImage... 28–9, 49, 181–5, 194–6
- CImageDecoder 28, 49, 143–96
 - see also* decoding...
- CImageDecoderPlugin 142–96
- CImageDisplay 143, 181–5
- CImageDisplayHandler 184–5
- CImageEncoder 28, 142, 143–4, 175–96
- CImageEncoderPlugin 142–4, 175–96
- CImageProcessor 194–6
- CImageReadCodec 142–96
- CImageTransform 28, 143, 182–9
- CImageTransformHandler 186–9
- CImageTransformPluginExtension 187–9
- CImageType-Description 144–96
- CImageWriteCodec 142–96
- CImplementation-InformationType 144–96
- CIOStreamAudio 108–10, 114–17
- CJPEGExifDecoder 144, 168
- CJPEGExifTransform-Extension 187–9
- classes
 - audio 105–39
 - Tuner API 30, 199
 - video 63–5, 89–90
- ClearPlayWindow() 121–2
- client APIs
 - see also* Multimedia Framework
 - concepts 10–12, 24–8, 57–8, 60–103, 105–39, 141–96, 223–4
 - client-server 11–12, 144–5
 - clipping region 75–6, 94
 - Close() 63, 68, 93, 110–17, 176–81, 201
 - closing a video 68, 93
 - CMdaAudioClipUtility 123–7
 - CMdaAudioConvert-Utility 26–7, 127–9
 - CMdaAudioInputStream 26–7, 106–39
 - CMdaAudioOutput-Stream 26–7, 106–39, 213–14
 - CMdaAudioPlayer-Utility 26–7, 106–39, 211–12, 215–16, 221–2
 - CMdaAudioRecorder-Utility 26–7, 106–39
 - CMdaAudioToneUtility 26–7, 106–39, 215
 - CMdaVideo... 26–7
 - CMidiClientUtility 139, 225–7
 - CMMFDevSound 134–5, 214
 - see also* DevSound
 - CMMFMeta... 100–1
 - CMMFMetaData-Entry 84
 - CMMRdsTunerUtility 199, 203–5
 - CMMTunerAudioPlayer-Utility 199–205
 - CMMTunerAudio-RecorderUtility 199–205
 - CMMTunerScanner-Utility 199–205

- CMMTunerUtility 30, 199–207
 - see also* Tuner API
- codecs
 - see also* decoding...; encoding...
 - concepts 56–7, 59–60, 91–3, 95, 106–7, 108–17, 134–5, 142–96, 214, 223
- CoeControl 183–5
- color depth transformations, performance costs 191
- COM 15
- communications 1–14
- Complete 111–17
- compression factors, storage formats 56, 59–60
- computing 1–14
- concurrency policy, audio 135–9, 223–4
- constraints, smartphones 2–3, 197–8
- ConstructL() 109–10, 118–22, 123–7, 130–5, 176–81, 184–5, 190–3
- constructors 108–10, 118–22, 123–7
- Content Access Framework (CAF) 15, 21–3
- ContentAccess 160
- ContinueConvert() 173
- ContinueOpenL() 173–4
- ContinueProcessing-HeadersL() 172–4
- contrast settings, onboard camera 42–3
- controller framework 10–12, 24–7
 - see also* Multimedia Framework
- controller plug-ins 10–12, 24–7, 216–17
 - see also* Multimedia Framework
- controllers 10–12, 24–7, 30, 57–60, 61–81, 93–100, 106–7, 200–1, 216–18, 222–3
 - audio 97–100, 106–7, 216–17
 - thread priorities 217–18, 222–3
 - Tuner API 30, 200–1
 - video 57–60, 61–81, 93–7, 216–17
- convergence
 - functions 2–3
 - introduction 1–14
 - statistics 4–5
 - trends 1–2
- Convert... 127–9, 145–6, 159, 161, 171, 173, 175–81, 225–6
- Copy() 163–5
- copyright infringements 3–4
- CPlayAudio 118–22
- CRecordAudio 123–7
- CropFromThe-BeginningL() 127, 129
- cropping concepts 76–7, 127, 129
- CSimpleDecodeHandler 156–7
- CurrentDataOffset 151–2
- custom commands, video 101–3
- CustomCommand... 101–2
- CVideoPlayerUtility 60–103, 211–12, 214
- CVideoRecorder-Utility 60–103
- DAB *see* digital radio
- data caging, concepts 19–21
- DataNewL() 146–8, 170–4, 175–81
- debugging 210
- decoding still images 9, 23–4, 28–9, 49, 141–74, 181–96, 224
 - see also* Image Conversion Library
- concepts 28–9, 141–74, 181–9, 224
- creation options 144, 158–60
- different sizes 153–7
- DRM content support 158–60
- error codes 160–2
- Exif Utility 142–4, 167–8, 187–9
- frame comment/metadata retrieval methods 162–5
- getting opened image information 148–52, 162–8
- initial parameters 146–8
- performance issues 174
- plug-in discovery 165–7
- simple examples 145–6
- size factors 150–7, 161
- streamed decoding 170–4
- Symbian decoder plug-ins 168–70, 189
- thumbnails 157–8, 186, 189
- decoding video 56–7, 59–60
- Delete() 129, 193
- delivering video, concepts 55–6, 66, 87–8
- demand paging 14
- descriptors 57–8, 91–2, 107, 117–22, 133–5, 163, 170–4, 209–10
- destructors 108–10
- developing world, smartphones 3
- DevSound
 - see also* audio; CMMFDevSound

- concepts 12, 17–18, 25–8, 30, 33, 57, 59, 106–7, 129–35, 197–8, 214–15, 222
 - definition 134–5
- DevVideo..., concepts 12, 25, 27–8, 33, 57–60
- different sizes, decoding still images 153–7
- digital cameras 1–2, 6–7, 10–12, 15, 29–30, 35–54
- digital music sales 3–5
- digital radio (DAB) 30, 197, 201–2, 206
 - see also* Tuner API
- Digital Rights Management (DRM) 19–20, 21–3, 31–2, 65–6, 70, 121–2, 147, 158–60, 221–2
- Digital Signal Processor (DSP) 56
- direct screen access
 - video screen output controls 72–3
 - viewfinder 44–5
- DisablePost-Processing 192
- display windows, video
 - screen output controls 71–2, 129
- displaying images *see* Image Display
- disruptive technologies 3–4
- DLLs 14, 19–21, 144–5, 224–5
 - see also* ECOM
 - capabilities 19–21
 - demand paging 14
- DMB 206–7
- downloads 3–4, 13
- DRM 19–20, 21–2, 121–2
 - see also* Digital Rights Management
- DSP *see* Digital Signal Processor
- Dual-Tone Multiple Frequency (DTMF) 26–7, 129–35
- duration 82, 94, 118–22, 127
- DVB-H TV 7, 206–7
- EAllowGeneratedMask 158–60
- EAlphaChannel 152
- EAutoScale... 79–80
- ECam 10–12, 24, 29–30, 35–54
 - see also* digital cameras; onboard camera
- ECanDither 152
- EColor... 149, 169, 174, 190–2
- ECOM
 - see also* Image Conversion Library; Multimedia Framework; plug-ins
 - concepts 10, 15–16, 20–3, 28–9, 31–2, 145–96, 220–1
 - platform security 20–1
- EConstantAspectRatio 152
- EFormat... 47–9
- EFrameInfoProcessingComplete 151–2
- EFullyScaleable 152, 154–7
- EGray... 192
- EImageCapture-Supported 38–9
- EImageTypeMain 157–8
- EImageTypeThumbnail 157–8
- EKern 15
- ELeaveInPlace 152
- electronic service guide (ESG) 206–7
- EMdaPriority... 137–8
- EMngMoreFramesTo-Decode 152, 170
- emulator 136, 210
- enabling audio playback 87, 98
- enabling audio recording 98
- enabling video playback 70
- enabling video recording 97
- encoding still images 9, 23–4, 28–9, 141–4, 174–81, 224
 - see also* Image Conversion Library
 - concepts 28–9, 141–4, 174–81, 224
 - defaults 179
 - definition 174–5
 - examples 176–7
 - initial parameters 177–81
 - Symbian decoder plug-ins 181
 - thumbnails 180–1
- encoding video 56–7, 59–60
- ENotReady 123–7
- EOption... 158–60, 172–4, 176–81
- EOrientation... 38–40
- EPartialDecode-Invalid 152, 161, 173
- EPOC32 15–16
- EPreferFastCode 160
- EPreferFastDecode 174
- ERecording 123–7
- ERestoreToBackground 151–2
- ERestoreToPrevious 152
- Ericsson 5–6
 - see also* Sony...
- error codes, decoding still images 160–2

- error handling 37–54, 60–1, 63–88, 111–17, 118–22, 124–7, 131–2, 136–7, 160–2, 219–20, 226–7
 - see also KErr...; leaves; panics
- ESG see electronic service guide
- ESock 15
- EState... 131–5
- ETel 15
- ETransparency-
 - Possible 152
- ETunerEvent 203
- EUsesFrameSize-
 - InPixels 151–2
- event-driven framework
 - see also active objects
 - concepts 9–10
- EventCategoryAudio-ResourceAvailable 86
- EVideoCapture-
 - Supported 38–9
- EXEs, capabilities 19–21
- Exif Utility 142–4, 167–8, 187–9
- exposure settings, onboard camera 43

- F32 22
- feature phones, definition 2
- FileNewL() 146–8, 158–61, 174, 175–81
- files
 - audio 26–7, 106–7, 117–29, 133–5, 210–11, 221–2
 - tone sequence files 133–4, 215
 - video 210–11, 221–2
- FixedSequenceCount 134
- flash settings, onboard camera 43

- FM/AM radio 12, 15, 23–4, 30, 197–207
 - see also Tuner API
- formats 10–12, 16, 41–3, 49–50, 56, 59–60, 87, 91–3, 107–8, 116–17, 124–7, 141–4, 211–13, 219–20, 230
- Forum Nokia 230
- FourCC code 56, 87, 91–2, 98, 116–17
- frame comment retrieval
 - methods 162–5
- frame metadata retrieval
 - methods 162–5
- frame rates 81–2, 93–4, 148–52
- frame size 82, 94, 102–3
- frame snapshots 70
- FrameCount 148–52, 169, 171–4
- FrameDataOffset 151–2
- FrameInfo 148–52, 158, 169
- FrameInfoStringsL... 163–5
- frames, getting opened
 - image information 148–52, 162–8
- FreeWay 13
- Fsm 131–2
- Fujitsu 5
- future prospects 12–14, 31–4, 206–7

- gain support, audio 99
- GainL() 99
- games 7, 53–4
- Games on Symbian OS (Stichbury) 139
- ‘general’ sounds 138
- GetCapabilities 199
- GetFileTypesL() 166–7
- GetFrameData() 164–5, 178–81
- GetFrameL() 70
- GetFrequency() 203
- GetDataImage() 178–81
- GetImageTypesL() 166–7
- GetImplementation-InformationL() 166–7
- GetMimeTypeDataL() 166–7
- GetMimeTypeFileL() 148, 166–7
- GetPlayRate-CapabilitiesL() 69–70
- GetPosition() 119–22, 136–7
- GetPriorityL() 85–7, 97
- GetRdsCapabilities() 205
- GetRdsData() 204–5
- GetRdsTunerUtilityL() 204–5
- GetScaleFactorL() 78–80
- GetSupportedAudio-ChannelsL() 99
- GetSupportedAudio-SampleRatesL() 99–100
- GetSupportedAudio-TypesL() 98
- GetSupported-Destination-DataTypesL() 125–7
- GetSupportedPixel-AspectRatiosL() 96
- GetSupportedVideo-TypesL() 95
- getting opened image information, ICL 148–52, 162–8
- getting video information 81–4
- GetTunerAudioPlayer-UtilityL() 202
- GetTunerAudioRecorderUtility() 203

- GetVideoFrameSize... 94
- GIF 28–9, 141–4, 148–52, 164–5, 169, 177, 180–1, 230
- GPS 1–2, 7
- graphics subsystem 7
- graphics surfaces 73
- GSM network 83
- H.263 format 56, 91–2
- H.264 format 56, 91–2
- HAL *see* hardware abstraction layer
- Handle() 53
- HandleCommandL() 40–1, 210
- HandleEvent() 53
- hardware
 - acceleration 12–13, 189–93
 - MMF 11–12, 25–8, 106–7, 216–17
- hardware abstraction layer (HAL) 11–12, 206–7
- HBufC 170–4
- heaps, multi-heap problem 49, 224–7
- high-bandwidth networks 13
- historical background, Symbian OS 5–12
- home networks 13
- horizontal positions, scaling 79–80
- HSDPA network 7, 83
- HTTP 88
- HWDevice 57, 59–60
 - see also* DevSound; DevVideo
- iBackgroundColor 151–2
- iBitsPerPixel 151–2
- iBufferPtr 112–17
- ICL Recognizer 142–4
- iDelay 151–2
- iFlags 151–2
- iFrameCoordsInPixels 149–52
- iFrameDisplayMode 149–52, 159, 174
- iFrameSizeInTwips 151–2
- iInputStream 108–10
- IMA ADPCM 125–7, 220–1
- Image Conversion
 - component
 - concepts 141–96
 - helper classes 193–6
- Image Conversion Library (ICL)
 - see also* multimedia subsystem
 - best practice 212–27
 - class diagram 143–4
 - concepts 10–12, 16, 24, 28–9, 36–54, 141–96, 212–27
 - decoder creation options 144, 158–60
 - decoding images 28–9, 141–74, 181–96
 - different size decoding 153
 - DRM content support 158–60
 - encoding images 28–9, 141–4, 174–81
 - error codes 160–2
 - examples 145–6, 176–7
 - Exif Utility 142–4, 167–8, 187–9
 - frame comment/metadata retrieval methods 162–5
 - getting opened image information 148–52, 162–8
- Image Display 143, 181–5
- Image Transform 142–4, 182–93
- initial parameters 146–8, 177–81
- performance issues 174, 222–3
- plug-in discovery 165–7
- simple examples 145–6
- size factors 150–7, 161
- streamed decoding 170–4
- structure 28–9, 141–4
- Symbian plug-ins 168–70, 179–81, 189–90
- threaded request
 - considerations 222–3
- thumbnails 157–8, 180–1, 186, 189
- uses 28–9, 141–4, 219–20
- Image Display, ICL 143–4, 181–5
- Image Processor, ICL 143–4
- Image Transform, ICL 142–4, 182–93
- ImageBufferReady 48
- ImageCommentL() 162–5
- ImageConversion 142–96
- ImageProcessorUseL() 195–6
- ImageProcessor-Utility() 194–6
- images
 - capturing still images 46–9
 - ECam 10–12, 24, 29–30, 35–54
 - ICL 10–12, 16, 24, 28–9, 36–54, 141–96, 212–27
 - Media Server 9–10
 - onboard camera basic image settings 41–3
- iMainBuffer 110–17
- ImplementationUId 166–7
- in-place transformations, bitmaps 190
- information
 - getting opened image information 148–52, 162–5
 - getting video information 81–4

- information (*continued*)
 - metadata 31–2, 83–4, 100–1, 162–5, 167–8, 203–5, 218
 - RDS 197–9, 203–5
 - references and resources 229–31
- InitializeL() 202–3
- innovative applications,
 - onboard camera 53–4
- input methods, constraints 2–3
- input/output streams, audio 107–17
- intellectual property rights 3–4, 21–3
- Internet 1–14, 55–6, 57–60, 66, 87–8, 91–2, 102–3, 170–4, 206, 213–14
 - historical background 1
 - streaming 3–5, 9, 10–12, 55–6, 57–60, 65, 87–8, 91–2, 102–3, 170–4, 206, 213–14
- iOutputStream 108–10
- iOverallSizeInPixels 150–2
- ISDB-T 1seg 206–7
- IsHeaderProcessing-Complete 171–4
- IsImageHeader-Processing-Complete 174
- IsRdsSignal 204–5

- Japan 4
- JPEG 16, 28–9, 47–9, 141–96, 230
 - see also Image Conversion Library
- Juniper Research 4

- KBMPDecoderImplementationUidValue 147–8, 169
- KBMPEncoderImplementationUidValue 181
- KERN-EXEC 3 110
- kernel 17–21
- KErrAbort 112–17
- KErrAccessDenied 53, 136–7
- KErrArgument 161
- KErrCorrupt 162, 220–1
- KErrDenied 136–7
- KErrInUse 85–6, 136–7
- KErrNoMemory 161, 227
- KErrNone 63–88, 111–17, 118–22, 124–7, 131–2, 145–6, 154–8, 171–3, 176–81, 184–5
- KErrNotFound 161, 168, 199–200
- KErrNotReady 52–3
- KErrNotSupported 37–54, 60–1, 71–3, 84, 162, 220
- KErrOverflow 113–17
- KErrPermissionDenied 20–1, 37–54
- KErrUnderflow 115–17, 119–22, 161, 170–4
- keys, DTMF 130
- KGIFDecoderImplementationUidValue 169
- KGIFEncoderImplementationUidValue 181
- Khronos 13, 32–3, 230
- KICODecoderImplementationUidValue 169
- KImageType... 147–8
- KJPGDecoderImplementationUidValue 169
- KJPGEncoderImplementationUidValue 181
- KMBMDecoderImplementationUidValue 169
- KMBMEncoderImplementationUidValue 181
- KMMFBalance... 87, 122
- KMMFErrorCategory... 65
- KMMFFourCCCodePCM... 116–17
- KNullDesc 111–17
- KNullUid 147–8, 216–17
- KOTADecoderImplementationUidValue 169
- KPNGDecoderImplementationUidValue 169
- KPNGEncoderImplementationUidValue 181
- KTIFFDecoderImplementationUidValue 169
- KUidICLJpegEXIF-Interface 148, 168
- KUidIclMng-PluginImplUid 169
- KWBMPDecoderImplementationUidValue 169
- KWMFDecoderImplementationUidValue 169

- latency problems 9, 226–7
- LeaveIfError 37–8, 150–2, 156, 176–81, 184–9, 190–3, 195–6
- leaves 36–8, 108–10, 150–2, 156, 176–81, 184–9, 190–3, 195–6, 226–7
- LG Electronics 5, 8
- licensees, Symbian OS 5–6, 10, 11, 23–4
- loudspeakers

- see also* audio
video 84–7
- LTE 13
- M600i, Sony Ericsson 197
- MaiscBufferCopied() 113–17
- MaiscOpenComplete() 110–17
- MaiscRecordComplete() 113–17
- malware 17–21
- manufacturers, Symbian decoder plug-ins 168–9
- MaoscBufferCopied() 115–17
- MaoscOpenComplete() 114–17
- MaoscPlayComplete() 115–17
- MapcInitComplete() 118–22
- MapcPlayComplete() 118–22, 136–7
- market share 4–5, 12
 - Nokia 4
 - Symbian OS 5, 12
- MatoPlayComplete() 132
- MatoPrepareComplete() 131–2, 133
- MAudioIOStreamObserver 108–10
- MaxGain 99, 222
- MaxVolume 86, 122, 131–5, 222
- MBM 34, 149–52, 169, 178–81
- MCamera... 37–8, 39, 45–9, 52–3
- MdaAudioTonePlayer.h 129
- MDF *see* Media Device Framework
- media, introduction 1–14
- Media Client image ICL component 142–4
- Media Device Framework (MDF)
 - concepts 11–13, 17, 25, 27–30, 32–3, 36–54, 57
 - new architecture 32–3
 - structure 11–12, 25, 27–8
- media industry
 - statistics 4–5
 - transformation 3–5
- Media Server 9–10, 214
- memory
 - see also* RAM
 - chunks 49, 224–7
 - demand paging 14
- memory management unit (MMU) 17
- metadata 31–2, 83–4, 100–1, 162–5, 167–8, 203–5, 218
- Metadata Utility Framework (MUF) 31–2
- MetaDataEntryL() 84
- Microsoft Windows Mobile 2
- MIDI 11–12, 139, 215–16, 225–6
- MiidoImageReady 183–5
- MIME types 56, 66, 82, 95, 144, 146–8, 166–7, 175–81, 211–12, 216–17
- Mitsubishi Electric 5
- MMdaObjectStateChangeObserver 122–7
- MMF *see* Multimedia Framework
- MMHP 33
- MMMFAudioResourceNotificationCallback() 85–6
- MMP files 198–9
 - see also* project files
- MMS applications 153
- MMU *see* memory management unit
- MNG 148–52, 170, 230
- MOAP(S) 6
- mobile phones
 - see also* smartphones
 - introduction 1–14
- mobile TV 33–4, 197, 206–7
- mono to stereo changes 127
- MoscoStateChangeEvent 122–7, 128–9
- Mosquitoes game, Ojom 53
- Motorola 5, 7–8
 - A920 7
- MOTORIZR Z8 7–8
- MOTO Z10 7–8
- MP3 11, 25–7, 56, 213
- MP4 56
- MPEG... 22, 56, 230
- MToTuneComplete 201
- MUF *see* Metadata Utility Framework
- multi-heap problem, best practice 49, 224–7
- multi-tasking systems,
 - concepts 135–9, 218, 223–4
- multimedia
 - definition 1–2
 - future prospects 31–4, 206–7
 - introduction 1–14
- multimedia architecture,
 - concepts 15–34
- Multimedia Framework (MMF)
 - see also* audio; controller...; ECOM; video
 - best practice 211–27
 - client APIs 10–12, 24–8, 57–8, 60–103, 105–39, 223–4
 - concepts 10–12, 23–30, 36–54, 55–103, 105–39, 197–8, 211–27
 - definition 10
 - hardware 11–12, 25–8, 106–7, 216–17
 - heaps 225–7

- Multimedia Framework
 - (MMF) (*continued*)
 - how it works 25–6
 - structure 10–12, 24–8
- multimedia subsystem
 - see also* ECam; Image Conversion Library; Multimedia Framework; Tuner API
 - best practice 209–27
 - concepts 4–5, 8–12, 15, 23–34, 35–54, 209–27
 - definition 4–5, 10–11, 23
 - diagrammatic overview 10–11, 23–4
 - evolution 8–12
 - Media Server 9–10, 214
- MultimediaDD 39, 84–5, 139, 202
- multiple threads 10–12, 24–8, 218
- ‘must have’ sounds 138
- MVideoPlayer... 63–88
- MVideoRecorder... 90–3
- MvloLoading... 88
- MvpuoEvent() 65–88
- MvpuoOpenComplete() 63–88
- MvpuoPlayComplete() 63–88
- MvpuoPrepareComplete() 63–88
- MvruoOpenComplete() 92
- MvruoPrepareComplete() 92
- MvruoRecordComplete() 92

- NAND Flash memory 14
- Netsize Guide 4–5
- NewDuplicateL() 53
- NewL() 36–8, 53, 63–93, 97–100, 118–22, 123–7, 130–5, 170–4, 176–81, 183–5, 189–93, 200–1, 225–6
- news announcements, RDS 205
- Nokia 4–5, 6–8, 45, 54, 102, 167, 230
 - see also* S60 7650 6–7
 - 9210 Communicator 6–7
 - market share 4
 - N96 7–8
 - Nseries range 7
- Notify() 204–5
- NTSC 95
- NumberOfFrame-Comments 163–5
- NumberOfImage-Comments 162–5
- NumberOfMetaData-EntriesL() 84

- Ojom, Mosquitoes game 53
- OMA DRM v 1.0 21–2
- on-the-fly pixel conversions 194–6
- onboard camera
 - see also* ECam
 - accessing issues 36–9
 - basic image settings 41–3
 - brightness settings 41–3
 - camera controls 39–43
 - capabilities 46–8, 49–51
 - capturing still images 46–9
 - capturing video 49–52
 - concepts 10–12, 23–4, 29–30, 35–54
 - contrast settings 42–3
 - error handling 52–3
 - exposure settings 43
 - flash settings 43
 - image format 41–3, 46–8
 - innovative applications 53–4
 - power control 39–41
 - secondary clients 53
 - video format/size/rate 49–50
 - viewfinder displays 43–6, 54
 - zoom settings 42–3
- online resources 229–31
- OnOpenL() 124–8
- Open... 22, 63–88, 91–3, 111–17, 133–5, 173–4, 209–10, 214, 216–17
- open standards 13
- OpenFileL() 63–93, 118–22, 123–7, 221–2, 225–6
- opening a video 66, 90–3
- OpenMAX 32–3
- OpenGL ES 33
- OpenUrlL() 214
- operating systems
 - see also* Symbian OS
 - smartphone/featurephone contrasts 2
- output/input streams, audio 107–17

- P800, Sony Ericsson 7
- P990, Sony Ericsson 197
- PAL 95
- Panasonic 5
- panics 173–4, 188–9, 226–7
- Pause... 67, 89–93, 120–2
- pause/stop sequences 62–5, 67–93, 103, 119–22, 126, 184–5
- PCM 11, 26, 59, 107–17, 124–7, 129, 214, 220–1
- PDA's 1–2, 6
- peer-to-peer file sharing 3–4
- performance issues 12–14, 174, 191–2, 209–27
- pixel aspect ratio 79, 95–6
- platform security
 - capabilities 19–21, 36–8, 46–8, 49–51, 84–5, 137, 138–9, 199–200

- concepts 11–12, 15, 16–21, 36–7, 137–9, 219
- data caging 19–21
- DTMF 132
- ECOM 20–1
- trust model 16–21, 138–9
- Play... 63–88, 118–19, 126–7, 133–5, 136–7, 183–5, 202
- playback of audio 9, 11–12, 15–16, 24–7, 87, 107–10, 114–17, 129–39, 199–205, 210–12, 214–16, 220–1
- playback of video 11–12, 15–16, 24–7, 55–61, 62–5, 67–88, 210–12, 214
- PlayFormats 91–3
- playing tones, concepts 132–4, 215
- PlayOrRecordComplete 126–7
- PlayVelocityL() 69–70
- plug-in discovery, decoding still images 165–7
- plug-ins
 - see also* ECOM
 - best practice 219–21
 - concepts 10–12, 15–16, 20–3, 144–96, 212, 216–17, 219–21
 - definition 15–16
 - media component library 10
 - Media Server 9–10, 214
- PNG 16, 28, 141–4, 149–52, 169, 178–81, 230
- portable code, controller UIDs 216–17
- Position... 68, 119–22, 126–7
- position-setting operations 68, 119–22, 126–7
- Power... 39–54
- power management 12–13, 39–40
- pre-emption considerations 135–9
- Prepare... 47–8, 50–3, 63–93, 130–5, 194–6
- PrepareImageCaptureL() 47–8, 52–3
- PrepareToPlayDesSequence() 133–5
- PrepareToPlayFileSequence() 133–5
- PrepareToPlayFixedSequence() 134
- PrepareVideoCaptureL() 50–3
- preparing to record a video 92
- preparing a video 66–7
- PriceWaterhouseCoopers 4
- priority settings 85–7, 97, 137–9, 200–1, 217–18
- PriorityL() 85
- privileges 11–12, 17–21
- project files 198–9
 - see also* MMP files
- Psion 5, 6
- Publisher ID 19–20
- QCIF 82–3, 102–3
- quality controls, video 96–7
- radio 12, 15, 23–4, 30, 197–207
 - see also* Tuner API
- Radio Data System (RDS) 197–9, 203–5
- RadioText 205
- RAM constraints 13–14
- RArray 46–8, 50, 62, 96, 98
- RAW 11
- RBuf8 117
- RChunk 224–5
- RDS *see* Radio Data System
- ReadL() 112–17
- ReadNextBlock() 111–17
- ReadUserData() 20
- RealEyes3d 53
- ReAlloc() 117, 170–4
- RecommendedImageSizes 183–5
- Record 89–93
- RecordFormats 91–3
- recording of audio 9, 11–12, 15–16, 23–31, 97–100, 106–7, 108–17, 122–7, 136–9, 199–205, 215–16, 220–1
- recording of video 11–12, 15–16, 23–31, 56–7, 58–61, 88–93, 220–1
- RecordingStopped 126–7
- RecordL() 110–17, 123–7, 139
- RecordTimeAvailable 94
- ReducedSize 153–7, 161
- ReductionFactor 153–7, 161
- references 229–31
- refreshing the frame 81
- regional links, RDS 205
- RegisterAudioResourceNotification 85–7
- RegisterForVideoLoadingNotification 88
- Release() 49, 51–3
- RemoveDisplayWindow() 72
- RemoveMetaDataEntryL() 100–1
- repeated clips, audio 121–2, 215–16
- ReplaceMetaDataEntryL() 100–1
- RequestStop() 114–17

- RequestTunerControl 200–7
- Reserve() 39–40, 52–3
- Reset() 174, 186–9
- resources 229–31
- RFile 118–22, 133–5, 161, 167, 211–12, 221–2
- RFs 145–6
- RIM's BlackBerry devices 2
- ringback tones, sales statistics 4
- ringtones, sales statistics 4
- Rotate() 193
- rotation 28, 57–8, 80–1, 141, 182–93
- RTP 24, 31, 230
 - see also* VoIP
- RTSP 65, 88, 230
- RunL() 145–6, 209–10, 223
- RWindowBase 71–2

- S60 6, 30, 37, 91, 102, 169, 182, 197, 214, 230–1
 - see also* Nokia
- sample rates, audio 99–100
- Samsung 5, 8
- scaling
 - see also* size factors
 - bitmaps 189–93
 - concepts 28, 76–80, 95–6, 141, 153–7, 182–93
 - performance issues 191–2
- scanning for frequencies, Tuner API 199, 203–5
- screen rectangle *see* video extent
- ScreenPlay 13
- screens 2–3, 6–7, 12–13, 71–81
 - high resolutions 12–13
 - size constraints 2–3
 - video controls 71–81
- SDKs 16, 23, 31–3, 53, 61, 67, 105, 135, 167–9, 197–8
- secondary clients, onboard camera 53
- SecureID 137
- security issues *see* platform security
- self-signed software 18–19, 20–1
- sequence files 133–5
- Series 5, 6
- servers
 - client–server 11–12, 144–5
 - Media Server 9–10, 214
- SetActive() 145–6, 176–81, 190–3
- SetAudioBitRateL() 98
- SetAudioChannelsL() 99
- SetAudioEnabledL() 87, 98
- SetAudioPropertiesL() 117
- SetAudioSampleRateL() 100
- SetAudioTypeL() 98
- SetAutoScaleL() 78–80
- SetBalanceL() 86–7
- SetCropRegionL() 77
- SetDataTypeL() 116–17
- SetDestinationDataTypeL() 125–7
- SetDisplayMode() 183–5
- SetDisplayWindowL() 71–6
- SetGainL() 99
- SetImageSource() 183–5
- SetImageTypeL() 157–8
- SetMaxClipSizeL() 94
- SetNewsAnnouncement() 205
- SetOptions() 183–5, 186–9
- SetPixel... 194–6
- SetPlayVelocityL() 69–70
- SetPlayWindow() 120–1
- SetPosition... 68, 120–2, 126–7
- SetPriorityL() 85–7, 97, 137–8
- SetQuality-Algorithm() 192
- SetRed() 196
- SetRegionalLink() 205
- SetRepeats() 121–2
- SetRotationL() 81
- SetScaleFactorL() 77–80
- SetSizeinPixels() 183–5
- SetSquelch() 203
- setting the position 68, 119–22
- SetTrafficAnnouncement() 205
- SetupL() 182–5
- SetVideoBitRateL() 94
- SetVideoEnabledL() 70, 97
- SetVideoExtentL() 75–81
- SetVideoFrameRate... 93–4, 96–7
- SetVideoFrameSize... 94
- SetVideoQualityL() 96–7
- SetVideoTypeL() 95
- SetVolume() 122, 131–5
- SetWindowClipRectL() 76
- Sharp 5
- Siemens 5
- signed software 18–21
- sine waves 132–5
- SIP 24, 31, 230
 - see also* VoIP
- Size() 150–2
- size factors
 - see also* scaling
 - decoding still images 150–7, 161

- smartphones
 - see also* mobile phones
 - constraints 2–3, 197–8
 - definition 1–2
 - developing world 3
 - examples 2–3
 - future prospects 12–14, 31–4, 206–7
 - historical background 6–12
 - introduction 1–14
 - statistics 4–5
 - Swiss Army knife analogy 2
- social networking 3–4, 13
- socket server 16
- Sony Ericsson 5, 7–8, 197, 207
 - M600i 197
 - P1 207
 - P800 7
 - P990 197
- sound types 138
 - see also* audio
- South Korea 4
- stack traces 210
- standards, references and resources 230
- StartDirectScreen-AccessL() 72–3
- starting to record a video 92
- StartVideoCapture() 51–2
- StartViewFinder... 44–6
- StationScan() 203
- StationSearchBy... 205
- StationSeek() 203, 205
- statistics 4–6
- StepFrameL() 70
- stereo to mono changes 127
- Stop 67, 89–93, 113–17, 119–22, 124–7, 202
- stop/pause sequences 62–5, 67–93, 103, 119–22, 126, 184–5
- StopAndPlay... 114–17, 126–7
- StopDirectScreen-AccessL() 72–3
- stopping a video 67, 93
- StopScan 203
- StopVideoCapture 50–3
- streaming 3–5, 9, 10–12, 55–6, 57–60, 65, 87–8, 91–2, 102–3, 106–39, 170–4, 206, 213–14, 216–17
- audio 26–7, 106–39, 206, 213–14
- best practice 213–14
- decoding still images 170–4
- delivering video 55–6, 66, 87–8
- high-performance demands 9
- statistics 4–5
- video 55–6, 66, 87–8, 91–2, 102–3, 216–17
- Swiss Army knife analogy, smartphones 2
- Symbian C++ for Mobile Phones Volume 3 (Harrison & Shackman) 105, 229, 231
- Symbian Developer 16, 60–1, 73, 209–10, 224–5, 229–30
- Symbian Foundation xvi, 6
- Symbian OS
 - background 4–14, 135, 229–31
 - best practice 209–27
 - books 105, 229, 231
 - decoder plug-ins 168–70, 189
 - encoder plug-ins 181
 - future prospects 12–14, 206–7
 - historical background 5–12
 - ICL plug-ins 168–70, 181, 189–93
 - innovative camera applications 53–4
 - licensees 5–6, 10, 11, 23–4
 - market share 5, 12
 - multi-tasking systems 135–9, 223–4
 - multimedia subsystem 4–5
 - owners 5
 - references and resources 229–31
 - statistics 5–6
 - video architecture 57–60
- Symbian OS v6.1 9, 10
- Symbian OS v7.0 9, 10, 15, 35
- Symbian OS v8 11
- Symbian OS v9 11–12, 15, 16–17, 21, 30, 31–3, 59–60, 68–9, 72, 78–9, 85–7, 138–9, 153, 167, 197, 207
- Symbian Signed 18–21
- symmetric multi-processing (SMP) 13
- synchronization issues, audio/video 59
- synchronous operations, definition 61
- syslibs 16
- SYSTEMINCLUDE 198
- TBitmapUtil 142–96
- TBmpCompression 178–81
- TBmpImageData 178–81
- TCamera... 38–54
- TCB *see* Trusted Computing Base
- TCE *see* Trusted Computing Environment
- TColorConvertor 194–6
- TDes... 163–5, 186–9
- technological developments 1–14, 206–7
- TField 204–5
- TFormat 46–8, 50

- TFrameDataBlock 164–5, 178–81
- TFrameInfo 148–52, 156–8, 162–5, 170, 172–4
- TFrameInfoState 151–2
- TGifBackgroundColor 164–5
- third-party software 3, 4, 6–7, 10, 193–6, 197, 207
- THorizontalAlign 79–80
- threads
 - controllers 217–18
 - ICL request considerations 222–3
 - multiple threads 10–12, 24–8, 218
- thumbnails
 - decoding 157–8, 186, 189
 - encoding 180–1
- TIFF 16, 28, 164–5, 169
- TImageBitmapUtil 194–6
- TImageDataBlock 164–5, 178–81
- TImageType 144
- TJpegImageData 178–81
- TJpegQTable 164–5, 178–81
- TMbmEncodeData 178–81
- TMMFDurationInfo 119
- TMMFile... 22–3, 160
- TMMFMessage-
 - Destination... 101–2
- TMMSource 22–3, 65–6, 121–2, 160, 220–1
- tone sequence files 133–4, 215
- tone utility 26–7, 129–35, 215
 - concepts 129–35, 215
 - DTMF 129–35
 - playing tone sequences 133–4, 215
 - playing tones 132–4, 215
 - security issues 132
- TOptions 144, 158–60, 176–81, 186–9
- touchscreens 6–7
- TPngEncodeData 178–81
- TPtrC 170, 211–12
- traffic announcements, RDS 205
- Transform 186–9
- TRAP 108–10, 124–7
- TRdsData 204–5
- TRect 46, 72, 149–52, 184–5
- TRequestStatus 145–96, 209–10
- TRgb 174, 195–6
- trick play modes, video 68–70
- TRotationAngle 193
- troubleshooting, video 102–3
- trust model, platform
 - security 16–21, 138–9
- Trusted Computing Base (TCB), concepts 17–21
- Trusted Computing Environment (TCE), concepts 17–21
- TSize 46, 50, 149–52, 155–7, 186–9
- TTimeInterval... 82, 94, 118–22, 133
- TUId 167–8
- Tune 200–7
- Tuner API
 - see also* radio...
 - basic use cases 201–5
 - capabilities 199–200, 205
 - classes 30, 199
 - cleanup 201
 - CMMTunerUtility 30, 199–207
 - concepts 30, 197–207
 - controllers 30, 200–1
 - DAB 30, 197, 201–2, 206
 - definition 30, 197
 - future prospects 206–7
 - getting started 198–201
 - initialization 199–200
 - mobile TV 33–4, 197, 206–7
 - playing the radio 201–2
 - priorities 200–1
 - RDS 197–9, 203–5
 - recording from the radio 202–3
 - sample code 207
 - scanning for frequencies 199, 203–5
 - tuning methods 201
 - uses 30, 198–9, 201–5
- tuner.lib 30, 198–9
- TunersAvailable 199–200
- TV 4–5, 7, 12, 33–4, 197, 206–7
 - mobile TV 33–4, 197, 206–7
 - statistics 4–5
- TVerticalAlign 79–80
- TVideoAspectRatio 96
- TVideoPlayRate-
 - Capabilities 69–70
- TVideoQuality 96–7
- types list
 - audio 98
 - video 95
- UDP 31
- UI 6, 23–4
 - see also* MOAP(S); S60; UIQ
- UIDs 61–2, 65, 66, 88, 91–3, 102, 146–8, 166–9, 175–81, 216–17
- UIQ 6, 30, 91, 169, 197, 201, 230–1
- unicast streaming 206
- Unicode 163
- unit of trust, definition 17

- universal plug and play (UPnP) 13
- URLs 57–8, 65, 87–8, 117–22
- UseLowMemory-Algorithm 192
- user-generated content 13
- UserEnvironment 20–1, 36–7, 139
- UseSharedHeap() 225–7
- ValidDecoder 173–4
- vertical positions, scaling 79–80
- VGA 82–3, 102–3
- video 2–3, 7, 10–12, 15–34, 49–52, 55–103, 210–27
 - see also* Multimedia Framework
 - APPARC 210–12
 - audio 55, 59, 84–7, 97–100, 210–12
 - balance 86–7
 - best practice 210–27
 - bit rates 83, 87, 94, 97–8, 102–3
 - capabilities 49–51, 84–5
 - capturing video 49–52
 - classes creation 63–5, 89–90
 - client APIs 57–8, 60–103, 223–4
 - clipping region 75–6, 94
 - closing a video 68, 93
 - concepts 49–52, 55–103
 - controllers 57–60, 61–81, 93–7, 216–17
 - cropping 76–7, 127, 129
 - custom commands 101–3
 - decoding video 56–7, 59–60
 - definition 55
 - delivering video 55–6, 66, 87–8
 - duration 82, 94
 - enabling video playback 70
 - enabling video recording 97
 - encoding video 56–7, 59–60
 - examples 102–3
 - files 210–11, 221–2
 - frame rates 81–2, 93–4, 148–52
 - frame size 82, 94, 102–3
 - frame snapshots 70
 - getting video information 81–4
 - graphics surfaces 73
 - metadata access 83–4, 100–1
 - MIME types 56, 65, 82, 95, 144, 146–8, 166–7, 175–81, 211–12, 216–17
 - onboard camera 49–52
 - opening a video 66, 90–3
 - pause/stop sequences 62–5, 67–93, 103
 - pixel aspect ratio 79, 95–6
 - playback of video 11–12, 15–16, 24–7, 55–61, 62–5, 67–88, 210–12, 214, 220–1
 - preparing to record 92
 - preparing a video 66–7
 - quality controls 96–7
 - recording time available 94
 - recording of video 11–12, 15–16, 23–31, 56–7, 58–61, 88–93, 220–1
 - refreshing the frame 81
 - rotation 28, 57–8, 80–1, 141, 182–93
 - scaling 28, 76–80, 95–6, 141, 182–93
 - screen output controls 71–81
 - setting the display window 71–2, 129
 - setting the video position 68
 - starting to record a video 92
 - stopping a video 67, 93
 - storage formats 56, 59–60, 87, 91–3, 212–13, 219–20, 230
 - streaming 55–6, 65, 87–8, 91–2, 102–3, 216–17
 - Symbian OS video architecture 57–60
 - synchronization issues 59
 - trick play modes 68–70
 - troubleshooting 102–3
 - types list 95
 - volume 86, 222
- video extent 73–5, 80–1
- video recorders 3
- VideoBitRateL() 83, 94
- VideoEnabledL() 97
- VideoFormatMimeType 82, 95
- VideoFrame... 82–3, 93–4
- videoplayer... 60
- VideoQualityL() 97
- viewfinder displays
 - bitmap-based viewfinder 45–6
 - concepts 43–6, 54
 - direct screen access viewfinder 44–5
- VoIP 24, 31
 - see also* RTP; SIP
- volume 86, 122, 131–5, 222
- WAV 11, 27, 107–17, 125–7, 211–12, 214, 220–1
- WBMP 28, 169, 213
- web browsing 2–3
 - see also* Internet
- white balance, onboard camera 43

- WiFi 7, 13, 83
- `WillResumePlay()` 86
- WiMAX 13
- window rectangle *see* video
 extent
- window server 71–81
- Windows
 - emulator 136, 210
 - Mobile 2
- WLAN network 83,
 102–3
- WMF 28, 169
- ‘wrapped audio’ 107
- `WriteL()` 214
- `WriteUserData()` 20
- XVID codec 56, 59, 60
- YouTube 13
- Z8, MOTORIZR
 7–8
- Z10, MOTO
 7–8
- zoom settings, onboard
 camera 42–3