WILEY

Programming PC Connectivity Applications for Symbian OS

Smartphone Synchronization and Connectivity for Enterprise and Application Developers

symbian

Ian McDowall

Programming PC Connectivity Applications for Symbian OS

Smartphone Synchronization and Connectivity for Enterprise and Application Developers

By

Ian McDowall

Reviewed by

Day Barr, Emlyn Howell, Helena Bryant, Paul Newby, Rob Falla, Simon Didcote, Tony Naggs, Zoë Martin

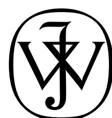
Symbian Press

Managing editor

Phil Northam

Project editor

Freddie Gjertsen



John Wiley & Sons, Ltd

Programming PC Connectivity Applications for Symbian OS

TITLES PUBLISHED BY SYMBIAN PRESS

- Programming PC Connectivity Applications for Symbian OS
Ian McDowall
0470 090537 477pp 2004 Paperback

- Symbian OS Explained
Jo Stichbury
0470 021306 416pp 2004 Paperback

- Symbian OS C++ for Mobile Phones, Volume 2
Richard Harrison
0470 871083 448pp 2004 Paperback

- Programming Java 2 Micro Edition on Symbian OS
Martin de Jode
0470 092238 498pp 2004 Paperback

- Symbian OS C++ for Mobile Phones, Volume 1
Richard Harrison
0470 856114 826pp 2003 Paperback

- Programming for the Series 60 Platform and Symbian OS
Digia, Inc.
0470 849487 550pp 2002 Paperback

- Symbian OS Communications Programming
Michael J Jipping
0470 844302 418pp 2002 Paperback

- Wireless Java for Symbian Devices
Jonathan Allin
0471 486841 512pp 2001 Paperback

Programming PC Connectivity Applications for Symbian OS

Smartphone Synchronization and Connectivity for Enterprise and Application Developers

By

Ian McDowall

Reviewed by

Day Barr, Emlyn Howell, Helena Bryant, Paul Newby, Rob Falla, Simon Didcote, Tony Naggs, Zoë Martin

Symbian Press

Managing editor

Phil Northam

Project editor

Freddie Gjertsen

John Wiley & Sons, Ltd

Copyright © 2005 by John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England
Telephone (+44) 1243 779777

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

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system for exclusive use by the purchaser of the publication. 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, 33 Park Road, 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, 22 Worcester Road, Etobicoke, Ontario,
Canada M9W 1L1

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

Library of Congress Cataloging-in-Publication Data

McDowall, Ian.
Programming PC connectivity applications for Symbian OS : smartphone synchronization and connectivity for enterprise and application developers / by Ian McDowall.
p. cm.
Includes bibliographical references and index.
ISBN 0-470-09053-7 (pbk. : alk. paper)
1. Cellular telephone systems – Computer programs. 2. Operating systems (Computers) 3. Computer input-output equipment. I. Title.
TK6570.M6M38 2004
005.26′8 – dc22
2004017257

British Library Cataloguing in Publication Data

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

ISBN 0-470-09053-7

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Biddles Ltd, King’s Lynn
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

Author Biography ix

Author's Acknowledgments xi

Symbian Press Acknowledgments xiii

1 Introduction 1

1.1 What is PC Connectivity and Why is This Book Different from Other Symbian OS Books ? 2
1.2 What This Book Will Tell You (and What It Will Not) 3
1.3 How This Book is Structured 4
1.4 Conventions Used in This Book 5
1.5 Developer Resources 5

2 A History of Symbian OS and PC Connectivity 7

2.1 A History of Symbian OS 7
2.2 PC Connectivity Using PLP 8
2.3 PC Connectivity Using TCP/IP 8
2.4 PC Connectivity Using OBEX 10

3 An Architectural Overview of PC Connectivity 11

3.1 The Bearers, TCP/IP and PPP 11
3.2 A Client-Server Model of PC Connectivity 12

4 The Symbian Connect Object Model 15

4.1 Overview 15
4.2 Functionality in SCOM and in PC Suites 15
4.3 SCOM and BAL 16
4.4 COM Programming and Language Choice 17
4.5 Error Handling 18

4.6 SCOM Class Reference 18
4.7 BAL Class Reference 33
4.8 Using SCOM in C++ and Visual Basic 37

5 An Example PC Connect Application – a File Browser 39

5.1 Overview 39
5.2 Connecting to a Phone or Emulator 39
5.3 Accessing SCOM and Connecting to a Device 48
5.4 Handling Differences Between Devices 52
5.5 Copying Files – Asynchronous Actions 53
5.6 Navigating the Filing System 58
5.7 A File Browser Application 60
5.8 Simple Actions on Files and Directories 66
5.9 Error Handling and Disconnection 77
5.10 Visual C++ Code for Application and Device Management 78
5.11 Visual C++ Code for Drive and Directory Navigation 86
5.12 Visual C++ Code for Synchronous and Asynchronous Operations 87

6 Programming for Symbian OS 89

6.1 Building a Project 90
6.2 Using the Emulator 96
6.3 Types and Naming Conventions 100
6.4 Error Handling 102
6.5 Descriptors 106
6.6 Arrays 108
6.7 Processes and Threads 109
6.8 Active Objects 110
6.9 Backwards Compatibility and Programming for Multiple Phone Types 113

7 Developing Custom Servers 117

7.1 Overview of Custom Servers 117
7.2 Limitations of Custom Servers 118
7.3 Custom Servers API 119
7.4 Protocol Conventions 123
7.5 Creating Your First Custom Server 124
7.6 Installing a Custom Server 129
7.7 Starting a Custom Server from SCOM 130
7.8 Communicating with a Custom Server 132
7.9 Asynchronous Communication 133
7.10 Debugging a Custom Server 136

8 Developing Socket Servers 137
8.1 Overview of Connectivity Socket Servers 137
8.2 An Introduction to the Server Socket Classes 138
8.3 Using the Service Broker API 141
8.4 Server Socket Classes 142
8.5 Developing an Echo Socket Server 151
8.6 Installing and Registering a Server Socket Service 161
8.7 Starting a Socket Service from SCOM 163
8.8 Communicating with a Socket Service 164
8.9 Asynchronous Communication 165
8.10 Debugging a Socket Service 165

9 Introducing SMS and Messaging Classes 167
9.1 The Message Server and MTMs 167
9.2 The Structure of Messages 170
9.3 Message Server Events and Sessions 173
9.4 SMS Specific Variations 174
9.5 Common Messaging Classes 175
9.6 SMS Specific Classes 187

10 Developing an SMS Management Connectivity Service 191
10.1 SMS Management Protocol 191
10.2 Packing and Unpacking Data 200
10.3 Obtaining Access to the Message Server and the SMS MTM 204
10.4 Listing SMS Messages and Returning Their Contents 206
10.5 Deleting and Creating SMS Messages 209
10.6 Handling Message Server Events 213
10.7 Putting the Messaging Code in a Connectivity Plug-in 215
10.8 A Command-line SMS Application 219

11 Using the Contacts Model 227
11.1 Databases and Models 227
11.2 The Contacts Model 228
11.3 Views 230
11.4 Contacts Observers 230
11.5 Synchronization and Performance Issues 231
11.6 Contacts Model API 231
11.7 A Contacts Connectivity Service 256

12 Using the Agenda Model 283
12.1 The Various Agenda Models 283

12.2 Types of Agenda Entries 284
12.3 Repeating Entries 285
12.4 Alarms 285
12.5 List and Filter Classes 286
12.6 Agenda Model API 286
12.7 An Agenda Connectivity Service 325

13 Developing a Specialized Connectivity GUI Application 347
13.1 What is Special About a GUI Application? 347
13.2 Managing Connections to Phones 347
13.3 Starting a PC Connectivity Service 351
13.4 Communicating and Managing Delays 351
13.5 A GUI SMS Application 358
13.6 A Contacts GUI Application 367
13.7 An Agenda GUI Application 384
13.8 Conclusion and Ideas for Further Development 396

14 Starting General Socket Servers 397
14.1 Communicating with a Socket Server 398
14.2 Starting a Server 400

15 Connectivity Dos and Don'ts 403
15.1 Protocol Design 403
15.2 Robustness and Defensive Design 406
15.3 Device and Service Management 407
15.4 General Development and Debugging Skills 410

Appendix 1 Developer Resources 413

Appendix 2 Specifications of Symbian OS Phones 421

Index 441

Author Biography

Ian joined Symbian in 2000 and is currently a technology architect responsible for connectivity. He has previously filled roles ranging from developer through project manager to technical manager by way of quality manager and process consultant (including presentation at international conferences).

He has an MA in Computer Sciences from Cambridge University and an MBA from Warwick University. As a software engineer for over twenty years he has been with a number of software companies and has worked on more than fifteen operating systems, developing software ranging from enterprise systems to embedded software. He is married to Lorraine and they have two children, Ross and Kelly, and a number of pets.

Author's Acknowledgments

I would like to thank the members of the PC Connectivity team and others in Symbian's Software Engineering Department who have made this book possible. In the PC Connectivity team Day Barr, Simon Didcote and Paul Newby have provided essential information and suggestions, and in other teams Emlyn Howell, Tony Naggs and David Cunardo have provided invaluable advice on the best use of Symbian's Messaging, Contacts and Agenda APIs.

I would like to thank Zoë Martin, Colin Turfus and Ian Weston for their support in promoting the wider use of PC Connectivity software.

The other reviewers have also been both diligent and constructive – Helena Bryant and Rob Falla (who also suggested the original idea for this book).

I must thank Freddie Gjertsen and Phil Northam of Symbian Press for promoting the concept of this book inside and outside Symbian and for their patient checking and support.

I would also like to thank all the engineers in Symbian and elsewhere who have made Symbian's PC Connectivity software what it is today.

Finally, I would like to thank my family who have put up with my work on this book for more than a year.

Symbian Press Acknowledgments

Symbian Press would like to thank Ian for his perseverance in adversity. And all those who reviewed the book, mentioned or otherwise. And those who worked 'behind the scenes' to allow this book to be realized. And, of course, the BA cabin-crew for always looking after the Symbian Press 'frequent flyer' so splendidly. And the wonderful Loza, Symbian Press Officer extraordinaire.

Cover concept by Jonathan Tastard.

1

Introduction

Welcome to this book on programming PC Connectivity applications for Symbian OS. PC Connectivity applications based on standard services and APIs can be created purely by programming on the PC, but more specialized applications involve programming on the Symbian OS smartphone as well as on the PC. This book will help you to create both types of application.

If you have created an application for Symbian OS, have you considered how to improve its usability by integrating it with a PC? Maybe your application could support a user interface on the PC when the Symbian OS smartphone is connected, or maybe your application could store or archive data on the PC.

If you have created an application for Windows PCs, have you considered how to improve your application by integrating it with Symbian OS smartphones? This has been considered difficult and expensive but, with the information in this book, it can be straightforward. You may be surprised at the quality of integration you can achieve just by creating PC software – for example, you could manage media files such as image, audio and video files just by using the APIs described in Chapters 4 and 5. If your application is more specialized then a small amount of Symbian OS programming may give you a unique service that increases its attractiveness (and therefore its sales).

All the examples in this book are of stand-alone PC Connectivity applications, but this is by no means the only way to create PC Connectivity applications. We will create a file browser that will provide a convenient user interface to the filing system on Symbian OS smartphones; we will create an application to read SMS messages on the smartphone and to send such messages by means of the smartphone; we will create applications to directly read and modify the Contacts and Agenda data on the smartphone.

These applications are potentially useful (I certainly use them extensively in favor of the other ways of accessing Contacts, Agenda and SMS

messages) but they are just examples of what can be done and I have deliberately kept them simple. If you want to create a fully-featured, integrated and commercialized version of these applications then I wish you good luck. However, I feel that the largest potential value of PC Connectivity applications lies in integration with other applications.

If you are an Enterprise or corporate developer this book is also aimed at you. Symbian OS provides a selection of methods to connect a smartphone to a server of which PC Connectivity is one of the cheapest and fastest.

1.1 What is PC Connectivity and Why is This Book Different from Other Symbian OS Books ?

A PC Connectivity application is an application with one part on the Symbian OS smartphone and one part on the PC. Usually, the software on the smartphone (commonly referred to as a server or service) will not have a user interface.

Most books on Symbian OS programming are concerned with developing applications with user interfaces (although some also cover server design). This book contains no information on user interface programming on Symbian OS, but it does provide specialist information on how to create services using the PC Connectivity plug-in and server APIs. As most PC Connectivity services are interfaces to existing servers on the smartphone, this book does not go into detail about server design.

The most obvious PC Connectivity software that most users will see is the PC suite that is supplied with their Symbian OS smartphone. Typical functions supported by such a PC suite include:

- synchronization with a Personal Information Manager (PIM) such as Microsoft Outlook or Lotus Notes
- backup and restore of data
- the ability to install software on the smartphone remotely from the PC.

Some manufacturers add extra features such as a configuration application or an interface to an MP3 player.

Some other books on Symbian OS programming touch on PC Connectivity, but they only describe the standard functions and do not provide information on how to create new PC Connectivity applications.

This book is aimed at software engineers creating additional applications or integrating other applications; it is not aimed at smartphone manufacturers (who have their own support channels). This book does not, therefore, cover the standard functions listed above, as I do not expect third-party developers to create alternative PC suites.

1.2 What This Book Will Tell You (and What It Will Not)

This book covers PC Connectivity for a wide range of Symbian OS smartphones but, unfortunately, not all. The reasons for the variations in PC Connectivity framework are discussed in Chapter 2. In summary, this book covers all Symbian OS smartphones that I know of based on Symbian OS v6.1 and Symbian OS v7.0 and some smartphones based on Symbian OS v7.0s. It will also apply to a large extent to smartphones based on Symbian OS v8.0 and later that include Symbian's TCP/IP PC Connectivity framework.

Because this book is aimed at PC Connectivity developers, it has less space for general-purpose Symbian OS programming than some other (commonly much larger) books. Chapter 6 explains the basics of Symbian OS programming and the later chapters cover PC Connectivity and some other APIs in detail. The developer resources that are available will provide more detail on the other APIs in Symbian OS. However, if you want to go further into Symbian OS programming then you may benefit from another book that concentrates on Symbian OS programming, such as other books published by Symbian Press.

For the Symbian OS programming I have used C++. This is the language used to develop Symbian OS and the language in which all the APIs are provided. Symbian uses C++ in ways that are slightly different from how the language is used with other operating systems, but a good grasp of object-oriented programming will be essential for any Symbian OS development.

It is certainly possible to program for Symbian OS using Java. Java is well supported by Symbian OS and is the most appropriate language for a range of purposes, but there is no PC Connectivity API for Java. However, Chapter 14 does show a way in which Java services might be accessed using a PC Connectivity framework.

For programming on the PC I have used C# as I think it is the best current language for development in Microsoft Windows. However, it is possible to use any Microsoft-supported development language as the PC Connectivity framework uses COM. The examples in this book would be easily understood by any Microsoft Visual C++ programmer and, I believe, also by Visual Basic programmers.

Because space in this book is limited, I have not provided any tuition in using C++ or C#. Many good books and online resources on both languages are available.

I have not included extensive information on using some of the programming tools such as Integrated Development Environments (in particular the debuggers). This is because I assume that the reader will already know how to use these tools.

This book is aimed at integrating Symbian OS smartphones with PCs running Microsoft Windows (multiple versions). What about other

operating systems such as Linux or Mac OS? Theoretically, there is no reason why PC Connectivity applications could not be created on these operating systems, but Symbian does not publish the protocols required and so the possibility remains more theoretical than practical. I have created some experimental PC Connectivity software on Linux using Perl, but that used undocumented protocols, required manual configuration and would not work on all bearers.

1.3 How This Book is Structured

This book is structured so as to be read from the start to the end, but it can also be used as a reference book and be dipped into.

Chapter 2 describes the history of Symbian OS in general and of Symbian OS PC Connectivity in particular. You do not need to read this to use the rest of the book, but it is only a short chapter and does provide some context if you want to understand how Symbian OS evolved to its current state.

Chapter 3 describes the architecture of Symbian OS PC Connectivity and is a basis for understanding later chapters.

Chapters 4 and 5 describe the APIs provided on the PC for Symbian OS PC Connectivity and show how to use these in creating a file browser without writing any new code on the Symbian OS smartphone.

Chapter 6 leads into the development of software on the Symbian OS smartphone. It contains a compressed tutorial on developing for Symbian C++ but it ignores aspects that will not be used in developing for PC Connectivity.

Chapters 7 and 8 describe how to create PC Connectivity services using specialized APIs. Chapter 7 covers custom servers which are used in Symbian OS v6.1 to Symbian OS v7.0s, while Chapter 8 covers the socket server APIs introduced in Symbian OS v8.0. In order to illustrate the APIs we will see how to create very simple PC Connectivity plug-ins.

Chapters 9, 10, 11 and 12 cover specific APIs and show how to create PC Connectivity services using them. These chapters cover SMS messaging, the Contacts Model and the Agenda Model and show how to create PC Connectivity plug-ins to expose these APIs to the PC.

Chapter 13 builds on the previous chapters and shows how to create an application on the PC that communicates with the services created in the previous chapters to present a GUI on the PC that integrates with the Symbian OS smartphone.

Chapter 14 is a slight diversion that discusses how to manage services that were not originally design for PC Connectivity with minimal changes.

Chapter 15 finishes the book with a selection of advice on designing and developing PC Connectivity applications based on Symbian's experience over a number of years.

1.4 Conventions Used in This Book

This book has very little in the way of conventions that are not obvious. Example code is presented in a fixed-width font and is normally highlighted:

```
void CSomeClass::someMethod()
  {
  someOtherMethod();
  }
```

The same convention is used for Symbian OS C++ as for C# or C++ for the PC. It should always be clear which is meant by the context.

In order to avoid pages of uninteresting code listings, this book shows only the example code that I believe is relevant to the text. Because Symbian OS is less commonly understood and because the code is more compact, this book normally shows more complete listings of Symbian OS C++. In a few cases I show an early version of a method and then return to it later to add more code. In these cases the unchanged code is not highlighted so that the new code can be clearly seen.

Where this book describes C# GUI code I have omitted all of the code that is created by wizards.

Where classes or members are referred to in the text of the book, they are normally shown in a `fixed-width font` to highlight them.

1.5 Developer Resources

In order to develop for Symbian OS in general and for PC Connectivity in particular, you will need compilers and other development resources.

For PC development you will need standard Microsoft development tools and developer resources: see ***www.msdn.microsoft.com***.

For Symbian OS development the tools and resources are more specialized and are not all provided directly by Symbian. The best starting point for resources and partners is ***www.symbian.com/developer*** which has links to partner websites and also those resources that Symbian does not provide.

Previously Symbian has not released software development kits (SDKs) directly to developers. Instead the smartphone manufacturers have created software development kits for their phones and released those. With the creation of platforms that span multiple phones there are now software development kits for those platforms. In a new departure, the CD that accompanies this book includes an SDK for Symbian OS v7.0s based on the TechView test user interface.

At the time of writing, Nokia provides a range of developer resources via its Nokia forum ***www.forum_nokia.com*** and ***www.series60.com***, Sony Ericsson provides SDKs and Symbian OS documentation and tools via ***http://developer.sonyericsson.com***, and Sendo provides information via ***www.sendo.com/dev***.

Another site that is worth a look is ***www.newlc.com*** which is an independent developer site. This has links to partners as well as tutorials and other resources.

The link between the Microsoft development tools and the tools for development on the Symbian OS smartphones is the PC Connectivity framework on the PC. This requires an SDK that is available directly from Symbian at ***www.symbian.com/developer/downloads/tools.html***. As this SDK is quite small we have not included it on the CD that accompanies this book; you can pick up the latest version in a few minutes from the Symbian website.

2

A History of Symbian OS and PC Connectivity

2.1 A History of Symbian OS

Symbian was formed in 1998 as an unprecedented joint venture between the largest players in the mobile telephone industry. From its inception, Symbian has been dedicated to making Symbian OS available to any smartphone manufacturer – it is not restricted to the original investors. Since 1998 the number of licensees, that is smartphone manufacturers who have licensed Symbian OS, has grown. The number of smartphone models based on Symbian OS has grown at an increasing rate and so have the numbers of actual Symbian OS smartphones shipped.

Over the last few years smartphones have become more advanced, and the spread of advanced messaging and multimedia features requires more advanced operating systems than the earlier, native, mobile phone operating systems, while increasing computing power has made the hardware required more affordable.

Symbian OS is not a complete and fully defined (and therefore limited) system – instead it allows smartphone manufacturers to develop user interfaces according to their own views and allows them to add extra hardware features such as cameras and FM radios. Current user interfaces include screens of differing sizes but all in color and include a variety of input devices such as touchscreens, jog dials, softkeys, joysticks and built-in or attachable keyboards. Symbian does not take a view on which user interface is best but allows smartphone manufacturers to innovate and compete.

One possible source of confusion is the name EPOC. This is the original name for Symbian OS when it was created by Psion. ER5 stands for EPOC Release 5 which was the first real version of Symbian OS. The name EPOC still appears in some documentation, either as an anachronism or as a reference to the underlying kernel.

Symbian OS is not static but has developed through multiple versions from ER5, Symbian OS v6.0, Symbian OS v6.1, Symbian OS v7.0,

Symbian OS v7.0s and Symbian OS v8.0, with further versions under development. Each version builds on the previous versions and as much consistency as possible is maintained, allowing for the new features in each version.

The variety of possibilities that come with Symbian OS can be bewildering and manufacturers have developed platforms that include a user interface, a selection of applications and hardware interface layers that can be used to develop multiple models of smartphones. This reduces the development cost and time for new smartphones and allows developers to create applications that can be deployed on a wide range of smartphones.

2.2 PC Connectivity Using PLP

In the first versions of Symbian OS (ER5) PC Connectivity was provided by a component called PLP, which stands for Psion Link Protocol. PLP was originally designed for RS232 serial connections and was later extended to support infrared. It used proprietary software both on the PDA or smartphone and on the PC, and it used limited-sized buffers.

Using PLP, Symbian and smartphone manufacturers were able to provide the same headline functionality that is part of PC suites in later versions: access to the filing system, backup and restore, PIM synchronization and remote software install.

From the start, PLP supported plug-ins on the smartphone to make the PC Connectivity extensible. These plug-ins were called 'custom servers'. Each function required was implemented by a plug-in (some with supporting libraries). Because of the use of plug-ins, the PC Connectivity framework allowed extra features to be added and smartphone manufacturers were able to extend their PC suites. For this reason, the PC suite for the Nokia 9210 supports plug-ins within the PC software.

2.3 PC Connectivity Using TCP/IP

PLP was used in Symbian OS v6.0, but during the implementation of Symbian OS v6.1 Symbian switched to a TCP/IP based PC Connectivity framework. The switch coincided with the introduction of Bluetooth as a bearer.

Symbian did not develop the TCP/IP based bearer but instead licensed a product named m-Router from Intuwave. m-Router has components on the smartphone and on the PC. The PC components provide a PPP implementation that is lacking from Microsoft Windows.

m-Router supports TCP/IP connections that can be used for any purpose, but it also supports its own framework for loading services. In order

to make use of existing components, m-Router is able to load custom servers (by means of a plug-in named `ectcpadapter` that we will encounter again in Chapter 7).

The use of TCP/IP connections rather than PLP allowed a number of underlying technical improvements, but the same functions were supported by the PC suites; the change was one of extended bearers and underlying technical improvements rather than extra headline functions.

The first product to use the m-Router based PC Connectivity was the Nokia 7650; its PC suite was a development of that from the 9210. Through Symbian OS v6.1, Symbian OS v7.0 and Symbian OS v7.0s, the PC Connectivity framework was improved in terms of performance and robustness but was not significantly extended. This is not to say that all the PC suites looked the same – the PC suites provided by Sony Ericsson with the P800 and P900 Symbian OS v7.0 smartphones are significantly different from those provided by Nokia. Other smartphone manufacturers have also tried different approaches to make their smartphones more attractive.

With the earliest Symbian OS smartphones, manufacturers were concerned with supplying an attractive and robust PC suite for each product, with a certain amount of innovation. When manufacturers released their second or third Symbian OS smartphones, they began to consider the value of standardization and moved towards developing PC suites to support a range of their smartphones rather than providing a different PC suite with each model of smartphone.

During 2002, Symbian introduced alternative PC software based around SCOM (Symbian Connect Object Model). SCOM was intended to require less resources when running than the previous PC software and was designed to provide standardized functionality across as many types of Symbian OS smartphone as possible. Subsequently, another layer of software called BAL (Bearer Abstraction Layer) was added to provide a standardized way of accessing connected phones and services. SCOM and BAL did not introduce any changes to any software on the smartphone – they used the existing protocols rather than adding new ones.

Since its creation, SCOM and BAL have been maintained to support as many Symbian OS smartphones as possible. At the time of writing, SCOM and BAL work with all the Symbian OS smartphones that use m-Router and the TCP/IP based services. It is impossible to guarantee this in the future as smartphone manufacturers continue to improve their products and some manufacturers will regard innovation as more important (that is, more attractive to consumers) than standardization. In some cases this will mean that individual models of Symbian OS smartphone will work well with SCOM, but will also have additional services, whereas in other cases the services may be so changed that SCOM and BAL may not work with them.

2.4 PC Connectivity Using OBEX

In the previous section I mentioned smartphone manufacturers seeking to standardize their PC suites. It is worth bearing in mind that all manufacturers of Symbian OS smartphones also make other mobile phones and smartphones, so there is an attraction to creating a PC suite that supports both Symbian OS smartphones and other, non-Symbian OS, smartphones.

However, most non-Symbian OS smartphones do not use TCP/IP to connect to Windows PCs. Instead, they use OBEX, which is well suited to exchanging small objects such as contact details and SMS messages and also supports larger objects such as file transfer.

Symbian OS includes some support for OBEX, but smartphone manufacturers have extended this and added services. This means that there is not currently a standard Symbian-supported PC Connectivity framework using OBEX, and for any development with such PC suites the developer will require support from the smartphone manufacturer.

3

An Architectural Overview of PC Connectivity

Most books on programming Symbian OS include a description of the architecture. This chapter describes the architecture of Symbian OS PC Connectivity without going into as much detail on the other Symbian OS internals.

Figure 3.1 is a deliberately simplistic view of PC Connectivity, but it shows the important features – a connection between the PC and the Symbian OS smartphone, a client application on the PC, and a server or service on the smartphone.

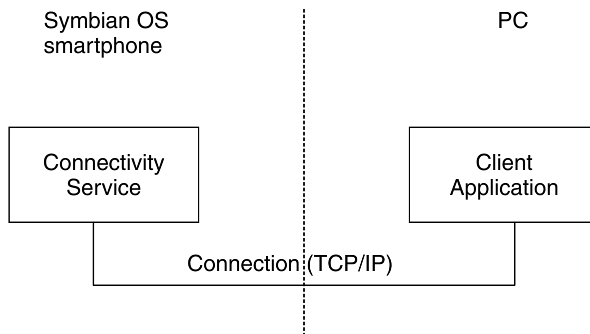


Figure 3.1 PC Connectivity

3.1 The Bearers, TCP/IP and PPP

All the PC Connectivity examples in this book run on Symbian OS smartphones that use a TCP/IP connection as described in the previous chapter. The TCP/IP connection between the relevant client and service runs over a PPP link that can be carried over RS232-serial, infrared, Bluetooth and USB. The use of PPP and TCP/IP to abstract the physical

bearers means that the rest of the PC Connectivity software (on both the PC and the smartphone) does not need to have any awareness of the actual bearer in use.

Although Microsoft Windows uses PPP to connect user PCs to servers, it does not include a PPP implementation that will natively run over infrared, Bluetooth and USB as well as RS-232 serial. That is why Symbian PC Connectivity uses m-Router to provide a PPP implementation.

The good news is that all the difficult problems (and handling the various Microsoft and third-party communications stacks can cause a range of problems) are hidden from the developer who is creating new PC Connectivity services.

3.2 A Client-Server Model of PC Connectivity

Given a TCP/IP connection, we need two more components – the software to run at each end of the connection. With a PC Connectivity application, one of the software components runs on the Symbian OS smartphone and the other runs on the PC.

The normal way of describing these components is to call the software running on the smartphone a service or a server and to call the software running on the PC a client. These names match the behavior of most PC Connectivity applications. It is possible to reverse these roles and have a client on the smartphone using a service on the PC, but this is less common and there is less support for it.

It is worth bearing in mind that this client-server model of PC Connectivity hides additional levels of complexity. Symbian OS uses the client-server model extensively as an internal pattern – servers are used to control access to shared resources throughout Symbian OS. Therefore, a PC Connectivity server or service running on the smartphone is almost certainly a client of one or more other servers on the smartphone.

In Chapters 4 and 5 we will use services that access the filing system on the smartphone; these use PC Connectivity services that make use of the Symbian OS file server. In Chapters 9–12 we will create further PC Connectivity services that make use of the Message server and the Contacts and Agenda servers.

Symbian OS provides a number of PC Connectivity services as standard in order to provide the functions expected from a PC suite. It is possible to access these directly from the PC, but this is not recommended for two reasons.

Firstly, Symbian has not published documentation for the protocols, and Symbian and the smartphone manufacturers have made improvements and alterations across the different versions of Symbian OS and even between different smartphone models based on one version of Symbian OS. Therefore, although it has been done, reverse engineering

the protocols is difficult and prone to unpredictable differences between smartphone models.

Secondly, Symbian provides a layer of middleware (called SCOM) described in Chapter 4 and used in Chapter 5 to access the standard services. SCOM has the task of handling protocol differences and also exposes an API that is much easier to use than driving the protocols directly.

If you want to create a new PC Connectivity service (as we will in later chapters) then you have a series of challenges:

1. You need to create the software on the Symbian OS smartphone to actually provide the service.
2. You need to create the software on the PC to use the service.
3. You need a way of starting the service on the smartphone when required (you do not want to have the service running when it is not required).
4. You need a way of establishing a connection between the PC software and the service.

Challenges 3 and 4 are addressed by the Symbian OS PC Connectivity framework. In Symbian OS v6.1 to Symbian OS v7.0s you can create the service on the smartphone as a custom server as described in Chapter 7. The PC Connectivity framework then provides methods to load the server and connect to it. In Symbian OS v8.0 onwards the custom server APIs are replaced by socket server APIs described in Chapter 8. In both of these cases I will show the commands required to use your services.

As an alternative, it is possible to create your service as a standard TCP/IP socket server that knows nothing about PC Connectivity. In this case the challenge is to start the service and connect to it, and this is covered in Chapter 14.

4

The Symbian Connect Object Model

4.1 Overview

SCOM (the Symbian Connect Object Model – pronounced 'escom') is a reusable software component that allows developers to more easily produce applications that incorporate connectivity with Symbian OS smartphones. While SCOM does this by abstracting core connectivity features, it also provides the ability for developers to access other services on the phone which may be developed either by themselves or by a third party. SCOM is an out-of-process COM server that supports multiple clients. SCOM is not an application that can be directly used by an end-user. Instead, some form of application must be created that uses SCOM in a way that is helpful to the end-user. This chapter describes the functionality provided by SCOM that can be used by an application.

4.2 Functionality in SCOM and in PC Suites

SCOM does not provide all the functions that a user might expect. It provides functionality to manage device connections and services, and it provides simple access to some core services that Symbian considers should be common to all Symbian OS smartphones.

SCOM was originally created with the needs of smartphone manufacturers in mind. These Symbian licensees have to provide a PC suite to accompany their smartphones. Typically, the suites include the following functionality:

- backup – copying files that include data, settings and installed applications from the smartphone to the PC
- restore – restoring the files that include data, settings and installed applications back to the smartphone to restore it to a previous state
- installation of new software (Symbian OS applications or Java applications) on the smartphone

- synchronization with PIMs on the PC to keep contacts and calendar data up to date
- some form of image or sound file management that requires the ability to copy files to and from the smartphone.

Often, Symbian OS smartphone manufacturers will provide additional functionality in order to give their smartphones a competitive advantage, but Symbian cannot predict this functionality and so SCOM cannot directly support it (although it does provide the means to access any additional services by means of stream interfaces).

SCOM provides the following functionality directly:

- backup and restore
- file management
- software install.

PIM synchronization is not directly provided by SCOM. It, instead, allows specialized synchronization software access to services on the Symbian OS smartphone.

It is possible for any developer with sufficient skill and resources to create a complete PC suite based on SCOM, but Symbian does not regard that as sensible. You should assume that the smartphone manufacturers or specialist partners will create their own PC suites and that it will not be sensible to compete directly with them. Instead, developers should focus their attentions on creating applications that complement the PC suites provided with the smartphones. Therefore, this book shows how to carry out functions useful to third-party developers and does not attempt to show how to create a full PC suite. The backup and restore and software installation functionality are omitted from this chapter, and the protocols used for them and for PIM synchronization are not covered.

The main areas of functionality covered by this book are:

- managing connections to devices and starting and using services on Symbian OS smartphones
- file management functions on Symbian OS smartphones.

4.3 SCOM and BAL

SCOM is the higher-level API provided to manipulate devices and their filing systems. BAL (the Bearer Abstraction Layer) is a slightly lower-level API that manages device connection and disconnection and services on the device. SCOM uses BAL to provide its own API (take a look at the

respective device properties and the mapping will become apparent). It is possible to use SCOM without making direct use of BAL – indeed, this is how SCOM was originally intended to be used. However, the BAL service API is slightly more efficient, in terms of performance, than that of SCOM and so the developer may choose to use BAL for some operations. In later chapters we will start up services on the phone, using SCOM and BAL, which are then accessed by means of Windows sockets in various guises.

Figure 4.1 gives a simplified view of the components that a PC Connect application interacts with.

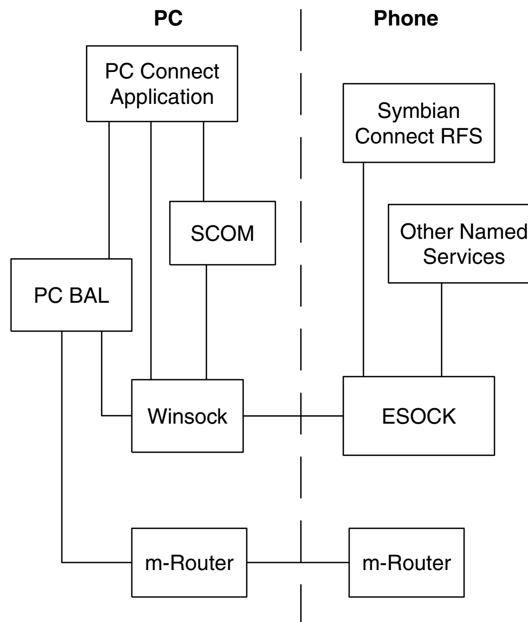


Figure 4.1 SCOM and BAL

4.4 COM Programming and Language Choice

SCOM and BAL are built as COM servers because this provides the simplest way to access their functionality. It also allows SCOM and BAL to be used from any COM-compatible language and so makes them available to the widest possible set of developers. On a PC this means that just about any development language can access SCOM and BAL.

This book does not attempt to provide an introduction to COM – the bookstores have shelves of books on COM (and all its variants) and it would be wasteful to reproduce their contents here. This book does include specific guidance on using SCOM and BAL in several languages,

so you will be able to use SCOM and BAL even if you have never used COM.

Similarly, this book is not intended as a tutorial in C#, C++ or Visual Basic because there are even more books on these subjects than on COM. The examples in the subsequent chapters are mostly written in C# and the logic should be apparent to any developer, although the way COM is used in different languages means that the actual class and method names will vary slightly. Sufficient examples are provided in Visual C++ and in VB to get developers started in those languages; standard IDE tools will then allow you to obtain the detail that you need on class and method names.

4.5 Error Handling

All the methods provided by SCOM and BAL can fail with bad `HRESULT`s and developers should check the return values. In C# these errors are thrown as exceptions that must be caught.

SCOM provides rich error information to clients by means of `IError-Info`, but these error descriptions are not localized. Therefore, they must not be displayed to the user – they are intended just as debugging aids.

4.6 SCOM Class Reference

This section lists the classes that make up SCOM and BAL that are intended for use by third-party developers and describes the API for these classes. It omits some classes and APIs that are intended only for use by smartphone manufacturers (these can be accessed using the type libraries, but I suggest that you ignore them).

You will see that some of the class names are of the form <name><number>, for example `ISCDevice2`. These are classes that have been extended. SCOM developers follow the rules for COM development and so, once the interface to a class has been defined and published, they will not change it. However, there have been cases where additional functionality has been desirable and so classes have been extended by defining a new class that replaces the old one. In such cases you should normally use the 'latest' class to have access to the most functionality.

All the classes and types listed are part of the `SymbianConnect` namespace. The types of members and arguments are given in C# terminology; it should be straightforward to convert these to the appropriate types for C++ or VB. In any case, the type libraries will provide information on the types in a language-specific form.

Figure 4.2 is a simplified view of the major SCOM classes and interfaces that you will use.

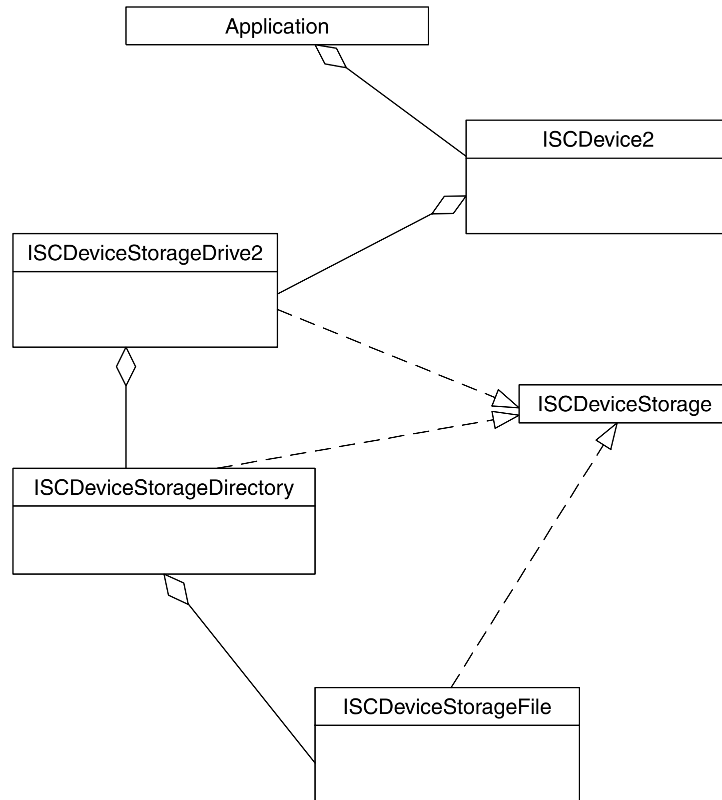


Figure 4.2 SCOM Classes

4.6.1 SCOM Application, Connection and Device Classes

These classes handle an SCOM application (which provides access to connected devices and provides a basis for event handlers) and connected devices.

Class `SymbianConnect.Application` – This class implements `ISCApplication` and `ISCEvents` and provides access to connected devices and to SCOM events.
Member Variables
`ISCDeviceCollection ConnectedDevices` This read-only member provides access to the set of currently connected devices. See also `DeviceConnected` and `DeviceDisconnected` events.
Event Handlers (part of `ISCEvents` interface – described in Section 4.6.3)

`DeviceConnected` This event handler is called when a new device is connected. See `ISCEvents_DeviceConnectedEventHandler`.
`DeviceDisconnected` This event handler is called when a device is disconnected. See `ISCEvents_DeviceDisconnectedEventHandler`.
`DeviceCopyStorageFileProgress` This event handler is called to report progress during a file copy operation. See `ISCEvents_DeviceCopyStorageFileProgressEventHandler`.
`DeviceCopyStorageFileError` This event handler is called when an error occurs during a file copy operation. See `ISCEvents_DeviceCopyStorageFileErrorEventHandler`.
`DeviceCopyStorageFileExistingFileFound` This event handler is called when a file copy operation encounters an existing file with the same name as the target. See `ISCEvents_DeviceCopyStorageFileExistingFileFoundEventHandler`.
`DeviceCopyStorageFileComplete` This event handler is called when a file copy operation is complete. See `ISCEvents_DeviceCopyStorageFileCompleteEventHandler`.

Class `ISCDevice2` – This interface provides access to the properties of a connected device and its storage.
Member Variables
`string ConnectionBearer` This read-only member provides a string that describes the transport used to connect the device.
`string Manufacturer` This read-only member provides the name of the manufacturer of the device.
`string Model` This read-only member provides the name of the model of the device. This should be used in conjunction with the manufacturer.
`string Id` This read-only member provides a unique identifier for the device. It is guaranteed to be different from any other device and can be used to identify it. Commonly, it is the IMEI number of the smartphone.

`ISCDeviceStorageDriveCollection StorageDrives` This read-only member provides the set of drives owned by the device. This member is the root for navigation of the filing system on the device.
`ISCDeviceStorage Storage [string]` This read-only member is used to directly access a directory or file on the device. It is possible to access any file or directory on the device by navigating through the directory tree, but this can be tedious. Instead, this member is used with the full path of the file or directory required (directories should be terminated with a trailing backslash, `\`) and returns an object that can be cast to an `ISCDeviceStorageDirectory` or `ISCDeviceStorageFile`.
`bool IsActive [string]` This read-only member is obsolete and should not be used. It is used by some legacy synchronization software.
Member Methods
`void SynchroniseDateTime ()` This method synchronizes the device date and time with that of the PC. It is commonly used as a part of synchronization and backup operations.
`ISCSequentialStream OpenDeviceService (string aServiceName)` This method attempts to open a service on the device by name and returns a stream that can be used to communicate with the service. This is how SCOM provides access to lower-level or third-party services on the smartphone. `aServiceName` – the name of the service to be started on the device. returns – a stream object if successful or null if unsuccessful.
`SCAsyncStreamSink OpenAsyncDeviceService(string aServicename)` This method attempts to open a service on the device by name and returns a stream that can be used to communicate with the service asynchronously. This is how SCOM provides asynchronous access to lower-level or third-party services on the smartphone. `aServiceName` – the name of the service to be started on the device. returns – an asynchronous stream object if successful or null if unsuccessful.
`void SetActive(string)` This method sets a device as the active device. This concept is used only by some legacy synchronization software and should not be used elsewhere.

Class `ISCDeviceCollection`
This class is a collection of `ISCDevice` objects. It can be accessed by standard iterators. Note that its first index is 1, not 0.

4.6.2 SCOM Storage Classes

Enumerated Type `ScStorageType`
• `scDrive`
• `scDirectory`
• `scFile`

Class `ISCDeviceStorage` – This interface is the base for drives, directories and files.
Member Variables
`ScStorageType Type` This read-only property gives the type of the device storage object.
`string Path` This read-only property gives the path of the device storage object. Note that paths always start with a drive letter and a colon, directory paths always terminate with a backslash (`\`), and file paths never terminate with a backslash.

Flag Type `ScDriveAttributes` Not all of these attributes may be applicable to a smartphone.		
• `scDriveAttLocal`	=	`0x01`
• `scDriveAttROM`	=	`0x02`
• `scDriveAttRedirected`	=	`0x04`
• `scDriveAttSubsted`	=	`0x08`
• `scDriveAttInternal`	=	`0x10`
• `scDriveAttRemovable`	=	`0x20`
• `scDriveAttRemote`	=	`0x40`
• `scDriveAttTransaction`	=	`0x80`

Enumerated Type `ScDriveBatteryState`
• `scBatteryGood`

- `scBatteryLow`
- `scBatteryNotSupported`

Flag Type `ScMediaAttributes`

- `scMediaAttVariableSize` = 0×01
- `scMediaAttDualDensity` = 0×02
- `scMediaAttFormattable` = 0×04
- `scMediaAttWriteProtected` = 0×08
- `scMediaAttLockable` = 0×10
- `scMediaAttLocked` = 0×20
- `scMediaAttHasPassword` = 0×40

Enumerated Type `ScMediaType`

- `scMediaCdRom`
- `scMediaFlash`
- `scMediaFloppy`
- `scMediaHardDisk`
- `scMediaNotPresent`
- `scMediaRam`
- `scMediaRemote`
- `scMediaRom`
- `scMediaUnknown`

Class `ISCDeviceStorageDrive2` – This interface provides access to the properties of a drive and access to the directories on the drive.

Member Variables

`int Attributes`
This read-only property is a combination of zero or more `ScDriveAttributes` flags. This property does not always provide meaningful values, so test it with specific devices.

`ScDriveBatteryState BatteryState`
This read-only property indicates whether or not the drive supports a battery and, if so, its state. This property does not always provide meaningful values, so test it with specific devices.

`int CapacityHigh`
This read-only property is the high 32-bits of the drive capacity in bytes. Because of COM automation compatibility, this is returned as a signed value and must be cast to an unsigned value before combining it with the other half of the value.

`int CapacityLow`
This read-only property is the low 32-bits of the drive capacity in bytes. Because of COM automation compatibility, this is returned as a signed value and must be cast to an unsigned value before combining it with the other half of the value.

`int FreeSpaceHigh`
This read-only property is the high 32-bits of the drive free-space in bytes. Because of COM automation compatibility, this is returned as a signed value and must be cast to an unsigned value before combining it with the other half of the value.

`int FreeSpaceLow`
This read-only property is the low 32-bits of the drive free-space in bytes. Because of COM automation compatibility, this is returned as a signed value and must be cast to an unsigned value before combining it with the other half of the value.

`long MediaAttributes`
This read-only property is a combination of zero or more `ScMediaAttributes` flags.
This property does not always provide meaningful values, so test it with specific devices.

`ScMediaType MediaType`
This read-only property gives the type of media mounted on the drive.
This property does not always provide meaningful values, so test it with specific devices.

`string Path`
This read-only property gives the path of the device storage object.

`ISCDeviceStorageDirectory RootDirectory`
This read-only property gives the directory at the root of the drive.

`ScStorageType Type`
This read-only property gives the type of the device storage object.

`int UniqueId`
This read-only property gives the unique identifier for the drive. This is meaningful only for some types of removable drive and it changes after formatting with some devices.

`string VolumeLabel`
This writable property gives the volume label of the drive. Setting this property will set the volume label of the drive.

Member Methods
`int Format ()` This asynchronous method initiates a format operation on the drive. The return value is the request ID that will be returned by event handlers. Please note that SCOM may not be able to format the `c:` drive because it may contain files which are necessary to maintain the connection. It is probably unwise to try to format the `c:` drive anyway, because it contains essential data.
`void Refresh ()` SCOM caches information about drives, directories and files to provide fast access to that information, as it would be slow to retrieve that information on demand whenever required. This method refreshes that stored information about a drive. It is likely to be most useful if the drive is a removable drive such as an MMC card.

Class `ISCDeviceStorageDriveCollection`
This class is a collection of `ISCDeviceStorageDrive` objects. It can be accessed by standard iterators. Note that its first index is 1, not 0.

Class `ISCDeviceStorageDirectory` – This interface provides access to the properties of a directory and the child directories and files that it owns, and supports a range of operations on the directory.
Member Variables
`ScStorageType Type` This read-only property gives the type of the device storage object.
`string Path` This read-only property gives the path of the device storage object.
`ISCDeviceStorage Parent` This read-only property gives the parent of the device storage object. This will be an `ISCDeviceStorageDirectory` for all directories except root directories for which it will be an `ISCDeviceStorageDrive`.
`ISCDeviceStorageDirectoryCollection ChildDirectories` This read-only property gives the collection of child directories.
`ISCDeviceStorageFileCollection ChildFiles` This read-only property gives the collection of files in the directory.

Member Methods

`int CopyFileFromPC (string aFileToCopy)`
This asynchronous method copies a file from the PC to the device.
- `aFileToCopy` – the name of the file on the PC to copy to the directory.
- returns – the request ID for the file copy operation. This will be returned by subsequent file copy events.

`void Rename (string aNewName)`
This method attempts to rename the directory.
- `aNewName` – the new name for the directory. This can be either a fully qualified directory name on the same drive ending with a backslash or an unqualified valid directory name in the same parent directory.

`void Delete ()`
This method attempts to delete the directory. If the directory is not empty it can still be deleted – in fact SCOM will recursively delete all child directories and files and then delete the empty directory. This method should be used with care.

Class `ISCDeviceStorageDirectoryCollection`

This class is a collection of `ISCDeviceStorageDirectory` objects. It can be accessed by standard iterators.
Note that its first index is 1, not 0.

Class `ISCDeviceStorageFile` – This interface provides access to the properties of a file and supports a range of operations on the file.

Member Variables

`ScStorageType Type`
This read-only property gives the type of the device storage object.

`string Path`
This read-only property gives the path of the device storage object.

`ISCDeviceStorage Parent`
This read-only property gives the parent of the device storage object. This will be an `ISCDeviceStorageDirectory` for all files.

`string FileName`
This read-only property is the name of the file. Although it is read-only as a property, it can be altered using the `Rename()` method.

`int Size`
This read-only property is the size of the file.

`DateTime LastModified` This read-only property is the date and time at which the file was last modified.
`int Attributes` This writable property is the attributes of the file. It has the standard Win32 meaning (see the `System.IO.FileAttributes` type), although not all the Win32 attribute bits are meaningful for files on a Symbian OS phone.
`string MIMEType` This read-only property is the MIME type of the file. This is not reliable and should not be used.
Member Methods
`int CopyToPC (string aToPath)` This asynchronous method copies a file from the PC to the device. `aToPath` – the name of the directory on the PC to which the file will be copied. returns – the request ID for the file copy operation. This will be returned by subsequent file copy events.
`void Rename (string aNewName)` This method attempts to rename the file (a read-only file can be renamed). `aNewName` – the new name for the file. This can be either a fully qualified file name on the same drive or an unqualified valid file name in the same parent directory.
`void Delete ()` This method attempts to delete the file (a read-only file cannot be deleted).

Class `ISCDeviceStorageFileCollection`
This class is a collection of `ISCDeviceStorageFile` objects. It can be accessed by standard iterators. Note that its first index is 1, not 0.

4.6.3 SCOM Device Connection and Storage Event Handling

All these events are associated with a `SymbianConnect.Application` object. Each event is associated with a delegate, an event handler type and a member variable of `SymbianConnect.Application`.

Device connection and disconnection events can occur at any time, but file copy and format events occur only in response to an asynchronous method call (`ISCDeviceStorageFile.CopyToPC` or `ISCDeviceStorageDirectory.CopyFromPC` for file copying events, or `ISCDeviceStorageDrive2.Format` for format events), and the Request ID returned with the event will match that returned by the asynchronous method.

4.6.3.1 Device Connection and Disconnection Events

Event	**`DeviceConnected`**
Event Handler	`ISCEvents_DeviceConnectedEventHandler`
Application Member Variable	`DeviceConnected`
Delegate `void OnDeviceConnected (ISCDevice aNewDevice)` `aNewDevice` – reference to newly connected device.	

Event	**`DeviceDisconnected`**
Event Handler	`ISCEvents_DeviceDisconnectedEventHandler`
Application Member Variable	`DeviceDisconnected`
Delegate `void OnDeviceDisconnected (string aDeviceId)` `aDeviceId` – device ID of disconnected device (cannot provide `ISCDevice` reference because it has been removed).	

4.6.3.2 File Copy Events

Event	**`DeviceCopyStorageFileProgress`**
Event Handler	`ISCEvents_DeviceCopyStorageFileProgress EventHandler`
Application Member Variable	`DeviceCopyStorageFileProgress`
Delegate `void OnDeviceCopyStorageFileProgress (int aRequestId, string aFrom, string aTo, int aPercentComplete, out bool aCancel)` `aRequestId` – request ID for the file copy operation. `aFrom` – name of the source file. `aTo` – name of the destination file. `aPercentComplete` – percentage complete (may include values of 0 or 100). `aCancel` – allows the copy operation to be canceled if set to true.	

Enumerated Type `ScErrorDescription`
• `scUnexpectedError` • `scInvalidPCArchivePath` • `scBackupServiceOpenFail` • `scCloseDeviceAppsFail` • `scBackupVersionCheckFail` • `scAnalyseDifferencesFail` • `scDeviceDisconnected` • `scPathNotFound` • `scFileAlreadyExists` • `scDiskFull` • `scRebootingPhoneNotFound`

Event	**`DeviceCopyStorageFileError`**
Event Handler	`ISCEvents_DeviceCopyStorageFileError EventHandler`
Application Member Variable	`DeviceCopyStorageFileError`
Delegate `void OnDeviceCopyStorageFileError (int aRequestId, ScErrorDescription aErrorDesc, int aErrorCode, bool aCanContinue, out bool aContinue)` `aRequestId` – request ID for the file copy operation. `aErrorDesc` – description (in enumerated form) of the error. Note that this enumerated type may be extended in the future and so an application should respond gracefully to unrecognized values. `aErrorCode` – further error code. This is an `HRESULT` code from whichever function caused the error. `aCanContinue` – set to true if the copy operation can continue. `aContinue` – allows the copy operation to continue if set to true.	

Enumerated Type `ScOverwrite`
• `scOverwriteNo` • `scOverwriteNoAll`

- scOverwriteYes
- scOverwriteYesAll

Event	**DeviceCopyStorageFileExistingFileFound**
Event Handler	ISCEvents_DeviceCopyStorageFileExisting FileFoundEventHandler
Application Member Variable	DeviceCopyStorageFileExistingFileFound

Delegate
```
void OnDeviceCopyStorageFileExistingFileFound(int aRequestId,
string aFileName, DateTime aTargetDate, int aTargetFileSize,
DateTime aSourceDate, int aSourceFileSize, out ScOverwrite
aOverwrite, out bool aCancel)
```
`aRequestId` – request ID for the file copy operation.
`aFileName` – name of the file being copied.
`aTargetDate` – date and time associated with the target file.
`aTargetFileSize` – size of the target file.
`aSourceDate` – date and time associated with the source file.
`aSourceFileSize` – size of the source file.
`aOverwrite` – set to control whether the file should be overwritten or not.
`aCancel` – allows the copy operation to be canceled if set to true.

Event	**DeviceCopyStorageFileComplete**
Event Handler	ISCEvents_DeviceCopyStorageFileComplete EventHandler
Application Member Variable	DeviceCopyStorageFileComplete

Delegate
```
void OnDeviceCopyStorageFileComplete
(int aRequestId, int aCompletionCode)
```
`aRequestId` – request ID for the file copy operation.
`aCompletionCode` – completion code for the file copy operation. This will be an `HRESULT` of `S_OK` for success or a failure code for a failed attempt. The best information about a failure can be derived from the error event handler.

4.6.3.3 Format Storage Events

Enumerated Type **ScFormatStorageProgress**
• scFormatStorageFormatting

Event	**`DeviceFormatStorageDriveProgress`**
Event Handler	`ISCEvents_DeviceFormatStorageDriveProgress EventHandler`
Application Member Variable	`DeviceFormatStorageDriveProgress`
Delegate `void DeviceFormatStorageDriveProgress (int aRequestId, ScFormatStorageProgress aProgress, string aAdditionalInformation, int aPercentComplete)` `aRequestId` – request ID for the format operation. `aProgress` – progress indicator. `aAdditionalInformation` – the drive being formatted (e.g. `"d:"`). `aPercentComplete` – format percentage complete (may include values of 0 or 100).	

Event	**`DeviceFormatStorageDriveError`**
Event Handler	`ISCEvents_DeviceFormatStorageDriveError EventHandler`
Application Member Variable	`DeviceFormatStorageDriveError`
Delegate `void DeviceFormatStorageDriveError (int aRequestId, ScErrorDescription aErrorDescription, int aErrorCode)` `aRequestId` – request ID for the format operation. `aErrorDesc` – description (in enumerated form) of the error. `aErrorCode` – further error code. This is an `HRESULT` code from whichever function caused the error. N.B. There is no option to deliberately cancel a format operation.	

Event	**`DeviceFormatStorageDriveComplete`**
Event Handler	`ISCEvents_DeviceFormatStorageDriveComplete EventHandler`
Application Member Variable	`DeviceFormatStorageDriveComplete`
Delegate `void DeviceFormatStorageDriveComplete (int aRequestId, int aCompletionCode)` `aRequestId` – request ID for the format operation. `aCompletionCode` – completion code for the format operation. This will be an `HRESULT` of `S_OK` for success or a failure code for a failed attempt. The best information about a failure can be derived from the error event handler.	

4.6.4 SCOM Services and Streams

Class `ISCSequentialStream` – This interface supports synchronously reading data from and writing data to a service running on the Symbian OS phone. It can be obtained using the `ISCDevice2.OpenDeviceService` method. This interface also supports the standard `ISequentialStream` interface which is not automation compatible but is easier to use from C++.
Member Methods
`int Read (int aBytesToRead, ref object aBuffer)` This method attempts to read data from an active sequential stream. It will block until there is data to read or until the read times out. `aBytesToRead` – the number of bytes expected. `aBuffer` – an object that boxes an array of type byte[] to receive the data. returns – the number of bytes read.
`int Write (object aBuffer)` This method attempts to write data to an active sequential stream. It will normally return immediately but the data may be buffered. `aBuffer` – an object that boxes an array of type byte[] containing the data to write. The size of the array controls how much data is written. returns – the number of bytes written.

Class `SCAsyncStreamSink` – This interface supports asynchronously reading data from and writing data to a service running on the Symbian OS phone. It can be obtained using the `ISCDevice2.OpenAsyncDeviceService` method.
Member Methods
`void Read (int aBytesToRead)` This method initiates a read operation on the active service. When the read has completed, the `OnRead` event will be triggered. `aBytesToRead` – maximum number of bytes to read from the service.
`void Write (object aBuffer)` This method initiates a write operation on the active service. When the write has completed, the `OnWrite` event will be triggered. `aBuffer` – an object that boxes an array of type byte[] to be written. The size of the array controls how much data is written.
Event Handlers
`ISCSequentialStreamSink_OnReadEventHandler OnRead` This event is triggered when a read operation has completed. See `ISCSequentialStreamSink_OnReadEventHandler`.

`ISCSequentialStreamSink_OnWriteEventHandler OnWrite` This event is triggered when a write operation has completed. See `ISCSequentialStreamSink_OnWriteEventHandler`.

Event	**`OnRead`**
Event Handler	`ISCSequentialStreamSink_OnReadEventHandler`
Delegate `void OnRead (object aBuffer, int aError)` This event is triggered when an asynchronous read operation has completed. `aBuffer` – an object that boxes an array of type byte[] containing the data read. `aError` – an error code for the read operation. This is a standard `HRESULT` and zero indicates success.	

Event	**`OnWrite`**
Event Handler	`ISCSequentialStreamSink_OnWriteEventHandler`
Delegate `void OnWrite (int aError)` This event is triggered when an asynchronous write operation has completed. `aError` – an error code for the write operation. This is a standard `HRESULT` and zero indicates success.	

4.7 BAL Class Reference

4.7.1 BAL Applications and Devices

These classes provide access to connected devices and to device connection and disconnection events. Although SCOM uses BAL, there is no direct access from SCOM to the BAL classes. If you wish to use both SCOM and BAL (probably to start services through BAL for performance reasons rather than via SCOM) then you will need to implement a framework to manage BAL devices in parallel to that managing SCOM devices. It is quite possible to use SCOM without ever directly using BAL.

Class `BALApplication` – This class provides access to connected devices and to BAL events.
Member Variables
`ISCBALDeviceCollection ConnectedDevices` This read-only member provides access to the set of currently connected devices. See also the `DeviceListChanged` event.

Event Handlers
`DeviceListChanged` This event handler is called when a device is connected or disconnected. See `ISCBALEvents_OnDeviceListChangedEventHandler`.

Event	**`DeviceListChanged`**
Event Handler	`ISCBALEvents_OnDeviceListChangedEvent-Handler`
Application Member Variable	`DeviceListChanged`
Delegate `void DeviceListChanged()` This event handler is called whenever a device is connected or disconnected. It is not provided with any information about which device(s) may have connected or disconnected – the event handler must use the `BALApplication.ConnectedDevices` member that will have been updated.	

Class `ISCBALDevice` – This interface provides access to the properties of a connected device and allows services to be started on it.
Member Variables
`string ConnectionBearer` This read-only member provides a string that describes the transport used to connect the device.
`string Manufacturer` This read-only member provides the name of the manufacturer of the device.
`string Model` This read-only member provides the name of the model of the device. This should be used in conjunction with the manufacturer.
`string Id` This read-only member provides a unique identifier for the device. It is guaranteed to be different from any other device and can be used to identify it. Commonly, it is the IMEI number of the smartphone.
`bool Active` This writable property sets the device as active – this is an obsolete concept that should not be used.
`ISCBALDeviceServiceCollection Services` This writable property contains the set of services running on the device.

Class `ISCBALDeviceCollection`
This class is a collection of `ISCBALDevice` objects. It can be accessed by standard iterators. Note that its first index is 1, not 0.

4.7.2 BAL Services and Streams

The behavior of services varies slightly with versions of Symbian OS. Before Symbian OS v8.0 the set of services returned by the `ISCBALDevice.Services` member is the set of pipe processors configured on the smartphone. The only one of these that is likely to be useful is `ectcpadapter` (see Chapter 8), which can be accessed without explicitly being started (calling `Start()` on them will not cause any errors – it is just unnecessary). From Symbian OS v8.0 the set of services is the set that can potentially be started using the `Start()` method.

An `ISCBALSequentialStream` object can be obtained using the `Services` member of an `ISCBALDevice`. The object can be used to read from and write to the device service synchronously, or it can be cast to an `ISCSequentialStreamAsync` object.

Class `ISCBALDeviceServiceCollection`
This class is a collection of `ISCBALDeviceService` objects. It can be accessed using standard iterators to find out which services are available. It can also be accessed by service name – this will return a service which can be `Start()` ed. Note that its first index is 1, not 0.

Class `ISCBALDeviceService` – This interface provides access to the properties of a service that can be `Start()` ed.
Member Variables
`string Name` This read-only property is the name of the service.
`int Version` This signed 32-bit integer is the version number of the service; the meaning of this depends on the definition of the protocol defined for the service. Note that version numbers are sometimes not well maintained for pre-v8.0 Symbian OS smartphones, so this value may not be reliable – check with specific devices.
`string IPAddress` This string is the IP address that can be used to communicate with the service using sockets directly. Note that this may not be the IP address that you expect – the connectivity transport may implement some form of address translation. The only guarantee is that connecting to the provided IP address and port number will connect to the service.

`ushort Port`
This unsigned 16-bit integer is the TCP port number that can be used to communicate with the service using sockets directly. Note that this may not be the same as the port number that the service is using on the device, owing to address translation by the connectivity transport.

Member Methods

`void StartService ()`
For devices from Symbian OS v8.0 onwards, this method starts the service. Once this method has been called, the IP address and port number can be obtained to access the service. On pre-Symbian OS v8.0 devices it has no effect.

`ISCBALSequentialStream StartServiceOnStream ()`
This starts the service and returns a sequential stream that allows direct `Read()` and `Write()` operations. This is likely to be the best way of accessing a service unless you have a specific need to access sockets directly. An asynchronous stream can be obtained by casting the result.

Class `ISCBALSequentialStream` – This interface supports reading data from and writing data to a service running on a Symbian OS phone.

Member Methods

`int Read (int aBytesToRead, ref object aBuffer)`
This method attempts to read data from an active sequential stream. It will block until there is data to read or until the read times out.

- `aBytesToRead` – the number of bytes expected.
- `aBuffer` – an object that boxes an array of type byte[] to receive the data.
- returns – the number of bytes read.

`int Write (object aBuffer)`
This method attempts to write data to an active sequential stream. It will normally return immediately, but the data may be buffered before being sent.

- `aBuffer` – an object that boxes an array of type byte[] containing the data to write. The size of the array controls how much data is written.
- returns – the number of bytes written.

Class `BALApplicationAsyncStream` – This interface supports reading data from and writing data to a service running on a Symbian OS phone.

Member Methods

`void Read (int aBytesToRead)`
This method initiates a read operation on the active service. When the read has completed, the `OnRead` event will be triggered.

- `aBytesToRead` – maximum number of bytes to read from the service.

`void Write (object aBuffer)` This method initiates a write operation on the active service. When the write has completed, the `OnWrite` event will be triggered. `aBuffer` – an object that boxes an array of type byte[] to be written. The size of the array controls how much data is written.
Event Handlers
`ISCBALSequentialStreamSink_OnReadEventHandler OnRead` This event is triggered when a read operation has completed. See `ISCBALSequentialStreamSink_OnReadEventHandler`.
`ISCBALSequentialStreamSink_OnWriteEventHandler OnWrite` This event is triggered when a write operation has completed. See `ISCBALSequentialStreamSink_OnWriteEventHandler`.

Event	**`OnRead`**
Event Handler	`ISCBALSequentialStreamSink_OnReadEvent-Handler`
Delegate `void OnRead (object aBuffer, int aError)` This event is triggered when an asynchronous read operation has completed. `aBuffer` – an object that boxes an array of type byte[] containing the data read. `aError` – an error code for the read operation. This is a standard `HRESULT` and zero indicates success.	

Event	**`OnWrite`**
Event Handler	`ISCBALSequentialStreamSink_OnWriteEvent-Handler`
Delegate `void OnWrite (int aError)` This event is triggered when an asynchronous write operation has completed. `aError` – an error code for the write operation. This is a standard `HRESULT` and zero indicates success.	

4.8 Using SCOM in C++ and Visual Basic

The class and member names listed above are correct for C++ and Visual Basic and the types map to corresponding C++ or Visual Basic types

(`int` variables in the above list are equivalent to Long for Visual Basic and `string` variables are BSTR variables that are translated to LPCTSTR for C++).

In Visual Basic the `SymbianConnect.Application` event handlers are automatically registered if they are created with a name that includes the name of the `SymbianConnect.Application` object and the name of the event member variable listed above (and if you declare the object with the keywords `with events`). For example, if the `SymbianConnect.Application` object is named `SCApp` then the device-connected event handler will have the form:

```
Private Sub SCApp_DeviceConnected(ByVal Device As
  SymbianConnect.ISCDevice)
  ...
End Sub
```

In C++ the obtaining of a `SymbianConnect.Application` object and the registration of event handlers are more complex, owing to the way COM is used. Examples that cover these areas are provided in Chapter 5.

5

An Example PC Connect Application – a File Browser

5.1 Overview

In this chapter, we are going to use the classes provided by SCOM to create a series of examples and then put them together into a very simple file browser application. I have chosen a file browser because it includes the functionality that many PC-only Connect applications are likely to require. With the information in this chapter you should be able to develop more specialized file management applications (such as image management applications or jukebox applications). The advantage of focusing on file management is that SCOM provides all the functions we need and so we do not have to write any new code to run on the phone. Later chapters will cover developing more specialized applications that include both PC-side and device-side code.

The major part of this chapter is based on using C# because I regard it as the development language of choice for modern PC development. I have a lot of experience using C++, but I find it much easier to develop COM and GUI applications using C# nowadays. However, C++ is still a powerful language and is still widely used, so I have included a section that describes more briefly how to use SCOM in Visual C++.

I have assumed that the reader has at least a basic familiarity with C# (or VC++ for the later sections) and COM, although even a complete novice to COM will be able to use SCOM and create a working Connect application.

5.2 Connecting to a Phone or Emulator

In order to test our connected applications, we will need to connect either to a Symbian OS phone or to the emulator. This is mostly straightforward, but I have included a step-by-step guide here just in case (after all, if you cannot get a connection then your development will not get very far).

In principle, obtaining a connection works in a similar way for all transports – you enable the connection on the PC, physically arrange the

connection and initiate the connection from the phone or emulator. There are variations depending on the transport. In all cases all three parts must be in place to make a connection.

There are two versions of m-Router in circulation: version 2 with simple configuration support and version 3 with a more helpful set of wizards to set up the connections. The instructions in this section apply to version 2, because that version requires more support.

Enabling a connection from the PC requires you to bring up the mRouterConfig application. Take a look at the task bar on your PC and you should see an icon that displays the tooltip 'mRouter' if you hover the cursor over it. This will normally have the m-Router icon displayed, although Symbian OS smartphone manufacturers may be able to change the icon. If you right-click on the mRouterConfig icon you get a short menu with a single choice labeled 'Properties'. Select this to bring up the dialog that allows you to select the ports on which you want m-Router to search for devices. If you see a menu with two entries – 'Change Device' and 'Properties' – then this means that you have an alternative application installed that was designed for an earlier PC suite. Don't worry, it will work just as well and you can select the 'Properties' entry to obtain the same dialog – just ignore the 'Change Device' entry.

If you have several PC suites installed then you may have two or even three icons in your system bar that allow m-Router to be configured. This is probably caused by using earlier PC suites, but you have no need to worry; all the icons and applications work on the same data and you can use any of them to configure your connection.

The list of ports that you see will depend on the physical configuration of your PC. Commonly, you will see one or two labeled 'Cable' for RS-232 serial ports. If you have a PC with built-in infrared functionality or an external infrared pod then you should see one port labeled 'Infrared'. If you have Bluetooth installed then you should see a number of ports labeled 'Bluetooth'. You will not see any ports labeled 'USB' until a phone tries to make a USB connection; this is described in more detail in the section on USB below.

Each port has a check-box. If this is checked then m-Router will try to connect over the port; if it is unchecked then m-Router will ignore the port. Normally, you will check the box(es) that correspond to the port(s) that you want to use.

There is a corresponding application on the phone that is used to configure and initiate a connection or to start and stop listeners. The exact form of the phone application depends on the phone, reference board or emulator. For development purposes on a reference board or emulator a single application (mRouterRefUI for Symbian OS v6.1, v7.0 and v7.0s, ConnectUI for Symbian OS v8.0) is used to control all transports. This application can be started from the Extras bar (on the emulator there is a screen shortcut; with a reference board look on the

Tools menu) or by navigating directly to the application. To navigate to the application, use the Ctrl-Q key to get to the `z:` drive and then navigate to the System\Apps directory where you will see a directory for the application; enter it and select the .app file to start the application. Once the application is started, you can use the Configure menu (use F1 on an emulator or the menu key on a reference board) to select the transport and port number you want, or to start or stop a listener, and then select 'Connect' on the File menu to initiate a connection. An active connection can be broken by selecting 'Disconnect' from the File menu or by stopping a listener from the 'Configure' menu.

Real Symbian OS smartphones do not necessarily have a single application to control all types of connection. On some phones, for example the Sony Ericsson P800, the connection is configured and managed using the control panel application. On other phones, for example Series 60 phones, there are separate applications for each type of connection – an infrared connection is initiated from the infrared application within the Connectivity folder, for example.

When you make a connection, there is some visual feedback. On the PC, the system tray icon changes and the port description in the mRouterConfig changes. It will go from a stable disconnected state to a moving state (with the description 'Connecting') while trying to connect, and then to a stable connected state when the connection is established – you will soon recognize the sequence. On most real smartphones, there is no visual feedback from the connection process, but on a reference board or on the emulator, the application that controls the connection displays the state. This should change from 'Disconnected' through 'Connecting' to 'Connected'.

Normally, when you make a connection between the PC and the phone, the connection allows SCOM to access the phone. However, on rare occasions you will appear to get a connection from m-Router but SCOM will not recognize that a device is connected. This can be caused by a configuration problem on the phone, normally on a reference board or on the emulator; you should not get this problem on a real smartphone. If you do get this problem then check for a file called `mrouterservices.ini` in the `z:\System\Data` directory (on phones before Symbian OS v8.0). If it is not present then it will need to be replaced, for which you may need to unpack a fresh copy of the emulator.

More specific details of how to use the different transports are included below.

5.2.1 Connecting over RS-232 Serial

The ability to connect over a serial cable was present in the earliest Psion PDAs, but most modern phones do not include an RS-232 serial port. However, an RS-232 serial connection is worth covering as it works well with the emulator and also with reference boards.

If you are connecting a PC to a reference board then a standard serial cable should be used. If you connecting to the emulator then a null-modem cable must be used (because the PC serial port behaves differently from that of a reference board). If you have the wrong type of cable then, however hard you try, you won't get a connection.

As well as having the correct type of cable, you need to ensure that the serial ports are not in use by other applications. On the PC you should be aware of what other software you have installed that might be configured to use the serial ports. You need to disable or reconfigure any other software so that the required ports are free.

Ensuring that the emulator or reference board has a serial port free is more complicated. Configuration files control the allocation of ports. On the emulator, these can be set using a text editor before the emulator is started, but for a reference board they need to be set before a ROM image is built.

The relevant configuration files can normally be found in the `z:\System\Data` directory and common candidates to take over serial ports are infrared and Bluetooth. Open the files `irda.esk` and `bt.esk` and look for a reference to a port number. Bear in mind that Symbian OS numbers serial ports from zero upwards, not from COM1 upwards. If you are using a reference board that has a built-in infrared capability then the `irda.esk` file is likely to be correctly set up and should be left alone. However, Bluetooth is commonly provided for development boards and for the emulator by an external Bluetooth pod that is connected via a serial cable (or a null-modem cable) and so the Bluetooth configuration may be set up to take over the port number that you want. If this is the case then you can find a line in the `bt.esk` file of the form:

```
port =0
```

and change the port number to another one (−1 works well, as does a higher number such as 10). Be aware that in editing these configuration files you may disable features and so it is always advisable to keep backup copies of any files that you alter and to exercise caution.

If you are using the emulator then infrared will often be configured to use a serial port, the expectation being that you will plug a serial infrared pod into the serial port. Therefore, check the port number in the `irda.esk` file and, if necessary, disable it as described above for Bluetooth.

Once you have freed up the required ports, plug in the correct type of cable, check the m-Router check box on the PC, select the relevant port on the emulator or reference board, and select 'Connect' from the File menu from the application on the emulator or reference board. The connection should then be made.

Once the connection has been made, you can break the connection by unplugging the cable, by unchecking the box associated with the port

for m-Router or by selecting 'Disconnect' from the File menu on the emulator or reference board.

When using the emulator with an RS-232 serial connection there is a catch that should be highlighted. The emulator is designed to take over as many serial ports on the PC as it can when it starts up. This is a positive design decision, but it has a drawback: it can grab RS-232 serial ports that you want to use with m-Router from the PC. The way to sidestep the problem is for m-Router on the PC to take control of the ports before you run the emulator. In order to run an emulator on the same PC as your Connect applications, the sequence of actions is important:

1. Bring up mRouterConfig and select one serial port, for example COM2.
2. Start the emulator. It will not be able to access COM2 because m-Router has already reserved it.

You can then plug a null-modem cable between COM1 and COM2 and make your connection as normal. It does not matter whether the cable is plugged in before the emulator is started or afterwards. If you plug a null-modem cable between COM1 and COM2 and select both ports in mRouterConfig then m-Router will try to talk to itself, with confusing results.

You can see that it is necessary to have two COM ports on a PC if you want to run an emulator and run Connect applications and connect using serial communications. If your PC has only one COM port (increasingly common nowadays) then you have a problem. There are a number of possible solutions:

- Use two PCs and run the emulator on one and the PC applications on the other.
- Use a specialized emulator bearer for m-Router that is provided with some SDKs.
- Use a third-party application to mimic multiple looped-back COM ports.
- If your PC has one COM port and an infrared port then you can plug an external infrared pod into the serial port for use by the emulator. This is described below.

5.2.2 Connecting over Infrared

Connecting over infrared follows a similar pattern to connecting over RS-232. First ensure that the infrared ports are available. On the PC, if you are using a built-in infrared port then there will be no need for extra

drivers, but you will need to configure it to be available for connecting to the phone. Select Control Panel – > Wireless Link and disable Image Transfer. If you are using an external infrared pod then you may need drivers (depending on the version of Windows that you are using) and you will need to set up the pod and select the port (serial or USB) that it is using. Again, you will need to disable Image Transfer.

If you are using a real phone then the infrared port will automatically be correctly configured and you should not need to take any further steps to configure it (in fact, you won't be able to, as all the configurations are normally held in ROM). If you are using a reference board then the infrared port is normally configured as part of any normal ROM image. It is possible to misconfigure it, but you would have to make an effort to do so.

One possible problem with using an infrared port on a reference board or emulator is that it may be configured as a connection to a mobile phone. Consider the needs of most developers working with Symbian OS: they write applications for smartphones, but the reference boards and emulators often do not include telephony hardware. For some applications this does not matter too much, but for other types of application it can be a serious problem (consider trying to develop an SMS or MMS application without access to telephony hardware). To help these developers, some SDKs have a configuration that allows a mobile phone (not normally a smartphone) to be connected via infrared. This is a very helpful configuration for these telephony applications, but it can be a real obstacle for Connectivity applications. If a reference board or emulator is set up in this way then when you try to make an m-Router connection over infrared an error will be generated on the device – it may mention etel which is the telephony engine – or you may just be unable to make a connection. If you have this configuration then you cannot use m-Router over infrared.

There are three ways to solve this problem. The first way is to change the configuration to free up the infrared port. However, this is contained in the communications database (known as CommDB) which is regarded as a dangerous thing to meddle with. The tools to set it and change it are different in different versions of Symbian OS and it is very easy to damage the configuration by mistake (the CommDB is regarded as worth avoiding by most Symbian developers unless they are experts in its use). The second way is to obtain an application that can disable the telephony engine's use of the infrared port. More details of the configurations and solutions for specific SDKs can be found on the website that is associated with this book. The third way is a bit crude, but can be effective. Symbian OS uses a number of components that are referred to as 'watchers' or 'listeners'. These are responsible for monitoring some communications channels and responding when traffic is detected. These cover areas such as an infrared link to a phone, incoming faxes and incoming SMS messages. If you delete or rename the watchers then they cannot hog the infrared

link (but other functions will not work, which is why this is a crude solution). The watchers can normally be found in a standard directory, `z:\System\Libs\Watchers`, and the directory can be simply renamed on the emulator or omitted from a ROM if you are building ROMs.

If you want to use infrared with an emulator then you will need an external serial infrared pod. Plug it into one of the serial ports and do not install any PC drivers – the infrared pod will not be used by Windows but by Symbian OS running in the emulator. Ensure that no other PC application is using the serial port. It is then necessary to configure Symbian OS to recognize that an infrared pod is available on that port. The emulator maps specific directories on the PC to the logical drives it uses. To edit the infrared configuration you need to go to the `z:\System\Data` directory. This can be found under either `\epoc32\wins` (for an emulator built for Microsoft Visual C++) or `\epoc32\winscw` (for an emulator built for Metrowerks CodeWarrior).

Within the `irda.esk` file, look for a line that sets the port number, and set the port to match the serial port into which you have plugged the infrared pod. Remember that Symbian OS numbers serial ports from 0, so the PC COM1 port is serial port 0 to Symbian OS, and the PC COM2 port is serial port 1 to Symbian OS.

Once you have configured the port to be used for infrared, ensure that you leave the infrared pod plugged in. Specifically, the emulator checks hardware when it starts, so if you do not have the pod plugged in when the emulator starts it may not be recognized.

Once you have configured the infrared port, the sequence for connection is the same as for RS-232 serial. Check the mRouterConfig check box for infrared, position the phone or ports so that the infrared ports line up and then initiate the connection from the phone, reference board or emulator.

Once you have made a connection, it can be broken by unchecking the check box for m-Router, by moving the PC or phone so the line-of-sight is broken, or by terminating the connection from the device.

If you have a PC with a single serial port and a built-in infrared port (such as most laptops) then you can make a useful connection with the emulator by plugging an external infrared pod into the serial port (and by configuring it as described above) for use by the emulator. Select ‘Infrared’ for m-Router and line up the external infrared pod with the built-in infrared Port and then make the connection. This is slower than making a loopback connection using a null-modem cable, but can get you going when you do not have two COM ports on your PC.

5.2.3 Connecting over USB

In many ways, USB is the easiest and fastest form of connection to use but it does have some drawbacks. There are no issues with port

configurations, but it does require special drivers and is not supported by the emulator or by some phones.

On the PC you will need to install USB drivers for the Symbian OS smartphone. The drivers contain knowledge about the smartphone manufacturer and model, so you will need to install the PC suite that accompanies the smartphone in order to get the drivers installed. Symbian does not provide alternative drivers for real smartphones, but if you have a reference board that supports USB then suitable drivers can be obtained.

When using a reference board that supports USB, use the Configure menu to start the USB listener and then plug a USB cable into the reference board and the PC. If the drivers have been correctly installed on the PC, m-Router will automatically create an enabled entry for a USB port and connect to it. The entry for the USB port does not exist until the USB cable is plugged in and the USB listener started. Once there is an active connection over USB, you can uncheck the port in mRouterConfig if you want to. The connection can be broken either by unchecking the port in mRouterConfig, or by unplugging the USB cable, or by stopping the USB listener.

When using USB to communicate to a smartphone, the USB cable will normally link to a cradle for the phone, and the phone will come with USB drivers.

The emulator does not support USB, so developers who want to make specific use of USB need to use a specialized reference board. Fortunately for PC Connectivity developers, if your application works over RS-232, infrared or Bluetooth it will also work over USB.

5.2.4 Connecting over Bluetooth

Bluetooth is faster than a serial or infrared connection and more convenient in that you do not need to maintain line-of-sight, but it can be slightly tricky to set up.

On the PC, you will need to install a supported Bluetooth card or dongle. At the time of writing a number of Bluetooth dongles and stacks are supported. Over time, extra support may be added (and support for old stacks may be withdrawn) so it is impossible to give a definitive list in this book.

If you are using a real Symbian OS phone then either it will support Bluetooth (at the time of writing all Symbian OS smartphones created since Symbian OS v6.1 have included Bluetooth, but it has to be disabled for some regions because of local laws) or it will not. If you are using a reference board or emulator then you will need to use an external Bluetooth pod. These external Bluetooth pods are specialized pieces of development kit (they are larger and more expensive than normal Bluetooth dongles) and so come with their own instructions. In order to

use one of these pods, you will need to set the `bt.esk` file to indicate the correct port.

Once you have a Bluetooth-enabled PC and phone, you connect by following a sequence:

1. Pair the PC and the smartphone. The pairing can be initiated either from the PC or from the device. This is a standard action when using Bluetooth: on one device search for other Bluetooth devices, select the one you want (assuming that you have set up names so you can recognize the one you want), and opt to pair. You will probably have to enter a pass code on both devices to make the connection. You can choose to allow automatic pairing in the future; this is normally helpful during development.

2. Check the relevant port number in mRouterConfig. Commonly, a Bluetooth dongle will provide more than one port and it is not clear in advance which one will be used. One way to select the correct port is to attempt to check all of them, make a connection and then uncheck the other ones.

3. If you are using a reference board or the emulator, use the Configure menu to start the Bluetooth listener.

4. On the PC, select the paired smartphone and attempt to open a Bluetooth serial port. The exact means of doing this will depend on the Bluetooth software provided with the dongle. Try selecting the device and right-clicking on the mouse.

5. If the opening of the serial port works then there may be a misleading error message. Because of an internal limitation of the serial communications software in some versions of Symbian OS, the port may have to be dropped and then reopened from the device. This leads the PC to report that the remote device has closed the connection. However, take a look at the m-Router status and you should see a successful connection. This behavior may be corrected in later versions of Symbian OS.

6. Once you have a connection, you may want to uncheck the other Bluetooth serial ports.

Successfully setting up a Bluetooth connection can be difficult for technically aware people (probably because of the misleading error message), let alone for normal consumers. There are plans to provide a more friendly user interface on the PC to make it easier (look out for m-Router 3).

Once you have a Bluetooth connection, you can break it by turning off Bluetooth on either the PC or the smartphone or by unchecking the mRouterConfig check box.

5.2.5 Connecting Using the Emulator Bearer

As well as the 'real' bearers described above, m-Router supports an emulator bearer. As the name suggests, this works only with the emulator. It is a plug-in that is not always supplied, but it is very useful if available. If it is available then it will be apparent as a bearer choice in the Configure menu and it avoids the need for a loopback cable.

5.3 Accessing SCOM and Connecting to a Device

SCOM is delivered as an automation-compatible out-of-process COM server. Any COM-aware language may use it, using either early or late binding. This section discusses how to use the SCOM type library and access SCOM from C#.

In order for C# to access SCOM, we need to import the type library. This is a standard technique for using COM in C# – SCOM requires nothing unusual. Obtain a command line prompt in the same directory as the SCOM COM server (`c:\Program Files\Symbian\Shared \SymbianConnectRunTime`) and type the commands:

```
tlbimp SymbianConnectRuntime.exe
tlbimp SCBAL.exe
```

These commands will generate two assemblies that C# can make use of – `SymbianConnect.dll` and `SymbianConnectBAL.dll`. These two files need to be referenced from your project (or from the command line using the/reference option if you try building the examples from the command line). Once we have been through these stages, our classes can make use of any of the SCOM classes simply by referencing them. The root (or route) to access SCOM is the `SymbianConnect.Application` class which provides access to the connected devices and the event handlers:

```
SymbianConnect.Application mySCOMApp = new SymbianConnect.Application();
```

At this point we could use the `ConnectedDevices` member to examine a currently connected phone. However, this would be slightly premature. We need to bear in mind that SCOM supports multiple connected phones and that phones can be connected or disconnected at any time. If we could rely on a single device being connected at all times then we could obtain the phone details from SCOM and just use them. Instead, we have to maintain a set of connected phones which may contain zero, one or more phones and we have to select a device for any operation.

In order to manage these devices, we will create some new classes: `ConnectedPhone` to contain the information that we want about each connected phone, and an `ArrayList` to manage the set of connected phones.

We store a reference to the SCOM device – `ISCDevice2` – and the phone identification properties. We use the device identifier (normally the phone IMEI number) to provide a unique identifier for the phone, because we might have more than one phone of the same type connected. We store the device name (on which more anon) as the name chosen by the user for the phone. We also store the device manufacturer and model number in order to decide what operations we can carry out (again, this will be expanded later in this chapter). None of these properties can be changed by the application so we only need to store them and provide accessor functions. We override the `ToString()` method for when the phone objects are placed in a list. This gives us an initial class declaration as follows:

```
public class ConnectedPhone
{
  // Private members for the device and related information
  private ISCDevice2 myDevice;
  private string myPhoneId;
  private string myPhoneManufacturer;
  private string myPhoneModel;
  private string myPhoneName;
...
  public string Id
  {
    get
    {return myPhoneId;}
  }

  public string Manufacturer
  {
    get
    {return myPhoneManufacturer;}
  }

  public string Model
  {
    get
    {return myPhoneModel;}
  }

  public string Name
  {
    get
    {return myPhoneName;}
  }

  public override string ToString()
  {
    return myPhoneName;
  }
  ...
```

The constructor is able to set all the member variables from the `ISCDevice2` provided. The device ID, manufacturer and model are simply attributes of the phone that can be retrieved.

The device name is slightly more complex – you can choose any convention that you please, although it may be helpful to follow that used by other applications (so as to share a name). When a phone is first connected to the PC, it is referred to by the name of the manufacturer and the model, for example ‘My Nokia 6600’ for a Nokia phone. The user can, however, set their own name for the phone and this is then stored in the registry, indexed by the phone’s device ID (IMEI number). If this name has been set then our application should attempt to use it, so that the user is presented with the familiar name. In this application we will not provide code to edit the name, although it would be straightforward to do so. If the user has not set the device name then we will use the standard format based on the manufacturer. Note that, because the device name is stored in the PC registry, not on the phone, if the phone is connected to another PC, the name will not be carried over.

```
public ConnectedPhone(ISCDevice2 device)
{
  // Device identification strings are retrieved from the device
  myDevice = device;
  myPhoneId = myDevice.Id;
  myPhoneManufacturer = myDevice.Manufacturer;
  myPhoneModel = myDevice.Model;

  // The user-specified device name is stored in the registry
  string nameKeyName =
    @"SOFTWARE\Symbian\Symbian Connect QI\Devices\" + myPhoneId;
  Microsoft.Win32.RegistryKey aKey =
    Microsoft.Win32.Registry.CurrentUser.OpenSubKey(nameKeyName);
  if(aKey != null)
  {
    myPhoneName = aKey.GetValue("Name").ToString();
    aKey.Close();
  }
  else
  {
    myPhoneName = "My "+ myPhoneManufacturer + " " + myPhoneModel;
  }
}
```

Now that we know what to do with a device reference, we need to fill in the code to obtain and manage them. We could set up a dictionary based on the device ID, but that is probably overkill. We are unlikely to have more than a handful of phones connected at one time (actually, we are unlikely to have more than one phone connected outside an IS department), so a simple `ArrayList` that we can step through will be adequate.

The actual work of creating `ConnectedPhone` objects, adding them to the array on creation and removing them from the array on destruction

is contained in the `AddPhone` and `RemovePhone` methods. These can be called from within the class as required. In this case, we call `AddPhone` on construction, based on the currently connected devices.

```
private System.Collections.ArrayList phoneArray;

public void AddPhone(ISCDevice2 device)
{
  lock(phoneArray)
  {
    bool alreadyPresent = false;
    foreach(ConnectedPhone phone in phoneArray)
    {
      if(phone.Id == device.Id)
      {
        alreadyPresent = true;
      }
    }
    if(!alreadyPresent)
    {
      phoneArray.Add(new ConnectedPhone(device));
    }
  }
}

public void RemovePhone(string deviceId)
{
  lock(phoneArray)
  {
    // Search for the phone by ID
    for(int i = 0 ; i < phoneArray.Count ; i++)
    {
      if(((ConnectedPhone)(phoneArray[i])).Id == deviceId)
      {
        phoneArray.Remove(i);
        break;
      }
    }
  }
}

public SCOMApp()
{
  // Get access to SCOM via the Application member
  mySCOMApp = new SymbianConnect.Application();
  phoneArray = new System.Collections.ArrayList();
  foreach(ISCDevice2 device in mySCOMApp.ConnectedDevices)
  {
    AddPhone(device);
  }
}
```

Note that in a real example the `RemovePhone` method will need to become more elaborate to handle termination of any pending operations as we progress through implementing the application.

Now that we have the array of connected devices, we need to set up event handlers so we can take the appropriate action when devices are connected or disconnected. SCOM uses a number of event handlers. We will be implementing the event handlers for device connection and disconnection here, but other event handlers are used for progress and error events for asynchronous functions; some of these will be considered later.

Because SCOM is primarily a COM server and was not originally written for C#, its event handlers do not conform to the common C# event handler prototype (two arguments, one object and one EventArgs). Instead, SCOM event handlers have prototypes that depend on their function. Otherwise, they work as expected. In this case, we make use of the `AddPhone` and `RemovePhone` methods that we defined previously:

```
public void OnDeviceConnected( ISCDevice2 newDevice)
{
  AddPhone(newDevice);
}
public void OnDeviceDisconnected( string deviceId)
{
  RemovePhone(deviceId);
}

...
// Add event handlers in constructor
mySCOMApp.DeviceConnected +=
  new ISCEvents_DeviceConnectedEventHandler(OnDeviceConnected);

mySCOMApp.DeviceDisconnected +=
  new ISCEvents_DeviceDisconnectedEventHandler(OnDeviceDisconnected);
```

By putting all these parts together, we have the pieces for a simple command line application that reports on connected devices and responds to connection and disconnection events.

5.4 Handling Differences Between Devices

The File Browser application that we will be building up during this chapter will be a general-purpose application that can work well with any Symbian OS phone that is compatible with SCOM. However, more specialized applications may not be so general. For example, if you are developing an image management application or a jukebox application then you will need to know where the relevant media files are stored on each phone. Unfortunately, different licensees choose to store files in different locations. For example, on a Nokia 7650 images are stored in the `c:\Nokia\Images` directory, and saved sound files (at least certain formats) are stored in the `c:\Nokia\Sounds\Digital`

directory. On a Nokia 6600 (at least on a pre-production prototype) `c:\Nokia\Images` and `c:\Nokia\Sounds\Digital` are still used, but we now have a `c:\Nokia\Videos` directory and comparable directories on the `e:` drive. On the Sony Ericsson P800 we have the directory `c:\Documents\Media` files with sub-directories audio, image and video, which in turn have sub-directories that the user creates and a corresponding `d:\Media` files directory.

If your application assumes a fixed set of directories then it may be right for one Symbian OS smartphone, but it has a good chance of being wrong for another manufacturer's smartphone, or even for another model by the same manufacturer. The same logic applies to other properties of the phone. For example, different phones support different media formats and settings.

To make allowances for different licensees' choices of directory structures and properties, your application needs to embody knowledge of these directories. The `ISCDevice2` class provides methods to access the device manufacturer and model, and then the application will require some form of table indexed by these properties.

However, the set of Symbian OS smartphones that your customers can buy will grow with time and, sooner or later, your application will be faced with a phone of a type that it does not recognize. In this situation, you have the following choices:

1. Use some default set of directories or properties in the hope that the new device will fit.
2. Use a default set based on the manufacturer, if you have encountered phones from that manufacturer before.
3. Refuse to work with that phone because it is unsafe to do so.
4. Implement some form of active querying to discover the properties of the device.

You should be very careful if you choose option 1 or option 2 – consider what might happen if you select the wrong directories. If the worst that can happen is that you copy files to the wrong place then you may feel safe, but you could fill up the drive on the phone without the user being able to see why. If the consequences could be more serious then you must be more careful.

5.5 Copying Files – Asynchronous Actions

Once we have selected a smartphone, we can start to carry out actions with it. In Section 5.6 we will cover navigating around the directories on the smartphone (which is best carried out using a graphical user

interface). This section introduces copying files between the PC and the smartphone. This is likely to be one of the actions that any real application wants to carry out and it introduces asynchronous actions and associated event handlers.

Because we are largely ignoring navigation of the phone filing system here, we will simply try to copy a file to the root directory of the `c:` drive on the phone. You will see that this is about as simple as it gets with SCOM.

We are using the classes defined above to manage the devices, and we will use a simple assumption that the first connected smartphone is the current smartphone. I will not show the code to manage the current smartphone in this way as it is not terribly interesting; the full source can be found on ***www.symbian.com/books***. A later section of this chapter – based around a GUI application – has more useful code for managing a selected smartphone. The following code gets the set of drives associated with that smartphone:

```
public class ConnectedPhone
{
  private ISCDeviceStorageDriveCollection myDrives;
...
  public ISCDeviceStorageDriveCollection Drives
  {
    get
    {return myDrives;}
  }
...
}

SCOMApp theApp = new SCOMApp();
ISCDeviceStorageDriveCollection drives =
  theApp.GetCurrentPhone.Drives;
```

We can then iterate through the drives to find the `c:` drive. This is simple and should work on any Symbian OS smartphone. By convention, the ROM is mapped to the `z:` drive, and the `c:` drive is a writable drive. There may or may not be additional drives which can be removable media. In a real application, we would either give the user the choice of drive or incorporate our knowledge of the actual smartphone. In this application, we can fix the choice to a `c:` drive without too many worries. Once we find the `c:` drive we can then copy a file from the PC to the root of the `c:` drive. The example shows that the `ISCDeviceStorageDrive2` class has a `RootDirectory` member that gives access to the root of the drive. Normally, we would use the child directories and files present in this directory to carry out some useful action, but in this case we will just copy a file:

```
public static void CopyFileToPhone(ISCDeviceStorageDrive2 drive,
                                   string fileName)
```

```
{
  int requestId = drive.RootDirectory.CopyFileFromPC(fileName);
}

...

foreach(ISCDeviceStorageDrive2 drive in drives)
{
  if(drive.Path == "C:")
  {
    CopyFileToPhone(drive, @"c:\config.sys");
    break;
  }
}
```

As you can see, we have chosen a file that should always be present on a PC but that has no particular relevance to a phone. The actual call to `CopyFileFromPC` returns immediately, regardless of the size of the file being copied. It would have been possible to make the call fully synchronous and delay returning until the copy was complete, but that would have a number of drawbacks:

- The copy operation could take a long time, depending on the size of the file and the speed of the transport in use, and the calling application would not be able to respond in the meantime.
- Although the call could return a success or failure result, it would be difficult to handle some events, such as what to do if a file already exists with the same name.
- It would not be possible to provide the user with any measure of progress during the copy, which would be unfriendly.

Therefore, the copy operation is asynchronous and we use a number of event handlers to manage progress, errors and success or failure. There are four event handlers associated with a file copy operation:

- **Copy progress** – this provides a percentage complete figure and is intended to allow you to provide a progress dialog so the user has visual feedback. The progress event also allows the copy to be canceled if the user gets bored because it is talking too long.
- **Copy complete** – this confirms that the copy operation is completed. The operation may have been completed because it was canceled by the user or was aborted because of some error, or it may have completed successfully.
- **Existing file found** – this event occurs when an existing file is found with the same name. It provides the size and date of the two files and

requires instruction on whether or not to overwrite the existing file and whether or not to cancel the copy operation.

- **Copy error** – this provides information when some other error occurs.

In this example we just output the results to the console and allow the operation to continue. In a real application, the existing file found event either could cause a dialog to be raised to give the choice to the user or could use fixed logic to decide whether or not to overwrite the existing file. In a real application, the error event would cause an informative dialog to be raised for the user, while the progress event would normally drive a progress dialog that would include a cancel button. Whatever the success of the copy operation, the copy complete event lets us close down dialogs and acknowledge that the copy is complete.

```
public void OnDeviceCopyStorageFileProgress(int requestId,
                                            string from,
                                            string to,
                                            int percentComplete,
                                            out bool cancel)
{
  System.Console.WriteLine("Progress reqId={0} from={1} to={2} "+
                           " percent={3}",
                           requestId, from, to, percentComplete);
  cancel = false;
}

public void OnDeviceCopyStorageFileError(int requestId,
                                        ScErrorDescription errorDesc,
                                        int errorCode,
                                        bool canContinue,
                                        out bool aContinue)
{
  System.Console.WriteLine("Error reqId={0} eDesc={1} eCode={2} "+
                           " canContinue={3}",
                            requestId, errorDesc.ToString(),
                            errorCode, canContinue);
  aContinue = canContinue;
}

public void OnDeviceCopyStorageFileExistingFileFound(
                              int requestId,
                              string fileName,
                              System.DateTime targetDate,
                              int targetFileSize,
                              System.DateTime sourceDate,
                              int sourceFileSize,
                              out ScOverwrite overwrite,
                              out bool cancel)
{
  System.Console.WriteLine("Existing file found requestId={0} "+
                           " fileName={1} targetDate={2} "+
                           " targetFileSize={3} sourceDate={4} "+
                           " sourceFileSize={5}",
```

```
                           requestId, fileName, targetDate,
                           targetFileSize,
                           sourceDate, sourceFileSize);
  overwrite = ScOverwrite.scOverwriteNo;
  cancel = false;
}

public void OnDeviceCopyStorageFileComplete(int requestId,
                                           int completionCode)
{
  System.Console.WriteLine("Copy complete requestId={0} "+
                           " completionCode={1}",
                           requestId, completionCode);
}

...
  // Install file copy event handlers
  mySCOMApp.DeviceCopyStorageFileProgress +=
    new ISCEvents_DeviceCopyStorageFileProgressEventHandler(
      OnDeviceCopyStorageFileProgress);
  mySCOMApp.DeviceCopyStorageFileError +=
    new ISCEvents_DeviceCopyStorageFileErrorEventHandler(
      OnDeviceCopyStorageFileError);
  mySCOMApp.DeviceCopyStorageFileExistingFileFound +=
    new ISCEvents_DeviceCopyStorageFileExistingFileFoundEventHandler(
      OnDeviceCopyStorageFileExistingFileFound);
  mySCOMApp.DeviceCopyStorageFileComplete +=
    new ISCEvents_DeviceCopyStorageFileCompleteEventHandler(
      OnDeviceCopyStorageFileComplete);
```

Bear in mind that a single SCOM application may be managing multiple connected smartphones and may be running multiple asynchronous operations at the same time. Therefore, the application might receive notification of events that correspond to one of several current operations. If these events were indistinguishable then this would make it impossible to provide accurate progress reporting or to handle events correctly. Therefore, the copy operation call returns a request identifier and each event handler includes the `RequestId` as an argument. This means that we can associate events with the copy operation that they relate to. If your application is intended to have only one copy operation active at a time then you might think that you could ignore these `RequestId`s, but you should consider carefully whether it really will be impossible to generate multiple simultaneous operations – normally it is better to assume that you will have multiple operations and design accordingly.

At this point it is worthwhile to build and run an application to see the event handlers go off. If you run the application more than once, you can see the file exists event. You could change the behavior to handle an existing file, and you could change the drive to try to copy to the `z:` drive (this should fail, but it will trigger the error event handler).

5.6 Navigating the Filing System

The simplest way of navigating the filing system is to navigate the set of device, directory and file collections that are included in the SCOM classes. These are described in the context of their classes in Chapter 4; this section will focus on these classes and members.

The `Application.ConnectedDevices` member is a collection of `ISCDevice2` objects that provides access to all currently connected Symbian OS smartphones. Remember that this collection can change as smartphones are connected or disconnected, and you should register for device connected and device disconnected events to keep track of connected smartphones.

The `ISCDevice2.StorageDrives` member is a collection of `ISCDeviceStorageDrive2` objects that provides access to all drives that exist on a device. Commonly a device will have a `z:` drive (the ROM) and a `c:` drive (the writable drive) and may have additional drives. The `ISCDeviceStorageDrive2.Path` member provides the name of the drive, which can help you to avoid attempting to write to the ROM drive.

The `ISCDeviceStorageDrive2.RootDirectory` member is the root directory for the drive.

The `ISCDeviceStorageDirectory.Parent` member is a reference to the parent object (either another directory or the drive) of a directory. The `ISCDeviceStorageDirectory.ChildDirectories` member is a collection of sub-directories within a directory; the collection may, of course, be empty. By repeated or recursive use of the `ChildDirectories` and `Parent` members, it is possible to navigate around the directory tree on a drive. The name of a directory can be found from the `ISCDeviceStorageDirectory.Path` member, but this includes the whole path of the directory from the drive onwards. If you need to separate out just the name of the directory then you will need to take a copy of the path string and manipulate it with reference to the backslash characters.

The `ISCDeviceStorageDirectory.ChildFiles` member is a collection of `ISCDeviceStorageFile` objects that contains references to all the files that are stored directly in that directory. Note that to get at the drives at the root of a drive you need to use the root directory.

The `ISCDeviceStorageFile.Parent` member is a reference to the directory in which the file resides. The `ISCDeviceStorageFile.Path` member is the name of the full file path, while `ISCDeviceStorageFile.FileName` is just the name of the file without that of the owning directory (it would have been nice to have something similar for directories).

The `Parent` member of any class refers to an `ISCDeviceStorage` object rather than to an `ISCDeviceStorageDirectory` object,

because the root directory on a drive will not be owned by another directory. As the `ISCDeviceStorage` class has `Type` and `Path` members, it is always possible to navigate usefully and to output the name of a parent, whatever it is.

In order to illustrate the use of some of these members, and some other effects, we can look at another example program. This is based on the connection example but uses very simple code to recurse through all directories on all drives and list the files and directory details. Because this is a command line application rather than a GUI application, it mindlessly recurses through the whole device.

```
public static void ListDirectory(ISCDeviceStorageDirectory directory)
{
  System.Console.WriteLine("{0} is a directory", directory.Path);

  // List files first
  foreach(ISCDeviceStorageFile aFile in directory.ChildFiles)
  {
    System.Console.WriteLine("{0} / {1} is of size {2} "+
                          " last modified {3} and attributes {4:X}",
                          aFile.Path, aFile.FileName, aFile.Size,
                          aFile.LastModified, aFile.Attributes);
  }

  // Recurse through child directories
  foreach(ISCDeviceStorageDirectory childDirectory in
          directory.ChildDirectories)
  {
    ListDirectory(childDirectory);
  }
}

public static void Main()
{
  SCOMApp theApp = new SCOMApp();
  System.Console.WriteLine( "Press return when a device is"+
                            " connected");
  string text = System.Console.ReadLine();
  System.Console.WriteLine("{0}", System.DateTime.Now);

  ISCDeviceStorageDriveCollection drives =
    theApp.GetCurrentPhone.Drives;
  foreach(ISCDeviceStorageDrive2 drive in drives)
  {
    ulong capacity  = (((ulong)drive.CapacityHigh)<<32) |
                       ((ulong)drive.CapacityLow);
    ulong freeSpace = (((ulong)drive.FreeSpaceHigh)<<32) |
                       ((ulong)drive.FreeSpaceLow);
    System.Console.WriteLine("Drive {0} has {1} free of {2}",
                             drive.Path, freeSpace, capacity);
    ISCDeviceStorageDirectory rootDirectory = drive.RootDirectory;
    ListDirectory( rootDirectory);
  }
  System.Console.WriteLine("{0}", System.DateTime.Now);
}
```

Note the casting and shifting to get the drive capacity and free space. These values are stored as 64-bit values in Symbian OS. This may seem excessive at the moment, but multi-gigabyte drives are already under development. Because of the limitations of COM, the two halves of each value are returned as signed 32-bit values, so you need to cast them to unsigned before combining them.

This can take some time (I just ran it on a Nokia 7650 and the output amounts to over 1,700 lines). If you have a real Symbian OS smartphone to hand then I recommend trying it out.

One of the effects that should become apparent is the speed (or otherwise) of the operations. Each time SCOM needs to obtain some information from the phone it has to send a command and receive the answer. SCOM hides the gory details from you, the developer, but it cannot always hide the time taken. Internally, SCOM implements some caching of information, but the information has to be retrieved once in order to be cached.

The example above also includes details on the time taken for the whole operation. Try running the application with an infrared connection and then with a Bluetooth connection and you will see the difference that the transport speed makes. Even with a fast transport, it makes sense to design your program to not ask for information unless it needs it. Often SCOM's caching will give you the best performance possible with the transport, but you will still need to consider hourglass icons and good design. We will touch on this when allowing our file browser to navigate through directories.

5.7 A File Browser Application

The previous examples have shown how to access and manage the currently connected devices, how to carry out some basic navigation of the directory tree and how to copy files. As a programmer with a Unix background I am quite comfortable with the idea of writing small command-line applications to manage my phone, but most consumers are not. Therefore, we will create a basic graphical program to view and manage the files on a Symbian OS smartphone. This application could easily be tailored for specific types of file.

We start with a form that is the main dialog of the application. I have included fields for device information, a large tree view to display the phone directories and files, and buttons for the supported operations.

5.7.1 Smartphone Connection, Disconnection and Management

We can use the `CConnectedPhone` class from the command-line applications we developed earlier in the chapter. We will need a new SCOM

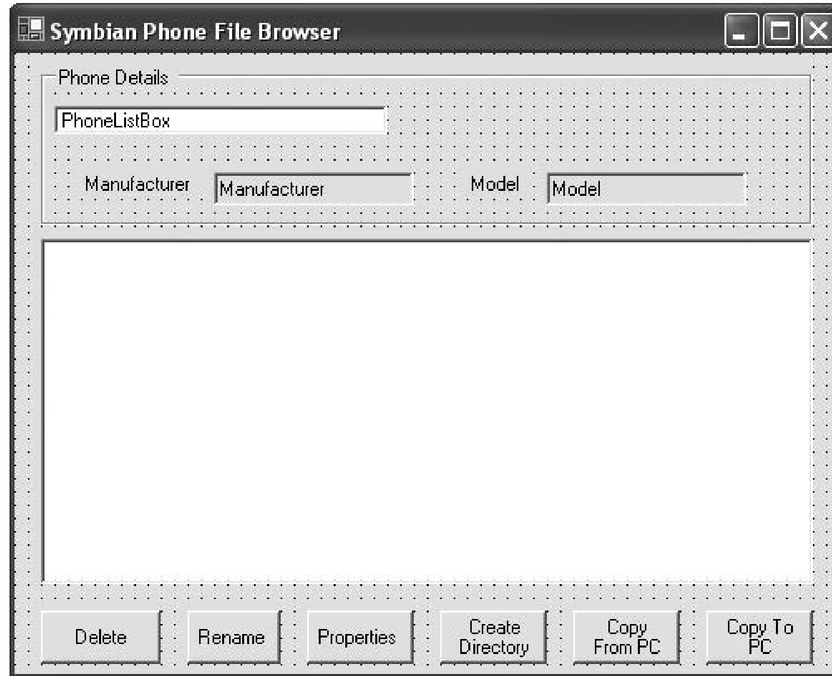


Figure 5.1 Symbian Phone File Browser form

application class. In this case, I have chosen to create a new class just for that purpose. This class needs quite close links to the form class to be able to pass events through. It would be possible to make the `EFBForm` handle the SCOM application directly, but I regard that as poor design – separating the management of the SCOM application from the form makes it clearer. Later in this chapter I will link the form more directly to SCOM functions that I am comfortable with, but in those cases the separation would require more complexity than is justified.

We know that we will need to handle events caused by smartphones being connected or disconnected (and later on, we will also need to handle file copy events). Therefore, we can define an interface that includes the events and make `EFBForm` implement it. In this way we have a chance of reusing the `SCOMApp` class at a later date.

```
/// <summary>
/// SCOMForm contains the interface methods required for event
/// handling, so a form can interact with an SCOMApp.
/// </summary>
public interface SCOMForm
{
  // Called when phones are connected or disconnected
  void UpdatePhoneList();
}
```

In this case, we just expose a single method – `UpdatePhoneList()` – that can be called whenever a phone connects or disconnects (later, we will add more methods to handle file copying events). This will cause the phone list to be refreshed. We don't directly refresh the phone list within the event handler. The event handler is being called indirectly from SCOM, and I have seen some problems caused by carrying out too much work in an event handler. There should not be any problem, because internally SCOM spawns off a new thread for each call to an event handler (before it did this, one badly behaved event handler could block the whole of SCOM), but in practice SCOM appears to work better if you are careful about how much work is carried out in its event handlers. Therefore, we will just set a private flag to indicate that the phone list needs to be updated and use the `OnPaint` event handler to do the work:

```
/// <summary>
/// Update display of list of connected phones.
/// </summary>
public void UpdatePhoneList()
{
  // This method can be called from an event when a phone connects
  // or disconnects.
  // Don't rewrite the file tree view here as we don't
  // want to do too much in an event so just prime it.
  pendingPhoneViewUpdate = true;
  Invalidate();
}

/// <summary>
/// Some operations (normally triggered by events) cannot be
/// carried out directly.  They are achieved by setting flags and
/// causing a repaint.
/// The use of OnPaint is a bit crude but it is an easy event to
/// trigger.
/// </summary>
protected override void OnPaint(
                        System.Windows.Forms.PaintEventArgs e)
{
  // If we have a pending update to the phone list then do it
  if(pendingPhoneViewUpdate)
  {
    ResetPhoneList();
    pendingPhoneViewUpdate = false;
  }
  base.OnPaint(e);
}
```

Once we have a set of connected smartphones, we can list them and show some of the phone properties.

```
/// <summary>
/// Called when a phone is connected or disconnected
/// </summary>
private void ResetPhoneList()
```

```
{
  bool wasCurrentPhone = false;
  bool currentPhonePresent = false;
  if( !initializing)
  {
    wasCurrentPhone = (currentPhoneId != "");
    updatingPhoneList = true; // Prevent selection events
    PhoneListBox.BeginUpdate();
    PhoneListBox.Items.Clear();
    foreach(ConnectedPhone phone in mySCOMApp.phoneArray)
    {
      PhoneListBox.Items.Add(phone);
      if(phone.Id == currentPhoneId)
      {
        currentPhonePresent = true;
        PhoneListBox.SelectedIndex = PhoneListBox.Items.Count-1;
      }
    }
    if(PhoneListBox.Items.Count > 0)
    {
      isEmptyPhoneList = false;
      if(!currentPhonePresent)
      {
        PhoneListBox.SelectedIndex = 0;
        currentPhone = (ConnectedPhone)mySCOMApp.phoneArray[0];
        currenthoneId = currentPhone.Id;
        ManufacturerTextBox.Text = currentPhone.Manufacturer;
        ModelTextBox.Text = currentPhone.Model;
        InitializePhoneTreeView();
      }
    }
    else
    {
      isEmptyPhoneList = true;
      currentPhone = null;
      currentPhoneId = "";
      ManufacturerTextBox.Text = "";
      ModelTextBox.Text = "";
      FileTreeView.Nodes.Clear();
      DisableButtons();
    }
    PhoneListBox.EndUpdate();
  }
  updatingPhoneList = false;

  // If we have lost the current phone then we
  // may have to cancel operations
  if(wasCurrentPhone && !currentPhonePresent)
  {
    if(progressForm != null)
    {
      progressForm.Close();
    }
  }
}
```

Once we have a selected smartphone, we can display the directories and files in the tree view. We can get the list of drives on the phone and

then add nodes for them. With each tree view node, we store an object that contains the type of the SCOM storage object and a handle for it. This allows us to use the tree view node directly to carry out operations.

```
/// <summary>
/// Contains information about the phone directory or file that is
/// related to a node
/// </summary>
public class PhoneViewNode
{
  public enum NodeType
  {
    Directory,
    File
  }

  private bool myAlreadyExpanded;
  private string myNodeName;
  private NodeType myNodeType;
  private object myStorage;

  public PhoneViewNode(string aNodeName, NodeType aNodeType,
                       object aStorage)
  {
    myNodeName = aNodeName;
    myAlreadyExpanded = false;
    myNodeType = aNodeType;
    myStorage = aStorage;
  }

  /// <summary>
  /// Name of associated node
  /// </summary>
  public string NodeName
  {
    get
    {return myNodeName;}
  }

  /// <summary>
  /// File or Directory object
  /// </summary>
  public object Storage
  {
    get
    {return myStorage;}
  }

  /// <summary>
  /// Is the Node a directory or a file?
  /// </summary>
  public NodeType NodeStorageType
  {
    get
    {return myNodeType;}
  }
```

```
  /// <summary>
  /// Has the list of child objects already been expanded?
  /// </summary>
  public bool AlreadyExpanded
  {
    get
    {return myAlreadyExpanded;}
    set
    {myAlreadyExpanded = value;}
  }
}
```

We could choose to traverse the whole directory and file tree of each drive and fully populate the tree view, but that would take too long. Remember how long it took a command line application to traverse the whole of a device filing system? Bear in mind that the user won't visit most of the directories, so traversing them all will be wasted time.

Instead, we will populate the tree view on a 'Just in Time' basis. When the user tries to expand a directory then we will hastily fill in the details.

```
/// <summary>
/// Add children before expansion of a node in the tree view
/// </summary>
private void FileTreeView_DoubleClick(object sender,
                                      System.EventArgs e)
{
  lock(FileTreeView)
  {
    FileTreeView.BeginUpdate();
    TreeNode tn = FileTreeView.SelectedNode;
    PhoneViewNode nodeInfo = (PhoneViewNode)tn.Tag;
    if((nodeInfo.NodeStorageType == PhoneViewNode.NodeType.Directory)
       && !nodeInfo.AlreadyExpanded)
    {
      // Clear any contents
      tn.Nodes.Clear();
      // Get a storage object from this
      try
      {
        ISCDeviceStorageDirectory myDir =
          (ISCDeviceStorageDirectory)nodeInfo.Storage;
        if(myDir != null)
        {
          Cursor.Current = Cursors.WaitCursor;
          // List directories before files
          foreach(ISCDeviceStorageDirectory dir in
                  myDir.ChildDirectories)
          {
            // Just take the last part of the name
            string dirName =
              dir.Path.Substring(dir.Parent.Path.Length);
            TreeNode dirNode = new TreeNode(dirName);
            dirNode.Tag = (object)
              (new PhoneViewNode(dirName,
                                 PhoneViewNode.NodeType.Directory,
```

```
                                 (object)dir));
            tn.Nodes.Add(dirNode);
          }
          foreach(ISCDeviceStorageFile file in myDir.ChildFiles)
          {
            TreeNode fileNode = new TreeNode(file.FileName);
            fileNode.Tag = (object)
              (new PhoneViewNode(file.FileName,
                                 PhoneViewNode.NodeType.File,
                                 (object)file));
            tn.Nodes.Add(fileNode);
          }
          nodeInfo.AlreadyExpanded = true; // don't try this again
          Cursor.Current = Cursors.Arrow;
        }
      }
      catch
      { // Do nothing but we cannot add any nodes
      }
      tn.Expand();
    }
    FileTreeView.EndUpdate();
  }
}
```

Note that I have not been able to use the handy '+' icon for directory expansion. This is a nice feature, but it has one big drawback for this use. In order for the tree view control to display the '+' correctly, the directory node list needs to be populated as soon as it is displayed. Therefore, we need to fill the node tree one level ahead of the user. If we did not do this, directories that have been visited and that contain child directories or files would have '+' icons, while directories that either are empty or have not been visited would lack the '+' icon. This would be confusing for the user.

Therefore, we use the double-click event to fill in the node lists before expansion. We take basic precautions to fill in the data only once.

5.8 Simple Actions on Files and Directories

Navigating around the filing system on the smartphone is a nice trick, but it will soon get boring. To make a useful application, we want to carry out some operations on the files and directories. Because this is a simple application, I have chosen to use straightforward buttons to carry out operations and display properties. If you feel more ambitious, you could display file properties within the tree view and you could implement a context menu and hang the operations off the right mouse button.

The operations that we will implement in this application are as follows:

- display file properties
- rename a file or directory

- copy a file from the PC to a directory
- copy a file from the smartphone to the PC
- create a new directory
- delete a file or directory.

You can see that not all of these operations apply to both files and directories. Therefore, we want to enable the buttons based on what type of entry is selected. We disable the buttons when the tree view is first populated and then selectively enable or disable when entries are selected.

```
/// <summary>
/// Disables button as if no selection
/// </summary>
private void DisableButtons()
{
  CopyFromButton.Enabled = false;
  CopyToButton.Enabled = false;
  PropertiesButton.Enabled = false;
  DeleteButton.Enabled = false;
  RenameButton.Enabled = false;
  CreateDirButton.Enabled = false;
}
/// <summary>
/// When a node is selected, enable or disable buttons selectively
/// </summary>
private void FileTreeView_AfterSelect(object sender,
                           System.Windows.Forms.TreeViewEventArgs e)
{
  // Enable or disable buttons based on the currently selected node
  if(!updatingPhoneList)
  {
    lock(FileTreeView)
    {
      PhoneViewNode selectedNode = null;
      if(FileTreeView.SelectedNode != null)
      {
        object myTag = FileTreeView.SelectedNode.Tag;
        if(myTag != null)
        {
          selectedNode = (PhoneViewNode)myTag;
        }
      }
      DisableButtons();
      if((selectedNode != null) &&
         (selectedNode.NodeStorageType ==
          PhoneViewNode.NodeType.Directory))
      {
        CopyFromButton.Enabled = true;
        DeleteButton.Enabled = true;
        RenameButton.Enabled = true;
        CreateDirButton.Enabled = true;
      }
      else if((selectedNode != null) &&
              (selectedNode.NodeStorageType ==
               PhoneViewNode.NodeType.File))
```

```
      {
        CopyToButton.Enabled = true;
        PropertiesButton.Enabled = true;
        DeleteButton.Enabled = true;
        RenameButton.Enabled = true;
      }
    }
  }
}
```

The simplest operations are deletion and renaming, so we will cover those first. Deletion could be very simple, but we will add the precaution of asking the user for confirmation. This can be annoying to the user if taken too far but is probably wise in this case. We also delete the node entry from the tree view.

```
/// <summary>
/// Event for delete button - delete file or directory
/// </summary>
private void DeleteButton_Click(object sender, System.EventArgs e)
{
  if (MessageBox.Show ("Are you sure you want to delete " +
                       FileTreeView.SelectedNode.FullPath,
                       "Confirm Delete",
                       MessageBoxButtons.YesNo,
                       MessageBoxIcon.Question,
                       MessageBoxDefaultButton.Button2 )
    == DialogResult.Yes)
  {
    Cursor.Current = Cursors.WaitCursor;
    FileTreeView.BeginUpdate();

    PhoneViewNode myNode =
      (PhoneViewNode)FileTreeView.SelectedNode.Tag;
    // Use a try loop because we may cause an exception
    // (like trying to affect ROM)
    try
    {
      if(myNode.NodeStorageType == PhoneViewNode.NodeType.File)
      {
        ISCDeviceStorageFile myFile =
          (ISCDeviceStorageFile)myNode.Storage;
        myFile.Delete();
      }
      else // directory
      {
        ISCDeviceStorageDirectory myDir =
          (ISCDeviceStorageDirectory)myNode.Storage;
        myDir.Delete();
      }
      FileTreeView.SelectedNode.Parent.Nodes.Remove(
                                      FileTreeView.SelectedNode);
    }
    catch( Exception ex)
    {
      MessageBox.Show("Unable to delete due to unexpected error"+
```

```
                 ex.ToString(), "Unexpected Error",
                 MessageBoxButtons.OK, MessageBoxIcon.Error);
    }

    FileTreeView.EndUpdate();
    Cursor.Current = Cursors.Arrow;
  }
}
```

Renaming is also straightforward: we put up a dialog to ask for the new name and then call `Rename()`. Note that directory names have to end in a backslash (\) – this is just a convention within SCOM. Note also that we amend the node entry to make the rename effect visible in the tree view.

```
/// <summary>
/// Event for rename button - rename file or directory
/// </summary>
private void RenameButton_Click(object sender, System.EventArgs e)
{
  PhoneViewNode vNode = (PhoneViewNode)FileTreeView.SelectedNode.Tag;
  string currentName;
  if(vNode.NodeStorageType == PhoneViewNode.NodeType.File)
  {
    ISCDeviceStorageFile myFile =
      (ISCDeviceStorageFile)(vNode.Storage);
    currentName = myFile.FileName;
  }
  else // directory
  {
    ISCDeviceStorageDirectory myDir =
      (ISCDeviceStorageDirectory)(vNode.Storage);
    currentName = myDir.Path.Substring(myDir.Parent.Path.Length);
    // Trim trailing slash
    currentName = currentName.Substring(0,currentName.Length-1);
  }
  renameForm = new RenameForm();
  renameForm.NameTextBox.Text = currentName;
  if(renameForm.ShowDialog() == DialogResult.OK)
  {
    try
    {
      Cursor.Current = Cursors.WaitCursor;
      if(vNode.NodeStorageType == PhoneViewNode.NodeType.File)
      {
        ISCDeviceStorageFile myFile =
          (ISCDeviceStorageFile)(vNode.Storage);
        myFile.Rename(renameForm.NameTextBox.Text);
        FileTreeView.SelectedNode.Text = renameForm.NameTextBox.Text;
      }
      else // directory
      {
        ISCDeviceStorageDirectory myDir =
          (ISCDeviceStorageDirectory)(vNode.Storage);
        string newName = renameForm.NameTextBox.Text;
        if(!newName.EndsWith(@"\"))
```

```
        {
          newName += @"\";
        }
        myDir.Rename(FileTreeView.SelectedNode.Parent.FullPath +
                     newName);
        FileTreeView.SelectedNode.Text = newName;
      }
    }
    catch( Exception ex)
    {
      MessageBox.Show("Unable to rename file or directory due to" +
                " unexpected error " + ex.ToString(),
                "Unexpected Error",
                MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    Cursor.Current = Cursors.Arrow;
    renameForm = null;
  }
}
```

The renaming or deleting operation may take some time, so we will change the cursor to an hourglass to give the user a cue that a delay is expected, but these functions are synchronous so we don't need any progress tracking.

Creating a directory is comparable, but we need to add a node for the new directory. We use the `Storage` member to get a handle for the newly created directory.

```
/// <summary>
/// Event for Create a directory button
/// </summary>
private void CreateDirButton_Click(object sender, System.EventArgs e)
{
  PhoneViewNode vNode = (PhoneViewNode)FileTreeView.SelectedNode.Tag;
  if(vNode.NodeStorageType == PhoneViewNode.NodeType.Directory)
  {
    ISCDeviceStorageDirectory myDir =
      (ISCDeviceStorageDirectory)(vNode.Storage);
    createDirForm = new CreateDirForm();
    createDirForm.NameTextBox.Text = "";
    if(createDirForm.ShowDialog() == DialogResult.OK)
    {
      try
      {
        Cursor.Current = Cursors.WaitCursor;
        string newDirName = createDirForm.NameTextBox.Text+ @"\";
        myDir.CreateDirectory(newDirName);
        // Now add the new directory to the view
        TreeNode dirNode = new TreeNode(newDirName);
        string fullPath =
          FileTreeView.SelectedNode.FullPath+newDirName;
        ISCDeviceStorageDirectory dir = (ISCDeviceStorageDirectory)
          currentPhone.device.get_Storage(fullPath);
        if(dir != null)
        {
```

```
          dirNode.Tag = (object)
            (new PhoneViewNode(newDirName,
                               PhoneViewNode.NodeType.Directory,
                               (object)dir));
          FileTreeView.SelectedNode.Nodes.Add(dirNode);
        }
      }
      catch( Exception ex)
      {
        MessageBox.Show("Unable to create new directory due to " +
                    " unexpected error "+ex.ToString(),
                    "Unexpected Error",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
      }
      Cursor.Current = Cursors.Arrow;
      createDirForm = null;
    }
  }
}
```

Displaying file properties is the reverse in one sense – we get the information from SCOM and then put up the dialog.

```
/// <summary>
/// Event for properties button - publish file properties
/// </summary>
private void PropertiesButton_Click(object sender,
                                    System.EventArgs e)
{
  PhoneViewNode vNode = (PhoneViewNode)FileTreeView.SelectedNode.Tag;
  try
  {
    ISCDeviceStorageFile myFile =
      (ISCDeviceStorageFile)(vNode.Storage);
    MessageBox.Show ("File " +FileTreeView.SelectedNode.FullPath +
      "\nFile size " + myFile.Size +
      "\nLast modified date " + myFile.LastModified,
      "File Properties",
      MessageBoxButtons.OK, MessageBoxIcon.Information);
  }
  catch // probably means the phone has disconnected
  {
  }
}
```

The copying functions are both asynchronous and so we need a slightly more complex framework, though not much more complex. In each case, we will need to put up a dialog to ask the user which file on the PC should be copied or which directory on the PC is to receive the file from the phone. In an early version of this application I used the common dialogs for opening and saving files, but they had excess baggage that didn't fit well with their usage here, so I replaced them with simple dialogs that display a tree view of the PC filing system and allow the user to select a

file or a directory. I don't include the code here because it is purely C# Win32 code; it can be found with the complete source code for the book.

Once we have the name of the desired file or target directory, we can call the relevant copy function. However, file copying is an asynchronous task and so we will get progress and completion events (at least). Therefore, before we try the copy, we will need to set up the event handlers. We extend the interface we used before:

```
public interface SCOMForm
{
  // Called when phones are connected or disconnected
  void UpdatePhoneList();
  // Called when there is progress during a file copy
  void FileCopyProgress(int aRequestId, int aPercentComplete);
  // Called when a copy attempts to overwrite a file
  void FileCopyExistingFileFound(int aRequestId, string aFileName,
    System.DateTime aTargetDate, int aTargetFileSize,
    System.DateTime aSourceDate, int aSourceFileSize,
    out ScOverwrite aOverwrite, out bool aCancel);
  // Called when a file copy is complete
  void FileCopyComplete(int aRequestId, int aCompletionCode);
  // Called when an error occurs during a copy
  void FileCopyError(int aRequestId, ScErrorDescription aErrorDesc,
    int aErrorCode, bool aCanContinue, out bool aContinue);

  // Called from a progress form to cancel an operation
  void CancelOperation();
  // Called to ascertain whether a copy operation is canceled
  bool IsOperationCanceled();
}
```

We could just put up an hourglass during the copy, but that would be a waste – the progress event handler provides a measure of progress and we can use that to keep the user informed when we are copying a large file. It also gives us a means of canceling the copy operation if desired. Therefore, we will have a progress dialog that includes a progress bar and a cancel button.

Within the copy function, we cannot complete the copy operation because of its asynchronous nature.

```
/// <summary>
/// Event for Copy From button - copy a file from the PC to a
/// directory
/// </summary>
private void CopyFromButton_Click(object sender, System.EventArgs e)
{
  ChooseFileForm ff = new ChooseFileForm(true);
  ff.SetTitle("Choose File");
  if((ff.ShowDialog()== DialogResult.OK) && (ff.ChosenName != ""))
  {
    string newFileName = ff.ChosenName;
    progressForm = new ProgressForm(this);
    progressForm.Caption.Text = "Copying " + newFileName +
```

```
      " to " + FileTreeView.SelectedNode.FullPath;
    progressForm.ProgressBar.Value = 0;

    PhoneViewNode selectedNode =
      (PhoneViewNode)FileTreeView.SelectedNode.Tag;
    try
    {
      ISCDeviceStorageDirectory myDir =
        (ISCDeviceStorageDirectory)(selectedNode.Storage);
      cancelOperation = false;
      doingCopyFromPC = true;
      int startPos = newFileName.LastIndexOf(@"\")+1;
      newCopyFileName = newFileName.Substring(startPos);

      int requestId = myDir.CopyFileFromPC(newFileName);
      progressForm.ShowDialog();
    }
    catch // probably means the phone has disconnected
    {
    }
  }
}

/// <summary>
/// Event for Copy To button - copy a file to the PC
/// </summary>
private void CopyToButton_Click(object sender, System.EventArgs e)
{
  PhoneViewNode selectedNode =
    (PhoneViewNode)FileTreeView.SelectedNode.Tag;
  ISCDeviceStorageFile myFile =
    (ISCDeviceStorageFile)(selectedNode.Storage);
  ChooseFileForm ff = new ChooseFileForm(false);
  ff.SetTitle("Choose Target Directory");
  if((ff.ShowDialog()== DialogResult.OK) && (ff.ChosenName != ""))
  {
    string newPath = ff.ChosenName;
    progressForm = new ProgressForm(this);
    progressForm.Caption.Text = "Copying " +
      FileTreeView.SelectedNode.FullPath + " to " + newPath;
    progressForm.ProgressBar.Value = 0;

    cancelOperation = false;
    doingCopyFromPC = false;
    newCopyFileName = "";
    try
    {
      int requestId = myFile.CopyToPC(newPath);
      progressForm.ShowDialog();
    }
    catch // probably means the phone has disconnected
    {
    }
  }
}
```

The `SCOMApp` class has very simple event handlers that just pass the events through to the supplied `SCOMForm`:

```
public void OnDeviceCopyStorageFileProgress(int aRequestId,
  string aFrom, string aTo, int aPercentComplete, out bool aCancel)
{
  mySCOMForm.FileCopyProgress(aRequestId, aPercentComplete);
  aCancel = mySCOMForm.IsOperationCanceled();
}
```

Then we can do a simple update in the `EFBForm` class:

```
/// <summary>
/// Progress event for asynchronous file copy
/// </summary>
public void FileCopyProgress(int aRequestId, int aPercentComplete)
{
  if(progressForm != null)
  {
    progressForm.ProgressBar.Value = aPercentComplete;
  }
}
```

The progress form also has code to handle the user canceling the operation. In this case all we do is remember that the operation is canceled for the next opportunity. At the time of writing, SCOM does not allow pre-emptive canceling of a copy operation. The `IsOperationCanceled()` method is used by the `SCOMApp` class.

```
/// <summary>
/// Button click event called when cancel is selected from the
/// progress form.
/// The cancel is stored for the next opportunity.
/// </summary>
public void CancelOperation()
{
  cancelOperation = true;
}

/// <summary>
/// Return whether an asynchronous operation is to be canceled
/// </summary>
public bool IsOperationCanceled()
{
  return cancelOperation;
}
```

Of course, the progress event handler is not the only event handler associated with file copying. As with the progress event, we have very simple event handlers in `SCOMApp` to pass events through:

```
public void OnDeviceCopyStorageFileComplete(int aRequestId,
                                            int aCompletionCode)
{
```

```
  mySCOMForm.FileCopyComplete(aRequestId, aCompletionCode);
}

public void OnDeviceCopyStorageFileError(int aRequestId,
  ScErrorDescription aErrorDesc,
  int aErrorCode, bool aCanContinue, out bool aContinue)
{
  mySCOMForm.FileCopyError(aRequestId, aErrorDesc, aErrorCode,
                           aCanContinue, out aContinue);
}

public void OnDeviceCopyStorageFileExistingFileFound( int aRequestId,
  string aFileName,
  System.DateTime aTargetDate, int aTargetFileSize,
  System.DateTime aSourceDate, int aSourceFileSize,
  out ScOverwrite aOverwrite, out bool aCancel)
{
  mySCOMForm.FileCopyExistingFileFound( aRequestId, aFileName,
                    aTargetDate, aTargetFileSize,
                    aSourceDate, aSourceFileSize,
                    out aOverwrite, out aCancel);
}
```

At the least, we will need a file copy complete event which we can use to close the progress dialog. This also completes the file copy operation by creating the new entry for the tree view. As with other events from SCOM, we route this through the `OnPaint` method.

```
/// <summary>
/// File copy complete event for asynchronous file copy
/// </summary>
public void FileCopyComplete(int aRequestId, int aCompletionCode)
{
  if((aCompletionCode == 0) && doingCopyFromPC)
  {
    // We will need to update the file tree view for the new file
    pendingFileAddition = true;
    Invalidate();
  }
  if(progressForm != null)
  {
  // Complete and clear the progress form
  progressForm.ProgressBar.Value = 100;
  progressForm.DialogResult = DialogResult.OK;
  progressForm.Close();
  }
}
```

```
/// <summary>
/// Some operations (normally triggered by events) cannot be
/// carried out directly.  They are achieved by setting flags and
/// causing a repaint.
/// The use of OnPaint is a bit crude but it is an easy event to
/// trigger.
/// </summary>
protected override void OnPaint(
```

```
                            System.Windows.Forms.PaintEventArgs e)
{
  // If we have a pending update to the phone list then do it
  if(pendingPhoneViewUpdate)
  {
    ResetPhoneList();
    pendingPhoneViewUpdate = false;
  }
  else if(pendingFileAddition)
  {
    UpdateForCopiedFile();
    pendingFileAddition = false;
  }
base.OnPaint(e);
}
```

```
/// <summary>
/// Update the tree view for a copied file.
/// This is called after a file copy completes successfully.
/// This code cannot be run directly from the copy complete event
/// as that is in a different thread and has access problems.
/// </summary>
private void UpdateForCopiedFile()
{
  lock(FileTreeView)
  {
    doingCopyFromPC = false;
    PhoneListBox.BeginUpdate();
    // Make a new entry for the directory we are copying to
    TreeNode fileNode = new TreeNode(newCopyFileName);
    try
    {
      string fullPath =
        FileTreeView.SelectedNode.FullPath+newCopyFileName;
      ISCDeviceStorageFile file =
     (ISCDeviceStorageFile)currentPhone.device.get_Storage(fullPath);
      if(file != null)
      {
        fileNode.Tag = (object)(new PhoneViewNode(newCopyFileName,
          PhoneViewNode.NodeType.File, (object)file));
        FileTreeView.SelectedNode.Nodes.Add(fileNode);
      }
    }
    catch // probably means the phone disconnected
    {
    }
    doingCopyFromPC = false;
    newCopyFileName = "";
    PhoneListBox.EndUpdate();
    Invalidate();
  }
}
```

We also have a dialog for attempts to overwrite an existing file and to report errors. The error handler could be elaborate, but in this example I have left it basic.

```
/// <summary>
/// File already exists event for asynchronous file copy
/// </summary>
public void FileCopyExistingFileFound( int aRequestId,
  string aFileName,
  System.DateTime aTargetDate, int aTargetFileSize,
  System.DateTime aSourceDate, int aSourceFileSize,
  out ScOverwrite aOverwrite, out bool aCancel)
{
  if (MessageBox.Show("File " + aFileName +
    " already exists. Do you want to overwrite it?",
    "Confirm Overwrite",
    MessageBoxButtons.YesNo, MessageBoxIcon.Question,
    MessageBoxDefaultButton.Button2 )
    == DialogResult.Yes)
  {
    aOverwrite = ScOverwrite.scOverwriteYes;
    aCancel = false;
  }
  else
  {
    aOverwrite = ScOverwrite.scOverwriteNo;
    aCancel = true;
  }
}

/// <summary>
/// File copy error event for asynchronous file copy
/// </summary>
public void FileCopyError(int aRequestId,
  ScErrorDescription aErrorDesc, int aErrorCode,
  bool aCanContinue, out bool aContinue)
{
  aContinue = false;
  if(aCanContinue)
  {
    if (MessageBox.Show("Error " + aErrorDesc.ToString() +
      " - do you want to continue?",
      "Copy Error", MessageBoxButtons.YesNo, MessageBoxIcon.Error,
      MessageBoxDefaultButton.Button2)
      == DialogResult.Yes)
    {
      aContinue = true;
    }
  }
  else
  {
    MessageBox.Show("Error " + aErrorDesc.ToString(), "Copy Error",
      MessageBoxButtons.OK, MessageBoxIcon.Error);
  }
}
```

5.9 Error Handling and Disconnection

Often the most difficult part of an application is handling errors: that certainly can be true of Connectivity applications. There are many possible

sources of error. Some are predictable – you could try to alter a ROM drive or to set an invalid name – while others are less predictable. The most common is disconnecting the smartphone – the user can disconnect it at any time.

In the case of the example file browser that we are developing, the design limits the damage to some degree. If any file operation is underway then the relevant dialog is modal so we cannot have more than one active at a time. If any of the synchronous actions are underway (delete, rename, properties display or directory creation) when the smartphone disconnects then an exception will be generated. If a file copy operation is underway then it will asynchronously complete with a failure error code. If the application was expanding directory contents then again an exception results.

It can be seen that the key is writing the code with failure in mind; if you write a long sequence of actions then be aware that any of the actions may fail and you may need to unwind. You should plan to test your application by disconnecting the smartphone at all possible points, but a complex sequence will make it almost impossible to ensure that you have tested every case.

Even though the file browser application is simple enough to make disconnection relatively painless, we still have to take some actions. We will need to update our internal records and any graphical display; this is done by the event handlers and the `ResetPhoneList` method that we saw earlier.

5.10 Visual C++ Code for Application and Device Management

This chapter (and later ones) pays most attention to C# as the programming language of choice. However, it is quite possible to develop an application using SCOM in Visual C++ or Managed C++. SCOM is implemented in Visual C++ and existing PC suites have been implemented in Visual C++. This is partly because the developers are used to Visual C++ and partly because it eases the deployment on to earlier versions of Microsoft Windows. This section shows how to use SCOM in Visual C++. It is less extensive than the C# example but should let a C++ programmer get up and running. Some of the details that were included in the C# examples will be omitted from this section; use this section just to get the C++ framework in place and then refer to the C# examples for the functional details.

As with the description of using C#, most of the learning curve is associated with using COM. I will not provide a full tutorial on using COM or on C++ because there are ample books on the subjects. However, this section should provide enough for most developers.

To begin with, we are going to use the #import directive to create the SCOM wrapper classes in C++. This is not the only possible method; the choice of which tool to use is up to the developer. It is also possible to have the ClassWizard generate these wrapper classes. However, the #import directive offers the advantage to C++ clients of providing access to the vTable side of the interface, rather than via IDispatch::Invoke.

To make the SCOM libraries available to the entire project, the type library is imported in the precompiled header file, StdAfx.h. The following lines should be added to the file StdAfx.h:

```
#import "SymbianConnectRuntime.exe"  named_guids no_namespace
#import "SCBAL.exe"  named_guids no_namespace
```

Here, we instruct VC++ to import the type library contained in SCOM's executables, assuming that the files SymbianConnect-Runtime.exe and SCBAL.exe are registered and are available in the system include path or the current directory. The options named_guids and no_namespace instruct #import to include true C++ GUIDs, rather than make use of the __uuidof operator, and to import the classes into the global namespace. It should be noted that classes imported without the raw_interfaces_only option throw errors on failure, rather than returning the error. The error checking code has been left out of some examples for clarity.

The #import directive will produce the files SymbianConnect-Runtime.tlh, SymbianConnectRuntime.tli, SCBAL.tlh and SCBAL.tli that contain the generated smart pointer classes. There will be one smart pointer class per interface defined by SCOM, with the extension 'Ptr' to its name. For example, the ISCApplication interface exposed by SCOM will be accessed via the smart pointer ISCApplicationPtr, which is defined by #import.

Once SCOM is available for use, we can obtain a connection to its top-level interface, ISCApplication. It is usual for the connection to the ISCApplication interface to exist for the lifetime of the program, and to be created by a globally available class. For example, in a Document/View application the connection can be obtained when the document is created; in a standard MFC application, the connection may be created when the window is created.

Once a suitable class is identified, we add a data member to it for SCOM's application interface:

```
ISCApplicationPtr m_spApplication;
```

Since the m_spApplication member above is a smart pointer, it will be automatically Release() d when the document object is destroyed. Then, in a suitable place, we can use the CreateInstance method to connect to the ISCApplication.

```
BOOL CImageBrowserDoc::Initialize()
{
  // Create instance of the SCOM application object
  HRESULT hr = m_spApplication.CreateInstance(CLSID_Application);
  if (FAILED(hr))
  {
    TRACE0("Failed to create Symbian connect application object\n");
    return FALSE;
  }
...
```

Once we have access to SCOM methods, we can gain access to connected phones. In essence, this is straightforward, but we have the complications explained above in the C# example:

- Smartphones can be connected or disconnected while the application is active.
- More than one smartphone can be connected at a time.

In order to manage these phones, we will create a new class, `CConnectedPhone`, to contain the information that we want about each connected phone.

During the creation of a real application, we would add attributes to the `CConnectedPhone` class, but initially we can set it up with a minimum of data. We will store a reference to the SCOM device `ISCDevice2Ptr` and the device identification properties. We use the device identifier (normally the phone IMEI number) to provide a unique identifier for the phone, because we might have more than one phone of the same type connected. We store the device name as the name chosen by the user for the smartphone, and we store the device manufacturer and model number in order to decide what operations we can carry out. None of these properties can be changed by the application, so we only need to store them and provide accessor functions. This gives us an initial class declaration as follows:

```
class CConnectedPhone
{
public:
  CConnectedPhone(ISCDevice2Ptr aspDevice);
  ~CConnectedPhone();

  ISCDevicePtr GetDevicePtr();
  CString GetDeviceId();
  CString GetDeviceName();
  CString GetDeviceManufacturer();
  CString GetDeviceModel();

private:
  ISCDevice2Ptr m_devicePtr;     // Pointer to SCOM device object
  CString m_deviceId;            // ID of device
  CString m_deviceName;          // Name of the device
```

```
  CString m_deviceManufacturer;  // Manufacturer of the device
  CString m_deviceModel;         // Model number of the device
};
```

The constructor is able to set all the member variables from the `ISCDevice2Ptr` provided. The device ID, manufacturer and model are simply attributes of the phone that can be retrieved.

The device name is slightly more complex. You can choose any convention that you please, but it may be helpful to follow that used by other applications (so as to share a name). When a phone is first connected to the PC, it is referred to by the name of the manufacturer, for example 'My Nokia device' for a Nokia phone. However, the user can set their own name for the phone and this is then stored in the registry, indexed by the phone's device ID (IMEI number). If this name has been set then our application should attempt to use it, so that the user is presented with the familiar name. In this example, we will not provide code to edit the name, but it would be straightforward to do so. If the user has not set the device name then we will use the standard format based on the manufacturer. Note that, because the device name is stored in the PC registry, not on the phone, if the phone is connected to another PC, the name will not be carried over.

```
CConnectedPhone::CConnectedPhone(ISCDevice2Ptr aspDevice)
{
  // Cache the device pointer
  m_devicePtr = aspDevice;

  // The device ID, manufacturer and model are returned from SCOM
  m_deviceId = LPCTSTR( m_devicePtr->Id);
  m_deviceManufacturer = LPCTSTR(m_devicePtr->Manufacturer);
  m_deviceModel = LPCTSTR(m_devicePtr->Model);

  // The device name is for the convenience of the user.
  // It may have been stored in the registry or may be generated.
  CString subKey =
    "Software\\Symbian\\Symbian Connect QI\\Devices\\");
  subKey += m_deviceId;

  HKEY hKey;
  bool nameFound = false;

  if(ERROR_SUCCESS == ::RegOpenKey(HKEY_CURRENT_USER, subKey,
                                   &hKey))
  {
    //device is in registry - attempt to retrieve name
    //first check type and size
    CString valueName = "Name";
    DWORD type;
    DWORD size;
    long errorCode = ::RegQueryValueEx( hKey, (LPCTSTR) valueName,
                                        NULL, &type, NULL, &size);
    if(ERROR_SUCCESS == errorCode && REG_SZ == type)
```

```
    {
      //now retrieve it
      TCHAR * pData = m_deviceName.GetBufferSetLength (size);
      errorCode = ::RegQueryValueEx (hKey, (LPCTSTR) valueName,
                                     NULL, NULL, (BYTE *) pData,
                                     &size);
      m_deviceName.ReleaseBuffer();
    }

    if(ERROR_SUCCESS == errorCode)
    {
      nameFound = true;
    }
  }

  if(!nameFound) //generate one from the device name
  {
    m_deviceName.Format( "%s %s", m_deviceManufacturer,
                         m_deviceModel);
  }
}
```

At this point we have the code that we require to cache useful information about the phone when it is connected. The accessors are straightforward and will not be listed here.

Now that we know what to do with a smartphone reference, we need to fill in some form of collection to manage one or more smartphones. Because the device ID is unique, we will set up a map with that as the index field. We need methods to populate the phone map when the application runs initially and then to handle events when devices are connected or disconnected. This gives us a class declaration as follows:

```
typedef std::map<CString,CConnectedPhone*> ConnectedPhoneMap;

class CConnectedPhoneMap
{
public:
  CConnectedPhoneMap();
  ~CConnectedPhoneMap();

  void InitializePhones(ISCDeviceCollectionPtr deviceCollection);
  void AddPhone(ISCDevice2Ptr aspDevice);
  void RemovePhone(CString deviceId);

  CConnectedPhone* GetPhoneById(CString deviceId);

private:
  ConnectedPhoneMap m_phoneMap; // Map of currently connected phones
};
```

The `m_phoneMap` member is straightforward map – refer to the STL for details. The `InitializePhones` method is called from the class which provides the collection of devices from an attribute of the `ISCApplication` class as follows:

```
// Fragment of code from CSCOMDoc::Initialize
ISCDeviceCollectionPtr spConnectedDevices =
  m_spApplication->ConnectedDevices;

if(spConnectedDevices->Count) //we have connected devices
{
  m_phoneMap.InitializePhones(spConnectedDevices);
}
...

void CConnectedPhoneMap::InitializePhones(
  ISCDeviceCollectionPtr deviceCollection)
{
  // Create a map based on which devices are currently populated
  long deviceNumber = deviceCollection->Count;
  while(deviceNumber)
  {
    ISCDevice2Ptr spDevice = deviceCollection->Item[deviceNumber];
    AddPhone(spDevice);
    --deviceNumber;
  }
}
```

The actual work of creating `CConnectedPhone` objects, adding them to the map on creation, and deleting them and removing them from the map on destruction, is contained in the `AddPhone` and `RemovePhone` methods which can be called from outside the class as required:

```
void CConnectedPhoneMap::AddPhone(ISCDevice2Ptr aspDevice)
{
  CConnectedPhone *phone = new CConnectedPhone(aspDevice);
  if(phone)
  {
    CString deviceId = LPCTSTR(aspDevice->Id);
    m_phoneMap[deviceId] = phone;
  }
}

void CConnectedPhoneMap::RemovePhone(CString deviceId)
{
  ConnectedPhoneMap::iterator ixPtr = m_phoneMap.find(deviceId);
  if(ixPtr != m_phoneMap.end())
  {
    m_phoneMap.erase(ixPtr);
  }
}
```

Note that the `RemovePhone` method will need to become more elaborate to handle termination of any pending operations in a real application.

As well as maintaining our map of connected phones, we will record the currently selected phone. This concept is not necessary for all applications, but it is useful for most. We could present the user with a list-box

of devices and let them select one to operate on. We choose to record the current phone by maintaining a device ID within our controlling class:

```
CString m_currentPhoneId;         //Current phone
```

and we will provide an accessor and a method to set the current phone by ID:

```
CString GetCurrentPhoneId() const {return m_currentPhoneId;}

void CImageBrowserDoc::SetCurrentPhone(CString deviceId)
{
  m_currentPhoneId = deviceId;
}
```

When the application starts up, if any smartphones are connected then we will set the first smartphone in the map to be the connected smartphone. This is not entirely satisfactory, as the devices will be ordered by device ID, which is not terribly meaningful to the user, but the user will be able to change the current phone easily enough.

The call to `CConnectedPhoneMap::InitializePhones` provides us with an initial set of connected phones (which may be empty), but we need to register for events on device connection and disconnection and then handle them.

In order to receive events from SCOM, we need to use COM Connection Points. A special dispatch interface, `ISCEvents`, is defined by SCOM that can be used to receive notifications when events occur. To use this, several steps need to be taken. First we must provide a COM object that implements the dispatch interface, `ISCEvents`. There are several ways of doing this but we will use MFC, create a class derived from `CCmdTarget` (either directly or indirectly such as `CDocument`) and use this class's dispatch map.

The following line is added to the class constructor:

```
EnableAutomation();
```

Next, method declarations are added to the class declaration:

```
void DeviceConnected (IDispatch* apDispatch);
void DeviceDisconnected (LPCTSTR apDeviceId);
```

and the method bodies are filled in:

```
void CExampleSCOMDoc::DeviceConnected (IDispatch* apDispatch)
```

```
{
  ISCDevicePtr spDevice (apDispatch);
  AddDevice(spDevice);
}

void CExampleSCOMDoc::DeviceDisconnected (LPCTSTR apDeviceId)
{
  _ASSERTE (apDeviceId);

  RemoveDevice(apDeviceId);
}
```

In fact, in a real application we would need to update any smartphone list and possibly change the currently selected smartphone.

The exact syntax for these event handlers can be found by using a tool such as OleView to examine the SCOM type library.

Having defined the event handlers, we need to include them in MFC's dispatch map. In the class header file we declare the dispatch and interface maps:

```
DECLARE_DISPATCH_MAP()
DECLARE_INTERFACE_MAP()
```

and in its source file we wire in the dispatch and interface maps:

```
BEGIN_DISPATCH_MAP(CExampleSCOMDoc, CDocument)
  //{{AFX_DISPATCH_MAP(CExampleSCOMDoc)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceConnected", 1, DeviceConnected,
VT_EMPTY, VTS_DISPATCH)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceDisconnected", 2,
DeviceDisconnected, VT_EMPTY, VTS_BSTR)
... Additional event handlers
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

BEGIN_INTERFACE_MAP(CExampleSCOMDoc, CDocument)
  INTERFACE_PART(CExampleSCOMDoc, DIID_ISCEvents, Dispatch)
END_INTERFACE_MAP()
```

The syntax used by MFC for dispatch maps can be confusing (and I am not convinced that the use of 'helpful' macros to hide details from the developer makes it any easier), but these constructs do not change, so you can probably copy and paste them from an example in the SCOM SDK.

Finally, with the dispatch interface implemented, we can advise the SCOM application object of our sink interface. Normally, this is done immediately after the `ISCApplication` connection has been made:

```
LPUNKNOWN pUnkSink = GetDispatch(FALSE);
AfxConnectionAdvise(m_spApplication, DIID_ISCEvents,
                    pUnkSink, FALSE, &m_dwCookie);
```

`m_dwCookie` is a `DWORD` data member that belongs to the class. When we have finished handling events (normally on destruction of the controlling class), we must call `AfxConnectionUnadvise` and pass in the cookie again.

In order to use AFX we need to add the following line to the application's `stdafx.h` file:

```
#include <afxctl.h>
```

At this point we can access connected smartphones. The next stage is to navigate through the smartphone filing system.

5.11 Visual C++ Code for Drive and Directory Navigation

Just as in the C# examples, we use the `StorageDrives` member of a device to access the drives on the phone and then we use the `RootDirectory`, `ChildDirectories` and `ChildFiles` members to navigate around the drives, as the following, deliberately incomplete, example shows:

```
CConnectedPhone *currentPhone = GetPhoneById(GetCurrentPhoneId());
ISCDeviceStorageDriveCollectionPtr spDrives;
spDrives = currentPhone->GetDevicePtr()->StorageDrives;
int numDrives = spDrives->Count;
// Do something with each drive based on its path
for(int idx = 1; idx <= numDrives ; idx++)
{
  ISCDeviceStorageDrive2 *spDrive = spDrives->Item[idx];
  CString driveName = LPCTSTR(spDrive->Path);
  ISCDeviceStorageDirectory* spRootDir = spDrive->RootDirectory;
  ...
}

RecurseDirectory(ISCDeviceStorageDirectory *apDirectory)
{
  ISCDeviceStorageFileCollectionPtr spFiles;
  spFiles = apDirectory->ChildFiles;
  int numFiles = spFiles->Count;
  for(int fidx = 1 ; fidx <= numFiles ; fidx++)
  {
    ISCDeviceStorageFile *spFile = spFiles[fidx];
    CString filePath = LPCTSTR(spFile->Path);
    ...
  }

  ISCDeviceStorageDirectoryCollectionPtr spDirs;
  spDirs = apDirectory->ChildDirectories;

  int numDirs = spDirs->Count;
  for(int didx = 1 ; didx <= numDirs ; didx++)
  {
```

```
    ISCDeviceStorageDirectory *spDir = spDirs[didx];
    CString dirPath = LPCTSTR(spDir->Path);
    ...
    RecurseDirectory(spDir);
  }
}
```

Extensions of this code will allow us to populate a tree or list view with directories and files, for example. As a shorthand for direct access, we can use the `ISCDevice::GetStorage()` method to obtain a generic storage object which we can then query for either a directory or file interface.

```
CConnectedPhone *currentPhone = GetPhoneById(GetCurrentDeviceId());
ISCDeviceStoragePtr spDeviceStorage;
spDeviceStorage =
  currentPhone->GetDevicePtr()->GetStorage(_bstr_t("C:\\");
ISDeviceStorageDirectoryPtr spDir;
HRESULT hRes = spDeviceStorage.QueryInterface(
  IID_ISCDeviceStorageDirectory, &spDir);
if(SUCCEEDED(hRes))
{
 ...
}
```

5.12 Visual C++ Code for Synchronous and Asynchronous Operations

If we want to copy files or format drives on the phone then we will need to set up event handlers for the file copy and format events. This is done in the same way that we set up the device connected and disconnected event handlers.

First we extend the dispatch map:

```
BEGIN_DISPATCH_MAP(CExampleSCOMDoc, CDocument)
  //{{AFX_DISPATCH_MAP(CExampleSCOMDoc)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceConnected", 1, DeviceConnected,
VT_EMPTY, VTS_DISPATCH)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceDisconnected", 2,
DeviceDisconnected, VT_EMPTY, VTS_BSTR)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceFormatStorageDriveProgress",10,
DeviceFormatStorageDriveProgress, VT_EMPTY, VTS_I4 VTS_BSTR VTS_I4)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceFormatStorageDriveError", 11,
DeviceFormatStorageDriveError, VT_EMPTY, VTS_I4 VTS_I4 VTS_I4)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceFormatStorageDriveComplete",
12, DeviceFormatStorageDriveComplete, VT_EMPTY, VTS_I4 VTS_BSTR VTS_I4)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceCopyStorageFileProgress", 13,
DeviceCopyStorageFileProgress, VT_EMPTY, VTS_I4 VTS_BSTR VTS_BSTR VTS_I4
VTS_BOOL)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceCopyStorageFileError", 14,
DeviceCopyStorageFileError, VT_EMPTY, VTS_I4 VTS_BSTR VTS_I4 VTS_BOOL
```

```
VTS_PBOOL)
  DISP_FUNCTION_ID(CExampleSCOMDoc,
"DeviceCopyStorageFileExistingFileFound", 15,
DeviceCopyStorageFileExistingFileFound, VT_EMPTY, VTS_I4 VTS_BSTR VTS_DATE
VTS_I4 VTS_DATE VTS_I4 VTS_PI4 VTS_PBOOL)
  DISP_FUNCTION_ID(CExampleSCOMDoc, "DeviceCopyStorageFileComplete", 16,
DeviceCopyStorageFileComplete, VT_EMPTY, VTS_I4 VTS_I4)
  //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
```

and we add declarations for the event handlers:

```
void DeviceCopyStorageFileProgress(long aRequestId, LPCTSTR aFrom,
  LPCTSTR aTo, long aPercentComplete, BOOL * apCancel);

void DeviceCopyStorageFileError(long aRequestId,
  ScErrorDescription aErrorMessage, long aErrorCode,
  VARIANT_BOOL aCanContinue, VARIANT_BOOL * apContinue);

void DeviceCopyStorageFileExistingFileFound(long aRequestId,
  LPCTSTR aFileName,
  DATE aTargetFileDateTime, long aTargetFileSizeBytes,
  DATE aSourceFileDateTime, longG aSourceFileSizeBytes,
  ScOverwrite* apOverwriteFile, VARIANT_BOOL * apCancel);

void DeviceCopyStorageFileComplete(long aRequestId,
  long aCompletionCode);

void DeviceFormatStorageDriveProgress(long aRequestId,
  LPCTSTR apInformation, long aPercentageComplete);

void DeviceFormatStorageDriveError(long aRequestId,
  ScErrorDescription aErrorMessage, long aErrorCode);

void DeviceFormatStorageDriveComplete(long aRequestId,
  LPCTSTR apInformation, long aCompletionCode);
```

Finally, we would define the event handlers themselves. In a real application they will need to update progress dialogs or raise informative dialogs.

Having defined the event handlers, copying files involves just calling either the `CopyToPC()` method on an `ISCDeviceStorageFile` object or the `CopyFromPC()` method on an `ISCDeviceStorageDirectory` object.

The synchronous methods, such as `Delete()` and `Rename()`, require no special techniques to invoke.

6

Programming for Symbian OS

The possibilities of PC Connectivity applications are far from exhausted by the creation of applications written solely in PC-side code. SCOM provides easy access to file management methods, but we can create additional services by writing servers on a Symbian OS smartphone. Doing this requires the ability to program for Symbian OS.

Programming for Symbian OS is potentially a huge subject. There are a number of books, some produced by Symbian Press and some by external companies with extensive experience of programming for Symbian OS. Each of these books makes its own assumption about the type of software that the reader wants to create. The most common assumption is that the reader wants to create user interface applications, because these are the most commonly written programs outside Symbian and the smartphone manufacturers. However, Symbian OS PC Connectivity services do not require a user interface – the user interface is normally located on the PC, and so this book covers Symbian OS programming without regard to the phone user interface. This has the added advantage of making the examples applicable to more models of phones – user interface applications are more restricted because of user interface variations between phones.

This chapter is necessarily a summary; it contains enough information to cover the services described in later chapters, but neglects some areas that a more advanced developer might want to know about.

This chapter assumes that the reader knows about C++. It is possible to develop for Symbian OS using other languages (such as Java, OPL and even Visual Basic), but it is difficult to create PC Connectivity services in these languages, and C++ is the 'native' language for Symbian OS. Symbian OS uses a high proportion of the C++ language and so a Symbian OS developer needs to be comfortable with Object Oriented principles and how those principles are implemented in C++.

6.1 Building a Project

The first stage is to create a project so we can build our software. There are existing build mechanisms for Microsoft Windows development tools and for other environments. However, when building software for Symbian OS we want to be able to build for the emulator as well as for the target (ARM) hardware. The emulator is an environment that runs on a PC that makes development and debugging easier, and we will cover it in more detail below. For now, we need to be able to build with two (or more) development environments and so Symbian has developed build tools that allow us to do this from a common starting point.

6.1.1 `.mmp` and `bld.inf` Files

Each executable that we build is specified in a `.mmp` file. The name comes from a tool, `makmake`, which has been replaced by friendlier tools. The `.mmp` file lists the name and type of the executable that we are building, sets up some of the environment (notably the directories to be searched for source and include files) and lists the source files and libraries to be linked.

We can directly build the executable for any of the desired target environments via the `.mmp` file, or we can generate project files for our desired IDE (Metrowerks CodeWarrior) so we can build directly in the IDE.

Here is an example `.mmp` file – in fact it is the one we will use for our first PC Connectivity plug-in in Chapter 7.

```
TARGET         echocs.cs
TARGETTYPE     DLL
TARGETPATH     \SYSTEM\LIBS
UID            0x10003D52 0x101FEAFD //KCustomServerRemoteServerUnicodeUid

SOURCEPATH     .
SOURCE         echocssvr.cpp
SOURCE         echocssess.cpp

USERINCLUDE    .
SYSTEMINCLUDE  \epoc32\include
LIBRARY        euser.lib
LIBRARY        ectcpadapter.lib

DEFFILE        .\echocs.def
```

- The `TARGET` line specifies the name of the executable. Most executables will be either `.exe` or `.dll` files, but some types of plug-in have specific file extensions.
- The `TARGETTYPE` line specifies that the executable is a DLL. This one is a plug-in, but a stand-alone executable could be a `.exe`.

- The TARGETPATH line is optional. It specifies where the target will be placed. This has no effect on ARM builds, because a separate mechanism is used to locate executables, but it is relevant for emulator builds. Some plug-ins have fixed locations.
- The UID line specifies the type of the executable. The first value (0×10003D52) specifies that this is a specific type of PC Connectivity plug-in, and the second value (0×101FEAFD) differentiates between executables of the same type.
- The SOURCEPATH line specifies where source files are located relative to the .mmp file. In this case they are located in the same directory ('.' means the current directory). Some developers use multiple source directories and this line supports that.
- The SOURCE line lists the source files to be built. In this case there are two. It is possible to list more than one source file per line, but I find one per line convenient for change tracking.
- The USERINCLUDE line specifies where local include files are located. Again, this example just refers to the current directory, but a value such as ..\inc would refer to a directory named inc at the same level.
- The SYSTEMINCLUDE line specifies where system include files are located. The conventional location for this is \epoc32\include.
- The LIBRARY line specifies libraries to be linked to the executable. As with SOURCE lines, it is possible to specify multiple libraries on one line.
- The DEFFILE line specifies a .def file to be used. This is not relevant to .exe files but it specifies the ordinals for a DLL.

Having created your .mmp file, you still cannot build directly from it. First, you need to create a component definition file. This is always called bld.inf and lists the .mmp files to be built.

Here is the bld.inf file that accompanies the .mmp file shown above.

```
PRJ_MMPFILES
echocs.mmp
```

The bld.inf file consists of sections introduced by lines of fixed text. The PRJ_MMPFILES line introduces a section that lists .mmp files, one per line. In this case we will build just the one executable, but we could build any number.

The bld.inf format supports a range of sections that are useful in more advanced situations. For example, the EXPORT line introduces a list of files to be copied to the\epoc32\include directory. This is vital for

system components that want to publish their public header files, but is not necessary for an executable that is not intended to be linked against by any other component. Similarly, it is possible to specify exactly which targets the executable is to be built for. Omitting this will allow the executable to be built for all supported targets.

6.1.2 Build Commands

Given a directory with the `bld.inf` and the `echocs.mmp` file, the command

```
bldmake bldfiles
```

reads the files and creates a command file `abld.bat` that can be used to actually build the executable. If you look inside `abld.bat` (do not edit it), you will see that it just calls a Perl script and passes on arguments. The command

```
abld build winscw udeb
```

will build the executable for the emulator. In fact `abld` supports several commands, and these commands have options. Full information on the build tools can be found in the SDK, but `abld help` gives a quick list of the commands. For now, we will just touch on two of the commands. The `abld build` command allows the platform (or target) to be specified and allows a build (`udeb` for debug, or `urel` for release) to be specified. If the build is not specified then both debug and release builds are built. If the platform is not specified then all supported platforms will be built.

For emulator use we will normally use the `udeb` build. In the above example, `winscw` specifies the CodeWarrior emulator platform while `wins` specifies the older Microsoft Visual Studio emulator platform. `arm4`, `thumb` and `armi` are alternative ARM builds and you need to find out which build a specific Symbian OS smartphone expects.

Although we normally use the debug build for the emulators, we may use both debug and release builds for ARM platforms. Obviously, the release build should be used for final software, but the debug build can be used to enable logging.

The other command that we will use is

```
abld makefile
```

to create a project file for the CodeWarrior IDE. The project file can be found in the `\epoc32\build` directory. This allows you to create a project and then use the IDE for normal editing and building. Bear

in mind that you must not add files to the project or remove files from it in the IDE; if you do then the `.mmp` file will not be correct. Instead, you must exit from the IDE, edit the `.mmp` file and run `bldmake` again. This process is slightly more involved than always working directly in the IDE, but it does maintain both the emulator and ARM builds together.

6.1.3 Common Directories

When we have built our component or generated the project file, we need to know where to find it. Symbian OS uses a standard set of directories and you will very quickly find your way around them. The SDK that you install will include directions on setting up a development drive (the original Symbian OS building tools were intended to be used with a separate mapped drive for each development environment, but more recent tools support a more flexible approach). On that development drive you will find a directory named `epoc32`. The key directories are located under this point.

- The `BUILD` directory contains projects created by the `abld makefile` command.
- The `include` directory is where Symbian OS header files are located.
- The `release` directory is where all binaries for Symbian OS are located. This directory has sub-directories for each target, for example `winscw` and `armi`. Within each target directory are `urel` and `udeb` directories for release and debug builds respectively.
- The `gcc` and `tools` directories contain standard tools for building and similar tasks. Your path will normally need to be set up to include these directories.
- The `data` and `winscw` directories contain data files for specific targets.

You need to know a little more about the layout of the `release`, `data` and `winscw` directories in order to fully use them. When you build your component, the binaries will automatically be placed in the correct directories. When Symbian or a smartphone manufacturer builds a ROM image they use a file (called a `.oby` file) to control the copying of files that include binaries, resource files and other data files to an image that can then be used to create the ROM. The `.oby` file does away with the need for all the files to start in the correct directories, but often the directories are maintained for reasons of clarity. As a developer, you will not be creating ROM images and so you will not need all the files that make up a ROM. However, when you build executables for installation on a real phone, they will be located in the relevant `epoc32\release` directory.

A real Symbian OS smartphone will have two or more physical drives. It will always have at least a ROM drive, which is conventionally the `z:` drive, and a writable drive, often the `c:` drive. More and more modern smartphones have additional drives. These may be removable media such as MMC cards or Memory Sticks, and some phones have a RAM drive.

In contrast, the emulator normally has access to only one physical drive – that of the PC on which it is running. In order to emulate the Symbian OS smartphone, the emulator maps certain directories to the `z:` and `c:` drives. In fact, you can configure additional mapped drives if you need to. The `z:` drive is mapped to the `epoc32\release` sub-directory that you run the emulator from; commonly the `epoc32\release\winscw\udeb` directory and executables that you build for the emulator will be placed here or in a relevant sub-directory. The `c:` drive is mapped to `epoc32\winscw\c` and you can directly place files here or copy files from here.

6.1.4 More on UIDs

The example `.mmp` file listed above included a UID for the component. This number is not just one that I chose at random. To apply for a UID, email ***uid@symbiandevnet.com*** and ask for a range of numbers, typically up to 10. You can use values in the range `0×01000000` to `0×0fffffff` for testing, but you must not use these values for any software that you release.

6.1.5 `.def` Files and Freezing

If you build a DLL then it exposes (or exports) a number of methods that can be called from other executables. Methods are exported only when necessary and this can be a very good way of helping to maintain backwards compatibility. You can just declare and define your methods and leave it up to the compiler and linker to order and declare your exported methods. However, if you subsequently create a new version of the DLL and add a new exported method or change or delete an existing one, then the order and type of the exported methods will change. This means that any other component linked against the DLL will need to be recompiled.

If, for example, if you remove an exported method then any other component that uses will definitely need to be changed. However, if you leave existing methods unchanged and just add one or more new methods, then it should be possible to add these in such a way as to allow other components to use the original methods without having to be recompiled. For an isolated developer or development team, maintenance of backwards compatibility may be irrelevant, but for Symbian and the smartphone manufacturers it can have a huge effect on the integration

costs of developing a smartphone. It also allows application developers to sell their applications for more than one model of smartphone.

One of the requisites to maintaining backwards compatibility is the ability to control the exported methods from a DLL (there are other requirements in terms of managing the details of changes). Therefore the .mmp file specifies a .def file that states the ordinal for each exported method. The .def file for the example built above is very simple:

```
EXPORTS
    EchoCSServerNewL @ 1 NONAME
```

This component exports only one method. Because this is a PC Connectivity plug-in, it is not intended to export an API to other components – more normal DLLs may export a large number of methods. However, the DLL has to export at least one method in order to be used. In this case the NewL() method is exported to allow an instance of the server to be created. This is standard for this type of PC Connectivity plug-in and all components of this type must export exactly the same type of method (the class name built into the mangled name will be different, of course).

The good news is that you do not have to create this file manually. The command abld freeze will generate a .def file for you based on your binaries. Creating the .def file is referred to as 'freezing' the exports, because it defines them; the .def file is then maintained under source control along with the rest of your source and project files. Until you have frozen the exports in this way, you will get warnings when you build.

6.1.6 Making Installer (.sis) Files

Having built your executable, you will need to get it on to the Symbian OS smartphone. With the emulator, this is not a problem as the executable will be placed in the correct location; however, this does not work with a real smartphone. Simply copying files into the right locations is not practical for most users and, in any case, you may want to deliver an application via a range of connections, including over the air. Therefore, Symbian OS supports a software installer. The engine for this runs on the phone and supports an application with a user interface. The same engine can also be driven from the PC; in this case the user interface runs remotely, but the underlying behavior is common. Regardless of the location of the user interface, the install file format is the same. The generated install file is referred to as a .sis file, and the file that is used to generate the .sis file is a .pkg (or package) file.

Here is a .pkg file for an echo custom server plug-in (this is the example that we will build in Chapter 7).

```
; SIS package file for Echo Custom Server
;
; Language - only English but no included text anyway
&EN

; Caption, UID and version
#{"Echo Custom Server"},(0x101FEAFD),0,0,0

; Files to install
"\epoc32\release\thumb\urel\echocs.cs" -
 "!:\System\Libs\echocs.cs"
```

This particular package file is aimed at the Nokia 6600 and so uses a thumb build. If you write a custom server that is source compatible with multiple phones, then you must expect to build it multiple times and potentially create a package for each target phone (in practice, you may find that a single package may be compatible with a family of phones).

The `.sis` file can be generated with the command

```
makesis echocs.pkg echocs.sis
```

This command omits a certificate for the `.sis` file and so a user who tries to install this will receive warnings of unsigned software. Creating a certificate uses the `makekeys` command – type the command without any arguments to obtain a list of options – but creation of a trusted certificate, as opposed to a self-signed one, requires a key signing authority. Symbian runs a signing program for which you can obtain details from the Symbian developer network. Once you have a signed certificate file, you can add a reference to it into the `.pkg` file.

Once you have the `.sis` file, it can be installed directly from the PC or copied to the phone and then installed using the built-in application manager.

It is also possible to use the same install application to install Java applications and MIDlets, but this book will not cover creating services in Java.

6.2 Using the Emulator

6.2.1 What is the Emulator?

Earlier sections have mentioned the emulator. This section provides some more detail on its use. Developing for embedded hardware is more difficult than developing for hardware such as a PC: the emulator provides a stepping stone. The emulator is, as its name suggests, an emulation of Symbian OS on a Windows PC. This allows software to be developed and tested on the emulator before deployment on the target hardware.

The emulator does not emulate the actual target hardware, so it needs software built specially for it – you cannot take an executable built for ARM and run it in the emulator. We saw the command

```
abld build winscw udeb
```

in an earlier section being used to build a component for the Metrowerks CodeWarrior emulator.

The emulator is easier to use than the real Symbian OS smartphone for development reasons, because it has a simpler deployment cycle – you can just build and run a component directly for the emulator, whereas for a real smartphone you need to build the component, deploy it on the smartphone and then run it. Also, any log or data files are directly accessible when using the emulator, but have to be copied off a real smartphone. The final real gain is that it is possible to use the debugger that comes with Metrowerks CodeWarrior or one of the other development environments, which can drastically reduce development times, though unfortunately this may not always be possible with PC Connectivity because of interaction between the debugger and the connections used.

However, in some aspects that can make a real difference, the emulator is not the same as the target hardware. The most obvious difference is that the emulator will not have access to the same hardware as a real smartphone. The emulator will not directly be able to use telephony hardware or infrared or Bluetooth hardware. In some cases, there are ways to work round these limitations, but they require specialized hardware (in the case of Bluetooth) or specialized configurations (to use separate telephony hardware). Therefore, some types of testing will inevitably require use of the real smartphone. For these reasons, some versions of the emulator do not include the software for PC Connectivity.

A less obvious difference is that the emulator is not a perfect emulation of Symbian OS. Currently available versions of the emulator run all the Symbian OS processes in a single process on the PC. This allows some behavior on the emulator (such as accessing memory owned by another process) that does not work on target hardware, and there can be other effects. For these reasons, it is wise not to rely on the emulator too much; by all means use it, but test your software on a real smartphone at regular intervals to avoid surprises.

6.2.2 Starting the Emulator

The emulator is started by calling a program called `epoc.exe` from the relevant directory. To start the debug version of the CodeWarrior emulator, you `cd` to the `epoc32\release\winscw\udeb` directory and invoke `epoc.exe`.

The emulator can take a few seconds to start up – effectively it is booting Symbian OS. When it has started, you will see a window containing a view of a Symbian OS smartphone. The actual graphics (the fascia) is a bitmap, and the size and other details of the screen can be configured. Figure 6.1 is a screen shot of the TechView emulator.

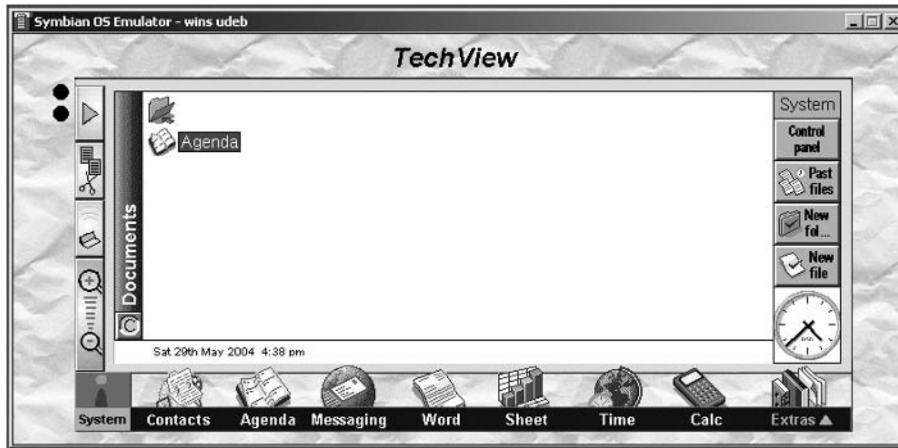


Figure 6.1 The TechView emulator

This emulator is based loosely on the old Psion Series 5 PDAs but is used for testing purposes. Nokia provides a Series 60 emulator and UIQ provides a UIQ emulator. The Series 60 emulator has the screen form factor of Series 60 smartphones as shown in Figure 6.2. The UIQ emulator has the screen form factor of UIQ smartphones as shown in Figure 6.3.

As you can see, the screen dimensions, icons and controls are different on all of these emulators. If you are creating a GUI application then these differences are crucial to good application design, but for servers without a user interface they are irrelevant.

You can change the configuration of your emulator, if desired, by changing the `epoc32\data\epoc.ini` file. This makes it possible to map extra drives or perform other simple emulator customizations if necessary.

The emulator also provides a console mode. In this mode no Symbian OS GUI is provided. A command-line based console runs instead, which is useful for automated testing purposes.

6.2.3 Driving the Emulator

The emulator provides a simulation of the screen of a Symbian OS smartphone and some simulation of input methods. The actual input methods supported by real Symbian OS smartphones vary and include

Figure 6.2 Series 60 emulator screen form factor

touch-sensitive screens, jog-dials, joysticks and a range of telephone keypad and other keys. If you are developing a GUI application then the input mechanisms are of critical importance, but PC Connectivity services normally do not expose a UI on the smartphone, so you need to know how to use the emulator only to run applications that you need. That is why this book contains almost no information on Symbian OS GUIs.

Figure 6.3 UIQ emulator screen form factor

You can use the Windows PC mouse to select screen areas or keys on the emulator, which is sufficient for most purposes. For some UIs, the F1 key is the menu key and will bring up the application menu. Once opened, use arrow keys to navigate around the menu.

6.3 Types and Naming Conventions

Symbian OS C++ has some naming conventions that you need to know in order to use it safely. At a minimum, you need to know what the

conventions tell you about system classes, but you would be well advised to follow the same conventions for your own code; your developers can just learn the conventions and follow them regardless of the source of classes.

The Symbian OS class naming conventions are based on the types of resources the classes own and how they should be destroyed. We will consider T-classes, C-classes and R-classes.

6.3.1 T-classes

T-classes are the simplest type of class; they do not own any separate data and do not need a destructor.

Examples of T-classes are the 'basic' types such as `Tint` and enumerated types (the enumerated class will be a T-class but the actual values conventionally all begin with E). More complex classes can be T-classes as long as they do not need a destructor. Although T-classes may not own external data, they may own pointers as long as they do not own the data pointed to.

Alongside T-classes for the basic types, constants conventionally begin with a K. The most common constant that you may see is `KErrNone`, which indicates success. The other common error values can be found in `e32std.h`.

6.3.2 C-classes

The next type of class is C-classes. These are the most common type of class. They all derive from `CBase` (either directly or indirectly) and they need a destructor. Because they have a destructor, they can own all manner of external data. One important feature of Symbian OS programming is the cleanup stack. This is covered in some detail in the next section, but for now it is sufficient to say that C-classes can be pushed on to the cleanup stack but must have a virtual destructor so that, when they are deleted as a `CBase*`, their destructors are called properly.

When a C-class is created, its contents are zeroed, so there is no need to set all the members if zero is an acceptable value.

6.3.3 R-classes

R-classes are classes that own resources. In a way, C-classes that have pointers to other objects on the heap that they own can be said to own resources, but the resources owned by R-classes are normally owned remotely. An example of an R-class is `RFile` which is a reference to a file. The file is not an object that is directly owned by the class. Instead, the `RFile` object owns a reference to the file that is maintained by the file server. References to objects owned by other servers are common

examples of R-classes. The contents of an R-class will be zeroed on creation and the object is attached to their resources by a method such as `Open()` or `Create()`; these may fail (if the server cannot be reached, for example), so do remember to check the return value.

In contrast to C-classes, R-classes do not derive from `CBase` and do not have a destructor. Instead, they have a method named `Close()` or something similar that must be called to free up the resources. The actual R-class will typically be small and resemble a T-class more than a C-class.

It is common to have R-classes as member variables of C-classes; the C-class destructor must call the `Close()` method of the R-class. However, remember that an R-class may fail to connect to its resource owner, and this must be taken account of in the C-class constructor.

6.3.4 M-classes

The final type of class to be covered here is M-classes. M-classes are interfaces (using the Java jargon) – that is, they are abstract classes that are used for multiple inheritance. The Symbian OS term for these is Mixin classes (hence the M). In later chapters we will encounter a number of Mixin classes, commonly for observer classes.

6.4 Error Handling

Error handling is a critical issue for Symbian OS programming. This is not because Symbian OS is more prone to errors than other embedded operating systems, but because Symbian takes robustness seriously.

There are a range of possible causes of error: running out of memory is the most obvious, but communications errors and other external events can occur. Programming properly for robustness is considerably harder than just programming for when everything goes according to plan, but Symbian OS provides some help in the form of `Leave`s and the cleanup stack.

6.4.1 Leaves and Traps

Leaving code is a form of exception handling. The `User::Leave()` method is equivalent to throwing an exception, and catching is achieved as part of a `TRAP` or `TRAPD` macro. The `User::Leave()` method takes one integer argument, which is the error code (this should not be `KErrNone`). If `User::Leave()` is called, control is immediately returned to the innermost `TRAP`. By convention, any method that can leave has a name that ends in `L` – this alerts developers who use the method to the possibility of a leave. Obviously, if a method does not itself call `User::Leave()` but calls other methods that can leave, then

it is itself a leaving function. There are a few exceptions to the naming convention – either because a meaningful method name ends with L but does not leave, or because a developer has neglected to put L on the end of a leaving function – but it is generally observed and you should follow it as well.

An alternative to leaving is to return an error value. Many methods in Symbian OS return a TInt error code rather than leaving. In general, a leave should be considered exceptional; if a method can commonly return an error then it should probably return an error code rather than leaving. Leaving code should normally be used where an error will cause control to unwind up a number of levels rather than just one. One reason for not using error codes everywhere is that setting up a TRAP macro and calling Leave has an overhead: it is slower than just returning an error code.

This is a fragment of code from a Connect plug-in from Chapter 10:

```
TRAPD(retVal, DoServiceL(aCmd, msgId));
if(retVal != KErrNone)
  {
  WriteErrorL(retVal, msgId);
  }
```

The TRAPD statement declares the variable retVal and then calls DoServiceL(). In this case DoServiceL() must be a method of type void. If DoServiceL() does not leave (which will be the normal case) then retVal will be set to KErrNone. If DoServiceL() does leave then retVal will be set to the leave code, which can then be handled. In this case, any error code will cause an error code to be returned, but other code might need to take different actions depending on the specific code.

6.4.2 The Cleanup Stack

At this point, the alert reader will be wondering what happens to any objects allocated on the heap when User::Leave() is called. Variables declared on the stack will be removed when the stack pointer is reset, but any objects left on the heap will constitute a memory leak and this is not acceptable.

To get around the problems of memory leaks in leaving situations, Symbian OS provides the cleanup stack. The cleanup stack is a stack of pointers (and some other information). When an object is created on the heap, a pointer to it can be pushed on to the stack. Whenever a TRAP macro is used, a marker is placed on the cleanup stack, and if a leave takes place then any objects left on the stack since the TRAP macro will automatically be deleted.

The theory is simple, but there are implications that need to be understood. Pointers to heap objects can be pushed on to the cleanup stack

but pointers to objects on the stack cannot. This means that an object on the stack will not get deleted during a leave, so you should place only T-class objects on the stack, not C-class objects.

The cleanup stack is a true stack – you push pointers on to the stack and pop them off. Therefore, it is not possible to pop an object off the stack unless it is the top object on the stack, since the sequence of pops must be the exact reverse of the sequence of pushes.

As one of the common causes of leaving is running out of memory, Symbian OS has evolved a pattern for construction of complex objects referred to as two-phase construction. In the first phase, an object is allocated on the heap, but the constructor must not leave itself and must not call any leaving functions. Once the object has been constructed, it can be pushed on to the cleanup stack and a second-phase constructor, normally called `ConstructL()`, can be called to set up the rest of the data. Often this behavior is wrapped up in a static method that will allocate and construct an object.

```
CRAgnCSServer* CRAgnCSServer::NewL(MCustomServerManager &aManager)
  {
  CRAgnCSServer* ecs = new (ELeave) CRAgnCSServer;
  CleanupStack::PushL(ecs);
  ecs->ConstructL(aManager);
  CleanupStack::Pop(ecs);
  return ecs;
  }
```

This example allocates a new object of type `CRAgnCSServer` on the heap and then pushes it on to the cleanup stack. Once it is safely on the cleanup stack, the `ConstructL()` method can be called. If this leaves then there will not be a memory leak, because the object will be deleted by means of the cleanup stack.

Note that the destructor for the class has to handle being called partway through the `ConstructL()` method in case of leaves. Therefore, it should not make any assumptions about the state of any of its members.

The `CleanupStack::Pop()` method can be called with no argument, in which case it just pops the top item off the cleanup stack. It can also be called with an integer argument, in which case it pops the specified number of items off the cleanup stack. However, it can also be called with a pointer as an argument. In release builds, this works in the same way as `CleanupStack::Pop()` with no arguments – it just pops the top item off the stack. In debug builds, it checks that the pointer matches the top item before popping it. If it does not, then it raises a panic (more on panics below). This means that any mismatched pushes and pops will be flagged at an early stage during development, which is why I recommend using this version of `Pop()`.

The example code above is about as simple as the use of the cleanup stack gets; the pop is very close to the push and it is clear what is going

on. Other patterns of use may have the Pop() further away from the PushL() and these can be trickier to develop.

It can be seen that PushL() is a leaving function (shown by ending in L). This is because it allocates memory to push the pointer on to the stack. What happens if the PushL() fails to allocate memory to push the pointer? That will itself cause a leave, so what happens to the pointer that we were trying to push? The answer is that the cleanup stack always maintains a spare entry at the end of the stack. The PushL() method pushes the pointer on to the stack and then tries to allocate the spare. If the allocation fails then at least the pointer has been pushed on and will be dealt with.

Having said that leaving functions end with L, one of the exceptions is functions that end in LC. The NewL() method shown above pushes the object under construction on to the cleanup stack and then pops it off before returning it. This leaves the cleanup stack in the same state that it found it. However, often the code calling NewL() to create an object wants to push the object on to the cleanup stack itself as part of its own logic. In this case, NewL() will push the object on to the stack and then pop it off again, and then the calling code will push it on again. Obviously, this is wasteful, so often classes have an alternative NewLC() method alongside the NewL() method. This creates the object but leaves it on the cleanup stack. In this case, NewLC() is a leaving method and the naming convention is extended to support LC as well as L.

Previously, we met three types of class: T-classes, C-classes and R-classes. The cleanup stack handles C-classes that are derived from CBase by calling their destructor when they are destroyed. This is why you must have a virtual destructor for any C-class, to allow the base destructor to access derived class destructors. It is also possible to push a pointer to a T-class on to the cleanup stack. Take a look in the SDK at the different overloaded versions of CleanupStack::PushL() and you will find a version that takes a TAny* pointer. However, if a T-class object is destroyed by the cleanup stack then its memory is simply freed – because it does not derive from CBase, there is no destructor that can be called.

Actually, the cleanup stack allows you to define the behavior required when an object is destroyed by explicit use of TCleanupItem. This will allow you to call CleanupClosePushL() to push a reference to an R-class object on to the stack. If the stack wants to delete the object, it will first call Close() on it.

Just as NewLC() was introduced to support a common situation, where a developer wants to create an object and leave it on the cleanup stack, sometimes a developer wants to leave an object on the stack until it is finished with and then pop it off and delete it. This can be done as a sequence, but Symbian OS provides a simpler method: CleanupStack::PopAndDestroy() pops an object off the stack and deletes it in one operation.

The cleanup stack methods are all static (that is, you do not need to provide a reference to a cleanup stack object), which avoids the need to pass a cleanup stack around your code. However, you do still need to create a cleanup stack at some point. If you are writing code that is a plug-in then the calling code will have created a cleanup stack, so you can just assume that one exists.

6.4.3 Panics

All the preceding discussion is about handling foreseeable runtime errors. Unfortunately, the same techniques do not work for programming errors. The key intention in such cases is to detect the error during development and debugging and, if absolutely necessary, to terminate a program neatly if the error occurs in a release environment.

Symbian OS provides a `User::Panic()` method that can be called. It takes a panic string of up to 16 characters and a panic code. If you call `User::Panic()`, your thread (actually your process) gets terminated and a message is provided. In a debugging environment, Symbian OS will provide you with as much information as it can; in a release environment, the information will be limited to the text and code that you provided. In a release environment, the end-user will be the person who sees the panic and may (you hope) report the error with the provided information.

The standard way of using panics is by means of asserts. Symbian OS provides the `__ASSERT_DEBUG` and `__ASSERT_ALWAYS` macros. They follow the same pattern of including a condition and an expression. The condition is evaluated and, if it is false, the expression is evaluated, which normally means calling `User::Panic()`.

There is a real difference of opinion about putting asserts into release code, but it is definitely good practice to use as many checks as possible in your debug code.

6.5 Descriptors

Descriptors are one of the unique aspects of Symbian OS and cause some confusion. In theory they are straightforward, but some developers have trouble working out which particular class to use for a specific purpose.

Descriptors were developed to address one of the common causes of errors with arrays in general and with C-style strings in particular. If you have a string (or an array) that is accessed by index, it is quite easy for a programming error to cause the index to go out of range. This can cause subtle and unpleasant defects and can be very difficult to trace.

Therefore, descriptors encapsulate both data and meta-data. They include (or refer to) an array of characters or other data along with the maximum size of the array and the current number of elements in the

array. Any attempt to access elements beyond current limits or to extend the array beyond its maximum size will result in an error.

As mentioned above, Symbian OS does not provide just one class for descriptors, but a range of classes. These all have more or less subtly different uses and it is necessary to understand them to work with Symbian OS.

The first distinction is between narrow and Unicode characters. Historically, Symbian OS was created for narrow, 8-bit character sets and was then successfully converted to support 16-bit Unicode characters. Symbian OS uses Unicode throughout in order to support a global range of locales and so in most cases you will use Unicode characters. However, there are some instances where true 8-bit data is used and so Symbian OS supports both. The descriptor class names incorporate the length of character used, so we have `TDesC8` and `TDesC16` classes. However, we mostly use Unicode and so `TDesC` is equivalent to `TDesC16`. If you want to use 8-bit characters you must do so explicitly.

The descriptor classes and their derivation are shown in Figure 6.4.

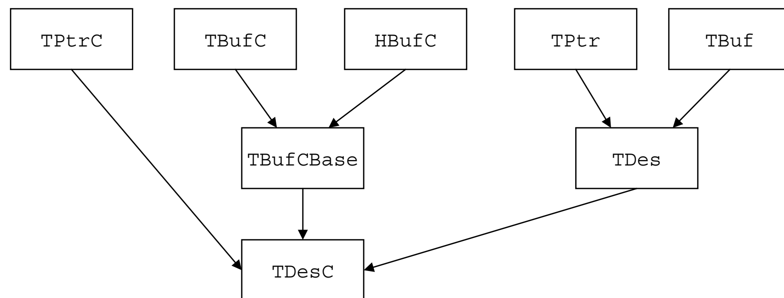


Figure 6.4 The descriptor classes

`TDesC` is an abstract base class for non-modifiable descriptors. It supports a range of methods that access but do not modify the contained data. Therefore, it has a length for the data but not a maximum length. The methods include a number of searching and comparison methods, and methods to provide the length of the data and to provide a pointer to the data. If you are using a Unicode descriptor then the length is the number of characters, not the number of bytes. `TDesC` also provides `Alloc()` methods to create an `HBufC` initialized from the descriptor.

`TDes` is an abstract base class for modifiable descriptors. It adds a maximum length and methods to modify the data. Data can be appended to the descriptor or inserted into it (as long as the maximum length is not exceeded) and there are a range of textual methods to alter the data in the descriptor.

The pointer classes, `TPtrC` and `TPtr`, contain a pointer to data, a length and (in the case of `TPtr`) a maximum length. These classes do not actually own the data pointed to. They are often used as a convenient

way to access data stored in another descriptor. They do not add any new methods to access or manipulate the data in the descriptor – these are provided by TDesC or TDes.

The TBuf and TBufC buffer descriptor classes contain their data as part of themselves and so they can be created on the stack. Because of the way they are declared, their maximum size has to be fixed at compile time.

The final type of descriptor is HBufC. An HBufC is also a buffer descriptor and contains its data within itself. However, it is allocated on the heap and so its size does not have to be fixed at compile time. This also means that you always refer to an HBufC* and never create an HBufC directly on the stack. Note that HBufC is a non-modifiable descriptor (hence the name ends in a C) and so you need to create a TPtr to point to its data in order to do anything useful with it.

One of the common uses of strings is to define literal strings. Symbian OS provides __LIT macros for this purpose. As with descriptors, there are 8-bit and 16-bit versions. If you omit a size, you will get a 16-bit version. The line

```
__LIT(KSomeText,"This is a test string");
```

declares a const, static TLitC variable called KSomeText of the correct length and sets its contents equal to the string provided. This constant can then be used wherever a reference to a TDesC is required.

While we are on the subject of text, it is worth mentioning briefly the CRichText class. This class allows you to build up a complex text object that includes embedded formatting. You will encounter it as the body type of message objects and it is ideal for HTML-type emails. We will not use any of its formatting properties in this book; if you want more information then refer to the SDK.

6.6 Arrays

Symbian OS does not support the Standard Template Library (STL) but has a number of container classes. One common need is for extensible arrays, and Symbian OS contains the useful classes RArray<class T> to hold fixed-size objects and RPointerArray<class T> to hold pointers to objects. There are other classes CArray... , but the RArray classes should be used in preference for efficiency reasons.

Full details of these classes can be found in the SDK, but they support appending, insertion and deletion of elements as well as direct access by index.

RArray classes do not own remote resources in the same way that typical R-classes do, but they do own a number of buffers on the local

heap, and the `Close()` or `Reset()` method must be called before the array is destroyed.

`RArray` classes also support sorting of their elements if you provide a comparison function.

6.7 Processes and Threads

Processes and threads are common to most operating systems and Symbian OS is no exception. Processes run in their own address space and are protected from each other. When a process is running, its memory is mapped so that local addresses map to global addresses. It is possible to communicate between processes, but this is relatively complex and relatively slow. Switching between processes requires the kernel to remap the memory for the new process. Therefore, passing a message from one process to another and receiving a reply requires two remapping operations as well as any other processing.

Threads run within one process and share the same memory. Switching between threads does not require memory to be remapped and so is faster, but because the threads share the same memory space they can affect each other directly. This can cause obscure defects and requires specific programming techniques to manage.

Symbian OS provides preemptive multitasking between threads. Whenever the kernel chooses to switch, it runs the highest-priority available thread (many threads may be suspended waiting for events) and manages CPU time between threads of the same priority.

6.7.1 Clients and Servers

One of the features of Symbian OS is the number of servers included. Symbian OS uses servers to manage scarce or shared resources. This includes fundamental resources, such as the filing system and the graphical display, as well more specialized ones, such as the Message system and communications peripherals.

In this book we will not create any full-blown servers ourselves; instead we will create a range of clients. However, we will still benefit from understanding how Symbian OS manages servers and clients.

Each server runs as a separate process and runs in its own address space. Although it is possible to create a server with clients in the same process, this is not common, and the clients are usually running in separate processes; therefore any communication between server and clients involves a context switch (or normally two). In addition, the raw APIs provided by Symbian OS for client-server communication are geared up to transfer small amounts of data. Therefore, frequent client-server communication or transfer of large amounts of data can be prohibitive.

There are techniques for easing client-server communications and these normally involve good design of the client API. Most servers provide such an API that is called from the client process and runs in that process. It is responsible for taking a meaningful API and packaging it up into raw client-server communication.

6.8 Active Objects

While concepts such as processes, threads and preemptive multitasking are common to many operating systems, Symbian OS includes another concept that is less common – cooperative multitasking. Preemptive multitasking may involve context switches and so has a cost, but Symbian OS is very heavily built on event handling, whether the events are user-generated or from external sources, such as incoming messages or data. Often, we do not need to respond to such an event immediately (at a lower level, interrupts do need to be serviced within a very short space of time, but further processing can often be delayed), and an approach that allows less preemption is likely to be more efficient.

Therefore, Symbian OS includes the concepts of an active scheduler and active objects. Each event handling thread has a single active scheduler. The active scheduler then manages a number of active objects and decides which one to run next. Active objects may not have any events pending, in which case they will not be considered for scheduling. Active objects may be assigned a relative priority if they are more or less urgent.

Because each active scheduler runs within a thread, the kernel will allocate CPU time between threads and thus between active schedulers.

6.8.1 `CActive` and `RunL()`

The key parts of the `CActive` class are as follows:

```
class CActive : public CBase
  {
public:
enum TPriority
  {
  EPriorityIdle=-100,
  EPriorityLow=-20,
  EPriorityStandard=0,
  EPriorityUserInput=10,
  EPriorityHigh=20,
  };
public:
  IMPORT_C ~CActive();
  IMPORT_C void Cancel();
  IMPORT_C void Deque();
  IMPORT_C void SetPriority(TInt aPriority);
  inline TBool IsActive() const;
```

```
  inline TBool IsAdded() const;
  inline TInt Priority() const;
protected:
  IMPORT_C CActive(TInt aPriority);
  IMPORT_C void SetActive();
// Pure virtual
  virtual void DoCancel() =0;
  virtual void RunL() =0;
  IMPORT_C virtual TInt RunError(TInt aError);
public:
  TRequestStatus iStatus;
private:
...
  };
```

Any active object must derive from `CActive` and must provide implementations of the `RunL()` and `DoCancel()` methods.

In explaining how to use active objects, I will work in reverse order. I will start with the `RunL()` method, which is the whole point of the active object, and then explain how to create and activate an active object. Finally, I will explain error handling and canceling of pending requests.

If an active object is successfully created and activated then the active scheduler will call the `RunL()` method. As you can see from the class declaration above, `RunL()` has no arguments. This should not be surprising if you consider that the active scheduler has to be generic and cannot include any knowledge of our specific implementation. The active scheduler does provide us with some information about the event that caused `RunL()` to be called, in that it puts a completion code in the `iStatus` member. In general, if you create an active object that can be used for more than one type of event, you will need to include additional member variables that hold some form of state, so you can work out in `RunL()` what the active object was supposed to be doing. One common pattern is to have a state or operation variable that is set when an event is queued, and then `RunL()` switches on the state or operation.

As the name indicates, `RunL()` is a leaving function and runs under a `TRAP` harness within the active scheduler. If it leaves then the `RunError()` method is called from the same object.

In order to run an active object, it must first be added to the active scheduler. In many cases the active scheduler will already exist, as it will be part of a framework. All the examples of PC Connectivity in later chapters have active schedulers already instantiated, but if one does not exist then it can be created by instantiating a `CActiveScheduler` object (or a derived object) and then calling `CActiveScheduler::Install()` to make it the current active scheduler.

Once you have an active scheduler, an instance of an active object can be added to it by calling `CActiveScheduler::Add()`. This is a static method, so you do not need a reference to the current active scheduler. Adding an active object to the active scheduler does not make

it a candidate for running – two more steps are necessary. Once an active object has been added to the active scheduler, it can be removed using `CActive::Deque()`.

First, an asynchronous operation must be triggered with the `iStatus` member of the active object. The way this is done depends on the event concerned, but any method that takes a `TRequestStatus& aStatus` argument is automatically an asynchronous method that will return immediately and call the active object whose `iStatus` member has been specified at a later time.

Second, the `SetActive()` method of the active object must be called. Once this has been called, the active object's `RunL()` method will be called when the relevant event takes place.

Note that an active object can only be associated with one event at a time – it has only one `iStatus` member. If you need to wait on multiple events simultaneously then you will need multiple active objects. The `CActive::IsActive()` method provides a way of checking whether or not the active object is already active, that is, waiting for an event.

Within the PC Connectivity plug-ins that we will create in later chapters, the `CActive` class is hidden behind other derived classes. Typically, we will have a method such as `ReadComplete()`, `Write-Complete()` or `ExecuteLD()` called to carry out some action. Most of the time, we do not need to be aware that we have been called from an active object's `RunL()`, but there are times when it will be relevant.

Many active objects have one-shot behavior. In these examples, the `RunL()` is called, carries out some action and returns, and the active object is inactive until it is triggered again by some external code. However, there are other possible patterns; an active object can reactivate itself or it can implement a long-running action in smaller chunks.

An active object might choose to reactivate itself to carry out an action on a more or less regular basis, for example. In this case, the `RunL()` method would carry out whatever action was required and then associate itself with an event and call `SetActive()`. This will relinquish control, but keep the active object active. If you use this pattern then consider how the active object will ever become inactive. Presumably there will be some condition under which it will not reactivate itself, or it will wait to be externally canceled.

The other pattern involves taking an algorithm that could be implemented in one sequence and breaking it up into smaller sections. This makes the algorithm and code more complex: the active object will require some form of state variable to store the progress, and the `RunL()` method will need to switch on the state. The advantage of such a design is that any other active objects in the same thread will get an opportunity to run, if they have a higher priority. This allows the long-running task to be carried out as a background task without making the whole thread unresponsive.

6.8.2 Canceling and Error Handling

In the previous section we mentioned that `RunL()` can leave. If an active object has a `RunL()` method that can leave then it must implement its own version of `RunError()` to handle the error. If an active object `RunL()` leaves then the active scheduler will call the `RunError()` routine of the same active object with the leaving code as the argument.

If the `RunError()` handles the error then it can return `KErrNone`. If it returns any other error then the `CActiveScheduler::Error()` method is called. The default implementation of this just panics, so if you expect your active objects to leave with unhandled errors then you should create your own active scheduler and override `Error()`.

Canceling an active object may be required for a number of reasons. Within the active object class, you must implement a derived `DoCancel()` method to cancel whatever request is pending. This means that the `DoCancel()` method must know which request might be pending if the active object supports more than one. `DoCancel()` is called from `CActive::Cancel()`, which checks that the active object is active; if it is then it calls `DoActive()` and then marks the object's request as complete (a cancel is thus really a request for early completion). If a request is canceled, the `RunL()` will not be called.

6.8.3 Active Scheduler Anti-Patterns

If you have had to create an active scheduler then the `CActiveScheduler::Install()` method will also start it. It is possible to create what are called nested schedulers or inner schedulers, but these are strongly deprecated. An active scheduler can be started and stopped using the `Start()` and `Stop()` methods, and it is possible to start an additional level of active scheduler and wait for it to complete. This is used to transform an asynchronous method into a synchronous one by waiting for it to complete.

Although there are apparent attractions to doing this, experience has shown that inner schedulers cause very obscure defects and more problems than they solve (in the case of premature termination, for example). Therefore, you should not use inner schedulers; instead use an alternative design that is not subject to the same problems.

6.9 Backwards Compatibility and Programming for Multiple Phone Types

When you develop a component for a Symbian OS smartphone, you may hope that it will work with all Symbian OS phones, but you may be disappointed. There are two reasons for differences: there are multiple

versions of Symbian OS, and smartphone manufacturers can make their own changes.

Symbian OS does not stand still, and newer versions have more features added to keep up with market needs. However, existing APIs are not changed unnecessarily. Therefore, if you are using an API in one version of Symbian OS, there is a good chance that it will be unchanged and usable in a later version – but this is not guaranteed. You should always test your software anew on each version or on each smartphone that you want to target.

The other reason for variation is the differences imposed by smartphone manufacturers. Although they build smartphones on a common base (Symbian OS) they each want to make better products than their competitors and so they may introduce new features which are not present in Symbian OS. Even if the features are introduced by more than one manufacturer, they are unlikely to create exactly the same API. This can be seen with the early camera-phones. The cameras were intended to provide competitive advantage and so they were not developed jointly with other manufacturers. Therefore, their APIs were incompatible.

Even where features are common to Symbian OS smartphones from a range of manufacturers, there may be variations. For example, file locations are often not mandated by Symbian OS and so manufacturers follow their own opinions. This leads to different phones having their agenda file or their camera images, for example, stored in different locations.

If you are writing a software component for exactly one model of Symbian OS smartphone then you just need to develop and test your software on that smartphone and not worry about other models. However, if you want to make the best use of your investment then you may want your software to be usable on as wide a range of phones as possible. In such a case, you need to be aware of the types of difference that may affect you and how to minimize their impact.

Because Symbian has created several versions of Symbian OS and the smartphone manufacturers have used them and wanted to reuse existing software, Symbian has evolved a relatively sophisticated view of the different types of compatibility.

At the most basic level, you may need to rebuild your software for different versions of Symbian OS or different makes or models of smartphone. Under some circumstances it may be possible to use an executable for one version of Symbian OS on a smartphone with a different version, but do not assume this.

The most likely level of compatibility is the level where you rebuild your software and it works. This can be true if all the APIs that you use are common and unchanged. The example PC Connectivity plug-ins created in later chapters all worked under both Symbian OS v7.0 and v7.0s. They should also work under Symbian OS v6.1 and possibly v8.0, but I have not tested these cases.

However, even though the executables were compatible with a range of Symbian OS v7.0 and v7.0s smartphones, I found that different manufacturers had located the agenda file in different locations. If I had made an assumption that the agenda file was always in the same place, my tests would work correctly on one smartphone but then fail on another. In fact, I built error checking into the code to make sure that I received a meaningful error in such cases. In order to handle the agenda file location, I could have built logic into the executable to work out the correct location. Instead, I deferred the problem to the PC software and just had the PC Connectivity plug-in accept the agenda file location as part of the protocol.

If you do need to make changes between versions of Symbian OS or between smartphones from different manufacturers then try to maintain protocol compatibility. If you were creating a GUI application then this might not be a relevant concept, but for PC Connectivity it is very important. If you maintain protocol compatibility then you can develop your PC software to interact with a wide range of smartphones, even if they have differences internally.

When you have written your software, you should try to test it on as many different smartphones as possible. If there are differences that you were unaware of, you want to be the one to find them – not your users.

7

Developing Custom Servers

7.1 Overview of Custom Servers

One of the common patterns that you will encounter in Symbian OS is that of plug-ins. Because Symbian OS is a platform as opposed to a one-off product, Symbian has to consider extensibility. If you were creating a closed phone that would have only a fixed set of features then you would probably fix them in libraries and avoid any need for expansion. However, Symbian does not know in advance what features each licensee wants to include on a phone or what features might be added after the phone is shipped.

Therefore, quite a few servers in Symbian OS provide their functionality by combining a central server, which exposes a common interface, with plug-in DLLs that can be loaded dynamically and provide support for specific features, protocols or media. In the PC arena, plug-ins have rightly become popular for supporting media formats: audio players may include plug-ins for MP3 or Ogg formats, and browsers support plug-ins to decode video and other media formats. In Symbian OS, media plug-ins are certainly used, but other servers also benefit from the concept. The Message Server uses Message Type Module (MTM) plug-ins (which we will meet in the next chapter) and the PC Connectivity subsystem uses plug-ins to provide connectivity services.

These were originally used in the days of PLP and have been carried forward with some modifications to allow custom servers designed for use with PLP to be used in later versions of Symbian OS with little modification. This results in APIs which are not necessarily ideal (and have some strong signs of their history – you will encounter some serious software archaeology), but remain usable. An alternative type of connectivity plug-in is a pipe processor. These have some advantages in design terms, but are more difficult to add to a phone unless you are creating the ROM for it. The custom servers are loaded by a process called `ectcpadapter`. This is itself a pipe processor invoked via m-Router and driven by commands sent from the PC.

The term 'custom server' is a poor choice of name, as Symbian OS is full of servers and 'custom' does not add much. However, as connectivity plug-ins can be used to provide a wide range of services, it might be difficult to find a better name (although 'connectivity server' springs to mind as a slight improvement). However, as with many software names, it is best just to accept it and not worry too much.

A custom server is a DLL which has an API that allows messages to be received from the PC and is able to pass responses back to the PC. The interfaces provided by Symbian OS are all concerned with loading the custom server and with communications. In order for a custom server to be useful, it must implement some actual functionality; exactly what is implemented is not constrained by the custom server interfaces – they are generic.

As well as providing interfaces that support receiving and sending of messages, the custom server infrastructure allows custom servers to be loaded by name at runtime based on commands from the PC. This means that it is not necessary for all the custom servers that are installed on the device to be loaded at the same time, with obvious savings in memory usage. Actually, the choice of which custom servers are loaded is made by the PC, so a PC suite could choose to load all custom servers. This would consume more memory on the device than loading the custom servers on demand but would produce a more rapid response when a specific custom server is required.

7.2 Limitations of Custom Servers

Some developers who have worked on or with custom servers have considered that they have undue limitations or drawbacks, one of which is that data is transferred between the smartphone and the PC in chunks of a fixed size. In fact, this is a reasonable restriction for embedded software. If a protocol allows indefinitely large packets then there is a risk of running out of memory (this has been seen in some components). Therefore, accepting that the protocol has a maximum packet size and planning for it is just good design.

One specific attempt to address some of the limitations of custom servers was the creation of `gencserv`, which is a custom server that itself loads plug-ins. `gencserv` was probably created to future-proof connectivity plug-ins by separating them from PLP. `gencserv` handles buffering and concatenation of large amounts of data and allows the creation of plug-ins that are one step further away from PLP or `ectcpadapter`. However, I find the custom server API pretty straightforward once it is stripped down to its basics, so I have not covered `gencserv` here. I hope that when you see the echo custom server you will feel the same way.

7.3 Custom Servers API

Custom servers are loaded by ectcpadapter (the commands are covered later) and then they receive and send data based on a simple API.

A custom server must implement classes that derive from two base classes (it may well have additional classes, but these are minimal). One class represents the custom server as a whole and the other represents a session. A single loaded custom server may support multiple sessions. This means that the sessions are responsible for most of the 'real' work.

A class derived from CCustomServer must be created that implements at least the NewInfoL() method to return information about the server, and the NewSessionL () method to generate a new custom server session.

A class derived from CCustomServerSession must be created that implements at least the AllocL () method to allocate a buffer for reading, and the ReadCompleteL () method to be called when a buffer has been received.

In addition to the two classes, the custom server must implement a standard E32Dll routine that does not have to do anything, and a single exported function, the first ordinal, that returns a pointer to a CCustomServer derived object.

We can see here the declaration for the CCustomServer class in full.

```
class CCustomServer : public CActive
  {
friend class CCustomServerSession;

public:
  IMPORT_C static void Delete(CCustomServer *aServ);

  inline TInt SessionCount() const;
  inline MCustomServerManager& Manager() const;
  inline void SetLib(const RLibrary &aLib);
  inline void DeferredDelete();
  inline TCustomServerServInfo& Info() const;
  inline RLibrary& Library() const;
  IMPORT_C virtual TCustomServerErrorResult Error(TInt aStatus,
                                                  TInt aRemErrorEvent);
  virtual CCustomServerSession *NewSessionL
    (MCustomServerWriter *aCSWriter, TVersion aVersion)=0;

protected:
  IMPORT_C void ConstructL(MCustomServerManager &aManager);
  IMPORT_C CCustomServer();
  IMPORT_C ~CCustomServer();

  IMPORT_C virtual void RunL();
  IMPORT_C virtual void DoCancel();
```

```
  virtual TCustomServerServInfo *NewInfoL()=0;

private:
  void ClearAllSessions();
  void DoStartL();

public:
  TDblQueLink iDblLink;

private:
  TDblQue<CCustomServerSession> iSessionList;
  TInt iSessionCount;
  TCustomServerServInfo *iInfo;
  MCustomServerManager *iManager;
  CCustomServerDeleter *iDeleter;
  RLibrary iLib;

protected:
  TDblQueIter<CCustomServerSession> iSessionIter;
  };
```

However, quite a few of the methods and members are not useful – they are historical leftovers or are exposed parts of the base class that are best ignored. For example, there is little that you can do with the `MCustomServerManager`, and the internals of the `CCustomServer` class are concealed within `ectcpadapter`, so there is insufficient information safely to override the `RunL()` method. The useful methods are covered below.

Class `CCustomServer`
Methods
`TCustomServerServInfo* NewInfoL ()` This pure virtual function is called when the custom server is instantiated and is used to return an object that contains information on the server priority, maximum buffer size, etc. The `TCustomServerServInfo` type is covered below.

`CCustomServerSession* NewSessionL`
  `(MCustomServerWriter *aCSWriter, TVersion aVersion)`
This pure virtual function is called to create a new `CCustomServerSession` derived object that does the actual work of the custom server.
  `aCSWriter` – a pointer to an `MCustomServerWriter` derived object that will be passed on to the custom server session.
  `aVersion` – the version of the custom server expected. This should be compared with the actual version of the custom server. If it is different then the routine should leave with error `KErrNotSupported`.

The `NewInfoL()` method returns a `TCustomServInfo` structure that contains priority and other information. The declaration for the class is as follows:

```
class TCustomServerServInfo
  {
public:
  TInt iDeleterPriority;
  TInt iReaderPriority;
  TInt iServerPriority;
  TInt iWriterPriority;
  TInt iMaxSessions;
  TInt iMaxPduSize;
  TPtrC8 iServerName;
  };
```

You can set the priorities to default values (all the Symbian standard custom servers use the default values, so that should be good enough for you as well), so you only need to attend to the maximum number of sessions, maximum PDU (Protocol Data Unit) size and server name. You can see how these are set in the example later in this chapter – but don't expect any surprises. The maximum PDU size should be chosen as a reasonable compromise unless you have specific needs: the PDUs should be large enough to allow most of your commands and replies to be passed in one unit, but they should not be so large as to impose an unreasonable burden on the Symbian OS mobile phone or so large as to reduce the responsiveness of commands. In future examples we will find cases where we may have an indeterminate amount of data to transfer (such as the content of all SMS messages or all contacts on the mobile phone) and we cannot make the PDU large enough for this, so we choose an arbitrary size.

Here is the declaration for the `CCustomServerSession` class in full.

```
class CCustomServerSession : public CActive
  {
friend class CCustomServer;

public:
  IMPORT_C ~CCustomServerSession();
  IMPORT_C virtual void Free(TDes8 *aPdu);
  virtual TDes8* AllocL(TInt aLen)=0;

protected:
  IMPORT_C void ConstructL(MCustomServerWriter *aCSWriter,
                          CCustomServer& aServ);
  IMPORT_C CCustomServerSession();
  IMPORT_C virtual void WriteCompleteL(TDes8 *aPdu);
  IMPORT_C virtual TCustomServerErrorResult Error(TInt aStatus,
                                                  TInt aRemErrorEvent);

  inline CCustomServer &Server() const;
  IMPORT_C virtual void RunL();
  virtual void DoCancel();

  inline void Write(TDes8 *aDes);
  inline TBool IsWriting() const;
public:
  virtual void ReadCompleteL(TDes8 *aPdu)=0;
```

```
private:

  enum {ECSIdle, ECSReading, ECSWriting} iMode;
  MCustomServerWriter *iCSWriter;
  CCustomServer *iServer;
  TDblQueLink iDblLink;
  };
```

Again, many of the methods are not useful. Here is information on the useful ones.

Class `CCustomServerSession`
Methods
`TDes8* AllocL (TInt aLen)` This pure virtual method is called to allocate a buffer for reading into. The buffer will be freed when the session is destroyed. `aLen` – the length of the buffer required. The maximum length that will be asked for has been set at the `CCustomServer` level.
`void ConstructL(CCustomServer& aServ, MCustomServerWriter *aCSWriter)` This method must be defined in order to call `ConstructL()` on the base class. `aServ` – a pointer to the owning `CCustomServer` object. Not necessary but it might be useful for data that is shared between sessions (if any). `aCSWriter` – this is required to pass to the base class constructor.
`TCustomServerErrorResult Error(TInt aStatus, TInt aErrorEvent)` This function is called to notify the session that an error has occurred. Little information is available on the meanings of the arguments, so there is little value in overriding it.
`void Free(TDes8 *aPdu)` This virtual function is called to free the buffer allocated with `AllocL()`. `aPdu` – the buffer to be freed.
`void ReadCompleteL (TDes8 *aPdu)` This pure virtual function is called when a complete buffer has been received. This is where most of the processing within the custom server will take place. `aPdu` – the incoming protocol data unit.
`void Write (TDes8 *aDes)` This function writes the data referred to by the descriptor to the PC. `aDes` – the outgoing protocol data unit.
`void WriteCompleteL (TDes8* aPdu)` This virtual function is called when a write operation is completed. It is not necessary to override it, but you should do so if you want to take action when the write action is complete. `aPdu` – the outgoing protocol data unit.

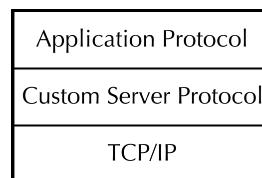
You can see that the `ReadCompleteL`, `Write` and `WriteCompleteL` methods manage receiving from the PC and sending data back to it. Internally, the data will be sent over a TCP/IP connection, but the details of this are hidden by these methods. Also hidden is a certain amount of buffering that goes on. This is dictated by the protocol used to communicate with custom servers.

7.4 Protocol Conventions

If you have any familiarity with communications protocols then you will know that they vary considerably and their design is based on the actual data to be transmitted. If the data is of a fixed size then the packets can also be fixed in their format and size. However, if the data is variable in size or content then a more complex protocol is necessary that allows the packets to completely define their contents. If the size variability is such that a single packet may not contain it, additional protocol elements are required to indicate continuation or termination packets.

The protocol used to communicate with custom servers has to support a totally extensible set of protocols built on top of it. By that I mean that the custom server infrastructure has no way of knowing what new custom servers will be written in the future. Therefore, it cannot really use fixed-sized packets (it could do so but this would probably be very wasteful). However, the protocol does not explicitly support messages that span more than one packet. If your protocol requires this then you need to build those features into your own protocol.

This will become clearer if you consider a 'stack' of protocols (a very common concept):



Here we have an underlying use of TCP/IP (which itself builds on lower layers that we do not need to consider here) on which the custom server protocol is built that allows packets to be transferred and that allows both sides to know when a packet is complete. On top of that lies the application protocol which is specific to the custom server. All custom servers use the same common protocol to receive and send packets, but they have different application protocols (actually, they may have some common elements such as command identifiers, but that is just because they are solving common problems).

The custom server protocol is solely concerned with reliably delivering packets of generic data and with indicating when a packet is complete. Therefore, the protocol simply relies on the first four bytes of a packet indicating the size of the packet (least significant byte first) and the rest of the packet being undifferentiated data.

However, the custom server infrastructure uses the packet length to ensure that the packet is complete and then strips it off before delivering it to the custom server. When you construct a packet on the PC, you must use the first four bytes to define the length of the rest of the data. However, the data that is received by the custom server is in the form of a descriptor, so the descriptor itself encapsulates information on the length of the data, and the four-byte length is not included in the data. In mirror fashion, when the custom server writes data back to the PC, it uses a descriptor whose length is set, so there is no need to explicitly include the length again. The custom server infrastructure constructs the packet to be received by the PC and puts the data length in the first four bytes.

Thus the PC always sees packets of the four-byte length form followed by data, while custom servers always see descriptors. We will see how this works in the example later in this chapter.

The use of descriptors in the custom server means that the length and the data are always consistent (the data may be garbage, but it will be of the chosen length and can be transmitted). However, because the PC software does not use descriptors, it is up to the developer to ensure that the data supplied matches the length. If you set a length and then send less data than the length promised, the custom server infrastructure will assume that the packet is incomplete and will buffer the data waiting for the rest of the promised data. If you set a length and then send more data than the length promised, the excess data is regarded as the start of the next packet.

If you try to send a data packet that is larger than the maximum PDU size that the custom server has set, it is not handled well; you will probably just lose the connection, so it is up to you to ensure that your packets conform to the protocol that you have designed.

7.5 Creating Your First Custom Server

Having presented the interfaces and discussed the protocols, we can now create our first custom server. For demonstration purposes we will build one that is about as simple as possible. We will create a custom server that just echoes data back to the PC. In later chapters we will see how to achieve more complex functionality, but for now we will introduce the custom server interface without any other code obscuring it.

For a non-trivial custom server, one of the first design tasks would be to define and document the protocol to be used. In this case, there really is no application protocol: whatever data is received in the packet is sent straight back.

This means that the echo custom server is effectively synchronous. We will see that the `ReadCompleteL()` method includes the code to respond. For many custom servers this will be the case, but some will trigger events that complete asynchronously and so need to respond in a different way.

Before we actually start writing code, we will need to include header files that declare the custom server classes that we will derive from:

```
#include <customserver.h>
```

Unfortunately, at least one SDK for Symbian phones misses out these header files. This is probably just an oversight. In this case, `customserver.h` also includes `customservershared.h` and `customserver.inl`. These three header files are in some SDKs and can be found on the website associated with this book, so that you can build your own custom servers. At the time of writing, these header files have not changed for any of the versions of Symbian OS for which they are valid, so you can just copy these three files into the `\epoc32\include` directory of your development drive and use them. If you do already have versions of these files then you should stick with those supplied with the SDK.

Diving straight in, here is the code for the echo custom server class `CEchoCSServer` constructor, etc.:

```
CEchoCSServer* CEchoCSServer::NewL(MCustomServerManager &aManager)
  {
  CEchoCSServer* ecs = new (ELeave) CEchoCSServer;
  CleanupStack::PushL(ecs);
  ecs->ConstructL(aManager);
  CleanupStack::Pop(ecs);
  return ecs;
  }

CEchoCSServer::CEchoCSServer()
  {
  }

CEchoCSServer::~CEchoCSServer()
  {
  }
```

This leaves us with two pure virtual functions to implement. The `NewSessionL` method is pretty standard:

```
CCustomServerSession* CEchoCSServer::NewSessionL
  - e(MCustomServerWriter *aCSWriter, TVersion aVersion)
```

```
  {
  // Check version
  if((aVersion.iMajor != KEchoCSMajorVersion) ||
     (aVersion.iMinor != KEchoCSMajorVersion))
    {
    User::Leave(KErrNotSupported);
    }

  CEchoCSSession* ecss = new (ELeave) CEchoCSSession;
  CleanupStack::PushL(ecss);
  ecss->ConstructL(*this, aCSWriter);
  CleanupStack::Pop(ecss);
  return ecss;
  }
```

One point to note is that the provided version is checked against the expected version and any mismatch causes the attempt to create a session to be aborted. Another approach would be to create the session anyway and pass in the requested version, but this requires the session code to handle the mismatch – the behavior above is simple and safe.

The `NewInfoL()` method here has a standard implementation that uses standard priorities and sets the maximum PDU size using a constant that is shared with the session class. It is very unlikely that we will need to set the priorities to any values except the standard ones.

```
TCustomServerServInfo* CEchoCSServer::NewInfoL()
  {
  TCustomServerServInfo *info = new (ELeave) TCustomServerServInfo;

  info->iServerPriority = ECustomServerConnectPriority;
  info->iReaderPriority = ECustomServerReadPriority;
  info->iWriterPriority = ECustomServerWritePriority;
  info->iDeleterPriority = ECustomServerLibDeleterPriority;
  info->iMaxSessions = 1;
  info->iServerName.Set(KEchoCSSvrNameDS);
  info->iMaxPduSize = KMaxEchoCSPduSize;

  return info;
  }
```

In addition to the class methods, we need two extra routines. We need the first because the custom server is a DLL:

```
GLDEF_C TInt E32Dll(TDllReason /* aReason */)
//
// DLL entry point
//
  {
  return KErrNone;
  }
```

and the other is the sole exported function:

```
extern "C"
  {
  EXPORT_C CCustomServer* EchoCSServerNewL
    (MCustomServerManager &aManager, const TDesC& /* aName */)
    {
    return CEchoCSServer::NewL(aManager);
    }
  }
```

This is the function that will be called by `ectcpadapter`. The `TDesC` argument is the name of the custom server. This is unlikely to be of much to you unless you want to have one DLL used under multiple names.

Any custom server class is likely to look very like this – there is little more to it than providing the configuration information and allowing a session to be created. The useful implementation details are included in the session class. This class also has a certain amount of ‘boiler plate’ code:

```
CEchoCSSession::CEchoCSSession()
  : iReadPtr(iBuffer, KMaxEchoCSPduSize)
  {
  }
void CEchoCSSession::ConstructL(CCustomServer& aServ,
                              MCustomServerWriter *aCSWriter)
  {
  CCustomServerSession::ConstructL(aCSWriter, aServ);
  }

CEchoCSSession::~CEchoCSSession()
  {
  }
```

You will see that the `ConstructL` method calls the base class `ConstructL` method. In this case there is no other local second-stage construction to carry out, but a more complex custom server might well require more here.

The next issue to address is the allocation of read/write buffers. In this case we can simply have a fixed-size buffer that is a member of the session class and can be used for reading.

```
private:
  // Buffer used for reading
  TPtr8 iReadPtr;
  TUint8 iBuffer [KMaxEchoCSPduSize];
```

Then the `iReadPtr` member can be initialized in the constructor and made available via the `AllocL()` method.

```
TDes8* CEchoCSSession::AllocL(TInt /*aLen*/)
  {
  return &iReadPtr;
  }
```

Obviously, we could implement more complex behavior here but, as we are only ever going to use the buffer for one purpose at a time with this custom server, this is quite sufficient.

At this point we have a custom server session that can be created and can allocate a buffer. All that remains is to do something with the data!

```
void CEchoCSSession::ReadCompleteL(TDes8 *aPdu)
  {
  Write(aPdu);
  }
```

As warned, this is about as simple as it gets. When a complete message is received, it is echoed back to the PC. In any real implementation, we would expect some real processing to happen before a response is returned. Note that the length of the data being sent is encapsulated in the descriptor and so does not need to be set in any other way.

At this point we can build the custom server. We will need a `.mmp` file, but for such a simple custom server there is not much to it:

```
TARGET        echocs.cs
TARGETTYPE    DLL
TARGETPATH    \SYSTEM\LIBS
UID           0x10003D52 0x101FEAFD //KCustomServerRemoteServerUnicodeUid
SOURCEPATH    .
SOURCE        echocssvr.cpp
SOURCE        echocssess.cpp

USERINCLUDE   .
SYSTEMINCLUDE \epoc32\include
LIBRARY       euser.lib ectcpadapter.lib

DEFFILE       .\echocs.def
```

There are some points to note here.

- The custom server is named `echocs.cs`. The `.cs` file extension is conventional for custom servers and differentiates them from other plug-ins.
- The target path is `\system\libs`. This is compulsory. If a custom server is not in this directory then it cannot be loaded.
- The two UIDs identify the custom server to Symbian OS. The first value, `0×10003D52`, is compulsory for all custom servers and identifies them as custom servers, while the second value must be unique.

Chapter 6 on developing for Symbian OS includes instructions on obtaining your own UIDs. In this case I have used one of a batch allocated for the writing of this book; please do not use any of these UIDs for your own production software.

- The `ectcpadapter.lib` includes the custom server libraries and must be linked against.
- A `.def` file must be defined as in Chapter 6.

Because the custom server is a DLL, it must have its exports properly defined and they must match what `ectcpadapter` expects. A custom server DLL exports only one function – the server `NewL()` function – so you can create the `.def` file manually rather than generating the `.def` file from the code by freezing. The `.def` file for the echo custom server is as follows:

```
EXPORTS
  EchoCSServerNewL @ 1 NONAME
```

Once you have sorted out building the custom server, just type the `bldmake` and `abld` commands and you should get `echocs.cs` built successfully.

7.6 Installing a Custom Server

Although the echo custom server has now been built successfully, you still need to carry out some more steps to get any response from it. First we will install it, and in the subsequent sections we will actually load it and communicate with it.

In order for a custom server to be loadable by `ectcpadapter`, it must be located in the `\system\libs` directory. The custom servers that are shipped with a Symbian OS phone are located on the `z:\system\libs` directory on the ROM, but that is no good to somebody developing add-ons. Luckily, it is also possible to load a custom server from `c:\system\libs` as well.

If you are running the emulator then building it will automatically put the binary in the right directory on the notional `z:` drive. If you are building a ROM for a reference board then you can add your binary to the ROM yourself. If you want to try your custom server out on a real Symbian OS phone then you have two ways to get it onto the device:

1. Use a file browser to copy the custom server directly. This is the easiest method for Connectivity developers.
2. Create a `.sis` installer file and then use the application installer.

The file browser option is quick and easy, and if you are writing Connectivity software then you have an advantage in that you may well already have a file browser available. However, you cannot expect an end-user to copy the binary manually. Using the file browser is perfectly fine during development, but eventually you will need to create a proper installer as described in Chapter 6.

7.7 Starting a Custom Server from SCOM

Once you have your custom server on the smartphone, you need to get `ectcpadapter` to load it. This is not difficult. You need to open a stream to communicate with `ectcpadapter` (which is always started when a connection is active) and send a command in the right format. The format is as follows:

Session Request Message

Op	Op	Op	Op	N[0]	...	N[31]	Maj	Min	Bld	Bld

Length: 40 bytes
Data Format:

- Op (4 bytes) – Operation Code. 0×00000000 (Start Session) is the only supported value
- N (32 characters) – Custom Server Name. The name of the custom server as an ASCII string (not Unicode) without the file extension and with the remainder of the string padded with zeros
- Maj (1 byte) – Major Version Number
- Min (1 byte) – Minor Version Number
- Bld (2 bytes) – Build Number

In response to this message, `ectcpadapter` will return a response message.

Session Response Message

Length: 4 bytes
Data Format:

- Result (4 bytes) – 0×00000000 indicates success, otherwise one of the error codes

If an error is returned then it will be one of the standard Symbian OS error codes – have a look in `e32std.h`. It is not possible to predict all the possible error codes, but two common ones are `KErrNotFound` (−1) if the custom server cannot be found, and `KErrNotSupported` (−5) returned by some custom servers if the version number is wrong.

Please note that there is no length field at the start of the message, because it is a fixed-length message and is not actually from a custom server.

If the session request is successful then the requested custom server will have been loaded and all subsequent data transfer on the socket/stream will be to the new instance of the custom server.

The custom server name is not case sensitive and omits the `.cs` file extension, so the extension is compulsory as you cannot load a binary with any other extension.

Given a connection to a Symbian OS smartphone, the following code will load our echo custom server:

```
// Read 32-bit integer LSB first
public int ReadInt32( ref byte[] aBuffer, int aStartPos)
{
  int retVal = aBuffer[aStartPos] | aBuffer[aStartPos+1]<<8 ;
  retval = retval | aBuffer[aStartPos+2]<<16 | aBuffer[aStartPos+3]<<24;
  return retVal;
}

...
try
{
  ISCBALDeviceService service = mDevice.Services["ectcpadapter"];
  System.Console.WriteLine("Ectcpadapter found - version {0} Address {1}
     Port {2}", service.Version, service.IPAddress, service.Port);
  ISCBALSequentialStream myStream = service.StartServiceOnStream();

  byte[] message = new byte[40]; // Zeroed by default
  message[4] = (byte)'E';
  message[5] = (byte)'C';
  message[6] = (byte)'H';
  message[7] = (byte)'O';
  message[8] = (byte)'C';
  message[9] = (byte)'S';
  // Operation and version left at zero
  int retval = myStream.Write(message);

  byte[] response = new byte[40];
  object oResponse = response;
  retval = myStream.Read(4, out oResponse);
  response = (byte[])oResponse;
  int responseCode = ReadInt32(ref response, 0);
  System.Console.WriteLine("session response {0} with {1} bytes read",
                           responseCode, retval);
}
catch
{
}
```

Try changing the name of the server or setting an invalid version to get one of the error codes.

There is no specific command to unload a custom server. It should be unloaded when the connection is broken, that is, when you destroy the stream used to communicate with it. It will certainly be unloaded when the connection between the PC and the phone is broken. If you use a lot of resources in your custom server and you want to ensure that they are freed up, you should add a command to your own protocol to free them rather than rely on unloading the custom server.

7.8 Communicating with a Custom Server

Once we have the custom server loaded, we can communicate with it. We will see in later chapters, in particular Chapters 10 to 13, that one of the major aspects of Connectivity programming at the lower level is concerned with putting data into packets to send to the phone and extracting data from packets received from the phone. We will develop some simple routines both on the PC and on the phone to handle the packing and unpacking, but this chapter will include only a minimum of data handling and I will concentrate on showing the other aspects of the communication.

As the `echocs` custom server just echoes data, we will simply allow the user to enter any text desired, send it to the custom server and then read it back.

To write the data to the phone, we receive text from the console, we write the length of the data, and then we write the data (remember that earlier I explained the protocol with the data length followed by the data).

```
// Write 32-bit integer LSB first
public void WriteInt32( int aValue, ref byte[] aBuffer, int aStartPos )
{
  aBuffer[aStartPos]   = (byte) (aValue & 0x000000ff);
  aBuffer[aStartPos+1] = (byte)((aValue & 0x0000ff00)>>8);
  aBuffer[aStartPos+2] = (byte)((aValue & 0x00ff0000)>>16);
  aBuffer[aStartPos+3] = (byte)((aValue & 0xff000000)>>24);
}

...
// establish connection to echocs
...
System.Console.WriteLine("Enter text to be echoed");
string text = System.Console.ReadLine();

// Write message length
int textLen = text.Length;
message = new byte[4];
WriteInt32(textLen, ref message, 0);
myStream.Write(message);

// Write the data
```

```
message = new byte[textLen];
for(int i = 0 ; i < textLen ; i++)
  {message[i] = (byte)text[i];}
myStream.Write(message);
```

As you can see, the `Write()` method in C# uses the fact that the boxed arrays know how long they are. Please note that the message is not sent (at least it is not received by the custom server) until it is complete, based on the four-byte length. This means that you can write the message in a number of sections, as above, or put together a single buffer and send that.

To read the data, we use the `Read()` method. We start by reading the four-byte message length and then we read the actual data. If your protocol is such that your packets are supposed to be of a fixed size then you might think that you could read the whole message in one operation, but I suggest that you stick with reading it in two stages.

```
retval = myStream.Read(message);(4, out oResponse);
int msgLen = 0;
if(retval == 4)
{
   byte[] buff = (byte[])oResponse;
   msgLen = ReadInt32(ref buff, 0);
   System.Console.WriteLine("Message length is {0}", msgLen);
}
else
{
   System.Console.WriteLine("Failed to read message length");
}
if(msgLen > 0)
{
  retval = myStream.Read(msgLen, out oResponse);
  if(retval == msgLen)
  {
    byte[] buff = (byte[])oResponse;
    for(int i = 0 ; i < msgLen ; i++)
    {System.Console.WriteLine("byte {0} = {1}", i, buff[i]);}
  }
  else
  {
    System.Console.WriteLine("Tried to read {0} but only read {1}",
      msgLen, retval);
  }
}
```

7.9 Asynchronous Communication

In this example, we have a very simple control flow: we carry out a synchronous read directly after the synchronous write operation. This will work if the custom server responds promptly and there are not too

many other operations taking up bandwidth to slow things down. In reality this approach may work satisfactorily for small operations, but some operations will take longer and we will need to allow for this. For these we will want to work asynchronously.

We can get an asynchronous stream simply by casting and then we associate event handlers with the stream. Handling writing of data is straightforward, so we will deal with it first. The `Write()` method is similar to that for synchronous streams except that it does not return the number of bytes written. The `OnWrite()` event handler is invoked when the bytes have been fed across the connection (actually, it is not really possible to know when they have reached the other end because of buffering at various stages). You could ignore the `OnWrite()` event handler and just pump data in with the `Write()` method. The value of the `OnWrite()` event handler becomes apparent when you want to transfer a large amount of data in multiple buffers. Then you can `Write()` the first bufferful and wait for the `OnWrite()` event before writing the next buffer. Other alternatives include just writing all of the buffers in turn with no delay (but the data will just get buffered up and you have a very real risk of either losing data or absorbing dangerous amounts of memory), or synchronously writing the data and waiting for a response message (which will work and is simple to program, but hogs the thread and makes it difficult to handle cancelation or other interruptions). The asynchronous `Write()` method is the most efficient solution.

Having made the case for a sophisticated use of asynchronous `Write()` operations, we are writing only one buffer but in two stages (length first and then data) in this example, so we will not really make use of the possibilities.

```
void OnWrite( int aError )
  {
    System.Console.WriteLine("OnWrite {0} error code", aError);
  }

...
private SymbianConnectBAL.BALApplicationAsyncStream mAStream;
...
//  Now create an asynchronous stream from the synchronous one
mAStream = (BALApplicationAsyncStream)myStream;
mAStream.OnWrite += new ISCBALSequentialStreamSink_OnWriteEventHandler
  (OnWrite);
```

```
System.Console.WriteLine("Enter text to be echoed");
string text=System.Console.ReadLine();
int textLen=text.Length;
```

```
message = new byte[4];
WriteInt32(textLen, ref message, 0); // Write message length
mAStream.Write(message);
```

```
message = new byte[textLen];
for(int i = 0 ; i < textLen ; i++)
{message[i] = (byte)text[i];}
mAStream.Write(message);
```

Having not really made use of the asynchronous possibilities when writing, we will try harder while reading. The actual API is straightforward, but effective use requires some planning. The `Read()` method returns immediately, but the data read (if any!) is not available until it appears in the event handler, so the handling of the data has to be included in the data handler. This means that we cannot have the simple sequential programming that we used with synchronous reads and writes, but this is part of the price we pay for the increased sophistication. This is also a pattern that is encountered in Symbian OS (in the custom servers, for example).

As with the synchronous `Read()` method, we need to set how many bytes we want to read. If we were using fixed-size buffers then we could set the whole read going in one operation, but it is more likely (and safer) that we will do it in two stages: we will read the length of the data first and then the rest of the data based on that length. In this case we just print out the received data to the console, but later chapters will cover more interesting tasks.

```
private bool mReadingLength;
...
mAStream.OnRead += new ISCBALSequentialStreamSink_OnReadEventHandler
  (OnRead);

...
// Try to read the length of data to come.
// The actual data will be received by the OnRead event handler
mReadingLength = true;
mAStream.Read(4);
...

public void OnRead( object aBuffer, int aError )
{
  System.Console.WriteLine("OnRead error {0} reading length {1}",
                           aError, mReadingLength);
  byte[] buff = (byte[])aBuffer;
  System.Console.WriteLine("Buffer length {0}", buff.Length);
  if(mReadingLength && buff.Length >= 4)
  { // We have just read the length of the buffer
    int msgLen = ReadInt32(ref buff, 0);
    System.Console.WriteLine("Message length is {0}", msgLen);
    // Trigger the read of the rest of the buffer
    if(msgLen > 0)
    {
      mReadingLength = false;
      mAStream.Read(msgLen);
    }
  }
```

```
  else if(!mReadingLength && buff.Length > 0)
  { // We have just read the body of the message
     for(int i = 0 ; i < buff.Length ; i++)
     {System.Console.WriteLine("byte {0} = {1}", i, buff[i]);}
  }
}
```

7.10 Debugging a Custom Server

In this chapter we have constructed a custom server and driven it from the PC. This is all quite easy when it works, but obviously not all development is flawless and sometimes we will have to debug our software. Unfortunately, there are some obstacles in the way to easy debugging of Connectivity software.

The first catch is that stopping on a breakpoint when using the debugger in the emulator stops all the processes in the emulator. This includes the keep-alives associated with the connection and so the connection between the PC and the phone is often dropped (some bearers allow for this – try it and see). This renders the debugger almost useless for Connectivity software, unless you are prepared to run to a breakpoint, stop and examine variables and then restart the whole process.

The second catch is that at least one SDK available actually omits the applications necessary to make a connection with the emulator at all. I assume that the people who put together the SDK didn't think it was possible to get an infrared or Bluetooth connection with the emulator and so did not try.

Either or both of these problems mean that you may well find yourself debugging using more primitive techniques than you might choose. However, it is still possible to debug Connectivity software.

The technique that I have used in writing this book is to use log files. I include some very simple logging functions in the custom server and call them when I want to know which methods are being called or what data is being passed around. The disadvantage of this approach is that I need to rebuild the custom server whenever I want to add or change the logging. This gives me a cycle of rebuild – copy cs to phone – run software – copy log file from phone – analyze log file – rebuild.

Some more sophisticated parts of Symbian OS include logging that is triggered by the existence of a specifically named directory. This is an elegant solution that allows logging to be switched on for selected components, but we will not need to use it as we are not creating components that will be used by other parts of the system.

One drawback that I have found is that errors in a custom server have a very good chance of crashing `ectcpadapter`. After all, custom servers are plug-ins and `ectcpadapter` does not appear to be very robust at handling error conditions.

8

Developing Socket Servers

8.1 Overview of Connectivity Socket Servers

Connectivity Socket Servers, sometimes referred to as Named Servers or Named Services (because they are accessed by name and because they provide a service to client software running on the PC), are used to provide PC Connectivity services from Symbian OS v8.0 onwards. They replace pipe processors and custom servers and try to make best use of the IP connection between the PC and the Symbian OS phone.

Connectivity Socket Servers use BSD-style sockets to run a TCP/IP connection directly rather than using a totally proprietary communication protocol. This is only sensible, as TCP/IP connections provide most of the functionality that they require and modern software development emphasizes standards and reuse wherever possible. The Connectivity Socket Servers do have some specialized, proprietary, aspects associated with service startup, but a developer familiar with sockets programming will have no problem creating Connectivity Socket Servers.

In essence, a Connectivity Socket Server is a Symbian OS server that listens on a TCP/IP socket and, when a connection is established by a client, responds to commands received from that client. The same description could apply to any server that uses TCP/IP, but there is a small specialization associated with starting a server.

Consider a file access server. If the server was listening on the Symbian OS phone on a fixed, known, socket number then a client running on the PC could connect to the server and start sending commands. However, if the server is started at boot time then it will constantly take up resources (RAM for sure, and probably a file server connection in this case) and it will be used for only a small fraction of the time. For some system services this is acceptable, but it would be wasteful and should be avoided if possible. Alternatively, the server could be manually started by the user, but that is inconvenient and would strike any user as clumsy. Therefore, we want a way to start the server on demand, so that it is running when the PC wants it, but not otherwise.

We also have an issue with using a fixed port number. The PC client needs to know which port number to connect on in order to access a server. A simple approach would be to use a fixed port number for each server, but this has some drawbacks. One is that it is frowned upon in the Internet community to use fixed port numbers unless they are registered (actually, any PC Connectivity services running on a Symbian OS phone may be specialized, and clashes with other services might not matter in practice). The other drawback is that it introduces a risk for third-party developers. Any third-party PC Connectivity developer has to create their server without knowing which other servers are under development. Therefore, there is a risk of two developers independently choosing the same port numbers (given the range of port numbers, the risk is small, but definitely non-zero).

The sockets library supports opening a random port number for listening, but the server then needs some way to communicate the port number to the PC client. In Symbian OS v8.0 the problems of starting a server and communicating a port number are solved by using a Service Broker. This is a server that listens on a fixed socket (with the port number set by the IANA) and supports commands that will start a server by name and then return the port number on which it is listening. In this way, the Service Broker needs to be active whenever the PC is connected and needs to have a fixed port number, but the same is not true of any other Connectivity Socket Server. Later in this chapter we will see how a socket server can register its name with the Service Broker and provide its port number, and how the PC client can use the Service Broker to make a connection to a Connectivity Socket Server.

It is possible to create a Connectivity Socket Server just by using the standard sockets API and the Service Broker, but some additional classes are provided that handle some of the tasks common to many Connectivity Socket Servers. Bearing in mind that these classes have been debugged and are already present on the Symbian OS phone, using them is likely to reduce your debugging time and reduce the ROM footprint of your service.

One side effect of using standard sockets is that a socket server could be accessed over a data connection other than from the PC (known as OTA or Over The Air). This has interesting possibilities (as well as raising some possible security issues), but, at the time of writing, there is no support for starting services in this way.

8.2 An Introduction to the Server Socket Classes

The major generic tasks that any Connectivity Socket Server will have are:

- open a socket for listening and inform the Service Broker of the port number

- wait on a socket for a connection from the PC and create a new connection on request
- when a message is received on a connection, decode it to decide what the command is, act on it and compose a response to send back to the PC.

In order to combine the generic behavior with specialized behavior, a set of base classes are used that encapsulate the standard functionality, along with a factory pattern and specialized classes to encapsulate nonstandard functionality.

8.2.1 The Server and Client Classes

The `CServerSocket` class is used to set up the listening socket and interact with the Service Broker. When a client connects to the listening socket, a `CClientSocket`-derived object is created and assigned to the connection. `CClientSocket` uses the `CMessage`class to send and receive data over the connection. The client socket stays connected until either the remote client disconnects or a write or read error occurs.

When designing a socket server, starting up can be delegated to the Service Broker, but the developer needs to consider closing the server. The Service Broker and the server socket classes make no rules about when a server should terminate. The server should not keep running indefinitely, otherwise it will absorb resources when the mobile phone is not connected to a PC, which would be wasteful.

The most common approach to closedown is to exit when the connection between the mobile phone and the PC is broken. This event can be detected using the `MServerSocketObserver`class. Other alternatives include defining a special command that allows the PC client to tell the server to close down, and terminating when all connections are dropped by the PC. Which one you choose depends on the protocol that you design and how you expect to use the server.

8.2.2 The Factory Class

The factory class (derived from `MFactory`) has the following three tasks.

- To create client socket objects derived from the `CClientSocket` class that can handle an incoming connection. Normally, developers do not need to worry about providing a client socket – they can just use `CClientSocket`, and `MFactory` by default returns an instance of `CClientSocket`. However, developers can extend `CClientSocket` for various reasons, such as to store session-based information.

- To create message classes. These classes are used to receive and send data on the socket. Messages are composed of a fixed-size header and a variable-length data trailer. Developers need to provide their own header class but can use `CMessage` for other purposes.
- To create command classes. These classes are used to handle messages received on the socket. The API is defined in `MCommand`. `CCommand` partially implements this API by providing some utility functions for sending errors and responses back to the remote client. When a message is received by `CClientSocket`, the factory is required to create a new command. The factory is passed a reference to `CClientSocket` and to the incoming message. The factory may therefore create different types of commands depending on the incoming messages. Typically, the factory has a table or a switch statement mapping message IDs to different command classes.

8.2.3 Command Classes

The key method in `MCommand` is `ExecuteLD()`. When a message is received on the socket and the factory has created a command that can handle the message, `CClientSocket` calls `MCommand::ExecuteLD()`.

Commands automatically take ownership of the incoming message and must destroy it when they no longer require it. The `CClientSocket` registers and retains ownership of `MCommand` objects (to allow for persistent commands, as we will see below), so when commands have finished executing they must unregister themselves by calling `CClientSocket::RemoveCommand ()` and then delete themselves. The `CCommand` class does this in the destructor, so developers need to be aware of this only if they are overriding the `MCommand` class directly.

Commands can send response messages back to the remote client by calling `CClientSocket::SendResponse ()` using an `MMessage`. Commands are responsible for managing the memory for the `MMessage` and typically reuse the incoming message. Only one message at a time can be sent via a `CClientSocket`. Messages are sent asynchronously, and when a message has been sent `MCommand::SendCompleteL()` is called to signal to the command that the message has been sent. Typically commands decide to delete themselves when their reply message has been completely sent. This is what is implemented in `CCommand::SendCompleteL()`. Developers need to override this if they intend to implement a different behavior.

As well as commands that send a reply and then unregister and delete themselves, it is possible to implement persistent commands, that is, commands that receive more than one incoming message and send zero or more reply messages. This sort of command might be used when it requires data that is too big for one message – the first message would set up the command and then the command persists and receives subsequent

messages until it has received all the data it requires, at which point it unregisters and deletes itself.

To support persistent commands, `CClientSocket` maintains a list of commands. When a new message is received, `CClientSocket` asks existing commands (in the order in which they were added) if they are interested in taking ownership of a new message by calling `MCommand::NewMessageReceivedL()`. Commands return true to `NewMessageReceivedL()` if they are interested in the new message (passed as a parameter in `NewMessageReceivedL()`). If no existing messages want the message or if there are no existing commands, `CClientSocket` asks the factory for a new command. The default `CCommand::NewMessageReceivedL()` always returns false, so persistent commands need to override this. Persistent commands that take ownership of a message have `ExecuteLD()` called as for new commands.

As a `CClientSocket` is destroyed when the socket is disconnected, it deletes the registered commands, so it is important that commands unregister themselves before deleting themselves.

8.3 Using the Service Broker API

This section covers the API exposed by the `RServiceBrokerClient` class. In addition, the Service Broker uses service registration files that are described in Section 8.6.

The `RServiceBrokerClient` class is part of the `conn` namespace.

Class `RServiceBrokerClient` – public `RSessionBase` Defined in `connect\servicebrokerclient.h`
This class is the client interface of the Service Broker. It allows named services to connect to the Service Broker and register the port on which they are listening. Note that if a server uses the `CServerSocket` classes then it does not need to use the `RServiceBrokerClient` class directly.
Member Methods
`RServiceBrokerClient()` The constructor takes no arguments.
`TInt Connect ()` This method connects the client session to the Service Broker server. returns – `KErrNone` or a standard error code.
`void Disconnect ()` This method disconnects the client session when it is no longer needed.

`TInt RegisterPort ( const TDesC& aServiceName, TUint aPortNumber )`
Informs the Service Broker that the named service is listening on the specified port.
`aServiceName` – the name of the service (must match the name used for registration).
`aPortNumber` – the port number on which the service is listening.
returns – `KErrNone` or a standard error code.

`TInt FailedToStart ( const TDesC& aServiceName, TUint aErrorCode )`
This method informs the Service Broker that the service failed to start. The error code will be returned to the client that tried to start the service.
`aServiceName` – the name of the service (must match the name used for registration).
`aErrorCode` – the error code to be passed back to the client.
returns – `KErrNone` on success, or a system-wide error code otherwise.

8.4 Server Socket Classes

The server socket classes are part of the `conn` namespace.

Class `MServerSocketObserver`
Defined in `connect\tserverinfo.h`

This mixin defines an observer for the server socket that gets notified if the socket is stopped due to an error.

Member Methods

`virtual void ServerSocketStoppedDueToErr (TInt aErr) = 0`
This pure virtual method is called when the server socket is stopping due to an error and can be used for cleanup tasks.
`aErr` – a system-wide error code for the error that is causing the server socket to stop.

Class `MFactory`
Defined in `connect\mfactory.h`

This mixin defines the methods that must be implemented by a message and command factory. The class derived from `MFactory` controls which classes derived from `CClientSocket` and `MMessage` will be created.

Member Methods

`virtual CClientSocket* NewClientL (CServerSocket* aServerSocket)`
This method returns a new instance of the `CClientSocket`-derived class associated with this command factory class.
`aServerSocket` – the base `CClientSocket`.
returns – an instance of a `CClientSocket`-derived class.

```
virtual MCommand* NewCommandL (MMessage* aMessage,
                                CClientSocket* aSocket) = 0
```

This method returns a new command based on a received message.

- `aMessage` – the received message that is assumed to include a command code.
- `aSocket` – the socket on which the message was received.
- returns – a new `MCommand`-derived object.

```
virtual MMessage* NewMessageL () = 0
```

This method returns a new instance of the `MMessage`-derived class associated with this command factory class.

Class `TServerInfo`

Defined in `connect\tserverinfo.h`

This class stores information regarding a server socket.

Constructor

```
TServerInfo(MFactory* aFactory, TUint16 aPortNumber,
            TUint16 aMaxConnectionQueue, const TDesC& aServiceName,
            MServerSocketObserver* aObserver)
```

- `aFactory` – pointer to a command factory for the server socket.
- `aPortNumber` – the TCP port number to be used for listening. If this is set to `0` then a dynamic port is used.
- `aMaxConnectionQueue` – the maximum number of pending connection requests. This is passed on to the listening socket.
- `aServiceName` – the name of the service.
- `aObserver` – pointer to an object derived from an `MSocketServerObserver` that will be informed if the socket is closed due to an error.

Member Variables

```
MFactory* iFactory
```

A concrete instance of a command factory.

```
TUint16 iPortNumber
```

The server port number.

```
TUint16 iMaxConnectionQueue
```

The maximum number of outstanding connection requests.

```
TPtrC iServiceName
```

The service name.

```
MServerSocketObserver* iObserver
```

The observer of the server socket.

Class `CServerSocket`
Defined in `connect\cserversocket.h`

The `CServerSocket` class handles connection requests and generates `CClientSocket` objects that use `CMessage` classes to send and receive data. `CServerSocket` hides a `CServerSocketImpl` class that is an Active Object that waits for connection attempts and processes them.

Static Methods

`static CServerSocket* NewL(const TServerInfo& aInfo)`
Creates a new `CServerSocket` object using information from the supplied `TServerInfo` object.
- `aInfo` – a `TServerInfo` object defining the desired port number, command factory, service name, etc.
- returns – a new `CServerSocket` object.

Class `CclientSocket`
Defined in `cclientsocket.h`

This class handles the creation of new commands based on received data. For normal, nonpersistent, commands it is not much used directly. If developing an unusual command, it provides direct access to the socket and the sending/receiving states (by means of methods not documented here).

Static Methods

```
static CClientSocket* NewL(CServerSocket* aServerSocket,
                           MFactory* aFactory)
```
This method creates a new `CClientSocket` instance and will be called by the socket server.
- `aServerSocket` – a `CServerSocket` instance.
- `aFactory` – an instance of a concrete command factory.

Member Methods

`void RemoveCommand (MCommand* aCommand)`
This method unregisters the command from the list owned by the `CClientSocket`. It should be called in the command destructor.
- `aCommand` – the command to be unregistered (normally the command being destroyed).

Protected Member Methods

```
virtual void ConstructL(CServerSocket* aServerSocket,
                        MFactory* aFactory)
```

Second-stage constructor for use by derived classes.
`aServerSocket` – a `CServerSocket` instance.
`aFactory` – an instance of a concrete command factory.

Enumerated Type `Serialise::TEndian` – This encapsulates whether messages are to be treated as Most Significant Byte (MSB) first or Least Significant Byte (LSB) first.
Defined in `connect\serialise.h`

- `EBig` – Big Endian – MSB first.
- `ELittle` – Little Endian – LSB first.

Module `Serialise`
This module consists of static methods to serialize variables into a buffer and to deserialize them out of a buffer. The methods can handle a range of data types and are intended to be used to unpack messages and pack responses.
Defined in `connect\serialise.h`

Module Methods

```
void Serialise (const TAny* aType, TInt aSize, TDes8& aBuffer,
                const TEndian& aEndian)
```
This method appends a variable to a buffer.
`aType` – the variable to be copied.
`aSize` – the size of the variable in bytes.
`aBuffer` – the buffer to which the data is to be appended.
`aEndian` – whether the data is stored MSB first or LSB first.

```
void DeserialiseL(TAny* aType, TInt aSize, TDes8& aBuffer,
                  const TEndian& aEndian)
```
This method copies data from a buffer into a variable.
`aType` – the variable to receive the value.
`aSize` – the size of the variable in bytes.
`aBuffer` – the buffer from which the data is to be removed. The data is removed from the beginning of the buffer.
`aEndian` – whether the data is stored MSB first or LSB first.

```
void Serialise(const TDesC& aString, TDes8& aBuffer,
               const TEndian& aEndian)
```
This method appends a descriptor to a buffer.
`aString` – the descriptor to be appended.
`aBuffer` – the buffer to which the data is to be appended.
`aEndian` – whether the data is stored MSB first or LSB first.

```
void DeserialiseL(TDes& aString, TDes8& aBuffer,
                  const TEndian& aEndian)
```
This method copies data from a buffer into a descriptor.
- `aString` – the descriptor to receive the value.
- `aBuffer` – the buffer from which the data is to be removed. The data is removed from the beginning of the buffer.
- `aEndian` – whether the data is stored MSB first or LSB first.

Class `MMessage`
Defined in `connect\mmessage.h`

This mixin defines the interface that must be implemented by concrete message classes. Such classes receive message header and data from the client socket. If the `CMessage` class can be used then it is not necessary to use `MMessage` directly.

Member Methods

`virtual TDes8& Data () = 0`
This method returns a pointer to the data that has been received or must be sent next. This might point to the header or to the body, depending on the receiving state.

`virtual void Receive () = 0`
This method initiates or resumes a receive operation.

`virtual void ReceiveCompleteL () = 0`
This method is called when a previous receive has completed. The state machine should be progressed in this function.

`virtual TBool IsReceiveComplete () const = 0`
This method returns True if the whole receive operation (header and body) has completed.

`virtual void Send () = 0`
This method initiates or resumes a send operation (header and body).

`virtual void SendCompleteL () = 0`
This method is called when a partial send has completed.

`virtual TBool IsSendComplete () const = 0`
This method returns True if the whole send operation (header and body) has completed.

`virtual void Reset() = 0`
This method resets an ongoing send or receive operation. Cancels any data content in the message body.

Class `MHeader`
Defined in `connect\mheader.h`

This mixin represents a message header. It allows clients to use `CMessage` instead of reimplementing the `MMessage` interface. A message header is normally concerned with unpacking data from part of the message into header variables and then packing up again for a response.

Member Methods

`virtual const TDes8& ReadPtr() const = 0`
This method returns the pointer for reading data from the socket. The buffer returned can be partially filled. The message class will try to read starting at the current length of this buffer up to its maximum length.

`virtual TBool ReadL(TUint aReadBytes) = 0`
The header is notified when the read operation completes. The number of bytes that have been received is passed as a parameter. The header can do any unpacking of data here.
 `aReadBytes` – the number of bytes received on the socket.
 returns – `ETrue` if all the header data has been received, `EFalse` otherwise.

`virtual void PrepareToWrite() = 0`
This method is called to prepare the header for sending, that is, data should be packed if necessary.

`virtual const TDes8& WritePtr() const = 0`
The pointer for writing data to the socket. Data will be written up to the current length of this buffer.

`virtual void SetBodyLength(TInt aLength) = 0`
This method sets the length of the message body. It is called by `CMessage` immediately before calling `PrepareToWrite()`.
 `aLength` – the length of the body.

`virtual TInt BodyLength() const = 0`
This method returns the current length of the message body.

`virtual TInt MaxBodyLength() const = 0`
This methods returns the maximum allowed length of the message body.

Class `CMessage` – public `CBase` , public `MMessage`
Defined in `connect\cmessage.h`

This class is a base implementation of a message derived from `MMessage`. It uses `MHeader` to encapsulate the message header.

Static Methods

`static CMessage* NewL(MHeader* aHeader, const TEndian& aEndian, CMessage* aMessage = NULL)`
This method returns a new `CMessage` object with the supplied header and message (if any) and with the 'endianness' of the data.
- `aHeader` – the message header to use. The `CMessage` takes ownership of it.
- `aEndian` – how to treat data in the message.
- `aMessage` – this argument is no longer used (it has not been removed for reasons of backwards compatibility).

Protected Construction Methods

`CMessage(const TEndian& aEndian)`
Constructor for the `CMessage` class.
- `aEndian` – how to treat data in the message.

`virtual void ConstructL(CMessage* aMessage, MHeader* aHeader)`
Second-stage constructor.
- `aMessage` – this argument is no longer used (it has not been removed for reasons of binary compatibility).
- `aHeader` – the message header to use.

Public Member Methods

`inline MHeader*& Header ()`
`inline const MHeader* const& Header() const`
This method provides access to the message header.

Protected Member Methods

`inline TState& State ()`
`inline const TState& State() const`
This method provides access to the current state of the message.

`inline HBufC8*& DataBuf ()`
`inline const HBufC8* const& DataBuf() const`
This method provides a pointer to the message data.

`inline TPtr8& DataPtr ()`
`inline const TPtr8& DataPtr() const`
This method provides direct access to the message data.

`inline const TEndian& Endian () const`
This method returns whether the message data is Big-Endian or Little-Endian.

Public Data Manipulation Methods
`virtual void Reset ()` This method resets an ongoing send or receive operation. It cancels any data content in the message body.
`virtual TDes8& Data ()` This method from `MMessage` provides direct access to the message data. It is probably better to use other methods to extract or add data.
`inline TUint Length ()` This method returns the length of the message data.
`template<class T> inline void Append (const T& aData)` This method appends data to the data trailer. `aData` – the data to be appended to the trailer.
`template<class T> inline void ExtractL (T& aData)` This method extracts data from the data trailer. `aData` – the data extracted from the trailer.
`template<class T> inline void PeekL (T& aData) const` This method peeks at data from the data trailer, that is, it returns the data but does not move the current data position, so the data can be read again. `aData` – the data peeked at from the data trailer.

`HBufC* ExtractLC ()`
`HBufC* ExtractL ()`
These methods extract an `HBufC` with a 16-bit length prefix. The caller takes ownership of the returned `HBufC`.

`HBufC* ExtractWithRawLenPrefixLC ()` This method returns an `HBufC` containing Unicode data. The data is prefixed with a 16-bit length (in bytes, not in Unicode characters). The caller takes ownership of the returned `HBufC`.
`HBufC8* ExtractDataL ()` This method extracts all the data from the message and returns it in one buffer. The caller takes ownership of the `HBufC`.
`TPtrC8 ExtractRawDataL ()` This method zeroes the data buffer and returns a pointer to it, so data can be added to the buffer.

Class `MCommand` Defined in `connect\mcommand.h`

This mixin defines the methods which a concrete command class must implement. It is created with a message, and the command is then responsible for the memory of this message. A pointer to the client socket is also passed as a parameter in the constructor. It can process more than one message.

Member Methods

`virtual void ExecuteLD() = 0`
Executes the command, called by the client socket when the command has been created, or after a persistent command has captured a message.

`virtual void SendErrorMessage(TUint aErrorCode) = 0`
This method sends an error message to the remote client.
`aErrorCode` – the error code to send to the remote client.

`virtual TBool NewMessageReceived(MMessage* aMessage) = 0`
This method is called by the client socket when a receive operation has completed. It gives persistent commands the opportunity of taking ownership of extra messages in addition to the one that created the command in the first place. The command implementing this method can take ownership of the pointer parameter, in which case `ETrue` should be returned. Alternatively, it may ignore the message (default behavior for all commands), in which case `EFalse` should be returned.
`aMessage` – the received message containing the data received.
returns – `ETrue` if the command has taken ownership of the message, `EFalse` otherwise.

`virtual void SendCompleteL() = 0`
This method is called when a send operation has completed. No error is reported by the send operation, because it is always assumed to succeed. Should it fail, the socket is closed and therefore `SocketClosing()` is called. Commands don't need to take any action on write failure – they simply cease to exist.

`virtual void SocketClosing() = 0`
This method is called when the socket is closing.

`virtual void TakeOwnershipOfMessage(MMessage* aMessage) = 0`
This method is called by `CClientSocket` after a new command has been created. The new command takes ownership of the message data.

Class `CCommand` – public `CBase`, public `MCommand`
Defined in `connect\ccommand.h`

This base class implements the interface specified in `MCommand`. Commands are created based on the message identifier using the command factory. It is created with a message, which is passed as a pointer, and the command is then responsible for the memory of this message. A pointer to the client socket is also passed as a parameter in the constructor. Persistent commands can process more than one message by registering with the client socket.

Constructor
`CCommand(CClientSocket* aSocket)` `aSocket` – the owning client socket.
Protected Member Methods
`virtual void SendResponse ()` This method sends the header followed by the rest of the message. It is the normal means of sending a response to the client.
`virtual void SendCompleteL()` This method is called when a send operation has completed. No error is reported by the send operation, because it is always assumed to succeed. Should it fail, the socket is closed and therefore `SocketClosing()` is called. Observers don't need to take any action on write failure – they simply cease to exist.
`TBool NewMessageReceived (MMessage* aMessage)` This method is called by the client socket when a new message is received to allow a persistent command to take ownership. The default version returns `EFalse`. This should be overridden only by those commands that receive the client request in more than one message. `aMessage` – this argument is not used. returns – `ETrue` if the command has taken ownership of the message, `EFalse` otherwise.
`virtual void SocketClosing ()` This method is called when the socket is closing.
`void TakeOwnershipOfMessage (MMessage* aMessage)` This method is called when the command is created to take ownership of the message.
`inline CClientSocket* Socket () const` This method returns a reference to the owning `CClientSocket`.

`inline CMessage*& Message ()`
`inline const CMessage* const & Message() const`
These methods provide access to the `CMessage` owned by the `CCommand`.

8.5 Developing an Echo Socket Server

In this section we will create an echo socket server that will accept a single type of command with accompanying data and return the same data as the response. We will build it from the bottom up by creating the command classes, then the factory, and finally the socket client and server.

This is the first place in this book where we will meet EKA2. The Symbian OS kernel developed in ER5 is designed for embedded devices – it is not a 'hard' real-time kernel. This means that it is not possible to

guarantee a response time. As smartphones incorporate multimedia and other increasingly demanding functions there is more need for such a kernel. From Symbian OS v8.0 onwards Symbian supports such a kernel – the name EKA2 stands for EPOC Kernel Architecture 2 – as an alternative to the existing kernel. The smartphone manufacturers choose which kernel to use and, at the time of writing, have not released details of which kernel will be used on their smartphones.

Apart from the internal changes, EKA2 has changes to the thread and process model, and the detailed APIs are slightly different in these areas. In this chapter the difference lies in the 'boiler-plate' used to create an `E32Main()`.

8.5.1 Command Classes

To deal with the incoming commands, we need classes for the echo command itself and for our header. We will also add a base class for commands. In this case the base command class is not strictly necessary for such a simple server, but it is useful when we have more commands, and we will include some code to handle unrecognized messages.

When we define the header class, we define at least part of our protocol, because the header class is responsible for unpacking and packing the message header. In this case we will use some fairly common parts. We will have a four-byte command identifier, a four-byte transaction identifier (so we can tell which command a response is linked to) and a four-byte length for the data trailer. In fact, we could use two-byte values for all practical purposes.

In this example, I will use the `conn` namespace explicitly.

This is the class declaration for our header class, with the members derived from `MHeader` and then our specific members.

```
class CHeader : public CBase, public conn::MHeader
{
public:
  /** The length of the message header */
  enum { KHeaderLength = 12 };

  /** The maximum length of the message data. */
  enum { KMaxMessageLength = 0xFFFF };

public: //From MHeader
  const TDes8& ReadPtr() const;
  TBool ReadL(TUint aReadBytes);
  void PrepareToWrite();
  const TDes8& WritePtr() const;
  void Reset();

  inline void SetBodyLength(TInt aLength);
  inline TInt BodyLength() const;
  inline TInt MaxBodyLength() const;
```

```
private:
  TUint32 iMessageId; // Message ID
  TUint32 iTransactionId; // Transaction ID
  TUint32 iLength; // Length of data trailer
  TBuf8<KHeaderLength> iData; // Buffer for header data

public:
  inline TUint32& Id();
  inline const TUint32& Id() const;

  inline TUint32& TransactionId();
  inline const TUint32& TransactionId() const;
};

// INLINE IMPLEMENTATION
void CHeader::SetBodyLength(TInt aLength)
  {iLength = STATIC_CAST(TUint32, aLength);}

TInt CHeader::BodyLength() const
  {return iLength;}

TInt CHeader::MaxBodyLength() const
  {return KMaxMessageLength;}

TUint32& CHeader::Id()
  {return iMessageId;}

const TUint32& CHeader::Id() const
  {return iMessageId;}

TUint32& CHeader::TransactionId()
  {return iTransactionId;}

const TUint32& CHeader::TransactionId() const
  {return iTransactionId;}
```

Then the methods derived from `MHeader` are straightforward. For reading and writing we provide access to the `iData` member.

```
const TDes8& CHeader::ReadPtr() const
  {
  return iData;
  }

const TDes8& CHeader::WritePtr() const
  {
  return iData;
  }

void CHeader::Reset()
  {
  iData.SetLength(0);
  iLength = 0;
  }
```

When our header data has been received, we need to unpack it. We use methods from the `Serialise` module and set up our data members.

```
TBool CHeader::ReadL(TUint aReadBytes)
  {
  iData.SetLength(aReadBytes + iData.Length());
  if (iData.Length() == KHeaderLength)
    {
    conn::UnpackL(iLength, iData, conn::ELittle);
    conn::UnpackL(iMessageId, iData, conn::ELittle);
    conn::UnpackL(iTransactionId, iData, conn::ELittle);

    return ETrue;
    }
  return EFalse;
}
```

We will use the same header for sending a response, so we need a corresponding method to put the data members back into the header part of a message.

```
void CHeader::PrepareToWrite()
  {
  iData.Zero();

  conn::Pack(iLength, iData, conn::ELittle);
  conn::Pack(iMessageId, iData, conn::ELittle);
  conn::Pack(iTransactionId, iData, conn::ELittle);
  }
```

Once we have our header class, we can define our base command. This has two purposes:

1. It provides implementations of methods that will be common to all commands, such as sending an error response.
2. It can be used by the factory class as a default command if it cannot recognize a received message. For this purpose, the `ExecuteLD()` method returns an error message to the client.

```
class CBaseCommand : public conn::CCommand
{
// Construction methods required for factory
public:
  static conn::CCommand* NewL(conn::CClientSocket* aSocket);
  CBaseCommand(conn::CClientSocket* aSocket);

// From MCommand
public:
  virtual void ExecuteLD();
protected:
  virtual void SendErrorMessage(TUint aErrorCode);

// Default routine to send a response
```

```
protected:
  virtual void SendMessage(TUint32 aMessageId, TUint32 aTransactionId);

// Client socket and access
private:
  conn::CClientSocket *iCs;
protected:
  inline conn::CClientSocket *Cs() {return iCs;}
};
```

The construction is straightforward and just saves the client socket. We do not have any second-stage construction in this case.

```
conn::CCommand* CBaseCommand::NewL(conn::CClientSocket* aSocket)
  {
  __ASSERT_DEBUG(aSocket, Panic(KNullPointer));
  return new (ELeave) CBaseCommand(aSocket);
  }

CBaseCommand::CBaseCommand(conn::CClientSocket* aSocket):
  conn::CCommand(aSocket)
  {
  iCs = aSocket;
  }
```

The `SendMessage()` method sets the message identifier and transaction identifier in the header and then uses the base `CCommand` to actually send the response.

```
void CBaseCommand::SendMessage(TUint32 aMessageId, TUint32 aTransactionId)
  {
  CHeader* hdr = STATIC_CAST(CHeader*, Message()->Header());
  __ASSERT_DEBUG(hdr, Panic(KNullPointer));
  hdr->Id() = aMessageId;
  hdr->TransactionId() = aTransactionId;
  CCommand::SendResponse();
  }
```

Given the `SendMessage()` method, the `SendError()` method just uses it with an error message identifier. The actual error code is set as the message body.

```
void CBaseCommand::SendErrorMessage(TUint aErrorCode)
  {
  CHeader* hdr = STATIC_CAST(CHeader*, Message()->Header());
  __ASSERT_DEBUG(hdr, Panic(KNullPointer));

  Message()->Reset();
  TInt32 error32 = aErrorCode;
  Message()->Append(error32);

  SendMessage(EREchoCmdError, hdr->TransactionId());
  }
```

As mentioned above, the `ExecuteLD()` method will be called only if a message is received that is not recognized.

```
void CBaseCommand::ExecuteLD()
  {
  SendErrorMessage(KErrNotSupported);
  }
```

Given the base command, the echo command is very simple (that was the whole idea of the base command class).

```
class CEchoCommand : public CBaseCommand
{
public:
  static conn::CCommand* NewL(conn::CClientSocket* aSocket);
  virtual void ExecuteLD();

private:
  CEchoCommand(conn::CClientSocket* aSocket);
};
```

Again, the construction is straightforward.

```
conn::CCommand* CEchoCommand::NewL(conn::CClientSocket* aSocket)
  {
  return new (ELeave) CEchoCommand(aSocket);
  }

CEchoCommand::CEchoCommand(conn::CClientSocket* aSocket):
  CBaseCommand(aSocket)
  {
  }
```

In the `ExecuteLD()` method we carry out the actual work of the command. Typically, this will involve reading data from the message body, doing some action and then sending a response. With this echo command, we will read text from the incoming message and then write it back into the message and send a response. (We could have left the message alone and sent it back, but I wanted to show the methods used to read and write data.)

```
void CEchoCommand::ExecuteLD()
  {
  CHeader* header = STATIC_CAST(CHeader*,Message()->Header());

  TUint32 messageId = header->Id();
  TUint32 transactionId = header->TransactionId();

  HBufC8* text = Message()->ExtractDataL();
```

```
Message()->Reset();
Message()->Append(text->Length());
for (TInt i=0;i<text->Length();i++)
  {
  Message()->Append(text->Ptr()[i]);
  }
delete text;

SendMessage(messageId, transactionId);
}
```

In a real implementation we would add further command classes, one for each possible command.

8.5.2 The Factory Class

The simpler tasks of the factory class are to create a new derived client socket and message objects on demand, but the more complex one is to create specific commands based on the message. Theoretically, you can do this in any way you please, but there is a standard way that works well and that you can copy. This involves defining a class that associates a message identifier with a method to create a new command.

```
typedef conn::CCommand* (*CreateFcn)(conn::CClientSocket*);

class TCreator
{
public:
  TCreator(CreateFcn aFcnL, TUint16 aId):
    iFcnL(aFcnL), iId(aId) { }

public:
  CreateFcn iFcnL;
  TUint32   iId;
};
```

This class will then be used to create an array of `TCreator` objects. This leads on to the declaration of our factory class:

```
class TFactory : public conn::MFactory
{
// From MFactory
public:
  conn::MMessage* NewMessageL();
  conn::MCommand* NewCommandL(conn::MMessage* aMessage,
    conn::CClientSocket* aSocket);

// Set of command creator objects
private:
  static const TCreator iCreators[];
  static const TUint iNumCreators;
};
```

In the factory class definition, we need to define our array of command creators. This does not need to be ordered in any way.

```
const TCreator TFactory::iCreators[] =
  {
  TCreator(CEchoCommand::NewL,EREchoCmdEcho),
  ...
  };

const TUint TFactory::iNumCreators =
  sizeof(TFactory::iCreators)/sizeof(TCreator);
```

This means that to add a new command we define the new command class and add a row to the array of command creators.

Given the array of command creators, the `NewCommandL()` method simply iterates through the array looking for a matching command identifier. If one is found then the command creator is used to obtain a command. If no match is found then we return a base command that we expect to return an error response (or just to leave when executed). By using the array of command creators, this code can just be reused unchanged for a new server.

```
conn::MCommand* TFactory::NewCommandL(conn::MMessage* aMessage,

                        conn::CClientSocket* aSocket)
  {
  __ASSERT_DEBUG(aMessage,Panic(KErrArgument));

  conn::CMessage* msg = STATIC_CAST(conn::CMessage*, aMessage);
  __ASSERT_DEBUG(msg, Panic(KNullPointer));

  CHeader* hdr = STATIC_CAST(CHeader*, msg->Header());
  __ASSERT_DEBUG(hdr, Panic(KNullPointer));

  for (TUint i = 0; i < iNumCreators; i++)
    {
    if (hdr->Id() == iCreators[i].iId)
      {
      return iCreators[i].iFcnL(aSocket);
      }
    }

  return new (ELeave) CBaseCommand(aSocket);
  }
```

The last part of the factory is responsible for returning a new derived message object.

```
conn::MMessage* TFactory::NewMessageL()
  {
  CHeader* hdr = new (ELeave) CHeader();
```

```
  CleanupStack::PushL(hdr);

  conn::CMessage* msg = conn::CMessage::NewL(hdr, conn::EBig);
  CleanupStack::Pop(); //hdr

  return msg;
  }
```

8.5.3 The Socket Client and Server Classes

In this case we do not have any need to create a derived client socket class. We will create a simple echo server socket class. This has to address several needs:

- It derives from `MServerSocketObserver` to be notified when the server socket stops.
- It owns a `CServerSocket` and a factory object.
- It provides a singleton reference to itself to allow global access. For the echo server we don't need this, but it can be useful when the server controls global resources.
- Otherwise, it is a standard server.

```
class CEchoServer : public CBase, public conn::MServerSocketObserver
{
public:
  static CEchoServer* NewLC();
  static void StartL();

  ~CEchoServer();

  // From MServerSocketObserver
  void ServerSocketStoppedDueToErr(TInt aError);

  // Singleton to provide global access to the server
  static CEchoServer*& EchoServer();

private:
  CEchoServer();
  void ConstructL();

private:
  conn::CServerSocket* iServerSocket;
  TFactory iFactory;
};
```

The singleton is straightforward:

```
CEchoServer*& CEchoServer::EchoServer()
  {
```

```
  static CEchoServer* _pThis = NULL;
  return _pThis;
  }
```

The creation and startup code is standard two-stage construction that creates and starts a `CServerSocket` object and runs it as an Active Object. We will need to create a `TServerInfo` object to configure the server socket. In this case we will set the port number to `0` to get a random, dynamically generated port number rather than a fixed number.

```
const TUint16 KEchoPortNumber = 0;
const TUint16 KServerQueueLength = 5;
_LIT(KEchoServerName, "com.symbian.echo");
```

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

CEchoServer::CEchoServer()
  {
  }

void CEchoServer::ConstructL()
  {
  conn::TServerInfo serverInfo(&iFactory, KEchoPortNumber,
    KServerQueueLength, KEchoServerName, this);

  iServerSocket = conn::CServerSocket::NewL(serverInfo);
  iServerSocket->StartL();
}

void CEchoServer::StartL()
  {
  CActiveScheduler* scheduler = new(ELeave)CActiveScheduler();
  CleanupStack::PushL(scheduler);
  CActiveScheduler::Install(scheduler);
  EchoServer() = CEchoServer::NewLC();
  scheduler->Start();
  CleanupStack::PopAndDestroy(EchoServer());
  CleanupStack::PopAndDestroy(scheduler);
  }

CEchoServer::~CEchoServer()
  {
  delete iServerSocket;
  }
```

In this case, if the socket server stops due to an error then we don't have anything to do except stop the Active Scheduler. This is normally called when the connection is dropped.

```
void CEchoServer::ServerSocketStoppedDueToErr(TInt aError)
  {
  CActiveScheduler::Stop();
  }
```

The final piece in the jigsaw is to set up the executable startup. The code for a Symbian OS EKA1 smartphone is:

```
TInt E32Dll(TDllReason)
  {
  return KErrNone;
  }

TInt E32Main()
  {
  __UHEAP_MARK;
  CTrapCleanup* cleanupStack = CTrapCleanup::New();
  TRAPD(error, CEchoServer::StartL());
  delete cleanupStack;
  __UHEAP_MARKEND;
  return 0;
  }
```

while the code for an EKA2 smartphone is:

```
GLDEF_C TInt E32Main()
  {
  __UHEAP_MARK;
  CTrapCleanup* cleanupStack = CTrapCleanup::New();
  TRAPD(error, CEchoServer::StartL());
  delete cleanupStack;
  __UHEAP_MARKEND;
  return 0;
  }
```

We can now build the echo socket server and try to run it.

8.6 Installing and Registering a Server Socket Service

Having built our echo socket server, it still needs to be installed and registered.

In order for a socket server to be started by the Service Broker, it must be located in the `\system\programs` directory and must be registered. The Connectivity socket servers that are shipped with a Symbian OS phone are located on the `z:\system\programs` directory on the ROM, so extra servers need to go in the `c:\system\programs` directory.

As with the custom server, building a socket server for the emulator will automatically put it in the right place on the mapped ROM drive. For

a real device or a reference board, we can either copy the binary into place or create a `.sis` installer file.

As well as placing the server in the right location, we need to create a service registration file for the Service Broker. This provides a name and version for the service and associates the name with the server executable. The service file is an XML file and has a very simple structure. One or more service files will exist in ROM on the mobile phone, but it is possible to create additional service files and place them on the `c:` drive. In fact, it is essential to create additional service files if we want to use additional services.

Here is a service file that will register our echo server.

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<service_registration>
  <service name="com.symbian.echo" exepath="echoss" version="1.0 0"
    processname="echoss" />
</service_registration>
```

The service name attribute is the name that will be published on the PC. You can use any name you like, but Symbian recommends the use of a reverse domain name convention (also seen in naming Java classes) to guarantee that your name is unique and will not clash with another server from another developer.

The `exepath` attribute is the name of the executable as fixed in your mmp file, but without the `.exe` or `.dll` extension.

The version attribute is a version string that will be provided to the PC as part of the information about the service. This can be used to detect different versions of the server and potentially enables the PC client to handle different versions by making allowances for the differences.

The process name attribute is used in EKA1 implementations to set the name of the loaded DLL.

You should note that building the server will not copy the service registration file into the right place; you need to place that manually if you are using the emulator. Service Broker registration files go into the `System\Data\ServiceBroker` directory, on either the `z:` drive or the `c:` drive.

The use of `makesis` was covered in Chapter 6, but here is a package file that will install the echo server socket with no certificate (you would provide a real certificate in practice). This is `echoss.pkg`.

```
; SIS package file for Echo Socket Server
;
; Language - only English but no included text anyway
&EN

; Caption, UID and version
#{"Echo Socket Server"},(0x101FEAFD),0,0,0
```

```
; Files to install
"\epoc32\release\arm4\urel\echoss.exe" -
  "!:\System\Programs\echoss.exe"
"echoservice.xml" -
  "!:\System\Data\ServiceBroker\echoservice.xml"
```

This package file is aimed at a reference board and so uses an ARM4 build. If you write a socket server that is source compatible with multiple phones then you must expect to build it multiple times and potentially create a package for each target phone (in practice, you may find that a single package may be compatible with a family of phones).

The `.sis` file can be generated with the command

```
makesis echoss.pkg echoss.sis
```

Once you have the `.sis` file, it can be installed directly from the PC or copied to the phone and then installed using the built-in application manager. In this case, I put the file in the `c:\System\Install` directory and was able to install it and then communicate with it.

8.7 Starting a Socket Service from SCOM

Once you have your socket server on the mobile phone, starting it is easier than with a custom server. On connection, SCOM (or BAL to be more precise) queries the Service Broker for the list of registered services. This means that the service can be accessed just by using the name to index the service collection.

Given a connection to a Symbian OS smartphone, the following code will load our echo custom server (it is much shorter than the equivalent code to load a custom server):

```
try
{
  SymbianConnectBAL.ISCBALDeviceServiceCollection serviceCol =
    mDevice.Services;
  SymbianConnectBAL.ISCBALDeviceService service =
    serviceCol["com.symbian.echoss"];
  SymbianConnectBAL.ISCBALSequentialStream myStream =
    service.StartServiceOnStream();
}
catch
{
}
```

It is possible to open more than one stream to a server. This is most likely to be useful if the server may be used by multiple clients (such as

the remote file server) and should be taken into account in the design of the server. When you close the stream, that particular connection will be destroyed, but any other connections will not be affected.

8.8 Communicating with a Socket Service

Once we have the socket server loaded and a connection established, we can communicate with it. As the `echoss` server just echoes data, we will simply allow the user to enter any text desired, pack it in a message with a valid header, send it to the server and then read it back.

To write the data to the phone, we receive text from the console, write the length of the data, and then write the data (remember that I explained the protocol with the data length followed by the data above).

```
// Establish connection to echoss

System.Console.WriteLine("Enter text to be echoed");
string text = System.Console.ReadLine();
int textLen = text.Length;

byte[] message = new byte[12];
int msgId = 0; // Message / command ID
int transId = 1; // Transaction ID
WriteInt32(textLen, ref message, 0); // Write message length
WriteInt32(msgId, ref message, 4); // message ID = echo
WriteInt32(transId,ref message, 8); // transaction ID
myStream.Write(message);

message = new byte[textLen];
or(int i = 0 ; i < textLen ; i++)
{message[i] = (byte)text[i];}
myStream.Write(message);
```

In this case, I have used literals for the message identifier and transaction identifier. In any real case we would not do this. We would have defined constants for the message identifiers and would probably have an incrementing transaction identifier (for debugging purposes).

To read the response, we use the `Read()` method and start by reading the 12-byte message header, including the message length, and then the actual data.

```
object oResponse;
int retval = myStream.Read(12, out oResponse);
int msgLen = 0;
if(retval == 12)
{
  byte[] buff = (byte[])oResponse;
  msgLen = ReadInt32(ref buff, 0);
  msgId = ReadInt32(ref buff, 4);
```

```
  transId = ReadInt32(ref buff, 8);
  System.Console.WriteLine("Message command {0} transaction {1}
                        length is {2}", msgId, transId, msgLen);
}
else // Didn't get the header correctly
{
  System.Console.WriteLine("Failed to read message length");
}

if(msgLen > 0)
{
  retval = myStream.Read(msgLen, out oResponse);
  if(retval == msgLen)
  {
    byte[] buff = (byte[])oResponse;
    for(int i = 0 ; i < msgLen ; i++)
    {System.Console.WriteLine("byte {0} = {1}", i, buff[i]);}
  }
  else
  {
    System.Console.WriteLine("Tried to read {0} but only read {1}",
      msgLen, retval);
  }
}
```

8.9 Asynchronous Communication

As with communicating with a custom server, we will probably want to communicate asynchronously for performance reasons. This works in just the same way using an `OnRead()` event.

8.10 Debugging a Socket Service

It is slightly easier to debug a socket server than it is to debug a custom server, because it is a separate process that can be debugged in its own right. The emulator bearer has a deliberately long timeout, so it is possible to run a socket server in the debugger and not lose the connection.

To debug a socket server using the Metrowerks IDE, load the socket server and run the debugger. This will start the debugger and the socket server will connect to the Service Broker (starting it in the process) and register itself before you connect the emulated mobile phone to the PC. This is not the usual sequence that you will find on a real mobile phone, where you will not get the socket server started until after a connection is made.

This has one drawback if you are not aware of the consequences. When you run the Connect UI application in the emulator to get the connection, it causes the `MServerSocketObserver::ServerSocketStopped`

DueToErr method to be called as part of a reset operation. Remember that for a normal implementation this will cause the server to exit. If we leave this in place for debugging purposes then we will connect the PC to the emulator and the socket server will immediately exit! To get around this problem, we want to either disable the MServerSocketObserver::ServerSocketStoppedDueToErr or, better still, make it use conditional compilation so that it is disabled only in a debug mode.

Once you have surmounted these hurdles, you can debug a socket server like any other Symbian OS process.

9

Introducing SMS and Messaging Classes

In this chapter, we are going to cover the Symbian OS Messaging subsystem in general and SMS in particular.

Messaging is one of the most important features of Symbian OS phones. It provides a good example of creating a Connectivity application that involves programming on the phone as well as on the PC. A bonus is that having an SMS management application is downright useful – I use the example I have built here to save me keying in messages using the numeric keypad.

The features that we are going to cover include:

- retrieving messages from the phone for display
- sending messages via the phone
- deleting messages on the phone
- picking up incoming messages to keep the list of displayed messages up to date.

This chapter will not be an exhaustive description of the Messaging APIs. Instead, it is intended to provide a general context and enough detail to write the SMS Management application.

The Message Server is a prime example of a system server in Symbian OS. It controls access to a system resource – the message store – and it supports plug-ins (called MTMs – Message Type Modules) for the specific types of messages. It also accepts requests from multiple clients; these requests can be synchronous or asynchronous.

Messages are stored as files in a dedicated directory and are accessed by means of Messaging APIs.

9.1 The Message Server and MTMs

The Message Server by itself cannot send or receive any messages, as it does not contain the message-type-specific knowledge required. Instead,

the Message Server uses MTMs that encapsulate the message-type-specific knowledge. MTMs have been created by Symbian and by phone manufacturers for a range of message types, including:

- email (IMAP, SMTP and POP3)
- fax
- SMS
- BIO messages (smart messages used for configuration and other purposes)
- infrared and Bluetooth OBEX messages
- MMS.

Over time, it has proved straightforward to add new MTMs for new message types – the architecture has proved its worth.

The MTMs are responsible for creating and sending messages, for receiving messages and for some aspects of displaying messages. Actually, some types of messages are received by means of watchers (SMS is one of these message types), but the general picture holds true.

Although I refer to an MTM, developers should be aware that, in fact, an MTM comprises a number of parts:

- a server-side MTM is a plug-in derived from the `CBaseServerMtm` class that is loaded directly by the Message Server and is responsible for the actual message transport, such as sending and receiving, if applicable
- a client-side MTM is derived from the `CBaseMtm` class and provides an API for messaging clients, such as the Messaging Application, that are specific to the message type
- a user interface MTM is derived from the `CBaseMtmUi` class and provides an API for use by a messaging application that has a user interface, making it easier to create a generic messaging application that can manage different types of message
- a user interface data MTM is derived from the `CBaseMtmUiData` class and encapsulates user interface data for the MTM such as icons for the Messaging Application.

These parts and their relationship are shown in Figure 9.1.

Only developers creating a new MTM or working on the Message Server need to care about server-side MTMs. The user interface and user interface data MTMs are used by GUI applications. As we are creating a non-GUI application, we will use only the client-side MTM. We are not creating a new MTM, so we do not need to go further into the structure of

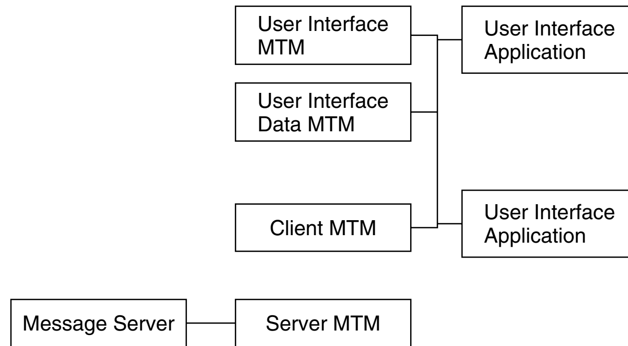


Figure 9.1 Message Type Module (MTM) general structure

MTMs. All we need to know is that we will use the SMS MTM for some of our tasks.

Although we are going to use only the SMS MTM, some applications need to use multiple MTMs and may need to use MTMs that their developers had not foreseen. Therefore the Message Server includes a registry of the available MTMs which can be used to iterate through the loaded MTMs. The MTM registry is responsible for loading MTMs on demand and for unloading them when they are no longer needed. We do not need to use these features.

One aspect to be aware of is that it is necessary to use a mixture of generic and specific APIs. It is hardly surprising that the Message Server does not provide APIs sufficient to manage all types of messages, and the MTMs do not provide full APIs of their own. This makes the learning curve for Messaging programming slightly steeper than might be expected.

Another aspect that affects the programmer starting to use the Messaging subsystem is that the Messaging APIs are powerful and sophisticated. They have been designed for functionality rather than ease of use, and they support a range of ways of working. For example, there are many asynchronous methods as well as the synchronous methods we will use. The asynchronous methods are very useful if you are writing a user interface: a synchronous method might leave the user interface unresponsive, so the asynchronous methods support a better user experience. However, the asynchronous methods are slightly more complex to program and we do not need them – we will make asynchronous calls from the PC, so we can handle synchronous behavior on the device.

There is one part of the Messaging subsystem that is generic and has been designed with ease of use in mind, and that is the Send-As API. This is intended for use by a wide range of applications to send data via a range of MTMs. However, the Send-As API does not cover retrieval of messages or other operations (nor does it fully send the message), so we will not use it.

9.2 The Structure of Messages

Although the Symbian OS Messaging subsystem supports a range of messages, all the messages share some common structures:

- a header that may include a subject and timestamp and from and to addresses
- a message body (normally Rich Text)
- zero or more attached files.

Not all types of message can have all these parts. For example, SMS and fax messages cannot have attachments, and MMS messages do not strictly have a body. Even when the parts are supported, they may be optional (for example, attachments are optional for email messages).

9.2.1 Headers and Bodies

All messages have some form of header data. This is most obvious for message types such as email and MMS, but even message types such as SMS and fax have information that is logically stored in a header, because it is meta-data as opposed to the actual content of the message. Header data includes:

- a message subject (actually an SMS message does not have a separate subject as such, but Symbian OS treats the start of the message text as a subject for compatibility with other message types)
- a 'from' address of some form
- one or more destination addresses
- a timestamp.

A header may include other data depending on the message type, such as encoding and character set or delivery priority.

In addition to the header, most messages have a body, which is stored in a `CRichText` object. Rich text is sufficiently versatile for most purposes, although it is excessive for an SMS. Exactly how the body text is encoded for sending or decoded when received depends on the message type and is the responsibility of the server-side MTM. For example, a fax message has its body rendered to a graphical form as it is sent.

The generic classes `TMsvEntry` and `CMsvEntry` expose structures that encapsulate much of the message structure. `TMsvEntry` contains information common to all message types that can be displayed in a message list in a Messaging Application. `CMsvEntry` provides access to the body text and message-type-specific headers and attachments. They

are supplemented by MTM classes that provide message-type-specific information and functions.

9.2.2 The Message Store and Caching of Data

Messages are persistent in Symbian OS. The user expects their SMS or email messages to be preserved, not simply displayed or sent and then discarded. Therefore, the Message Server manages the Message Store where all messages are stored. As this is a system resource that is used by multiple clients, the Message Server is responsible for controlling access and maintaining its integrity.

Regardless of the format used, all data can be stored in files, so the Message Store is a directory that contains sub-directories and files – the files are the message entries. You can see the Message Store by looking at the `/System/Mail` directory (by default on the `c:` drive, though it can also be created on another drive with more space). You will quickly see that the directory and file names are not meant for the casual reader. In fact, the Message Store is an indexed collection of files designed for rapid access. You should not directly alter the Message Store unless you know what you are doing, as you will almost certainly corrupt one or more messages or even the whole store. However, it can be useful for debugging purposes to take a complete copy of the Message Store, particularly with the emulator.

The data in message headers and bodies is accessed by means of APIs that expose the structure of the message. However, internally message headers and bodies are contained in stores in the files of the message entries. A store is a file containing one or more streams that are written to or read from by some software. Because the format and contents vary between message types, the MTMs are responsible for writing the messages into stores and for reading them out again. Unless you are implementing an MTM, you will not directly access the store for a message entry.

When required, the store for a message entry is available by means of the `CMsvEntry::ReadStoreL()` and `CMsvEntry::EditStoreL()` methods. The body text can then be extracted using the `CMsvStore::RestoreBodyTextL()` method and stored using the `CMsvStore::StoreBodyTextL()` method.

The data contained in `TMsvEntry` and `CMsvEntry` objects is not written to the Message Store whenever changes are made. This would be inefficient, particularly if a number of changes are being made to an object. Instead, these objects act as a local cache and the developer is responsible for writing the changes to the Message Store (normally when all changes have been made). For SMS messages, the `CSmsClientMtm::LoadMessageL()` method reads message data from the Message Store into objects, and the `CSmsClientMtm::SaveMessageL()` method writes changes back to the Message Store.

9.2.3 Attachments

Some types of messages can have attachments. These are just files; the messaging system has no interest in the content or format of these files, but it has to know that they exist or it cannot send them with the message.

For each attachment the Message Server creates a separate directory in the Message Store and a message entry (actually, this is not true for at least some MMS MTMs which are optimized to store multiple, small, attachments and which store them all in a single directory). The Message Server is considered the owner of the entry and the directory, but the client application is expected to copy the attached files directly into the directories. This exposure of the Message Store makes some operations very efficient – for example, the attachments of a received message can be accessed directly for reading purposes.

The message entries for attachments are created with the `CBaseMtm::CreateAttachmentL()` method and are children of the message entry.

Due to filing system overheads, the creation of a separate directory and message entry file for each attachment can be inefficient in terms of storage space if messages have many small attachments (such as MMS messages), so some MMS MTMs may have an alternative implementation which is more efficient.

9.2.4 Indices and Navigating

Although message data is heavily dependent on the type of the message, navigation around the Message Store is a generic requirement and so the `TMsvEntry` and `CMsvEntry` classes provide generic ways of navigating around the message store. A `TMsvEntry` object contains some standard pieces of information about a message, but it also contains the index identifier for the entry. Many of the MTM methods apply to a current context, that is, a currently selected message entry that corresponds to a `TMsvEntry` object and a `CMsvEntry` object, and obviously all non-static `TMsvEntry` and `CMsvEntry` methods apply to specific message entries.

Not all entries in the Message Store are messages. Message folders, such as the Inbox, Outbox, Drafts folder and Sent Messages folder, are also entries, as are attachments. In these cases they are entries that have child entries (the messages themselves). Message services are also entries, but we do not need to examine them in this chapter. Some entries have other entries as children (folder entries have message entries as children, and message entries may have attachments as children), and the Message Store can be navigated either as a tree of entries or directly to a desired entry.

The index identifier in each `TMsvEntry` object is a unique identifier that identifies each entry. There are some standard identifiers that are defined by constants. These are defined in `msvids.h` and include:

- `KMsvNullIndexEntryId` for a null, unused, entry

- KMsvGlobalInBoxIndexEntryId for the Inbox folder
- KMsvGlobalOutBoxIndexEntryId for the Outbox folder
- KMsvDraftEntryId for the Drafts folder
- KMsvSentEntryId for the Sent Messages folder.

Given these folder index values, it is possible to access all their child entries and so access all the messages in a selected folder. It is also possible to receive an entry identifier in isolation, for example as part of an event, and then navigate to the entry by means of CMsvEntry or MTM methods.

Given a CMsvEntry object with children, the CMsvEntry::Count() method provides the number of children, while the CMsvEntry::[] operator, CMsvEntry::Children(), CMsvEntry::ChildrenWithMtmL(), CMsvEntry::ChildrenwithTypeL(), and CMsvEntry::ChildDataL() or CMsvEntry::ChildEntryL() methods provide access to the children or a selection of them.

The CMsvEntry::SetEntryL() method allows navigation directly to an entry given its index identifier (for example, if the index identifier has been provided as part of an event). Similarly, the CMsvSession::GetEntry() and CMsvSession::GetEntryL() methods allow access to a TMsvEntry or CMsvEntry object given an index identifier.

New message entries can be created as children of the currently selected entry by means of the CMsvEntry::CreateL() method or MTM-specific methods. Deletion of message entries has to be carried out by means of the parent using the CMsvEntry::DeleteL() method.

9.3 Message Server Events and Sessions

The Message Server makes extensive use of asynchronous methods and is also designed to receive asynchronous external events, such as receiving some types of message. The most common types of messaging application require to be informed of these events. For example, the Messaging Application requires its display to be kept up-to-date when messages come in or are sent.

It would be possible for messaging clients to keep up to date by polling the Message Server on a regular basis, but this would be inefficient and would not guarantee a prompt response. Instead, the Message Server uses an observer pattern, whereby a client implements a class that implements an MMsvSessionObserver mixin and registers with the Message Server. Then, whenever an event occurs, the Message Server informs all registered observers. This is a common pattern within Symbian OS, and other servers use observers.

Like any other Symbian OS server, the Message Server is accessed by clients by means of a session, using the `CMsvSession` class. An object of this class is then passed to any other messaging method that requires access to the Message Server.

In fact, because the `MMsvSessionObserver` events include error events that all messaging clients need to know about, it is necessary to create an observer in order to get a `CMsvSession` object.

9.4 SMS Specific Variations

SMS messages have some differences in comparison with some other message types.

- Destination and From addresses are telephone numbers, which may be associated with contact names. These are stored in `CSmsNumber` objects that are owned by a `CSmsHeader` object. Note that incoming messages do not have recipients set (presumably the phone was the recipient), but outgoing and sent messages do have recipients.
- The message subject is taken as being the first 32 characters of the message.
- SMS messages cannot have any attachments.
- SMS messages have a limited length (e.g. 160 7-bit characters, some of which are used by the message header), but longer messages can be constructed by means of concatenation. This means that multiple messages are sent and are reassembled when they are received. Not all phones are capable of handling concatenated SMS messages, but the Symbian OS SMS MTM and SMS stack handle this, so an application developer can largely ignore the length of an SMS message (except for billing purposes!).
- SMS messages are sent to an SMS Service Center for forwarding to their final destination. This is equivalent to an email server. However, the user must have already configured their SMS, so an application developer is unlikely to need to touch it. To actually send a message, it is copied to the SMS Service Index entry. This can theoretically be done synchronously, but in this case the call will block until the message is sent, which may take some time. For this reason it is normal to copy the message asynchronously.
- SMS messages are received by 'push' rather than 'pull'. This means that the SMS Service Center sends the SMS message directly to the Symbian OS phone, where a watcher (just an active object in the watcher thread that monitors some incoming communications channel and acts on received data) intercepts it and places it in the Message Store. This contrasts with the 'pull' method whereby the Server MTM requests

the message (for example, POP3 email works in this way and the Symbian OS smartphone requests messages from the email server).

- The SMS Client-side MTM class `CSmsClientMtm` uses the `CBaseMtm::SetCurrentEntryL()` and `CBaseMtm::SwitchCurrentEntryL()` methods to set the current context and then provides `LoadMessageL()` and `SaveMessageL()` methods to load or save a message. It also provides `SmsHeader()` methods to enable direct access to the SMS message header.

9.5 Common Messaging Classes

Not all members and methods are described in this section: only those that are relevant to the work in this book are covered. If you want complete information then refer to the SDK or the relevant header files.

One aspect that I do not cover in this chapter is asynchronous operations. The Messaging subsystem has some sophisticated APIs to support asynchronous operations, but I have opted to use the simpler synchronous methods. Actually, I have to use one asynchronous operation to send an SMS message, because there are no synchronous methods available.

More detailed information on the Messaging APIs, including the extra methods for the classes covered here and the classes that are not covered here, can be found in SDKs.

9.5.1 Generic Messaging Classes

Class `TMsvId` Defined in `msvstd.h`
This type is a 32-bit integer that identifies a Message Server entry uniquely. See also `msvids.h` for standard 'well-known' identifiers such as `KMsvGlobalInBoxIndexEntryId`.

Class `TMsvEntry` Defined in `msvstd.h`
This type holds index entry values. It is a local cache – changes to it are not written into the index until the `CMsvEntry::ChangeL()` method is called.
Member Variables
`TUid iType` Set to `KUidMsvMessageEntry` for message entries.
`TUid iMtm` Set to `KUidMsgTypeSMS`.

`TTime iDate` The date and time when the message was originally created.
`TInt32 iSize` The size in bytes of the message.
`TPtrC iDescription` This member usually stores the message subject. For SMS messages this is the first 32 characters of the message.
`TPtrC iDetails` This member usually stores the recipient or sender address. For SMS messages this is the full telephone address of the first recipient or the sender.
`TMsvId iServiceId` This is the index identifier for the service associated with the entry. For SMS messages this will be set to the local SMS service.
Constructors
`TMsvEntry()` Initializes the new object to null values: `0` for integer values, `KMsvNullIndexEntryId` for IDs, and `KUidMsvNullEntry` for UIDs.
`TMsvEntry(const TMsvEntry& aEntry)` Creates a simple copy of the entry, so the `TPtrC` members `iDescription` and `iDetails` will point to the same descriptor data in the original and new objects. `aEntry` – the `TMsvEntry` to be copied.
Read-only Accessors
`TMsvId Id()` This read-only method returns the `TMsvId` of the entry.
`TMsvId Parent()` This read-only method returns the `TMsvId` of the parent entry.
Receiving and Sending State Members While an SMS is being received, its message entry in the Inbox is set to `InPreparation()==ETrue`, `Complete()==EFalse` and `Visible()==EFalse`. The message will be marked as complete, visible and not in preparation only when the entire SMS message (including all concatenated) has been received.
`inline TBool Complete() const` This method returns the complete flag for the entry.
`inline void SetComplete(TBool aComplete)` This method sets the complete flag for the entry. `aComplete` – the flag value to be set.

`inline TBool Visible() const` This method returns the visible flag for the entry.
`inline void SetVisible(TBool aVisible)` This method sets the visible flag for the entry. `aVisible` – the flag value to be set.
`inline TBool InPreparation() const` This method returns the in preparation flag for the entry.
`inline void SetInPreparation(TBool aInPreparation)` This method sets the in preparation flag for the entry. `aInPreparation` – the flag value to be set.
`inline TUint SendingState() const` This method returns the sending state for the entry.
`inline void SetSendingState(TUint aSendingState)` This method sets the sending state for the entry. `aSendingState` – the flag value to be set.
Unread Members
`inline TBool Unread() const` This method returns the value of the unread flag for the entry. This is set to `ETrue` when the message is received and should be set to `EFalse` when the message has been read.
`inline void SetUnread(TBool aUnread)` This method sets the unread flag for the entry. `aUnread` – the flag value to be set.

Class `CMsvEntry` – public `CBase`, public `MMsvSessionObserver`, public `MMsvStoreObserver` Defined in `msvapi.h`
Accesses and acts upon a particular Message Server entry. The current entry that a `CMsvEntry` object relates to is referred to as its context. It may be helpful to consider `CMsvEntry` functions in two broad groups. The first provides a means to access the various types of storage associated with an entry. The second provides a means to discover and access other entries that the entry owns (its children). A `CMsvEntry` object is relatively expensive in RAM usage, as it caches its children, updating them as necessary when notified of changes. Such objects should therefore be created sparingly.

Construction
`static CMsvEntry* NewL(CMsvSession& aMsvSession, TMsvId aMsvId, const TMsvSelectionOrdering& aOrdering)` This method obtains access to an existing entry. To create a new entry, use the `CreateL()` method. `aMsvSession` – the message server session to be used. `aMsvId` – the ID for the desired entry. `aOrdering` – the initial sort order for the children of the entry. It can be changed subsequently by using the `SetSortTypeL()` method.
Index Entry Access
`void SetEntryL(TMsvId aId)` Sets the context to the specified message entry ID. This message entry may be a message, service entry, attachment or folder. `aId` – the ID for the new context.
`void ChangeL(const TMsvEntry& aEntry)` Sets the context's index entry to the specified values. `aEntry` – the new context values.
Current Entry Member Methods
`TMsvId EntryId() const` This method returns the identifier of the context.
`const TMsvEntry& Entry() const` This method returns the index entry for the context.
Store Access
`TBool HasStoreL() const` This method returns `ETrue` if the entry has a store, `EFalse` if not.
`CMsvStore* ReadStoreL()` This method returns a reference to the store for the entry in read-only mode (can be shared). If the store is in exclusive use by another client, the function leaves with `KErrAccessDenied`. The store must be deleted when finished with.
`CMsvStore* EditStoreL()` This method returns a reference to the store in exclusive mode. If the store is in use by another client, the function leaves with `KErrAccessDenied`. If the entry does not have a store then it is created. The store must be deleted when finished with.

Child Entry Access Member Methods
`const TMsvSelectionOrdering& SortType() const` This method returns the current sort order type of the list of children.
`void SetSortTypeL(const TMsvSelectionOrdering& aOrdering)` Sets the sort order type of the list of children. `aOrdering` – the new sort order type.
`TInt Count() const` This method returns the number of child entries.
`const TMsvEntry& operator[](TInt aIndex) const` This method returns the index entry for a child entry. The array of child entries is a zero-based array.
`CMsvEntrySelection* ChildrenL() const` This method returns a selection including the index identifiers of all the children of the `CMsvEntry`. If it has no children then an empty list is returned. The caller is responsible for deleting the returned selection.
`CMsvEntrySelection* ChildrenWithMtmL(TUid aMtm) const` This method returns a selection including the index identifiers of all the children of the `CMsvEntry` that have a specific MTM type set. If it has no such children then an empty list is returned. The caller is responsible for deleting the returned selection. `aMtm` – the UID of the desired MTM.
`CMsvEntrySelection* ChildrenWithTypeL(TUid aEntryType) const` This method returns a selection including the index identifiers of all the children of the `CMsvEntry` that are of the specified type. If it has no such children then an empty list is returned. The caller is responsible for deleting the returned selection. `aEntryType` – the desired entry type.
`const TMsvEntry& ChildDataL(TMsvId aMsvId) const` This method returns the index entry of the child with the specified index identifier. `aMsvId` – the index identifier of the desired child entry. returns – the index entry of the specified child entry.
`CMsvEntry* ChildEntryL(TMsvId, aMsvId) const` This method returns a `CMsvEntry` object for the specified child entry. The caller must delete the `CMsvEntry` object when it is no longer required. `aMsvId` – the index identifier of the desired entry. returns – a new `CMsvEntry` object for the specified entry.

Child Entry Manipulation Member Methods
`void CreateL(TMsvEntry& aEntry)` This method creates a new child entry owned by the current context. `aEntry` – the index entry value for the new entry.
`void MoveL(TMsvId aMsvId, TMsvId aTargetId)` This synchronous method moves the specified child entry to be a child of the target entry. There is also an asynchronous version that takes a `TRequestStatus` argument and returns a `CMsvOperation`. `aMsvId` – the ID of the child entry to be moved. `aTargetId` – the new parent for the child entry.
`CMsvOperation* CopyL(TMsvId aMsvId, TMsvId aTargetId, TRequestStatus& aStatus)` This asynchronous method copies the specified child entry to be a child of the target entry. All stores and descendants are also copied. This method can be used to send an SMS message by copying the message to the SMS Service Index entry. `aMsvid` – the ID of the child entry to be moved. `aTargetId` – the parent for the new entry. `aStatus` – the request status to be completed when the copy is complete. returns – an `MsvOperation` for the copy. This can be used to get progress information or to cancel the operation.
`void DeleteL(TMsvId aId)` Deletes the specified child entry. `aId` – the ID of the child entry to be deleted.

Class `CMsvStore` Defined in `msvstore.h`
A `CMsvStore` object encapsulates a store used for message entry data. Often their access will be hidden by MTM functions but they can be used for access to SMS body text. The store for a `CMsvEntry` can be accessed by means of the `EditStoreL()` or `ReadStoreL()` method.
Member Methods
`void RestoreBodyTextL(CRichText& aRichTextBody) const` This method reads the body text from the store into the rich text object. `aRichTextBody` – the rich text object to receive the body text.
`void StoreBodyTextL(const CRichText& aRichTextBody)` This method stores the rich text in the store and commits the change so the store can be deleted. `aRichTextBody` – the rich text object that contains the body text to write.

Enumerated Type `TMsvSessionEvent`
This type indicates the type of event reported to an `MMsvSessionObserver`. The meanings of the other arguments to `HandleSessionEventL` depend on the type of event.

- `EMsvEntriesCreated` – One or more entries have been created.
  `aArg1` is a `CMsvEntrySelection` of the new entries.
  `aArg2` is the `TMsvId` of the parent entry.
- `EMsvEntriesChanged` – One or more index entries have been changed.
  `aArg1` is a `CMsvEntrySelection` of the index entries.
  `aArg2` is the `TMsvId` of the parent entry.
- `EMsvEntriesDeleted` – One or more entries have been deleted.
  `aArg1` is a `CMsvEntrySelection` containing the IDs of the deleted entries.
  `aArg2` is the `TMsvId` of the parent entry.
- `EMsvEntriesMoved` – One or more entries have been moved.
  `aArg1` is a `CMsvEntrySelection` containing the IDs of the moved entries.
  `aArg2` is the `TMsvId` of the new parent.
  `aArg3` is the `TMsvId` of the old parent entry.
- `EMsvMtmGroupInstalled` – A new MTM has been installed.
  `aArg2` points to a `TUid` for the new MTM.
- `EMsvMtmGroupDeInstalled` – A MTM has been uninstalled.
  `aArg2` points to a `TUid` of the removed MTM.
- `EMsvGeneralError` – This code is not currently in use.
- `EMsvCloseSession` – The client should immediately close the session with the Message Server.
- `EMsvServerReady` – Received after a client has used `CMsvSession::OpenAsyncL()` to create a session. The session can now be used.
- `EMsvServerFailedToStart` – Received after a client has used `CMsvSession::OpenAsyncL()` to create a session. The server could not be started.
  `aArg1` points to the error code.
- `EMsvCorruptedIndexRebuilt` – The Message Server index had been corrupted and had to be rebuilt. All local entries are recovered, but all remote entries have been lost.
- `EMsvServerTerminated` – The Message Server has been terminated. All clients must close their sessions immediately.
- `EMsvMediaChanged` – The Message Server has automatically changed the index location to use the internal disk.
  `aArg1` is a `TDriveNumber` value that identifies the drive used by the Message Server to hold the index prior to the change.
  `aArg2` is also a `TDriveNumber` value; it identifies the new drive that the Message Server is using. `CMsvEntry` contexts either refresh themselves or mark themselves invalid.
- `EMsvMediaUnavailable` – The media (disk) containing the Message Server index has been removed. Future requests may fail with `KMsvMediaUnavailable`. An `EMsvMediaChanged` event may be received in the future, as the Message Server switches back to the internal drive.
  `aArg1` is a `TDriveNumber` value that identifies the drive that is no longer available.
- `EMsvMediaAvailable` – The disk containing the Message Store is available again. The Message Server can now operate as normal. No client action is necessary.
  `aArg1` is a `TDriveNumber` value that identifies the drive that is being used.

- `EMsvMediaIncorrect` – An incorrect disk is inserted. Some requests may fail with `KMsvMediaIncorrect`. Clients may get an `MsvMediaChanged` event in the future telling them that the Message Server has switched back to the internal drive.

`aArg1` is a `TDriveNumber` value that identifies the drive in which the incorrect disk has been inserted.

- `EMsvCorruptedIndexRebuilding` – The Message Server has started to rebuild its index after it has been corrupted.

Class `MMsvSessionObserver`
Defined in `msvapi.h`

Provides the interface for notification of events from a Message Server session. The types of event are given in the enumeration `TMsvSessionEvent`. Clients must provide an object that implements the interface, and set it to be notified through `CMsvSession::OpenSyncL()` or `CMsvSession::OpenASyncL()`.

Member Methods

```
virtual void HandleSessionEventL(TMsvSessionEvent aEvent,
                                 TAny* aArg1, TAny* aArg2,
                                 TAny* aArg3) =0;
```

This method is called by the Message Server when an event occurs.

`aEvent` – the type of event that has occurred.

`aArg1`, `aArg2` and `aArg3` – the meanings of the `TAny*` arguments depend on the `aEvent` value.

Class `CMsvSession`
Defined in `msvapi.h`

Represents a channel of communication between a client thread (Client-side MTM, User Interface MTM, or message client application) and the Message Server thread.
A message client application must use `OpenSyncL()` or `OpenASyncL()` to create a session object, before it can instantiate any MTM or `CMsvEntry` object. Only a single session should be created within a thread.
To close a session, delete all objects relying on that session, and then the session object itself.

Static Methods

```
static CMsvSession* OpenSyncL(MMsvSessionObserver& aObserver)
```

This method creates a new session that can subsequently be used for other Messaging APIs.

`aObserver` – a reference to an object that implements the `MMsvSessionObserver` mixin (interface). The object will be notified of message events.

Utility Methods

`inline CMsvEntry* GetEntryL(TMsvId aEntId)`
This method returns a new `CMsvEntry` corresponding to the supplied index identifier. The client is responsible for deleting the `CMsvEntry` after use.
`aEntId` – the index identifier of the desired entry.
returns – a new `CMsvEntry` object.

`inline TInt GetEntry(TMsvId aId, TMsvId& aService, TMsvEntry& aEntry)`
This method returns the `TMsvEntry` for an index identifier along with the index identifier of the owning service.
`aId` – the index identifier of the desired entry.
`aService` – on return this is the index identifier of the owning service.
`aEntry` – on return this is the index entry.
returns – `KErrNone` or a system-wide error code.

Enumerated Type `TMsvSorting`
This type defines the sort order options for a `TMsvSelectionOrdering` object. Options are set through `TMsvSelectionOrdering::SetSorting()`.
Defined in `msvstd.h`

- `EMsvSortByNone` – don't sort
- `EMsvSortByDate` – sort by date (earliest to latest)
- `EMsvSortByDateReverse` – sort by date (latest to earliest)
- `EMsvSortBySize` – sort by size (smallest to largest)
- `EMsvSortBySizeReverse` – sort by size (largest to smallest)
- `EMsvSortByDetails` – sort by message details ('From' address) (A to Z collated)
- `EMsvSortByDetailsReverse` – sort by message details ('From' address) (Z to A collated)
- `EMsvSortByDescription` – sort by description (A to Z collated)
- `EMsvSortByDescriptionReverse` – sort by description (Z to A collated)
- `EMsvSortById` – sort by message ID (lowest to highest)
- `EMsvSortByIdReverse` – sort by message ID (highest to lowest)

Enumerated Type `TMsvGrouping`
This type defines the grouping options for a `TMsvEntrySelection` object. These values are bitmasks that can be combined and set through the `TMsvEntrySelection` constructor.
Defined in `msvstd.h`

- `KMsvNoGrouping` = 0 – no grouping
- `KMsvGroupByType` = 0×2 – group by type (Folder, Message or Attachment)
- `KMsvGroupByStandardFolders` = 0×6 – group standard folders first (must have sorting by type set as well)
- `KMsvGroupByPriority` = 0×8 – group by priority (High, Medium or Low)
- `KMsvGroupByMtm` = 0×10 – group by MTM ID value in increasing UID value

Class `TMsvSelectionOrdering`
Defined in `msvstd.h`

This type encapsulates the sorting and grouping type for a `CMsvEntrySelection`. The grouping and sorting values can be set either on construction or subsequently. When a selection is accessed, grouping is applied and then sorting is applied within the groups.

Constructors

`TMsvSelectionOrdering()`
The default constructor sets no grouping, and sorting is set to `EMsvSortByNone`.

`TMsvSelectionOrdering(TInt aGroupingKey, TMsvSorting aSorting, TBool aShowInvisible)`
This constructor allows the sorting and grouping values to be initialized.
`aGroupingKey` – the grouping value to be set.
`aSorting` – the sorting value to be set within groups.
`aShowInvisible` – whether or not to show invisible entries (defaults to `EFalse`).

Member Methods

`TMsvSorting Sorting() const`
This method returns the current sorting value within groups.

`void SetSorting(TMsvSorting aSortType)`
This method sets the current sorting value within groups.

`TBool GroupingOn() const`
This method returns `ETrue` if any grouping option is set.

`TBool GroupByType() const`
This method returns `ETrue` if group-by-type is set.

`void SetGroupByType(TBool aFlag)`
This method sets group-by-type.
`aFlag` – `ETrue` if group-by-type is to be set.

`TBool GroupStandardFolders()`
This method returns `ETrue` if group-by-standard-folders is to be set.

`void SetGroupStandardFolders(TBool aFlag)`
This method sets group-by-standard-folders.
`aFlag` – `ETrue` if group-by-standard-folders is to be set.

`TBool GroupByPriority() const`
This method returns `ETrue` if group-by-priority is set.

`void SetGroupByPriority(TBool aFlag)` This method sets group-by-priority. `aFlag` – `ETrue` if group-by-priority is to be set.
`TBool GroupByMtm(TBool aFlag)` This method returns `ETrue` if group-by-MTM is set.
`void SetGroupByMtm(TBool aFlag)` This method sets group-by-MTM. `aFlag` – `ETrue` if group-by-MTM is to be set.
`inline TBool ShowInvisibleEntries() const` This method returns `ETrue` if invisible entries are to be included in a selection.
`void SetShowInvisibleEntries(TBool aFlag)` This method sets whether invisible entries are to be included in a selection. `aFlag` – `ETrue` if invisible entries are to be included in a selection.

Class `CMsvEntrySelection` Defined in `msvstd.h`
This class is an array of `TMsvId` which is used to pass a selection of entries. An object of this type can be generated or returned with an event. The array base class `CArrayFixFlat<TMsvId>` provides methods to access the members of the array.
Member Methods
`TInt Find(TMsvId aId) const` This method finds an entry by entry ID. If the entry is present in the array then the index of the entry is returned. If the entry is not present then the method returns `KErrNotFound`. `aId` – the index identifier of the desired entry.

9.5.2 Generic MTM Classes

Class `CClientMtmRegistry` Defined in `mtclreg.h`
This class provides access to the registry that holds details of all client-side MTMs currently available on the phone. This class is used to obtain access to a specific client-side MTM.

Constructor
`static CClientMtmRegistry* NewL(CMsvSession& aMsvSession, TTimeIntervalMicroSeconds32 aTimeoutMicroSeconds32)` This method returns a new `CClientMtmRegistry` object. `aMsvSession` – reference to an `MsvSession` for this thread. `aTimeoutMicroSeconds` – time to wait before unloading MTMs in microseconds (defaults to 30 seconds).
Member Methods
`CBaseMtm* NewMtmL(TUid aMtmTypeUid)` This method returns a pointer to an object derived from a `CBaseMtm`. `aMtmTypeUid` – the UID of the desired MTM.

Class `CBaseMtm` Defined in `mtclbase.h`
This class provides methods for accessing and manipulating a Message Server entry; it also provides access to server MTM functionality through the `InvokeAsync` function. There are other classes specific to SMS messages.
Context Methods Context functions: the `SetCurrentEntryL()` and `SwitchCurrentEntryL()` functions change the context – the entry on which later actions are performed. After creating a new Client-side MTM object, a message client application should set an initial context before using other functions. Note that any changes made to an existing context are not automatically saved. The message client application should ensure this itself by calling `SaveMessageL()`. No message data for the new context is retrieved from the Message Server. To retrieve entry data, call `LoadMessageL()` (a pure virtual function in this class – for SMS messages it is implemented in the `CSmsClientMtm` class) after setting the context. Calling `Body()` immediately after setting the context returns an empty `CRichText` object, because the private cache of context body text that the base class maintains is reinitialized to an empty value.
`void SetCurrentEntryL(CMsvEntry* aEntry)` This method sets the context to the specified entry. `aEntry` – the context on which subsequent actions will be performed.
`void SwitchCurrentEntryL(TMsvId aId)` This method sets the context to the entry with the specified ID. `aId` – the ID of the context on which subsequent actions will be performed.
`CMsvEntry& Entry() const` This method returns a `CMsvEntry` for the current context.

`TBool HasContext() const`
This method returns `ETrue` if the object has a current context.

Body Text Methods
The base class maintains a private `CRichText` object cache to store the body text for the current context. This can be accessed for reading and writing by message client applications through `Body()`. Loading and saving the body text are achieved by methods in the derived class.

`CRichText& Body()`
`const CRichText& Body() const`
This method returns the body text of the context (which must be set to a message entry).

Address Methods
There are methods in the specific MTM classes to handle addresses, but there are also generic address handling methods that can be used.

`virtual void AddAddresseeL(const TDesC& aRealAddress)`
This method adds a new message recipient using the supplied address. The address is not validated by this routine.
`aRealAddress` – the address (syntax depends on message type).

`virtual void AddAddresseeL(const TDesC& aRealAddress, const TDesC& aAlias)`
This method adds a new message recipient with an address and an alias. The address is not validated by this routine.
`aRealAddress` – the address (syntax depends on message type).
`aAlias` – an alias for the address (meaning depends on message type).

`virtual void RemoveAddressee(TInt aIndex)`
This method removes an address from the current address list. The address is specified by a zero-based index into the address list. If the index is not known, applications can use `AddresseeList()` to retrieve the entire list to find the item.
`aIndex` – the index of the address to be deleted.

`inline const CDesCArray& AddresseeList() const`
This method returns a list of addresses stored with the message.

9.6 SMS Specific Classes

As well as the generic messaging classes, there are a range of classes that are specific to SMS messages. It should be noted that there are a range of types of SMS messages, but only the APIs relating to 'conventional' SMS messages are covered in this section.

Class `CSmsClientMtm`
Defined in `smsclnt.h`

This class is the Client-side SMS MTM and provides a range of methods specific to SMS messages. A copy of an instantiation of this class can be obtained from the MTM registry by using the MTM UID `KUidMsgTypeSMS`.
There are methods to create reply or forward messages, but they are not described in this section. There are methods to add or remove recipients, but this is just as easily done by means of the header.

Store and Restore Methods
The changes that a message client application makes to a message context through Client-side MTM functions, such as altering the body text obtained through `Body()`, are, for efficiency, cached in memory by the Client-side MTM. The message store and restore functions are concerned with transferring data between that cache and committed storage.

`void SaveMessageL()`
This method commits cached changes to the storage controlled by the Message Server. It can be called only on message contexts. It should be called to preserve changes when the context is changed, or when the Client-side MTM object is deleted or before sending a message.
The function panics for non-message contexts.

`void LoadMessageL()`
This method loads the cache with the message data for the current context. It can be called only on message contexts. It is typically used after the context has been set with `SetCurrentEntryL()` or `SwitchCurrentEntryL()`.
The function should panic for non-message contexts.

Member Methods

`inline TInt ServiceId() const`
This method returns the ID of the current SMS service.

`CSmsHeader& SmsHeader()`
`const CSmsHeader& SmsHeader() const`
This method returns the header of the SMS message.

`inline CSmsSettings& ServiceSettings()`
`inline const CSmsSettings& ServiceSettings() const`
These methods provide access to the SMS MTM's service settings.

`void CreateMessageL(TMsvId aServiceId)`
This method creates a new outgoing SMS message as a child of the current context. The context is set to the newly created message.
`aServiceId` – the ID of the service to own the message (see the `ServiceId` method).

Class `CSmsSettings` – public `CSmsMessageSettings`
Defined in `smutset.h`

An SMS service entry stores an object of this type in its message store. These settings define the default settings for standard SMS messages. They also provide some global settings. This object provides the default Service Center address which is required for new messages.

Member Methods

`inline TInt NumSCAddresses() const`
This method provides the number of Service Center addresses stored.

`inline CSmsNumber& SCAddress(TInt aIndex) const`
This method provides access to all stored SMS Service Center addresses.
 `aIndex` – which Service Center address to return.
 returns – `CSmsNumber` of the selected Service Center.

`TInt DefaultSC() const`
This method returns the index of the default Service Center address. It can be used in a call to `SCAddress()` to return the address of the default Service Center.

Class `CSmsHeader`
Defined in `smuthdr.h`

This class encapsulates the header for an SMS message. It includes data relating to all the different types of SMS message. It is useful to access the From address and the recipients (for outgoing messages). It also provides methods to access the lower-level information relating to an SMS message, but there are more convenient methods to access the body text, such as using the methods in `CBaseMtm`.

Construction Methods

`static CSmsHeader* NewL(CSmsPDU::TSmsPDUType aType, CEditableText& aText)`
This method creates a new `CSmsHeader` object.
 `aType` – the type of the SMS (`ESmsSubmit` for a conventional SMS message).
 `aText` – the message text for the message.

Member Methods

`const CArrayPtrFlat<CSmsNumber>& Recipients() const`
`CArrayPtrFlat<CSmsNumber>& Recipients()`
This method returns the list of message recipients.

`void SetFromAddressL(const TDesC& aAddress)`
This method sets the From address (phone number) for the SMS message.
 `aAddress` – the new From address.

`TPtrC FromAddress() const` This method provides non-modifiable access to the From address (phone number) of the SMS message.
`void RestoreL(CMsvStore& aStore)` This method restores the object from the Message store, but it may be ignored if the `CSmsClientMtm::RestoreMessageL()` method is used. `aStore` – the store to be read from.
`void StoreL(CMsvStore& aStore) const` This method writes the object to the Message store. `aStore` – the store to be written to.
`void SetSmsSettingsL(const CSmsMessageSettings& aSettings)` This method sets the SMS Message Settings for the message. `aSettings` – the SMS Message Settings to use.
`inline void SetServiceCenterAddressL(const TDesC& aAddress)` This method sets the SMS Service Center address for the message. `aAddress` – the address of the Service Center.

Class `CSmsNumber` Defined in `smutset.h`
This class stores the name and number for an SMS recipient.
Construction Methods
`static CSmsNumber* NewL()` This method creates a new, empty `CSmsNumber` object.
Member Methods
`TPtrC Address() const` This method returns a non-modifiable reference to the address (phone number) currently stored.
`void SetAddressL(const TDesC& aAddress)` This method sets the address (phone number). `aAddress` – the new address to use.
`TPtrC Name() const` This method returns a non-modifiable reference to the name currently stored.
`void SetNameL(const TDesC& aName)` This method sets the name. `aName` – the new name to use.

10

Developing an SMS Management Connectivity Service

In this chapter we will work through the code for a custom server and a socket server to manage SMS messages. This requires code to interact with the messaging system and code to pack and unpack the data exchanged between the Symbian OS smartphone and the PC. We will also work briefly through a command-line C# application that can be used to drive the servers.

We will aim to make our implementation as common as possible on different versions of Symbian OS. The custom server or socket server may need to be rebuilt for specific versions of the OS, but at least we can keep the maintenance burden as low as possible. At the same time, we will make sure that by using the same protocol we can keep the PC software as common as possible. Our aim is to hide any differences between devices from the user and again to keep maintenance as simple as possible.

The command-line C# application is good for debugging purposes but is not intended for real use, so we will work through a GUI application in Chapter 13. Once we have the command-line application working, the GUI aspects can easily be handled in isolation.

While I was writing this chapter, it became clear that using the term 'message' for the SMS messages that we are considering as well as for the messages exchanged between the Symbian OS smartphone and the PC caused some confusion. Therefore, I have used the term PDU (Protocol Data Unit) for the messages between the smartphone and the PC and reserved the term message for the SMS messages.

10.1 SMS Management Protocol

In this section I describe the protocol that we are going to use between the Symbian OS smartphone and the PC. It is important to document the protocol for maintenance reasons – if you do not document it then nobody knows whether any deviations are correct or incorrect. Having

said that, the protocol may undergo some prototyping and evolution before it settles down.

You can use any protocol that you wish, but I will set out some factors that I think make a better protocol, which you may ignore.

We will build our protocol mostly around a command–response sequence (the exception is event handling that will be covered at the end). The PC will send a command and the Symbian OS smartphone will carry out some action and send a response. I prefer a protocol to have as little state built in as possible. This makes it easier to implement the code to handle commands or responses, and it also eases debugging based on PDU logs. For the same reason we will include a PDU identifier in commands that will be echoed in the responses.

This means that for the simplest commands we have the following fields:

- the PDU length
- the command opcode
- the transaction identifier.

For more complex commands we will add some more fields. Responses have the same structure, with a response code rather than a command opcode. The transaction identifier is the echo of the PDU identifier of the command that prompted the response.

The fundamental starting point for our protocol is to consider the operations that we want to perform:

- display all SMS messages from the Symbian OS mobile phone
- send an SMS message via the Symbian OS mobile phone
- delete SMS messages on the Symbian OS mobile phone so as to manage the Message Store
- be informed of events so we can display new incoming and other changed messages.

We have not included any reply or forward functions (these can be implemented on the PC if required) and we have not made any provision for creating a draft SMS message and then amending it later.

Given this set of requirements, we can list the command opcodes and the response codes. The following is an extract from the Connectivity plug-in code on the Symbian OS smartphone:

```
enum TRSmsCmdCode
  {
  ERSmsCmdNone = 0, // Reserved for internal usage
```

```
ERSmsCmdQueryVersion = 1, // Query version
ERSmsCmdVersionReply = 2, // Version reply

ERSmsCmdGetAllSms = 10,  // Get messages in Inbox or folder
ERSmsCmdGetMoreSms = 11, // Get remaining messages from Inbox or folder
ERSmsCmdGetSmsById = 12, // Get one message by identifier
ERSmsCmdReceiveSms = 13, // Receive one or more SMS
ERSmsCmdReceiveNoMoreSms = 14, // Got no more messages

ERSmsCmdReturnEvents = 20, // Return message events
ERSmsCmdMsvEvent = 21, // Message created etc. event

ERSmsCmdSendSms = 30, // Send a message
ERSmsCmdSentSms = 31, // Message has been sent

ERSmsCmdDeleteSms = 40, // Delete a message
ERSmsCmdDeletedSms = 41, // Message has been deleted

ERSmsCmdError = 50 // An error has occurred
};
```

As you can see, we have some ‘boiler-plate’ codes – the no-op and the version query code. We have also included an error response `ERSmsCmdError`, as it is possible that any command may prompt an error.

The SMS-related commands are as follows:

- `ERSmsCmdGetAllSms` – get all SMS messages in a folder
- `ERSmsCmdGetMoreSms` – get any unfetched SMS messages
- `ERSmsCmdGetSmsById` – get one SMS by index identifier
- `ERSmsCmdSendSms` – create and send an SMS
- `ERSmsCmdDeleteSms` – delete an SMS

We have provided a command to get all the SMS messages from any folder, but we have also allowed for the fact that we may not be able to get all the SMS messages that we want in one response. When programming for an embedded device such as a smartphone, we can never assume that we can have an arbitrarily large buffer for a PDU. Any protocol that ignores the question of maximum PDU size is a good way to crash the smartphone. We assume later that the data for any single SMS can be fitted in a PDU. This is probably safe, given the limited size of an SMS, although if we concatenated sufficient messages together we could break any limit. If we were dealing with email messages with attachments then we would not be able to make this assumption and we would have to handle breaking messages across PDUs.

The delete and send SMS commands are clearly useful, and the command to fetch one SMS by itself will be useful when we handle events. When a new SMS arrives, we do not want to have to fetch all the other messages at the same time.

The response codes are also fairly obvious. I have chosen to define a response code for no more messages, but we could have sent just an empty set of messages.

The reason for the event command and response will become clear later, but, for now, note that the `ERSmsCmdReturnEvents` command puts a stream or session into an event handling mode.

Note that the actual values used for the command opcodes and response codes are arbitrary. We could have spread the values out or reserved one bit for responses, but these values are good enough for now.

As well as defining opcodes, etc., we need to define the maximum packet size. Normally, we would set this as big as we can afford, but for this example we will set it smaller so that we can test fetching more SMS messages than will fit into one packet.

Some of the commands (`ERSmsCmdNone`, `ERSmsCmdQueryVersion`, `ERSmsCmdGetMoreSms` and `ERSmsCmdReturnEvents`) need no further arguments, but the others need one or more additional arguments. The `ERSmsCmdFetch`, `ERSmsCmdGetSmsById` and `ERSmsDeleteSms` commands each need a single identifier.

The `ERSmsCmdSendSms` command is the one command that requires a lot of data. We will pack all the data required to create and send the SMS into one command. We could implement some form of storage of draft messages and then edit them, but we might as well implement that on the PC side if we need it.

The response PDUs follow a similar pattern. The `ERSmsCmdError` PDU has a four-byte error number (a standard Symbian OS error code). The `ERSmsCmdSentSms` and `ERSmsCmdDeletedSms` PDUs include a single four-byte message index identifier – the newly sent message or the newly deleted message respectively. The `ERSmsCmdReceiveNoMoreSms` PDU needs no extra data – it just signifies that all messages from the last specified get request have been returned. The `ERSmsCmdVersionReply` PDU includes standard version and build data. The `ERSmsCmdMsvEvent` PDU includes a series of sets of event data. These include the event type and the relevant message index identifiers. The `ERSmsCmdReceiveInitialSms` and `ERSmsCmdReceiveMoreSms` messages are the complex PDUs, as they include one or more SMS messages with all details.

Query Version Command

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdQueryVersion (=1)`
Transaction ID	Int32	PDU transaction identifier

Version Reply

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdVersionReply (=2)`
Transaction ID	Int32	PDU transaction identifier
Major Version	Int32	Major version number
Minor Version	Int32	Minor version number
Build Number	Int32	Build number

Fetch SMS from Folder

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdGetAllSms (=10)`
Transaction ID	Int32	PDU transaction identifier
Folder ID	Int32	Index identifier of the folder required

Fetch More SMS

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdGetMoreSms (=11)`
Transaction ID	Int32	PDU transaction identifier

Fetch SMS by Identifier

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdGetSmsById (=12)`
Transaction ID	Int32	PDU transaction identifier
Message ID	Int32	Index identifier of the message required

Receive SMS

This PDU contains data from one or more messages. The message data is repeated for each new message until a message identifier of zero indicates the end of the data.

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdReceiveSms (=13)`
Transaction ID	Int32	PDU transaction identifier

. . .

Message ID	Int32	If the message index identifier is zero then this is the last message in the buffer and there is no more message data
Parent ID	Int32	Index identifier of the parent folder
Date and Time Stamp	7-byte time value	Date and time stamp of the message
Description Length	Int16	Length in ASCII characters of message description
Message Description	Unicode data	Message description – up to the first 32 characters of the body text
Detail Length	Int16	Length in characters of message detail
Message Detail	Unicode data	Message detail – first recipient address
From Address Length	Int16	Length in characters of From address
From Address	Unicode data	Address of the message sender
Number of Recipients	Int16	Number of recipients for the message. The next four fields are included once for each recipient

. . .

Name Length	Int16	Length in characters of recipient name
Recipient Name	Unicode data	Recipient name
Address Length	Int16	Length in characters of recipient address
Recipient Address	Unicode data	Recipient address (i.e. mobile phone number)

. . .

Body Length	Int16	Length in characters of body text
Body Text	ASCII data	Body text of the message

Receive No More SMS

This reply is sent when the client requests SMS messages and none are to be returned.

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdReceiveNoMoreSms (=14)`
Transaction ID	Int32	PDU transaction identifier

Send SMS

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdSendSms (=30)`
Transaction ID	Int32	PDU transaction identifier
Body Length	Int16	Length in characters of body text
Body Text	ASCII data	Body text (length set in previous field)
Address Count	Int16	Number of sending addresses (the next two fields are repeated this number of times)

. . .

Address Length	Int16	Length of address text
Address Text	Unicode data	Address text (length set in previous field)

Sent SMS Reply

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdSentSms (=31)`
Transaction ID	Int32	PDU transaction identifier
Message ID	Int32	Message index identifier for the newly created and sent SMS

Delete SMS by Identifier

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdDeleteSms (=40)`
Transaction ID	Int32	PDU transaction identifier
Message ID	Int32	Index identifier of the message to be deleted

Deleted SMS Reply

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdDeletedSms (=41)`
Transaction ID	Int32	PDU transaction identifier
Message ID	Int32	Message index identifier for the deleted SMS

Enable Event Reporting

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdReturnEvents (=20)`
Transaction ID	Int32	PDU transaction identifier

Message Event Reply

This message contains details of one or more events. When an event type of −1 is returned, it indicates the last event in the message. Events of type `EMsvEntriesMoved` have an additional two index identifiers. Each event has a list of one or more message identifiers terminated with a zero index identifier. Therefore, a Message Event Reply contains nested lists of message identifiers within a list of events.

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdMsvEvent (=21)`
Transaction ID	Int32	PDU transaction identifier

. . .

Event Type	Int32	`EMsvEntriesCreated (=0)`, `EMsvEntriesChanged (=1)`, `EMsvEntriesDeleted (=2)`
Message ID	Int32	Message index identifier (this field may be repeated)
Null Message ID	Int32	Terminating message index identifier (=0)

. . .

Event Type	Int32	`EMsvEntriesMoved (=3)`
Message ID	Int32	Message index identifier (this field may be repeated)
Null Message ID	Int32	Terminating message index identifier (=0)
Old Parent ID	Int32	Index identifier for the message original parent
New Parent ID	Int32	Index identifier for the message new parent

Error Reply

Field	Type	Meaning
Opcode	Int32	`ERSmsCmdError (=50)`
Transaction ID	Int32	PDU transaction identifier
Error Code	Int32	Symbian OS error code

10.2 Packing and Unpacking Data

One of the standard types of functionality that we will need is to be able to unpack data from a buffer and pack fresh data into it as a response. This is common to all connectivity services. In fact, there are classes to manage this packing and unpacking (not surprisingly, given the prevalence of the need). The class developed for custom servers is `TRemsvr` and the `Serialise` module is provided with Socket Servers. However, we want to create a class that can be used with as wide a range as possible of Symbian OS smartphones. Therefore, we create our own class that we can use with any version of Symbian OS.

This code is not particularly clever, but it does not need to be – it just needs to be easy to use and reliable. We use descriptors and we check for running out of data at every point. This catches buffer overruns and some forms of data corruption. This makes it easier to use these methods. The alternative would be to check the length of remaining data whenever we are about to unpack some more. This would make for clumsier and less readable code, so delegating the checks to the unpacking class is the better alternative.

To begin with, we want methods to unpack 32-bit integers. For most purposes, we will just read the integer and delete it from the buffer, but sometimes we will want to read the integer nondestructively. Originally, I implemented these methods by using values from the front of a descriptor and then deleting them. However, the deletion involves copying the rest of the buffer over the deleted entries. If we have a large buffer containing many values then we may be copying the data literally thousands of times, which is inefficient. To avoid this, we define a simple class to contain a reference to the descriptor and a current read position variable. This allows us to step through the buffer simply by incrementing the position.

```
class TConnBuff
{
public:
  TConnBuff(TDes8& aBuffer);

  TDes8& mBuffer;  // Buffer to read from
  TInt   mReadPos; // Position to read from next
};

TConnBuff::TConnBuff(TDes8& aBuffer): mBuffer(aBuffer)
  {
  }

// Read a 4-byte integer from the start of a buffer
TInt32 CConnPack::PeekInt32L(TConnBuff& aBuffer)
  {
  if((aBuffer.mReadPos + 4) > aBuffer.mBuffer.Length())
    {
    User::Leave(KErrArgument);
```

```
    }
  TInt32 ret = (((TInt32)aBuffer.mBuffer[aBuffer.mReadPos+3])<<24)|
               (((TInt32)aBuffer.mBuffer[aBuffer.mReadPos+2])<<16)|
               (((TInt32)aBuffer.mBuffer[aBuffer.mReadPos+1])<<8)|
                ((TInt32)aBuffer.mBuffer[aBuffer.mReadPos]);
  return ret;
  }

// Read a 4-byte integer from the start of a buffer and remove it
TInt32 CConnPack::ReadInt32L(TConnBuff& aBuffer)
  {
  TInt32 ret = PeekInt32L(aBuffer);
  aBuffer.mReadPos += 4;
  return ret;
  }
```

The counter of these methods is to append a 32-bit integer to a buffer. As appending does not require copying, we can simply use the raw descriptor. This is slightly inconsistent, as we are using one type for reading and another for writing, but I prefer that to using an unnecessary level of indirection.

```
// Append a 4-byte buffer to the end of a buffer
void CConnPack::WriteInt32L(TInt32 aValue, TDes8 &aBuffer)
  {
  if(aBuffer.Length() + 4 > aBuffer.MaxLength())
    {
    User::Leave(KErrArgument);
    }
  aBuffer.Append((TUint8)(aValue&0x000000FF));
  aBuffer.Append((TUint8)((aValue&0x0000FF00)>>8));
  aBuffer.Append((TUint8)((aValue&0x00FF0000)>>16));
  aBuffer.Append((TUint8)((aValue&0xFF000000)>>24));
  }
```

We can set up similar routines for 16-bit and 8-bit integers as well. I will not include the code here, but it is available with the rest of the source code for this book.

The other data type that we want to read is a buffer of text. Because we are dealing with SMS data, we can be content with ASCII data coming in; however, as Symbian OS deals with Unicode text, we need to expand it as we unpack it.

```
// Read ASCII data. The length is the first 2 bytes and then each
// character comes in as a single 8-bit value. We expand it to a
// Unicode value for compatibility reasons.
TInt CConnPack::ReadASCIIDataL( TDes& aOutBuffer,
  TConnBuff& aSourceBuffer)
  {
  TUint16 len = ReadUint16L(aSourceBuffer);
  if(((aSourceBuffer.mReadPos + len) > aSourceBuffer.mBuffer.Length())
      || (len > aOutBuffer.MaxLength()))
```

```
    {
    User::Leave(KErrArgument);
    }

  aOutBuffer.Zero();
  for(TInt i = 0 ; i < len ; i++)
    {
    aOutBuffer.Append((TChar)aSourceBuffer.mBuffer[aSourceBuffer.
      mReadPos+i]);
    }
  aSourceBuffer.mReadPos += len;
  return len;
  }
```

The method to pack ASCII data works the same way, but in reverse:

```
// Take a buffer of Unicode data and write the length as a 2-byte value,
// then append the data but shrinking each character to a 1-byte value.
void CConnPack::WriteASCIIDataL( TDesC &aSourceBuffer, TDes8 &aOutBuffer)
  {
  TUint16 len = (TInt16)aSourceBuffer.Length();
  if((aOutBuffer.Length()+len+4 > aOutBuffer.MaxLength()))
    {
    User::Leave(KErrArgument);
    }
  WriteUint16L(len, aOutBuffer);
  for(TInt i = 0 ; i < len ; i++)
    {
    aOutBuffer.Append((TChar)aSourceBuffer[i]);
    }
  }
```

We will have Unicode versions of these methods when we want data that really can be Unicode, which is more normal, and these will be used in later chapters.

As SMS messages have date and time stamps, we need to be able to read and write these. In Symbian OS, as in most other operating systems, a date and time are stored as an offset in some fractions of a second since a logical start time. The basic type in Symbian OS is a `TTime`. The offset is stored as an eight-byte integer and we could access it and read and write it as two 32-bit integers. However, Windows does not use quite the same system, although conversions are possible. For the sake of my own understanding, and to ease debugging, I have chosen to break up a `TTime` into its component parts (ignoring any values smaller than a second) and send those across. On the PC side the same process is possible in reverse. One complication that is not relevant in this chapter, but will affect us in later chapters, is that Symbian OS allows a `TTime` to be a null value and that is treated separately (it means that a time is not set). We want to preserve this ability and so we treat a null `TTime` as one with all values set to zero, which would otherwise be illegal.

```
TTime CConnPack::ReadTTimeL(TConnBuff& aBuffer)
  {
  TInt tempYear = ReadInt16L(aBuffer);
  TInt tempMonth = ReadInt8L(aBuffer);
  TInt tempDay = ReadInt8L(aBuffer);
  TInt tempHour = ReadInt8L(aBuffer);
  TInt tempMinute = ReadInt8L(aBuffer);
  TInt tempSecond = ReadInt8L(aBuffer);

  if((tempYear==0) && (tempMonth==0) && (tempDay==0) &&
     (tempHour==0) && (tempMinute==0) && (tempSecond==0))
    {
    return(Time::NullTTime());
    }
  else
    {
    TDateTime dt;
    dt.Set(tempYear, (TMonth)tempMonth, tempDay,
           tempHour, tempMinute, tempSecond, 0);
    TTime tt(dt);
    return tt;
    }
  }

void CConnPack::WriteTTimeL(const TTime &aTT, TDes8 &aBuffer)
  {
  if(aTT == Time::NullTTime())
    {
    WriteInt16L(0, aBuffer);//year
    WriteInt8L(0, aBuffer);//month
    WriteInt8L(0, aBuffer);//day
    WriteInt8L(0, aBuffer);//hour
    WriteInt8L(0, aBuffer);//minute
    WriteInt8L(0, aBuffer);//second
    }
  else
    {
    WriteInt16L((TInt16)aTT.DateTime().Year(), aBuffer);
    WriteInt8L((TInt8)aTT.DateTime().Month(), aBuffer);
    WriteInt8L((TInt8)aTT.DateTime().Day(), aBuffer);
    WriteInt8L((TInt8)aTT.DateTime().Hour(), aBuffer);
    WriteInt8L((TInt8)aTT.DateTime().Minute(), aBuffer);
    WriteInt8L((TInt8)aTT.DateTime().Second(), aBuffer);
    }
  }
```

The final method we will provide here is to simply append one buffer of undifferentiated data to another. We will use this when we have created a temporary buffer and want to add it to the output buffer.

```
// Append one buffer of 1-byte data to another
void CConnPack::WriteBufferL( TDesC8 &aSourceBuffer, TDes8 &aOutBuffer)
  {
  TInt len = aSourceBuffer.Length();
  if((len < 0) || (aOutBuffer.Length()+len+4 > aOutBuffer.MaxLength()))
    {
    User::Leave(KErrArgument);
```

```
    }
  WriteInt32L(len, aOutBuffer);
  aOutBuffer.Append(aSourceBuffer);
  }
```

10.3 Obtaining Access to the Message Server and the SMS MTM

In order to write any messaging code, we will need to get a session with the Message Server. Like most good servers, the Message Server provides a good client API. The only requirement to obtain a Message Server session is an implementation of the `MMsvSessionObserver` mixin that includes a `HandleSessionEventL()` method. As a minimum, we can address just those error events that mean we would lose access to the Message Server.

```
class CRSmsCSServer : public CCustomServer, public MMsvSessionObserver
{
...
  // From MMsvSessionObserver
  virtual void HandleSessionEventL(TMsvSessionEvent, TAny*, TAny*, TAny*);
```

```
void CRSmsCSServer::HandleSessionEventL(TMsvSessionEvent aEvent,
                                 TAny *arg1, TAny *arg2, TAny *arg3)
  {
  // Handle global events
  switch(aEvent)
    {
    //Close session in case of error
    case EMsvGeneralError:
    case EMsvCloseSession:
    case EMsvServerFailedToStart:
    case EMsvServerTerminated:
      User::Leave(KErrGeneral);
      break;

    default: // Do nothing
      break;
    }
  }
```

In Section 10.6, we will add to this implementation code to handle other types of event. In this case we choose to have the `HandleSessionEventL()` method at the server level because we are going to have two communication sessions and we need only one Message Server session.

The other preparatory work we will do is set up some member variables that we will require later on. These are:

- a reference to an SMS client MTM
- a Rich Text object that we will use for body text
- a 'Waiter' object that we will need in order to manage asynchronous Message Server operations.

We create these in a class `CRSmsMsg` that we will be able to use in both types of PC Connectivity server. This class will use standard two-phase construction.

```
class CRSmsMsg : public CBase
{
public:
  static CRSmsMsg *NewL(CMsvSession *aMsvSession);
  CRSmsMsg();
  void ConstructL(CMsvSession *aMsvSession);
  virtual ~CRSmsMsg();

  private:

  CMsvSession* iMsvSession; // Message Server Session
  CClientMtmRegistry* iMtmRegistry;
  CSmsClientMtm* iSmsMtm;
  CParaFormatLayer* iParaFormatLayer;
  CCharFormatLayer* iCharFormatLayer;
  CRichText* iRichText;
  CMsvOperationWait* iWaiter;
};
```

The SMS Client MTM reference is accessed by means of the MTM registry using a fixed UID. Creating a Rich Text object requires some intermediate objects that we don't really care about, so we just view this code as 'boiler-plate'.

```
void CRSmsMsg::ConstructL(CMsvSession *aMsvSession)
  {
  iMsvSession = aMsvSession;

  iMtmRegistry = CClientMtmRegistry::NewL(*iMsvSession);
  iSmsMtm = static_cast<CSmsClientMtm*>(iMtmRegistry->
    NewMtmL(KUidMsgTypeSMS));

  iParaFormatLayer = CParaFormatLayer::NewL();
  TCharFormat format(_L("Arial"),150);
  format.iFontPresentation.iTextColor=KRgbWhite;
  format.iFontPresentation.iHighlightColor=KRgbBlack;
  format.iFontPresentation.iHighlightStyle=
    TFontPresentation ::EFontHighlightRounded;
  TCharFormatMask mask;
  mask.SetAttrib(EAttColor);
  mask.SetAttrib(EAttFontHighlightColor);
  mask.SetAttrib(EAttFontHighlightStyle);
  iCharFormatLayer = CCharFormatLayer::NewL(format,mask);
```

```
iRichText = CRichText::NewL(iParaFormatLayer, iCharFormatLayer,
                            CEditableText::ESegmentedStorage, 2048);

iWaiter = CMsvOperationWait::NewLC();
CleanupStack::Pop(); // iWaiter
}
```

10.4 Listing SMS Messages and Returning Their Contents

The first function that we will implement is retrieving details of one or more SMS messages. We want to be able to retrieve the contents of a message folder, such as the Inbox or the Sent Messages folder. We also want to be able to retrieve a single message, since we will need this when we handle events to retrieve individual incoming messages without needing to retrieve a whole folder.

We start with a routine to write the details we want based on a `TMsvEntry` object. We pass in the buffer in which to write the data.

```
void CRSmsMsg::GetOneSmsL(TMsvEntry aEntry, TDes8 &aBuffer)
  {
  // Check that the entry is an SMS message
  if((aEntry.iType == KUidMsvMessageEntry) &&
     (aEntry.iMtm == KUidMsgTypeSMS))
    {
    CConnPack::WriteInt32L(aEntry.Id(), aBuffer);
    CConnPack::WriteInt32L(aEntry.Parent(), aBuffer);

    // The message timestamp can be sent as two x 32-bits
    TInt64 msgTime = aEntry.iDate.Int64();
    CConnPack::WriteInt32L(msgTime.High(), aBuffer);
    CConnPack::WriteInt32L(msgTime.Low(), aBuffer);

    // Retrieve the message description and details
    // directly from the TMsvEntry
    TPtrC msgDesc = aEntry.iDescription;
    CConnPack::WriteASCIIDataL(msgDesc, aBuffer);
    TPtrC msgDetails = aEntry.iDetails;
    CConnPack::WriteASCIIDataL(msgDetails, aBuffer);

    // We want to access some fields using the SMS-specific
    // APIs so we set the Client SMS MTM context
    iSmsMtm->SwitchCurrentEntryL(aEntry.Id());
    iSmsMtm->LoadMessageL();

    // Write the From address using the SMS header
    TPtrC tempPtrC;
    tempPtrC.Set(iSmsMtm->SmsHeader().FromAddress());
    CConnPack::WriteASCIIDataL(tempPtrC, aBuffer);

    // Write the recipients - start with the number of
    // recipients and then write the name and number for each
    TInt numRecipients = iSmsMtm->SmsHeader().Recipients().Count();
```

```
    CConnPack::WriteInt32L(numRecipients, aBuffer);
    for (TInt ii = 0; ii < numRecipients; ii++)
      {
      tempPtrC.Set(iSmsMtm->SmsHeader().Recipients().At(ii)->Name());
      CConnPack::WriteASCIIDataL(tempPtrC, aBuffer);
      tempPtrC.Set(iSmsMtm->SmsHeader().Recipients().At(ii)->Address());
      CConnPack::WriteASCIIDataL(tempPtrC, aBuffer);
      }

    // The body text is accessible via the SMS MTM.
    // We do not know how big it might be so we allocate
    // a buffer especially
    TInt bodyLen = iSmsMtm->Body().DocumentLength();
    HBufC* body = HBufC::NewLC(bodyLen);
    TPtr tempPtr(body->Des());
    iSmsMtm->Body().Extract(tempPtr);
    CConnPack::WriteASCIIDataL(tempPtr, aBuffer);
    CleanupStack::PopAndDestroy(body);
    }
  }
```

Given this, handling the command to retrieve one SMS message just requires us to unpack the message index identifier and use it to generate a `TMsvEntry` to pass to our `GetOneSmsL()` method. We will create a response PDU that is suitable for one or many message details, so we will append a null message identifier to signify the end of the response.

```
void CRSmsMsg::GetSmsByIdL(TInt aMsgId, TDes8 &aBuffer)
  {
  TMsvEntry msgEntry;
  TMsvId owningServiceId;
  User::LeaveIfError(iMsvSession->GetEntry(aMsgId, owningServiceId,
    msgEntry));

  GetOneSmsL(msgEntry, aBuffer);

  // Send the complete PDU with a terminating null ID
  CConnPack::WriteInt32L(KNullUidValue, aBuffer);
  }
```

Handling the request to retrieve all messages in a folder has two extra complications:

1. We need to retrieve the set of message index identifiers rather than just one message.
2. Because we do not know how many messages we may retrieve or how long they are, we cannot guarantee that they will all fit in one response.

The first requirement is dealt with by using the `ChildrenWith-TypeL()` method of the folder index entry to access a `CMsvEntrySe-lection`.

The second requirement is addressed by writing the details of each individual message into a temporary buffer and then appending it to our main buffer, if it will fit. Because we cannot tell how large the message data will be until after we have retrieved it, we have to potentially retrieve it and then throw it away. To allow for folders with many messages, we keep the set of message index identifiers as a member variable and we have another request to retrieve more messages. This means that the client can keep asking for messages until no more are returned. This code is more complicated than just hoping that all the messages will fit into one buffer, but it is more reliable.

Because we support two commands with similar behavior – fetch all messages in a folder and fetch more messages after filling an initial response – we can put the logic for retrieving SMS messages in one routine and use the other just to initialize the set of message identifiers.

```
// Fetch all SMS messages from Inbox or other folder and return them
void CRSmsMsg::GetAllSmsL(TInt aBoxId, TDes8 &aBuffer,
  TDes8 &aTempBuffer)
  {
  // Get access to the selected folder with default grouping
  // and sorting
  TMsvSelectionOrdering order(KMsvNoGrouping, EMsvSortByNone, ETrue);
  if(iBoxEntry != NULL)
    {delete iBoxEntry;}
  iBoxEntry = CMsvEntry::NewL(*iMsvSession, aBoxId, order);

  // get the list of message entries in the folder
  if(iSelectedMessages != NULL)
    {delete iSelectedMessages;}
  iSelectedMessages = iBoxEntry->ChildrenWithTypeL(KUidMsvMessageEntry);

  // Traverse the messages
  iNextSmsToSend = 0;
  GetMoreSmsL(aBuffer, aTempBuffer);
  }

// Fetch remaining Sms messages from Inbox or other folder
// and return them
void CRSmsMsg::GetMoreSmsL(TDes8 &aBuffer, TDes8 &aTempBuffer)
  {
  // Set up for saying that there are no more messages
  TBool gotMessages = EFalse;

  // If there are pending messages then send them, otherwise, say
  // there are no more
  TInt msgCount = iSelectedMessages->Count();
  // Traverse the messages
  while(iNextSmsToSend < msgCount )
    {
    gotMessages = ETrue;
    // Get the message details in a temporary buffer
    TMsvEntry msgEntry;
    TMsvId owningServiceId;
    User::LeaveIfError(iMsvSession->GetEntry((*iSelectedMessages)
```

```
      [iNextSmsToSend], owningServiceId, msgEntry));
    aTempBuffer.Zero();
    GetOneSmsL(msgEntry, aTempBuffer);

    // The entry may not have been useful so check that data has been
    // added
    if( aTempBuffer.Length() > 0)
      {
      // If the message will fit then append it, otherwise break out
      if(aTempBuffer.Length()+aBuffer.Length()+4 < aBuffer.MaxLength())
        {
        CConnPack::WriteBufferL(aTempBuffer, aBuffer);
        iNextSmsToSend ++;
        }
      else
        { // Run out of room - leave it till next message
        break;
        }
      }
    else
      { // Not an SMS so skip it
      iNextSmsToSend ++;
      }
    } // endwhile

  if(gotMessages)
    {
    // Send the complete message with a terminating null message ID
    CConnPack::WriteInt32L(KNullUidValue, aBuffer);
    }
  }
```

10.5 Deleting and Creating SMS Messages

The previous sections show how to read SMS messages. In this section we will show how to delete a message and how to create and send one.

Deleting a message is relatively straightforward, as long as the developer is aware that it has to be done with the context of the `CMsvEntry` set to the parent of the entry to be deleted.

```
void CRSmsMsg::DeleteSmsL(TInt aDeleteId, TDes8 &aBuffer)
  {
  CMsvEntry* entry = CMsvEntry::NewL(*iMsvSession, KMsvRootIndexEntryId,
                                     TMsvSelectionOrdering());
  CleanupStack::PushL(entry);
  entry->SetEntryL(aDeleteId);
  // Check that it is an SMS message
  if((entry->Entry().iType == KUidMsvMessageEntry) &&
     (entry->Entry().iMtm == KUidMsgTypeSMS))
    {
    TInt parentId = entry->Entry().Parent();
    entry->SetEntryL(parentId);
    entry->DeleteL(aDeleteId);
    CleanupStack::PopAndDestroy(entry);
```

```
    CConnPack::WriteInt32L(aDeleteId, aBuffer);
    }
  else
    {
    // Don't need to pop entry as the leave will take care of it
    User::Leave(KErrArgument);
    }
  }
```

Creating a message is more complex, but it will enable us to send messages from the PC. This is the function that I find most useful, as it allows me to use the PC keyboard to compose the message – for some reason my SMS messages have grown more verbose since I developed this software!

Because this code is relatively complex, it deserves more explanation than the earlier routines, so I will go through it in sections. To begin with, we create the message entry. As for deletion, creation is done by means of the parent. In this case the parent is the Outbox. Once we have the empty message, we will set the body text from the incoming data. We can then set the message description – for an SMS message this is the first 32 characters of the body text.

```
TInt CRSmsMsg::SendSmsL(TConnBuff &aBuffer)
  {
  iSmsMtm->SwitchCurrentEntryL(KMsvGlobalOutBoxIndexEntryId);
  iSmsMtm->CreateMessageL(iSmsMtm->ServiceId());
  TMsvEntry newEntry = iSmsMtm->Entry().Entry();

  TInt bodyLen = CConnPack::PeekUint16L(aBuffer);
  HBufC* bodyBuff = HBufC::NewLC(bodyLen);
  TPtr tempPtr(bodyBuff->Des());
  CConnPack::ReadASCIIDataL(tempPtr, aBuffer);
  CRichText& mtmBody = iSmsMtm->Body();
  mtmBody.Reset();
  mtmBody.InsertL(0,tempPtr);

  // Set the description from the first 32 chars of the body
  TBuf<KMaxDescriptionLength> description;
  if (bodyLen > 32)
    bodyLen = 32;
  tempPtr.SetLength(bodyLen);
  description.Copy(tempPtr);
  newEntry.iDescription.Set(description);
  CleanupStack::PopAndDestroy(bodyBuff);
```

The other information that we supply from the PC is the phone number to which the message is addressed. We could use the `Recipients` member of the `CSmsHeader` class as we did when reading an SMS message, but, in fact, we can use the simpler API from the `CBaseMtm`

class. In SMS messages, the first recipient address is also regarded as the message detail.

```
// Read and set addressees
TInt addressCount = CConnPack::ReadUint16L(aBuffer);
for(TInt i = 0 ; i <addressCount ; i++)
  {
  TInt addressLen = CConnPack::PeekUint16L(aBuffer);
  HBufC* addressBuff = HBufC::NewLC(addressLen);
  TPtr addTempPtr(addressBuff->Des());
  CConnPack::ReadUNCDataL(addTempPtr, aBuffer);
  iSmsMtm->AddAddresseeL(addTempPtr);
  // Set detail as first recipient address
  if(i==0)
    {
    TBuf<KMaxDetailLength> detail;
    if(addressLen > KMaxDetailLength)
      {
      addTempPtr.SetLength(KMaxDetailLength);
      }
    detail.Copy(addTempPtr);
    newEntry.iDetails.Set(detail);
    }
  CleanupStack::PopAndDestroy(addressBuff);
  }
```

The final part of creating the draft message is to set the SMS Service Center and the SMS service settings. These can just be set to the default values currently set in the SMS MTM, but they must be set. We also set some of the attributes of the `TMsvEntry` – the message timestamp and others.

```
iSmsMtm->SmsHeader().SetSmsSettingsL(iSmsMtm->ServiceSettings());
TInt defaultIndex = iSmsMtm->ServiceSettings().DefaultSC();
iSmsMtm->SmsHeader().SetServiceCenterAddressL(
  iSmsMtm->ServiceSettings().SCAddress(defaultIndex).Address());

newEntry.iDate.HomeTime();
newEntry.SetInPreparation(EFalse);
newEntry.SetSendingState(KMsvSendStateScheduled);
newEntry.SetScheduled(ETrue);
newEntry.SetVisible(ETrue);

// Save changes
iSmsMtm->Entry().ChangeL(newEntry);
iSmsMtm->SaveMessageL();
```

At this stage the message exists as a draft message and the next stage is to actually send it. Exactly how a message is sent depends on the MTM, but for an SMS message it needs to be copied to the SMS Service Center. The Message Server provides synchronous and asynchronous methods to copy messages. In most cases the asynchronous method is preferable, because it allows an application to remain responsive, and in this case

the synchronous method simply does not work (because the developers did not see any value in a synchronous version).

This causes us a problem. In the cases of a custom server or a socket server, the command is handled in a synchronous manner – we are expected to finish processing and return. In fact, it is possible to extend the socket server methods to allow for asynchronous behavior, but `ectcpadapter` is not so flexible. This means that we have two choices: use an inner scheduler or trigger an Active Object with no way of obtaining the result.

An inner scheduler is the term used for the practice of incrementing the level of the current Active Scheduler. Unless you are familiar with the extensive use of Active Objects in Symbian OS, that explanation may not leave you any the wiser. Essentially, it is a technique for running an asynchronous action in a synchronous way. In this case we could use it to wait for the asynchronous send operation to complete and then return the result to the PC client. However, inner schedulers are tricky and can cause some very complicated defects unless you know what you are doing. In this case, I have chosen to avoid inner schedulers and trigger another Active Object.

Active Objects were briefly described in Chapter 6 and we are going to create one that does nothing except delete itself.

```
class CMsgWait: public CActive
  {
public:
  static CMsgWait* NewLC(TInt aPriority=EPriorityStandard);
  ~CMsgWait();
  void Start();
protected:
  CMsgWait(TInt aPriority);
  void RunL();
  void DoCancel();
};
```

This class has the standard `CActive` methods plus a `Start()` and `NewLC()` methods. The `Start()` method just activates the object and the `RunL()` deletes itself.

```
void CMsgWait::Start()
  {
  iStatus = KRequestPending;
  SetActive();
  }

void CMsgWait::RunL()
  {
  delete this;
  }
```

We can then use this class to asynchronously send the SMS message. When the sending completes, the `CMsgWait` object will be notified, but it will not care. This means that we cannot tell whether or not the send was successful, but we can use the event handler for this purpose.

```
// Send the actual message
CMsvEntry* parentEntry = CMsvEntry::NewL( *iMsvSession,
  newEntry.Parent(), TMsvSelectionOrdering());
CleanupStack::PushL(parentEntry);

CMsgWait* waiter = CMsgWait::NewLC();
waiter->Start();
parentEntry->CopyL( newEntry.Id(), iSmsMtm->ServiceId(),
  waiter->iStatus);

CleanupStack::Pop(waiter); // It will destroy itself
CleanupStack::PopAndDestroy(parentEntry);

return(newEntry.Id());
}
```

10.6 Handling Message Server Events

The final part of our messaging code is to handle events. The primary aim here is to pick up incoming messages without having to poll the mobile phone regularly. In fact, the Message Server provides us with events whenever a new message is created or a message is changed or deleted. Each time the event handler is called, we get a number of events reported.

The messaging part of the code is relatively straightforward. If we have a new, changed or moved message, we verify that it is an SMS message before informing the PC of the details. If it is a deleted message then we cannot check the details, because the message no longer exists. We assume that the PC client will be able to handle being told about the deletion of a message that it does not have.

Because we receive events for all types of entry in the Message Store, we have to allow for the fact that we may ignore all the events and not report any. This might be the case if they concern email or MMS message, for example. Therefore, we may need to discard the partial PDU.

```
void CRSmsMsg::GetMsvEventL
  ( MMsvSessionObserver::TMsvSessionEvent aEvent,
  TAny *arg1, TAny *arg2, TAny *arg3, TDes8 &aBuffer)
  {
  // Preserve the initial length
  TInt startLength = aBuffer.Length();
  CConnPack::WriteInt32L(aEvent, aBuffer);

  CMsvEntrySelection* eventEntrySelection = (CMsvEntrySelection*)arg1;
```

```
TInt msgCount = eventEntrySelection->Count();

TBool gotData = EFalse; // goes true if we have an SMS message
for (TInt jj = 0; jj < msgCount; jj++)
  {
  TMsvId id = eventEntrySelection->At(jj);
  // For deletion - don't check type, for others do
  if(aEvent == MMsvSessionObserver::EMsvEntriesDeleted)
    {
    CConnPack::WriteInt32L(id, aBuffer);
    gotData = ETrue;
    }
  else
    {
    TMsvEntry msgEntry;
    TMsvId owningServiceId;
    User::LeaveIfError(iMsvSession->GetEntry(id, owningServiceId,
      msgEntry));
    // Check that it is an SMS message
    if((msgEntry.iType == KUidMsvMessageEntry) &&
       (msgEntry.iMtm == KUidMsgTypeSMS))
      {
      CConnPack::WriteInt32L(id, aBuffer);
      gotData = ETrue;
      }
    }// endif
  }// endfor

if(gotData)
  {
  // Write terminating null ID
  CConnPack::WriteInt32L(0, aBuffer);

  // If a moved event then we have extra data in arg2 and arg3
  if(aEvent == MMsvSessionObserver::EMsvEntriesMoved)
    {
    CConnPack::WriteInt32L(*(TInt32*)arg2, aBuffer);
    CConnPack::WriteInt32L(*(TInt32*)arg3, aBuffer);
    }
  }
else
  {
  // No real data (messages were not SMS messages) so reset back
  // the offset
  TInt newLength = aBuffer.Length()-startLength;
  aBuffer.Delete(startLength-1,newLength);
  }
```

One point to note is that we might be tempted to get all the message data in this routine, but this would be a mistake. As a rule, we should not do too much work in any event handler, so I do not want to do anything more than unpack the event details. Also, experience shows that we may get several events for the same message in quick succession. If we get message data for each event then we will be sending the same data repeatedly. Instead, we just send the raw event details, and the PC

client will be responsible for filtering out repeated events and retrieving the data only once.

10.7 Putting the Messaging Code in a Connectivity Plug-in

The previous sections have shown the messaging code. We also need the code in the custom server or socket server to invoke these methods.

Most of the commands are straightforward. We need to identify the incoming command, call a messaging method and return a response. For a socket server the factory does this, but for a custom server we build our own code – this is a standard pattern.

```
void CRSmsCSSession::ReadCompleteL(TDes8* /*aPdu*/)
  {
  TRSmsCmdCode aCmd = (TRSmsCmdCode)CConnPack::ReadInt32L(iReadPtr);
  TInt msgId = CConnPack::ReadInt32L(iReadPtr);

  TRAPD(retVal, DoServiceL(aCmd, msgId));
  if(retVal != KErrNone)
    {
    WriteErrorL(retVal, msgId);
    }
  }

void CRSmsCSSession::DoServiceL(TRSmsCmdCode aCmd, TInt aMsgId)
  {
  switch(aCmd)
    {
    case ERSmsCmdQueryVersion:
      QueryVersionL(aMsgId);
      break;
    case ERSmsCmdGetAllSms:
      GetAllSmsL(aMsgId);
      break;
    case ERSmsCmdGetMoreSms :
      GetMoreSmsL(aMsgId);
      break;
    case ERSmsCmdGetSmsById:
      GetSmsByIdL(aMsgId);
      break;
    case ERSmsCmdDeleteSms:
      DeleteSmsL(aMsgId);
      break;
    case ERSmsCmdSendSms:
      SendSmsL(aMsgId);
      break;
    default:
      break; //Do Nothing
    }
  return;
  }
```

We call the service routine via a `TRAP` so we can catch errors and return an error PDU to the PC client.

Then the individual service routines here call one of the messaging routines and return the response. They are all quite similar, so I will just show one here.

```
void CRSmsCSSession::GetSmsByIdL(TInt aMsgId)
  {
  TInt msgId = CConnPack::ReadInt32L(iReadPtr);

  // Start a PDU with a response
  iWritePtr.Zero();
  CConnPack::WriteInt32L(ERSmsCmdReceiveInitialSms, iWritePtr);
  CConnPack::WriteInt32L(aMsgId, iWritePtr);

  iMsgUtil->GetSmsByIdL(msgId, iWritePtr);

  Write(&iWritePtr);
  }
```

The one aspect missing from this PC Connectivity plug-in is the handling of Message Server events. The connection (for the custom server and the socket server) cannot be used for reading and writing simultaneously. Therefore, we will need a second connection just for the events. In fact, we could avoid this by caching event details and requesting them on a regular basis, but that would impose a delay on receiving incoming messages. Therefore, we have defined the command to put a connection into event reporting mode.

We are supposed to have only one Message Server session per thread, and both the connections will share the same thread (certainly in the case of a custom server). Therefore, we have made the `CMsvSession` owned by the top-level server. When an event handling command comes in, the server is informed that this session is to be used for event reporting. Subsequently, when an event occurs, it is redirected to the relevant session.

```
void CRSmsCSServer::SetEventSession(CRSmsCSSession* aEventSession)
  {
  iEventSession = aEventSession;
  }
```

```
...
void CRSmsCSSession::DoServiceL(TRSmsCmdCode aCmd, TInt aMsgId)
  {
  switch(aCmd)
    {
    case ERSmsCmdQueryVersion:
      QueryVersionL(aMsgId);
      break;
    case ERSmsCmdGetAllSms:
      GetAllSmsL(aMsgId);
      break;
    case ERSmsCmdGetMoreSms :
```

```
      GetMoreSmsL(aMsgId);
      break;
    case ERSmsCmdGetSmsById:
      GetSmsByIdL(aMsgId);
      break;
    case ERSmsCmdDeleteSms:
      DeleteSmsL(aMsgId);
      break;
    case ERSmsCmdSendSms:
      SendSmsL(aMsgId);
      break;
    // This session is now supposed to return events
    case ERSmsCmdReturnEvents:
      iEventMsgId = aMsgId;
      iPendingEventWrite = EFalse;
      iRSmsCSServer->SetEventSession(this);
      break;

    default:
      break; //Do Nothing
    }
  return;
  }
...
void CRSmsCSServer::HandleSessionEventL(TMsvSessionEvent aEvent,
                                  TAny *arg1, TAny *arg2, TAny *arg3)
  {
  // Handle global events
  switch(aEvent)
    {
    //Close session in case of error
    case EMsvGeneralError:
    case EMsvCloseSession:
    case EMsvServerFailedToStart:
    case EMsvServerTerminated:
      User::Leave(KErrGeneral);
      break;

    default: // Do Nothing
      break;
    }

  if(iEventSession)
    {
    TRAPD(retVal, iEventSession->WriteMsvEventL(aEvent, arg1, arg2,
      arg3));
    if(retVal != KErrNone)
      {
      iEventSession->WriteErrorL(retVal, 0);
      }
    }
  }
```

The next complication we encounter is that we may get one event from the Message Server, report it to the PC client and then get another event before we have finished writing the first message. We cannot write again until the first write is complete. Therefore, we need to keep track with a flag of whether or not we have a write operation pending. If we do

not have a pending write then we just send the PDU straight to the PC. If we do have a pending write then we can only append the event details to a buffer and wait for the last write to complete – remember that we may have more than one event while we have a pending write. This is a good example of real asynchronous event handling, but we are fortunate that the buffering solution is adequate.

```
void CRSmsCSSession::WriteMsvEventL
  (MMsvSessionObserver::TMsvSessionEvent aEvent,
  TAny *arg1, TAny *arg2, TAny *arg3)
  {
  // We may get one event before the previous one has been fully
  // written out. Therefore, we check for this. If there is no pending
  // write then we write directly to the standard buffer.
  // If there is a pending write, we append to the temporary buffer.
  // If there is data in the temporary buffer when we get WriteComplete
  // then it will be flushed.
  if( !iPendingEventWrite)
    {
    iWritePtr.Zero();
    CConnPack::WriteInt32L(ERSmsCmdMsvEvent, iWritePtr);
    CConnPack::WriteInt32L(iEventMsgId, iWritePtr);
    iMsgUtil->GetMsvEventL(aEvent, arg1, arg2, arg3, iWritePtr);

    // The PDU may be empty if there were no SMS-related events
    if(iWritePtr.Length() > 8)
      {
      CConnPack::WriteInt32L(KNoMoreEvents, iWritePtr);
      Write(&iWritePtr);

      // Set up for pending event writes
      iPendingEventWrite = ETrue;
      iTempBuffer.Zero();
      CConnPack::WriteInt32L(ERSmsCmdMsvEvent, iTempBuffer);
      CConnPack::WriteInt32L(iEventMsgId, iTempBuffer);
      }
    } // endif
  else // Got pending write
    {
    iMsgUtil->GetMsvEventL(aEvent, arg1, arg2, arg3, iTempBuffer);
    }
  }
```

Finally, when a write operation completes, we need to check for data waiting to be sent to the PC.

```
void CRSmsCSSession::WriteCompleteL(TDes8* /*aPdu*/)
  {
  // If we have a pending event write, it is now safe to write it out.
  if(iPendingEventWrite && (iTempBuffer.Length() > 8))
    {
    CConnPack::WriteInt32L(KNoMoreEvents, iTempBuffer);
    iPendingEventWrite = EFalse;
    Write(&iTempBuffer);
    }
  }
```

10.8 A Command-line SMS Application

Having created the SMS software on the Symbian OS smartphone, we need to create a client to communicate with it. For real usage we will want a graphical program, but for development and debugging purposes a command-line application is adequate. For one thing, we can just debug one software component at a time and then, when the software on the mobile phone is working, we can more easily develop the GUI application.

Initially, we will need methods to pack and unpack text data. We saw how to read and write integers in Chapter 7, and textual data is handled similarly. Reading data is done simply by extracting data from an array of bytes, but for writing we will now append to an expandable `ArrayList` and then convert it into an array of bytes when necessary. We do this because in some later routines we will find it difficult to predict the size of our data in advance.

```
public void ReadASCIIData( ref  byte[] aBuff, ref int aStartPos,
  out StringBuilder aString)
{
  int dataLength = ReadInt32(ref aBuff, ref aStartPos);
  aString = new System.Text.StringBuilder();
  aString.Length = dataLength;
  for(int j = 0 ; j < dataLength ; j++)
  {
    char myChar = (char)aBuff[aStartPos];
    aStartPos++;
    aString[j] = myChar;
  }
}

public static void WriteInt32( int aValue, ref ArrayList aBuffer )
{
  aBuffer.Add((byte) (aValue & 0x000000ff));
  aBuffer.Add((byte)((aValue & 0x0000ff00)>>8));
  aBuffer.Add((byte)((aValue & 0x00ff0000)>>16));
  aBuffer.Add((byte)((aValue & 0xff000000)>>24));
}

public static void WriteUint16( ushort aValue, ref ArrayList aBuffer )
{
  aBuffer.Add((byte) (aValue & 0x00ff));
  aBuffer.Add((byte)((aValue & 0xff00)>>8));
}

public static void WriteASCIIData( string aString,
  ref  ArrayList aBuff )
{
  ushort dataLength = (ushort)aString.Length;
  WriteUint16(dataLength, ref aBuff);
  for(int j = 0 ; j < dataLength ; j++)
  {
    aBuff.Add((byte)aString[j]);
```

```
  }
}

public static byte[] AsByteArr( ref ArrayList aBuffer)
{
  int len = aBuffer.Count;
  byte[] ret = new byte[len];
  for(int i = 0 ; i < len ; ++i)
  {
    ret[i] = (byte)aBuffer[i];
  }
  return ret;
}
```

These text-handling routines assume ASCII data in the buffer. Unicode data is handled in the same way.

In order to read or write date and time values, we use the C# `DateTime` type and handle null values explicitly.

```
public static void ReadDateTime( ref byte[] aBuffer, ref int aStartPos,
                   out bool aNullTime, out System.DateTime aDateTime)
{
  short year = ReadInt16(ref aBuffer, ref aStartPos);
  short month = ReadInt8(ref aBuffer, ref aStartPos);
  short day = ReadInt8(ref aBuffer, ref aStartPos);
  short hour = ReadInt8(ref aBuffer, ref aStartPos);
  short minute = ReadInt8(ref aBuffer, ref aStartPos);
  short second = ReadInt8(ref aBuffer, ref aStartPos);

  if((year==0) && (month==0) && (day==0) &&
     (hour==0) && (minute==0) && (second==0))
  {
    aNullTime = true;
    aDateTime = System.DateTime.Now;
  }
  else
  {
    aNullTime = false;
    aDateTime = new System.DateTime(year, month+1, day+1, hour,
      minute, second);
  }
}

public static void WriteDateTime( bool aNullTime,
  System.DateTime aDateTime, ref ArrayList aBuffer )
{
  if(aNullTime)
  {
    WriteInt16(0, ref aBuffer);
    WriteInt8(0, ref aBuffer);
    WriteInt8(0, ref aBuffer);
    WriteInt8(0, ref aBuffer);
    WriteInt8(0, ref aBuffer);
    WriteInt8(0, ref aBuffer);
  }
  else
```

```
  {
    WriteInt16((short)aDateTime.Year, ref aBuffer);
    WriteInt8((short)(aDateTime.Month-1), ref aBuffer);
    WriteInt8((short)(aDateTime.Day-1), ref aBuffer);
    WriteInt8((short)aDateTime.Hour, ref aBuffer);
    WriteInt8((short)aDateTime.Minute, ref aBuffer);
    WriteInt8((short)aDateTime.Second, ref aBuffer);
  }
}
```

Loading the custom server or socket server is done in the same way as for the echo plug-ins described in Chapters 7 and 8.

Creating the command PDUs is simply a matter of packing the arguments into a buffer and sending the buffer. The following example would create a command to request the contents of the Inbox.

```
ArrayList message = new  ArrayList();
WriteInt32(ERSmsCmdGetAllSms, ref message);
WriteInt32(mNextPDUId, ref message);
WriteInt32(KMsvGlobalInBoxIndexEntryId, ref message);
mNextPDUId++;
mAStream.Write(ConnPack.AsByteArr(ref message));
mReadingLength = true;
mAStream.Read(4);
```

As we see, once the PDU is sent we start a read operation to obtain the reply PDU.

As discussed, we need to open a second connection for event handling. As soon as we open the stream, we send a command to enable event reporting on it and then initiate a read and wait for events to pop up. Clearly, this behavior would not be possible without asynchronous reads. In this example, we are using the custom server – hence the use of `ectcpadapter`.

```
private void EnableEvents()
{
  // Create a second stream for event handling
  SymbianConnectBAL.ISCBALDeviceService service =
    mDevice.Services["ectcpadapter"];
  // Load RSmsCS
  byte[] message = new byte[40];
  message[4] = (byte)'R';
  message[5] = (byte)'S';
  message[6] = (byte)'M';
  message[7] = (byte)'S';
  message[8] = (byte)'C';
  message[9] = (byte)'S';

  SymbianConnectBAL.ISCBALSequentialStream myStream2 =
    service.StartServiceOnStream();
  int retval = myStream2.Write(message);
```

```
  byte[] response = new byte[40];
  object oResponse = response;
  retval = myStream2.Read(4, out oResponse);
  response = (byte[])oResponse;
  int readPos = 0;
  int responseCode = ReadInt32(ref response, ref readPos);
  if(responseCode != 0)
  { System.Console.WriteLine("Response code when starting event
                             session {0}", responseCode);
  }

  //  Now create an asynchronous stream from the synchronous one
  mEventStream = (SymbianConnectBAL.BALApplicationAsyncStream)myStream2;
  mEventStream.OnRead += new
    ISCBALSequentialStreamSink_OnReadEventHandler(OnEventRead);
  mEventStream.OnWrite += new
    ISCBALSequentialStreamSink_OnWriteEventHandler(OnWrite);

  // Send a command to activate event reception
  ArrayList eMessage = new  ArrayList();
  ConnPack.WriteInt32(8, ref eMessage);
  ConnPack.WriteInt32(ERSmsCmdReturnEvents, ref eMessage);
  ConnPack.WriteInt32(mNextMessageId, ref eMessage);
  mAStream.Write(ConnPack.AsByteArr(ref eMessage));
  mEventReadingLength = true;
  mEventStream.Read(4);
  mNextMessageId ++;
}
```

Handling a reply is slightly more complex than sending PDUs. Whenever we expect a reply, we may get an error reply. When we are expecting a set of SMS message data, we may get a PDU that indicates that there are no messages to be returned. Therefore, our design may well be based around a single method to read and parse any reply PDUs. This also reduces the amount of state that we need to build into our application.

The following example just prints out the data to the console. In a production version we would update displays of messages. We would also build in more checks for corrupted messages.

```
private void ServiceRead(int aOpCode, int aMessageId, ref byte[] aBuff,
                         ref int aReadPos)
{
  System.Console.WriteLine("Read Op code {0} Message Id {1}", aOpCode,
    aMessageId);
  switch(aOpCode)
  {
  case(ERSmsCmdVersionReply):
    int majorVersion = ReadInt32(ref aBuff, ref aReadPos);
    int minorVersion = ReadInt32(ref aBuff, ref aReadPos);
    int buildVersion = ReadInt32(ref aBuff, ref aReadPos);
    System.Console.WriteLine("Reported version {0}:{1}:{2}",
                             majorVersion, minorVersion, buildVersion);
    break;
```

```
case(ERSmsCmdReceiveSms):
  bool carryOn = true;
  while(carryOn)
  {
    carryOn = ReadSms( ref aBuff, ref aReadPos);
  }
  break;
case(ERSmsCmdReceiveNoMoreSms):
  System.Console.WriteLine("No more messages to retrieve");
  break;
case(ERSmsCmdMsvEvent):
  int eventType = ReadInt32(ref aBuff, ref aReadPos);
  while(eventType != KNoMoreEvents)
  {
    switch(eventType)
      {
      case(EMsvEntriesCreated):
        System.Console.WriteLine("Entries created event");
        break;
      case(EMsvEntriesChanged):
        System.Console.WriteLine("Entries changed event");
        break;
      case(EMsvEntriesDeleted):
        System.Console.WriteLine("Entries deleted event");
        break;
      case(EMsvEntriesMoved):
        System.Console.WriteLine("Entries moved event");
        break;
      default:
        System.Console.WriteLine("Unrecognized event type {0}",
                                 eventType);
        break;
      }
    int eventId = ReadInt32(ref aBuff, ref aReadPos);
    while(eventId != 0)
    {
      System.Console.WriteLine("Id {0}", eventId);
      eventId = ReadInt32(ref aBuff, ref aReadPos);
    }
    // If a move event then we have two more IDs
    if(eventType == EMsvEntriesMoved)
    {
      int oldParentId = ReadInt32(ref aBuff, ref aReadPos);
      int newParentId = ReadInt32(ref aBuff, ref aReadPos);
      System.Console.WriteLine("Moved from {0} to {1}",
                               oldParentId, newParentId);
    }
    // Read the next event
    eventType = ReadInt32(ref aBuff, ref aReadPos);
  }
  // Re-initiate reading
  mEventReadingLength = true;
  mEventStream.Read(4);
  break;
case(ERSmsCmdSentSms):
  int sentSmsId = ReadInt32(ref aBuff, ref aReadPos);
  System.Console.WriteLine("Sms {0} sent", sentSmsId);
  break;
```

```
  case(ERSmsCmdDeletedSms):
    int delSmsId = ReadInt32(ref aBuff, ref aReadPos);
    System.Console.WriteLine("Sms {0} deleted", delSmsId);
    break;
  case(ERSmsCmdError):
    int errno = ReadInt32(ref aBuff, ref aReadPos);
    System.Console.WriteLine("Error {0}", errno);
    break;
  default:
    System.Console.WriteLine("Unexpected op-Code {0} buffer length {1}",
                             aOpCode, aBuff.Length);
    break;
  }
}

public bool ReadSms(ref byte[] aBuff, ref int buffOffset)
{
  int smsId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
  bool retVal = true;
  if( smsId != 0)
  {
    int parentId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
    // Read the date-time
    System.DateTime msgDate;
    bool nullDate;
    ConnPack.ReadDateTime( ref aBuff, ref buffOffset, out nullDate,
                           out msgDate);

    System.Text.StringBuilder description;
    ConnPack.ReadUNCData( ref aBuff, ref buffOffset, out description);
    System.Text.StringBuilder detail;
    ConnPack.ReadUNCData( ref aBuff, ref buffOffset, out detail);
    System.Text.StringBuilder fromAddress;
    ConnPack.ReadUNCData( ref aBuff, ref buffOffset, out fromAddress);

    System.Console.WriteLine("Sms Id {0} parent {1} from {2} date {3}",
                             smsId, parentId, fromAddress, msgDate);

    int recipientCount = ConnPack.ReadUint16(ref aBuff, ref buffOffset);
    for(int i = 0 ; i < recipientCount ; i++)
    {
      System.Text.StringBuilder recName;
      ConnPack.ReadUNCData( ref aBuff, ref buffOffset, out recName);
      System.Text.StringBuilder recAddress;
      ConnPack.ReadUNCData( ref aBuff, ref buffOffset, out recAddress);

      System.Console.WriteLine("Recipient {0}:{1}", recName,
                               recAddress);
    } //endfor

    System.Text.StringBuilder bodyText;
    ConnPack.ReadASCIIData( ref aBuff, ref buffOffset, out bodyText);
    System.Console.WriteLine("{0}\n", bodyText);
  }
```

```
  else
  {
    System.Console.WriteLine("End of received sms");
    retVal = false;
  }
  return retVal;
}
```

It will be clear from some details of this code that this was intended to work with the custom server. In fact, most of the protocol and code can be common with an application intended to talk to a socket server. In Chapters 11 and 12 we will develop code to interact with the Contacts and Agenda models and then we will present a simple GUI application to use these servers.

11

Using the Contacts Model

In this chapter we are going to cover the Symbian OS Contacts Model. This is the first example of a specific API that is relevant to synchronization or Enterprise operations. In this chapter we address the use of the Contacts Model for a normal user and we also include optimizations for large Contacts databases.

For the handling of SMS messages, Chapter 9 included the description and the API, and Chapter 10 included the code (or most of it) for a PC Connectivity service. In this chapter and the next, we combine the description, the API and example code. We can build on the basic code on the PC, included in Chapter 10, to drive the service, but we will not repeat that code in this chapter.

Contacts information is one of the key parts of a Personal Information Manager (PIM) and is vital to most users. The PC suites shipped with Symbian OS smartphones normally include software to synchronize the smartphone with a PIM running on the PC, such as Outlook. For many users this is the most important part of a PC suite.

However, not all PIMs are supported by all synchronization software, and some users may not want the master copy of their data to be maintained on the PC. Therefore, an alternative approach is to maintain the master version of contacts data on the phone and make sure that it gets backed up.

The Contacts Model is not difficult to access. This chapter shows the APIs used and includes example software to create a Connectivity plug-in to allow direct manipulation of Contacts data on the mobile phone from the PC.

11.1 Databases and Models

Symbian OS includes an SQL-based database management system. SQL databases are used for a number of purposes. The Symbian OS DBMS is impressive for a mobile phone, but it is not as powerful as those designed to run on PCs and servers with many times the memory and CPU power.

In some cases, a database is hidden behind a specialized API. This is true of the Contacts database, which is hidden behind a Contacts Model

API. It is still possible to access the database directly, but this is not wise and is not necessary. The Contacts Model API provides all the functions that are required and includes a range of checking and utility features. Therefore, this chapter will describe the Contacts Model API without regard to database concepts.

11.2 The Contacts Model

The Contacts Model is structured around a Contact Database that includes a number of Contact Items. Each Contact Item contains a number of fields.

A Contact Item can represent more than just a single address card. Contact Items can represent:

- an address card
- the address card of the mobile phone owner
- a group of cards
- a card template used to create new cards.

Although address cards are the most common type of item, the others are also meaningful and we will use them in this chapter.

Contact Items use a common base class, `CContactItem`, and derived classes for the specialized types. Similarly, fields have a base type, `CContactFieldStorage`, for the storage type and derived classes for the specific types (Figure 11.1).

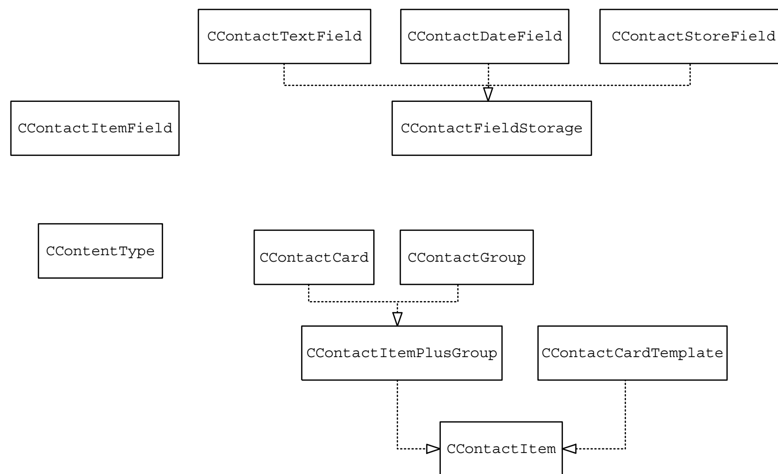


Figure 11.1 Contacts Model

An address card is the most common type of contact item. The name derives from the use of physical business cards, and the term has proved

useful for electronic purposes – synchronization uses the term vCard for a standard way of storing contact information.

An address card is represented by a `CContactCard` object but requires very little additional functionality – most of its behavior is contained in the `CContactItem` class. A `CContactCard` has a unique identifier and a set of fields that contain the actual card data. It can belong to zero or more groups of cards. A `CContactCard` can be found either directly by means of its identifier or by filters or searches.

The address card that belongs to the mobile phone owner is a standard card in most ways. It is identified separately because it can be used to identify the phone. It can be beamed to another phone and it could be used for an information screen. Therefore, the `CContactOwnCard` class is derived from `CContactCard` rather than directly from `CContactItem`.

Because the set of fields that an address card owns can be relatively complex and can be extended, it is necessary to have some method of setting the fields that a newly created card is given. This is done using templates. A `CContactCardTemplate` object contains details of fields and can be used when creating a new `CContactCard`.

Going beyond address cards, the `CContactGroup` class represents a group of address cards (actually, a group can be a member of other groups as well). A group has a label and a list of the `CContactItems` that it contains. Note that a group does not own the `CContactItems` that it contains – a `CContactItem` can be a member of more than one group. It may help to regard a `CContactGroup` as a club – a person can be a member of any number of clubs from zero upwards. However, as one group can be a member of another group, the analogy is limited. What use is made of groups depends on applications.

The fields owned by a `CContactCard` are more complex than might first be expected. Each field has a label, a type and a value, and attributes that may indicate a hidden or read-only field. The field type is also a complex construct and goes beyond the storage type (text, date/time or binary) of the field. Various applications need to extract specific data from an address card, such as first and last names, mobile phone number, email address or fax number. It would be possible to have the `CContactCard` defined with a fixed set of fields, which would make this form of access simple. However, this would make it inconvenient to add new fields and so would tend to limit the future use of the Contact Model. Therefore, these types of field are given an identifier (stored in `cntdef.h`) and each field is associated with one or more of these field type identifiers. This means that an application does not access the first name field but instead accesses the field that has the first name field type identifier (if any). Some of the field type identifiers may be associated with more than one field in a contact. Going beyond the applications on the Symbian OS mobile phone, various PIMs have their own field definitions and so additional field type identifiers have been created for synchronization/mapping

purposes. A field type can have a mapping identifier set if the other field types are not sufficient to identify it uniquely. Two field types are considered to be equal if they have the same set of field type identifiers and the same mapping identifier, if any. When comparing two fields, we must compare the complete set of field type identifiers, not just a subset.

11.3 Views

The Contact Database can be very large and complex in some cases. As described above, it includes different types of objects; it is possible to add new fields for specialized purposes, and it is possible to include a large amount of data. Therefore, we rarely want to access every single record in the database. We normally want a subset of some form.

We can reduce the set of fields that each item presents to us and the number of items that we see by using a View (note the unsurprising use of database terminology). A view contains a set of field types; only Contact Items that have one or more of those field types will be accessed, and then only those fields that have one or more of those field types will be made visible. In addition, views can be sorted without having to sort the whole of the Contact Database. This means that a view is almost certainly the best way to access a large set of Contact items. When we need to access the whole of a Contact item, we can do so without the view interfering.

The fields included in the view and the sort order are set when a view is constructed. If you need to change them then you need to recreate the view. When a view is created, the Contact Model has to carry out a range of tasks (notably sorting the contents of the view) and this is done asynchronously. Therefore, in order to use a view, you need to implement a view observer and note when the view is ready for use, otherwise a panic will ensue.

In this chapter, I have used just the local contact view, but there are other types of views that can be shared between clients or that support filtering.

11.4 Contacts Observers

As with other parts of Symbian OS, the Contact Model supports observers. By registering an observer, a process can be informed when data is added, deleted or changed, among other events.

When we were developing Messaging software we were concerned with learning when messages arrived. This is less of an issue for our use of the Contact Model. If the user is accessing the Contact Model by means of the PC then it is very unlikely that new entries will be spontaneously created or existing ones changed or deleted. Therefore, we will make no use of a general Contact Model observer.

However, we need a view observer, whether we like it or not. Creating a view requires some measure of filtering and sorting, and in Symbian OS this is carried out asynchronously. Therefore, when we create a view we need to specify a view observer in order to be informed when the view is ready for use.

11.5 Synchronization and Performance Issues

The methods used in this chapter are aimed at small numbers of changes, for example reading, editing or deleting one Contact card at a time. This is aimed at running a GUI on the PC with operations directly driven by the user. However, if you are editing or deleting many Contact items at a time then there are some patterns which can be useful. If you access Contact items in order of their identifier then performance will be improved.

Bulk deletion of Contact items should use the `CContactDatabase::DeleteContactsL()` method which takes an array of identifiers (in which case the order of identifiers in the array does not matter).

11.6 Contacts Model API

As with other APIs covered in this book, we will not cover all aspects of the Contact Model. It is fully documented in Symbian OS SDKs, but the sheer volume of information can make it more difficult to get started than you might expect. In particular, this chapter will not cover classes and methods concerned with specialized database manipulation (such as rollbacks and recovery), synchronization, field views and specialized phone number handling.

11.6.1 Contact Database

Class `CContactDatabase` Defined in `cntdb.h`
`CContactDatabase` provides access to a Contact Database and its members. It is the starting point for access to the Contact Model.
Creation Methods
`static CContactDatabase* OpenL(TThreadAccess aAccess =ESingleThread)` `static CContactDatabase* OpenL(const TDesC& aFileName, TThreadAccess aAccess =ESingleThread)`

These methods open access to an existing Contact Database. A new database can be created using the `CreateL()` methods. The method that does not specify a file name opens the default Contact Database.

`aFileName` – the name of the database to be opened if the default database is not wanted.

`aAccess` – determines whether the database should be opened for single or multi-thread access. The default `ESingleThread` setting is probably correct unless the database is to be used by a multi-threaded application.

returns – a pointer to a `CContactDatabase` object.

Sorting and Searching Methods

```
CContactIdArray* FindLC(const TDesC& aText,
                        const CContactItemFieldDef *aFieldDef)
```

This method searches for a text string in a defined set of fields in all contacts in the database.

`aText` – the text to search for.

`aFieldDef` – the set of fields to search.

returns – array of contact identifiers for contact items that contain the search string. The caller takes ownership of the array.

```
void SortL(CArrayFix<TSortPref>* aSortOrder)
```

This method sorts the database. After the method returns, the contact item identifiers can be accessed using the `SortedItemsL()` method. Usually, it will be better to use a view rather than sort the whole database.

`aSortOrder` – array of sort preferences. The database is sorted by the first sort preference. Any identical matches are then sorted by the next sort preference and so on. If the array is of zero length then the database is not sorted.

```
const CContactIdArray* SortedItemsL()
```

This method returns an array of contact items sorted by `SortL()`. The caller does not take ownership of the array. The array remains valid only until the database is changed or until the database's Active Object runs. If the caller wants the array after this then they need to take a copy of the array.

```
void SetDbViewContactType(const TUid aUid)
```

This method sets the type of contact items to be included in sorted views of the database. This value is initialized to `KUidContactItem` when the database is opened. This means that all `CContactItem` -derived types (cards, nonsystem templates, groups, own cards) are included in database views.

`aUid` – specifies a contact type which should be one of the following: `KUidContactCard` (contact cards), `KUidContactGroup` (contact item groups), `KUidContactOwnCard` (own cards), `KUidContactCardTemplate` (templates which are not system templates, i.e. which have been added to the database), or `KUidContactItem` (all of the above).

Contact Item Methods

```
CContactItem* ReadMinimalContactL(TContactItemId aContactId)
CContactItem* ReadMinimalContactLC(TContactItemId aContactId)
```

These methods provide read-only access to a contact item accessed by contact item identifier. They are faster than the `ReadContactL()` methods because they do not read template and group information.

`aContactId` – the contact item identifier.

returns – a `CContactItem`. the caller takes ownership of the item.

```
CContactItem* ReadContactL(TContactItemId aContactId)
CContactItem* ReadContactLC(TContactItemId aContactId)
```

These methods provide read-only access to a contact item accessed by contact item identifier.

`aContactId` – the contact item identifier.

returns – a `CContactItem`. The caller takes ownership of the item.

```
CContactItem* ReadContactL(TContactItemId aContactId,const
CContactItemViewDef& aViewDef)
CContactItem* ReadContactLC(TContactItemId aContactId,const
CContactItemViewDef& aViewDef)
```

These methods provide read-only access to selected fields of a contact item accessed by contact item identifier.

`aContactId` – the contact item identifier.

`aViewDef` – the contact item fields to be included in the item.

returns – a `CContactItem`. The caller takes ownership of the item.

```
CContactItem* OpenContactL(TContactItemId aContactId)
CContactItem* OpenContactLX(TContactItemId aContactId)
```

These methods open a contact for editing. The item remains open until `CommitContactL()` or `CloseContactL()` is called. The `LX` version of the method leaves the lock record on the cleanup stack.

`aContactId` – the contact item identifier.

returns – a `CContactItem`. The caller takes ownership of the item.

```
CContactItem* OpenContactL(TContactItemId aContactId,const
CContactItemViewDef& aViewDef)
CContactItem* OpenContactLX(TContactItemId aContactId,const
CContactItemViewDef& aViewDef)
```

These methods open a contact for editing while accessing only selected fields. Care is necessary because when the contact is committed any fields not loaded by the field view will be deleted from the item. The item remains open until `CommitContactL()` or `CloseContactL()` is called. The `LX` version of the method leaves the lock record on the cleanup stack.

`aContactId` – the contact item identifier.

returns – a `CContactItem`. The caller takes ownership of the item.

```
void CloseContactL(TContactItemId aContactId)
```

This method closes a contact item without committing any changes. Closing a contact that is not open or does not exist does not cause an error.

`aContactId` – the item identifier of the contact to be closed.

`void CommitContactL(const CContactItem& aContact)`
This method overwrites a locked contact item opened for editing.
`aContact` – the contact to be committed.

`void DeleteContactL(TContactItemId aContactId)`
`void DeleteContactsL(const CContactIdArray& aContactIds)`
These methods delete one contact or a set of contacts. `DeleteContactsL()` should not be used for single contacts and is most efficient for 16 contacts or more. Contacts may be deleted in a different order from the array of IDs.
`aContactId` or `aContactIds` – the contact item identifier(s) to be deleted.

Generic Contact Card Methods

`TInt CountL()`
This method returns the number of items in the database, excluding system templates and deleted items.

`TContactItemId AddNewContactL(CContactItem& aContact)`
This method adds a new contact item to the database.
`aContact` – the contact item to be added.

Group Methods

`inline TInt GroupCount() const`
This method returns the number of contact group items in the database.

```
CContactItem* CreateContactGroupL(TBool aInTransaction =EFalse)
CContactItem* CreateContactGroupLC(TBool aInTransaction =EFalse)
CContactItem* CreateContactGroupL(const TDesC& aGroupLabel,
              TBool aInTransaction =EFalse)
CContactItem* CreateContactGroupLC(const TDesC& aGroupLabel,
              TBool aInTransaction =EFalse)
```
These methods create a new, empty contact group and add it to the database. If the group label is not specified then a default label 'Group Label' is used.
`aGroupLabel` – the label to be used for the group.
`aInTransaction` – this argument should be ignored.
returns – a new contact group item. The caller takes ownership of the item.

`CContactIdArray* GetGroupIdListL() const`
This method returns an array containing the identifiers of all group items in the database. The caller takes ownership of the array. If there are no groups in the database then a NULL pointer is returned rather than a zero-length array.

```
void AddContactToGroupL(TContactItemId aItemId,
     TContactItemId aGroupId)
void AddContactToGroupL(CContactItem& aItem, CContactItem& aGroup)
void AddContactToGroupL(TContactItemId aItemId,
     TContactItemId aGroupId,TBool aInTransaction)
```

These methods add a contact item to a group.
`aItem` or `aItemId` – the contact item to be added to the group.
`aGroup` or `aGroupId` – the group to receive the item.
`aInTransaction` – this argument should be ignored.

```
void RemoveContactFromGroupL(CContactItem& aItem,
                             CContactItem& aGroup)
void RemoveContactFromGroupL(TContactItemId aItemId,
                             TContactItemId aGroupId)
```

These methods remove a contact item from a group.
`aItem` or `aItemId` – the contact item to be removed from the group.
`aGroup` or `aGroupId` – the group to lose the item.

Own Card Methods

```
CContactItem* CreateOwnCardLC()
CContactItem* CreateOwnCardL()
```

These methods create a new contact card and make it the own card.

```
TContactItemId OwnCardId() const
```

This method returns the identifier of the own card – returns `KNullContactId` if no own card has been set.

```
void SetOwnCardL(const CContactItem& aContact)
```

This method sets a contact item which already exists to be the own card. The item can be of any contact item type.
`aContact` – the contact item to be set as the own card.

Template Methods

```
inline TInt TemplateCount() const
```

This method returns the number of template items, not including the system template, in the database.

```
CContactIdArray* GetCardTemplateIdListL() const
```

This method returns an array containing the identifiers of all template items (except for the system template) in the database. The caller takes ownership of the array. If there are no nonsystem templates in the database then a NULL pointer is returned rather than a zero-length array.

```
CContactItem* CreateContactCardTemplateL(const TDesC&
aTemplateLabel,TBool aInTransaction =EFalse)
CContactItem* CreateContactCardTemplateLC(const TDesC&
aTemplateLabel,TBool aInTransaction =EFalse)
CContactItem* CreateContactCardTemplateL(const CContactItem*
aTemplate, const TDesC& aTemplateLabel,TBool aInTransaction
=EFalse)
CContactItem* CreateContactCardTemplateLC(const CContactItem*
aTemplate, const TDesC& aTemplateLabel,TBool aInTransaction
=EFalse)
```

These methods create a new template. If the template is created based on an existing contact item then the fields are based on that. Otherwise the fields are based on the system template. The fields are always empty.

`aTemplate` – a contact item on which to base the fields.
`aTemplateLabel` – the label for the template.
`aInTransaction` – this argument should be ignored.

Class `TSortPref`
Defined in `cntdb.h`

This class specifies a field to sort the Contact Database by.

Constructors

```
inline TSortPref()
inline TSortPref(TFieldType aFieldType,TOrder aOrder=EAsc)
```

The default constructor has the order initialized to `EAsc` and the field type to `KNullUid`.

`aFieldType` – the field type for sorting.
`aOrder` – specifies whether the sort is to be ascending or descending.

Member Variables

`TOrder iOrder` – specifies whether the sort is to be ascending (`EAsc`) or descending (`EDesc`).

`TFieldType iFieldType` – the first field matching this type will be sorted on.

11.6.2 Contact Item Classes

Class `CContactItem`
Defined in `cntitem.h`

The abstract base class for contact cards, templates and groups. All contact items are identified by a contact identifier (`TContactItemId`), have a last modified date/time, and own one or more fields (the field set). Contact items also have an access count and attributes (e.g. hidden). Note that fields in a contact item also have attributes. Attribute values specified in the contact item override those in the contained fields. The access count is a record of the number of objects referencing a contact item. A contact item cannot be fully deleted until its access count is zero.

Attribute Methods
`virtual TUid Type() const` This virtual method returns the type of the contact item. This can be one of the following: • `KUidContactCard` for a contact card • `KUidContactGroup` for a contact group • `KUidContactTemplate` for the system template • `KUidContactCardTemplate` for a template • `KUidContactOwnCard` for the own card
`TContactItemId Id() const` This method returns the contact item identifier.
`TTime LastModified() const` This method returns the date/time when the contact item was last modified.
`void SetLastModified(const TTime& aLastModified)` This method sets the date/time when the contact item was last modified.
`TBool IsHidden()` This method returns the item's hidden attribute. Hidden means that the item is not displayed if the view definition excludes hidden fields.
`void SetHidden(TBool aHidden)` This method sets the item's hidden attribute. Hidden means that the item is not displayed if the view definition excludes hidden fields. `aHidden` – `ETrue` if the item is to be hidden.
`inline TBool IsDeletable()` This method tests whether the contact item is deletable. This is true if the contact item's access count is zero.
`TBool IsDeleted() const` This method returns the deleted attribute. If the attribute is set, this means that an attempt has been made to delete the contact item, but because the item's access count is greater than zero, its data persists and the item is just marked as deleted.
`void SetDeleted(TBool aDeleted)` Sets the value of the contact item's deleted attribute. If the attribute is set, this means that an attempt has been made to delete the contact item, but because the item's access count is greater than zero, its data persists and the item is just marked as deleted. `aDeleted` – `ETrue` if the item is set as deleted.

`TContactItemId TemplateRefId() const`
This method returns the ID of the template on which this item is based or `KNullContactId` if the item is not based on a template.

`void SetTemplateRefId(TContactItemId aUid)`
This method sets the template on which this contact item is based.
`aUid` – the identifier of the template.

`inline TInt AccessCount() const`
This method returns the item's current access count.

Field Methods

`CContactItemFieldSet& CardFields() const`
This method returns a reference to the item's field set.

`void AddFieldL(CContactItemField& aField)`
This method appends a new field to the end of the item's set of fields. The item takes ownership of the field.
`aField` – the field to be added to the field set.

`void InsertFieldL(CContactItemField& aField,TInt aFieldPos)`
This method inserts a new field into the item's set of fields at the specified position.
`aField` – the field to be inserted.
`aFieldPos` – the position at which to insert the field. A value of zero inserts the field at the start of the set. A value equal to or greater than the number of fields causes the field to be appended to the set.

`void RemoveField(TInt aFieldPos)`
This method removes a field from the item's set. This method generates a panic if the indexed field does not exist.
`aFieldPos` – the position in the field set of the field to be deleted.

`void UpdateFieldSet(CContactItemFieldSet* aNewFieldSet)`
This method replaces the item's complete set of fields with a new set.
`aNewFieldSet` – the new set of fields for the item.

Class `CContactItemPlusGroup` – public `CContactItem`
Defined in `cntitem.h`

This is the abstract base class for `CContactGroup`, `CContactCard` and `CContactOwnCard`.
The purpose of this class is to avoid duplication of group functionality in its derived classes.

Public Methods

`const CContactIdArray* GroupsJoined() const`
This method returns a pointer to an array containing the identifiers of the groups of which the item is a member. The method returns NULL if the item does not belong to any groups.

`CContactIdArray* GroupsJoinedLC() const`
This method returns an array containing the identifiers of the groups of which the item is a member. The method returns an empty array if the item does not belong to any groups. The caller takes ownership of the array.

Class `CContactCard` – public `CContactItemPlusGroup`
Defined in `cntitem.h`

This class is a concrete implementation of `CContactItemPlusGroup` that represents a contact card. Apart from implementing virtual functions, it only adds creation methods.

Public Methods

```
static CContactCard* NewL()
static CContactCard* NewLC()
```
These methods create a new contact card using the system template.

```
static CContactCard* NewL(const CContactItem *aTemplate)
static CContactCard* NewLC(const CContactItem *aTemplate)
```
These methods create a new contact card based on a template.
`aTemplate` – the template to be used to create the contact card.

Class `CContactOwnCard` – public `CContactItemPlusGroup`
Defined in `cntitem.h`

This class is a concrete implementation of `CContactItemPlusGroup` that represents a contact card which contains information about the device's owner. Apart from implementing virtual functions, it only adds creation methods.

Public Methods

```
static CContactOwnCard * NewL()
static CContactOwnCard * NewLC()
```
These methods create a new contact card using the system template.

`static CContactOwnCard * NewL(const CContactItem *aTemplate)`
`static CContactOwnCard * NewLC(const CContactItem *aTemplate)`
These methods create a new contact card based on a template.
`aTemplate` – the template to be used to create the contact card.

Class `CContactCardTemplate` – public `CContactItem`
Defined in `cntitem.h`

This class is a concrete implementation of `CContactItem` that represents a contact card template. In addition to implementing virtual functions, it implements methods to manage the template label.

Public Methods

`TPtrC GetTemplateLabelL()`
This method returns the label for a contact card template.

`void SetTemplateLabelL(const TDesC& aLabel)`
This method sets the label for a contact card template. The label is initialized when the template is created. The template label is stored in a text field in the template. This field has a unique content type mapping of `KUidContactFieldTemplateLabel`. By default, this field is the first field in the field set; it must not be moved from this position.

Class `CContactGroup` – public `CContactItemPlusGroup`
Defined in `cntitem.h`

This class is a concrete implementation of `CContactItemPlusGroup` that represents a contact card group. In addition to implementing virtual functions, it implements methods to manage the group label and access the list of included contact items.

Public Methods

`static CContactGroup* NewL()`
`static CContactGroup* NewLC()`
These methods create a new contact group.

Contained Items Methods

`const CContactIdArray* ItemsContained() const`
This method returns a constant list of the items contained by the group.

`CContactIdArray* ItemsContainedLC() const`
This method returns a list of the items contained by the group and leaves it on the cleanup stack.

`TBool ContainsItem(TContactItemId aContactId)`
This method checks whether a contact item is contained in the group.
- `aContactId` – the contact item identifier to check.
- returns – `ETrue` if the item is contained in the group.

Group Label Methods

`TBool HasItemLabelField()`
This method tests whether the group has a label field (of type `KUidContactFieldTemplateLabel`).

`TPtrC GetGroupLabelL()`
This method returns the group label (if any).

`void SetGroupLabelL(const TDesC& aLabel)`
This method sets the group label.
- `aLabel` – the new group label to set.

Class `CContactIdArray` – public `CBase`
Defined in `cntdef.h`

This class is an array of contact item IDs (`TContactItemIds`). It owns a `CArrayFixFlat<TContactItemId>`. It is commonly returned by various methods.

Creation Methods

`static CContactIdArray* NewL()`
`static CContactIdArray* NewLC()`
These methods create a new empty array.

`static CContactIdArray* NewL(const CContactIdArray* aArray)`
`static CContactIdArray* NewLC(const CContactIdArray* aArray)`
These methods create a new array by copying an existing one.

Array Methods

`inline TInt Count() const`
This method returns the number of elements in the array.

`inline const TContactItemId& operator[](TInt aIndex) const`
`inline TContactItemId& operator[](TInt aIndex)`

These operators allow indexed access to array elements. `aIndex` – the array index.
`TInt Find(TContactItemId aId) const` This method searches for a specific contact identifier in the array. `aId` – the contact identifier to search for. returns – the index of the contact if it is found, or `KErrNotFound` if not found.
`inline void Reset()` This method removes all contact identifiers from the array.
`void AddL(TContactItemId aId)` This method appends a contact identifier to the array. `aId` – the contact identifier to be added.
`inline void InsertL(TInt aIndex,TContactItemId aId)` This method inserts a contact identifier at a specific point in the array. If the index is not valid then a panic occurs. `aIndex` – the position at which the identifier should be inserted. `aId` – the contact identifier to be inserted.
`void MoveL(TInt aOldIndex,TInt aNewIndex)` This method moves an identifier from one location in the array to another. If either index is invalid then a panic occurs. `aOldIndex` – the old position. `aNewIndex` – the new position.

`inline void Remove(TInt aIndex)`
`inline void Remove(TInt aIndex,TInt aCount)`
These methods remove either a single entry or a block of entries. If the index or number of entries is invalid then the method panics.
`aIndex` – the index (or starting index) to delete.
`aCount` – the number of entries to delete.

`void ReverseOrder()` This method reverses the contents of the array.

11.6.3 Contact Field Classes

Class `CContactItemField` – public `CBase` From `cntfield.h`
This class represents a field in a contact item. Each field has a content type, attributes, a label and field data. Most of the attributes are specialized or are accessed by other routes and so are not covered here.

Creation Methods

```
static CContactItemField* NewLC()
```
This method creates a new field. Its storage type and field type are unset.

```
static CContactItemField* NewL(TStorageType aType)
static CContactItemField* NewLC(TStorageType aType)
```
These methods create a new field and set the storage type.
`aType` – the storage type for the new field.

```
static CContactItemField* NewL(TStorageType aType,
                               TFieldType aFieldType)
static CContactItemField* NewLC(TStorageType aType,
                                TFieldType aFieldType)
```
These methods create a new field and set the storage type and field type.
`aType` – the storage type for the new field.
`aFieldType` – the field type for the new field.

```
static CContactItemField* NewL(TStorageType aType,
                          const CContentType &aContentType)
static CContactItemField* NewLC(TStorageType aType,
                          const CContentType &aContentType)
```
These methods create a new field and set the storage type and content type.
`aType` – the storage type for the new field.
`aContentType` – the content type for the new field.

```
static CContactItemField* NewL(const CContactItemField
                               &aField)
static CContactItemField* NewLC(const CContactItemField
                               &aField)
```
These methods create a new field based on an existing one. All details are copied from the existing field.
`aField` – the contact field on which to base the new one.

Attribute Methods

```
TInt Id() const
```
This method returns the field's identifier.

```
inline TBool IsReadOnly() const
```
This method returns `ETrue` if the field is read-only.

```
inline TBool UserAddedField() const
```
This method returns `ETrue` if the field has the user-added attribute set.

Field Type Methods
`const CContentType &ContentType() const` This method returns the content type of the field.
`void AddFieldTypeL(TFieldType aFieldType)` This method appends a field type to the field's set. `aFieldType` – the field type to be added.
`void RemoveFieldType(TFieldType aFieldType)` This method removes a field type from the field's set. `aFieldType` – the field type to be removed.
Storage Methods
`TStorageType StorageType() const` This method returns the field's storage type. The caller should use this method to determine the type of the field's storage before accessing it using one of the type-specific methods.
`CContactTextField * TextStorage() const` This method returns a pointer to the field's storage as a `CContactTextField*`. If the field storage type is not `KStorageTypeText`, this function raises a panic.
`CContactDateField * DateTimeStorage() const` This method returns a pointer to the field's storage as a `CContactDateField*`. If the field storage type is not `KStorageTypeDateTime`, this function raises a panic.
`CContactStoreField * StoreStorage() const` This method returns a pointer to the field's storage as a `CContactStoreField*`. This indicates field data stored in a descriptor or descriptor array. If the field storage type is not `KStorageTypeStore`, this function raises a panic.
`void ResetStore()` This method resets the field storage. The field's store is deleted, then reallocated.
Label Methods
`TPtrC Label() const` This method returns the field label. If no label is set then its length is zero.
`TBool LabelUnspecified() const` This method returns `ETrue` if the label has been specified either in a template or directly.

`void SetLabelL(const TDesC& aLabel)`
This method sets the field label. The new label is set using `TDesC::AllocL` to copy the value from the supplied descriptor and so can leave.
`aLabel` – the new label for the field.

`void SetLabel(HBufC* aLabel)`
This method sets the field label. The field takes ownership of the `HBufC` and so the method cannot leave.
`aLabel` – the new label for the field.

`static TBool IsValidLabel(const TDesC& aLabel,TInt& aInvalidPos)`
This method tests whether a field label is valid. The label is invalid if it contains any of the following characters:
[] (left or right square bracket)
= (equals sign)
. (dot)
: (colon)
, (comma)
`aLabel` – the label text to be checked.
`aInvalidPos` – the position in the label that contains the first invalid character (if any).
returns – `ETrue` if the label contains any invalid characters.

Class `CContentType` – public `CBase`
From `cntfield.h`

The `CContentType` class represents the content type for a contact item field. It is a set of field types, as each field can represent or be mapped to more than one field type. `CContentType` owns an array of `TUid` values for the field types. The UIDs are defined in `cntdef.h`. Each field is uniquely identified by the combination of UIDs contained in the content type. In addition to the field types, it is possible for a field to have an extra UID, which is the vCard mapping that identifies the vCard property to which the field maps.

Creation Methods

`static CContentType* NewL()`
This method creates a default content type. It has no field types and the mapping is set to `KNullUid`.

```
static CContentType* NewL(TFieldType aFieldType,
                          TUid aMapping=KNullUid)
```

This method creates a new content type with an initial field type and, optionally, a mapping. `aFieldType` – the initial field type for the new `CContentType`. `aMapping` – the field mapping.
`static CContentType* NewL(const CContentType &aContentType)` This method creates a new `CContentType` as a copy of an existing one. `aContentType` – the existing value to copy.
Field Type Methods
`TInt FieldTypeCount() const` This method returns the number of field types owned by the `CContentType`.
`TFieldType FieldType(TInt aIndex) const` This method returns the indexed field type. `aIndex` – the position of the field type required. If this index is negative or greater than or equal to the number of field types then the method raises a panic.
`TBool ContainsFieldType(TFieldType aFieldType) const` This method returns `ETrue` if the `CContentType` owns the specified field type, either as a field type or as the mapping. `aFieldType` – the field type being checked for.
`TBool operator ==(const CContentType &aType) const` This operator allows two `CContentType` objects to be compared. The two are considered to be equal if they have the same mapping and the same set of field types, although they do not need to be in the same order.
`void AddFieldTypeL(TFieldType aFieldType)` This method appends a field type to the set of field types owned by the `CContentType`. `aFieldType` – the field type to be added.
`void RemoveFieldType(TFieldType aFieldType)` This method removes a field type from the set owned by the `CContentType`. `aFieldType` – the field type to be removed.
Mapping Methods
`TUid Mapping() const` This method returns the vCard mapping for the `CContentType`.
`void SetMapping(TUid aMapping)` This method sets the vCard mapping for the `CContentType`. `aMapping` – the vCard mapping to be set.

typedef TUid TFieldType

TFieldType is a UID which identifies a contact item field's type. The possible values are defined as UIDs in cntdef.h.

Class CContactItemFieldDef – public CArrayFixFlat<TUid>
Defined in cntfield.h

This class represents an array of field types. It is used to specify a subset of fields when searching. CContactItemFieldDef is derived from CArrayFixFlat<TUid> and adds no methods apart from a constructor.

Constructor

```
inline CContactItemFieldDef() : CArrayFixFlat<TUid>(8)
```
This constructor creates the CContactItemFieldDef object, with an array granularity of 8.

Class TStorageType
Defined in cntdef.h

This class is a TUInt that represents the storage type of a field. The allowed values are defined in cntdef.hrh:

- KStorageTypeText (used by CContactTextFields)
- KStorageTypeStore (used by CContactStoreFields)
- KStorageTypeContactItemId (used by CContactAgentFields)
- KStorageTypeDateTime (used by CContactDateFields)

Note that numeric fields are not supported. Numbers (e.g. telephone numbers) are stored in the database using text fields.

Class CContactFieldStorage
Defined in cntfldst.h

This abstract class is the base class for the different types of field classes.

Member Methods

```
virtual TBool IsFull() const=0
```
This method returns ETrue if the field contains some data, EFalse otherwise.

Class `CContactTextField` – public `CContactFieldStorage`
Defined in `cntfldst.h`

This class provides access to a text storage field. It can be accessed by using the `TextStorage()` method of `CContactItem`.

Public Methods

`TPtrC Text() const`
This method returns a pointer to a copy of the text stored in the field.

`TPtrC StandardTextLC()`
This method takes a copy of the text stored in the field and converts it from Symbian editable text into standard text.

`void SetTextL(const TDesC& aText)`
This method creates a new descriptor (freeing any previously stored text) and copies the supplied text into it.
`aText` – the text to be copied into the field.

`void SetText(HBufC *aHbuf)`
This method takes ownership of the supplied descriptor as the contents of the text field.
`aHbuf` – the descriptor to replace any existing text.

`void SetTextArray(MDesCArray* anArray)`
This method sets the text which is stored in the field from a descriptor array. Each descriptor in the array is appended to the text field storage. They are separated by paragraph delimiters (`CEditableText::EParagraphDelimiter`). Any existing text is replaced.
`anArray` – an array of pointers to descriptors to use to set the field text.

`void SetStandardTextL(const TDesC& aText)`
This method converts a text string from plain text into Symbian editable text, and sets this as the text which is stored in the field.
`aText` – the text to store in the field.

`void SetStandardTextArray(MDesCArray* anArray)`
This method converts an array of text strings from plain text into Symbian editable text, appends them to a single descriptor, separating them with a new line character, and sets this as the text which is stored in the field. Any existing field text is replaced.
`anArray` – an array of pointers to descriptors to use to set the field text.

Class `CContactDateField` – public `CContactFieldStorage`
Defined in `cntfldst.h`

This class provides access to a date storage field. It can be accessed by using the `DateTimeStorage()` method of `CContactItem`.

Public Methods

`TTime Time() const`
This method returns the date/time value from the field.

`void SetTime(TTime aTime)`
This method sets the date/time stored in the field from a `TTime` value.
`aTime` – the new date/time value for the field.

`void SetTime(TDateTime aDateTime)`
This method sets the date/time stored in the field from a `TDateTime` value.
`aDateTime` – the new date/time value for the field.

Class `CContactStoreField` – public `CContactFieldStorage`
Defined in `cntfldst.h`

This class provides access to a binary data storage field. It can be accessed by using the `StoreStorage()` method of `CContactItem`.

Public Methods

`void SetThingL(const TDesC8& aDes)`
This method sets the store field contents.
`aDes` – the data to be stored.

`HBufC8* Thing() const`
This method returns a pointer to the binary data stored in the field.

11.6.4 Contact Views

Class `CContactItemViewDef`
Defined in `cntdb.h`

This class defines a view definition. A view definition can be used when reading or opening a contact item. A view definition includes a set of field types, a use flag that indicates whether the specified field types are to be included in or excluded from contact items, and a mode that indicates whether fields with the hidden attribute should be included or excluded.

Creation Methods

`static CContactItemViewDef* NewL(TUse aUse, TMode aMode)`
`static CContactItemViewDef* NewLC(TUse aUse, TMode aMode)`
These methods create a new view definition with the specified use and mode.
`aUse` – the value of the use attribute.
`aMode` – the value of the mode attribute.

Field Type Methods
`inline TInt Count() const` This method returns the number of field types in the view definition.
`inline TUid operator[](TInt aIndex) const` This method accesses the specified field from the view definition. `aIndex` – the array index for the required field type. If this value is negative or greater than or equal to the number of entries then this method will cause a panic.
`TInt Find(TFieldType aFieldType) const` This method searches the set of field types for one that matches the specified field type. `aFieldType` – the field type being searched for. returns – the index of the field type if found, otherwise `KErrNotFound`.
`TInt Find(const CContentType &aContentType) const` This method searches the set of field types for any field type that matches the content type. `aContentType` – the content type containing the set of field types being searched for. returns – the index of the first matching field type if found, otherwise `KErrNotFound`.
`TBool MatchesAll() const` This method tests whether the view definition contains a field type with the value `KUidContactFieldMatchAll`. If this is the case, all fields in the contact item are retrieved, regardless of the other field types specified in the view definition.
`inline void Reset()` This method deletes all field types from the view definition's set of field types.
`void AddL(TFieldType aFieldType)` This method appends a field type to the view definition's set of field types. `aFieldType` – the new field type to be added to the set.

<table>
<tr><td><code>void Remove(TFieldType aFieldType)</code>
<code>void Remove(TInt aIndex)</code>
These methods delete an entry from the set of field types, either by index position or by field type. If the specified entry is not present in the set of field types then these methods cause a panic.
  <code>aFieldType</code> – the field type to be removed from the set.
  <code>aIndex</code> – the position of the field type to be removed from the set.</td></tr>
<tr><td>Use and Mode Methods</td></tr>
<tr><td><code>inline TUse Use() const</code>
This method returns the view definition's use attribute. This will be either <code>EIncludeFields</code> (where fields in the view are included) or <code>EMaskFields</code> (where fields in the view are excluded).</td></tr>
<tr><td><code>inline void SetUse(TUse aUse)</code>
This method sets the view definition's use attribute.
  <code>aUse</code> – the use attribute to set.</td></tr>
<tr><td><code>inline TMode Mode() const</code>
This method returns the view definition's mode. This will be either <code>EIncludeHiddenFields</code> (in which case hidden fields will be included in the view) or <code>EMaskHiddenFields</code> (in which case hidden fields will be excluded).</td></tr>
<tr><td><code>inline void SetMode(TMode aMode)</code>
This method sets the view definition's mode attribute.
  <code>aMode</code> – the mode attribute to set.</td></tr>
<tr><td>Matching Methods</td></tr>
<tr><td><code>TBool Contains(const CContactItem& aItem)</code>
This method tests whether a contact item will be included in the view, i.e. whether any of its fields match those in the view definition (taking account of the use and mode attributes).
  <code>aItem</code> – the contact item to be tested.
  returns – <code>ETrue</code> if the contact item would be included in the view.</td></tr>
</table>

<table>
<tr><td>Class <code>RContactViewSortOrder</code>
Defined in <code>cntviewbase.h</code></td></tr>
<tr><td>This class specifies the sort order for a contact view. It is implemented as an array of <code>TFieldType</code> UIDs, which define the fields whose contents are used to sort on, and their order.</td></tr>
</table>

Creation and Destruction Methods
`RContactViewSortOrder()` This is the default constructor that creates an empty sort order.
`void CopyL(const RContactViewSortOrder& aSortOrder)` This method discards any existing field type data and replaces it with that from the specified sort order. `aSortOrder` – the sort order whose field types are to be used.
`void Close()` This method closes the sort order and frees up all space allocated to it.
Field Type Methods
`inline TInt Count() const` This method returns the number of field types used by the sort order.
`inline void AppendL(TFieldType aField)` This method appends a field type to the set owned by the sort order. `aField` – the field type to be appended.
`inline TFieldType operator[](TInt aIndex) const` This method returns the field type at the specified index. `aIndex` – the desired index.

Enumerated Type `TContactViewPreferences` This is used to specify the types of contact item that should be sorted and included in the view, and the behavior for items that cannot be sorted because they do not have content in any of the fields specified in the view's sort order. The default behavior is to include contact cards only and to sort contact cards without content in any of the sort order fields using the first available field containing any text.
• `EContactsOnly` (= `0×00000000`) – Only contact cards (of type `KUidContactCard` or `KUidContactOwnCard`) are included in the view. This is the default. • `EGroupsOnly` (= `0×00000001`) – Only contact groups (of type `KUidContactGroup`) are included in the view. • `EContactAndGroups` (= `0×00000002`) – Contact groups and contact cards (of type `KUidContactGroup`, `KUidContactCard` or `KUidContactOwnCard`) are included in the view. • `EIgnoreUnSorted` (= `0×00000004`) – Excludes contact items from the view which don't have content in any of the fields specified in the sort order.

- `EUnSortedAtBeginning` (= `0×00000008`) – Includes contacts in the view which don't have content in any of the fields specified in the sort order. These contacts are placed in an unsorted contact list which is located before the sorted list.
- `EUnSortedAtEnd` (= `0×00000010`) – Includes contacts in the view which don't have content in any of the fields specified in the sort order. These contacts are placed in an unsorted contact list which is located after the sorted list.
- `ESingleWhiteSpaceIsEmptyField` (= `0×00000020`) – Fields containing a single white space only are treated as empty, and therefore unsortable.
- `EICCEntriesOnly` (= `0×00000040`) – Only ICC entries (of type `KUidContactICCEntry`) are included in the view.
- `EICCEntriesAndContacts` (= `0×00000080`) – Only contact cards and ICC entries (of type `KUidContactCard`, `KUidContactOwnCard` or `KUidContactICCEntry`) are included in the view.

Enumerated Type `TContactViewEvent::TEventType`
This is used to specify the type of event to a view observer. Some of these event types cannot apply to a local view. Some of the event types indicate that further information is available in the relevant `TContactViewEvent` object.

- `EUnavailable` – The observed view's state has changed from `EReady` to either `ENotReady` or `EInitializing`, so is not available for use.
- `EReady` – The observed view's state has changed from `ENotReady` or `EInitializing` to `EReady` so is available for use.
- `ESortOrderChanged` – The observed view's sort order has changed, so observer views need to update themselves.
- `ESortError` – An error occurred when sorting the observed view or when appending an observer to its observer array. The error code is provided in `iInt`.
- `EServerError` – An error occurred in the contacts server. The error code is provided in `iInt`.
- `EIndexingError` – An error occurred when setting the range for a `CContactSubView`. The error code is provided in `iInt`.
- `EItemAdded` – An item has been added to the observed view. The identifier of the added item is provided in `iContactId`, and the index into the underlying view of the added item is provided in `iInt`.
- `EItemRemoved` – An item has been removed from the observed view. The identifier of the removed item is provided in `iContactId`, and the index into the underlying view of the item is provided in `iInt`.
- `EGroupChanged` – A change has occurred in a contact group, for instance a contact item has been moved into or out of the group. The identifier of the group affected is provided in `iContactId`.

Class `TContactViewEvent`
This class contains information on a contact view event. It is passed to a contact view observer – see `MContactViewObserver::HandleContactViewEvent()`.
Member Variables
`TEventType iEventType` – the type of event.
`TInt iInt` – the error code (where relevant) or the index of the contact item added to or removed from the underlying view.
`TContactItemId iContactId` – the identifier of the contact item or group that has been added to or removed from the view.

Class `MContactViewObserver` Defined in `cntviewbase.h`
This mixin has a single method that is called when events occur on a contact view.
`virtual void HandleContactViewEvent(const CContactViewBase& aView, const TContactViewEvent& aEvent) = 0` This method is called when an event occurs on the view that is being observed. `aView` – the view on which the event has occurred (the observer could be observing multiple views). `aEvent` – the event that has occurred.

Class `CContactViewBase` Defined in `cntviewbase.h`
This class is the base class for contact views. It includes the `Close()` method that must be used before a view is deleted.
Member Methods
`TBool Close(const MContactViewObserver& aObserver)` This method removes an observer from the view's set of observers. Any outstanding notifications for the observer are first canceled. `aObserver` – the observer to be removed. returns – `ETrue` if the view's set of observers is now empty, `EFalse` if the observer is not in the view's set of observers or if the set is not empty.

Class `CContactLocalView` – public `CContactViewBase`, public `MContactDbObserver`
Defined in `cntviewbase.h`

This class holds a view for one client. The data associated with a local view is allocated within the client's memory space; `CContactRemoteView` should be used in preference if the view is likely to be shared between multiple clients. It is kept up to date by receiving change events from the underlying `CContactDatabase` object which it observes. The view preferences and sort order are specified on construction.

Construction Method

```
static CContactLocalView* NewL(MContactViewObserver&
                                aObserver,
  const CContactDatabase& aDb,
  const RContactViewSortOrder& aSortOrder,
  TContactViewPreferences aContactTypes)
```

This method creates a view on a contact database. The view is created asynchronously and is not ready until the observer has reported.

- `aObserver` – an observer that receives notifications when this view is ready for use and when changes take place in it. The observer receives a `TContactViewEvent::EReady` event when the view is ready. An attempt to use the view before this notification causes a panic.
- `aDb` – the underlying database that contains the contact items. The view observes the database, so that it handles change events sent from the database.
- `aSortOrder` – specifies the fields to use to sort the items in the view.
- `aContactTypes` – specifies which types of contact items should be included in the view and the behavior for items that do not have content in any of the fields specified in the sort order.

Properties Methods

```
const RContactViewSortOrder& SortOrder() const
const RContactViewSortOrder& SortOrderL() const
```

These methods provide access to the view's sort order, as set during construction. Contrary to the name, `SortOrderL()` cannot leave.

```
TContactViewPreferences ContactViewPreferences()
```

This method returns the contact view preferences as set during construction.

Contact Item Access Methods

```
TInt CountL() const
```

This method returns the number of contact items in the view.

`TContactItemId AtL(TInt aIndex) const` This method returns the identifier of the contact item at the specified position in the view's array of contact items. `aIndex` – the desired position.
`const CViewContact& ContactAtL(TInt aIndex) const` This method returns data on the contact item at the specified position in the view's array of contact items. `aIndex` – the desired position.
`TInt FindL(TContactItemId aId) const` This method searches for the contact item with the specified identifier in the view's array of contact items. `aId` – the identifier of the desired contact item. returns – the index of the contact item in the view's array of contact items, or `KErrNotFound`.

11.7 A Contacts Connectivity Service

We now need to plan the functions that we want our Contacts connectivity service to support. We are not trying to compete directly with PIM synchronization software; what we aim to do is to provide a window on the Contacts data from the PC.

This code builds on custom servers or socket servers and the routines to pack or unpack data used in previous chapters. This code includes some quite sophisticated possibilities, even though we actually only use a subset of these when we build a GUI PC application.

11.7.1 Protocol Description

Some of the functions that we are going to require are relatively obvious. We will want to fetch all contacts from the mobile phone for display on the PC, we will want to edit or delete existing contacts and to create new contacts, and we will want to manage groups. In addition, there are some functions that we will find valuable that were not obvious at first. You can try to brainstorm these functions, but I have found that prototyping works well. When I started to create the software presented later in this chapter, I had not thought of some of these functions, but they soon made their value clear. Therefore, we need to be open to changes, but remember to document them!

The extra features were mostly optimizations to avoid having to copy all the contacts data to the PC in order to be able to display it. If the user has a relatively small address book then this may be viable, but

a corporate user may have several hundred addresses or even more. If our application spends several minutes transferring contacts whenever it starts up then it will not be well received. If we consider that we could transfer only a subset of contacts data (first and last name, for example) initially and then retrieve full information when required, we can see that the ability to filter the fields returned will be useful. We can then also add access to the search functions. If we were going to transfer all the data to the PC then the search could be more effectively carried out on the PC.

I have chosen not to edit or create new templates, but simply to use the system template. Similarly, I have not created new fields. If you are creating a specialized application then you can add these features.

This then allows us to propose a set of commands for our protocol and a set of responses. The following defines the message operation codes.

```
enum TRCntCmdCode
  {
  ERCntCmdNone = 100,      // Reserved for internal usage
  ERCntCmdQueryVersion = 101, // Query version
  ERCntCmdVersionReply = 102, // Version reply
  ERCntCmdOpenDatabase = 103, // Open Contacts Database
  ERCntCmdOpenDatabaseReply = 104, // Open database reply

  ERCntCmdFetchAllContacts = 111, // Request for all contacts
  ERCntCmdFindContacts = 112, // Request to find contacts
  ERCntCmdFetchMoreContacts = 113, // Request for more contacts
  ERCntCmdContactsReply = 114, // Reply containing contacts
  ERCntCmdFetchContactsCount = 115, // How many contacts in the db
  ERCntCmdContactsCountReply = 116, // Reply containing number of
                                    // contacts
  ERCntCmdFetchContactSet = 117, // fetch a set of contacts by ID

  ERCntCmdFetchAllGroups = 121, // Request for all groups
  ERCntCmdFetchMoreGroups = 122, // Request for more groups
  ERCntCmdGroupsReply = 123, // Reply containing groups
  ERCntCmdFetchContactsInGroup = 124, // Fetch list of contacts
                                      // in a group
  ERCntCmdContactsInGroupReply = 125, // Reply containing contacts
                                      // in a group

  ERCntCmdDeleteContact = 131, // Delete a contact
  ERCntCmdDeleteGroup = 132, // Delete a group
  ERCntCmdDeleteReply = 133, // Reply after deletion

  ERCntCmdFetchTemplateFieldInfo = 140, // Fetch default template data
  ERCntCmdTemplateFieldInfoReply = 141, // Data on default template
                                        // fields
  ERCntCmdFetchOwnCardId = 142, // Fetch the ID of the owners card
  ERCntCmdOwnCardIdReply = 143, // Reply with the own card ID
  ERCntCmdSetOwnCardId = 144, // Set ID of own card
  ERCntCmdSetOwnCardIdReply = 145, // Reply to set own card ID

  ERCntCmdCreateContact = 150, // Create a new contact card
  ERCntCmdCreateContactReply = 151, // Reply after contact creation
  ERCntCmdCreateGroup = 152, // Create a new contact group
```

```
ERCntCmdCreateGroupReply = 153, // Reply after group creation

ERCntCmdEditContact = 160, // Edit a contact card data
ERCntCmdEditContactReply = 161, // Reply after editing a contact card
ERCntCmdEditGroup = 162, // Edit a contact group
ERCntCmdEditGroupReply = 163, // Reply after editing a group
ERCntCmdEditContactInGroup = 164, // Add to or delete a card
                                  // from a group
ERCntCmdEditContactInGroupReply = 165, // Reply after editing a card
                                       // in a group

ERCntCmdError = 170 // An error has occurred
};
```

As with the SMS service, some of the commands require no specific parameters. Requesting all contacts or groups, or remaining contacts or groups, are complete commands without any further information.

Other commands require a single identifier (a 32-bit integer) or a set of identifiers. Deleting a contact or group, or requesting data on a contact or set of contacts or a group, requires just a sequence of 32-bit identifiers. Where we allow a set of values, we will normally use −1 as a terminating 'magic number'.

Creating a view requires resources, so we will not support commands to make frequent changes. Instead, we provide a command to open the Contact Database and create a view at the same time. If we want to change the view then we will need to close the connection and reopen the database. The command to open the Contact Database includes a list of field types to be used to set up the view; these field types will control the sorting order of the view.

The other commands that require additional data are those to create or edit a contact (this needs to include the field data) or group (this needs to include the group label). For contact creation and editing, we support a sequence of field identifiers followed by text data. I have avoided fields with other types of data, both for the sake of simplicity (specialized types need specialized handling) and because I believe that text data covers 99% of normal usage. However, if you want to handle binary store fields, such as those that store photographs, then you would need to extend the protocol.

The responses follow a similar pattern. Some of the responses need no data beyond their reply code. The delete or edit responses fall into this category. Other responses include one or more 32-bit identifiers, such as the reply that lists the contacts in a group. The more complex responses include those that list groups (an identifier and a label) and contacts (an identifier and a sequence of fields). Each field definition includes the field identifier, the field types, the storage type (even though we support only text in this case I made some allowance for expansion) and the actual field data.

Whenever we retrieve field data we provide a flag to select whether we return all fields and label data or just a subset. This allows us to retrieve bulk data rapidly or the whole of a smaller number of Contact items.

Field types are handled as a sequence of field type identifiers terminated with a −1 value and then a single mapping identifier.

Query Version Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdQueryVersion (=101)`
Transaction ID	Int32	PDU transaction identifier

Version Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCmdVersionReply (=102)`
Transaction ID	Int32	PDU transaction identifier
Major Version	Int32	Major version number
Minor Version	Int32	Minor version number
Build Number	Int32	Build number

Open Database Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdOpenDatabase (=103)`
Transaction ID	Int32	PDU transaction identifier
Field Type ID This field may be repeated	Int32	Field type identifier

...

Terminator	Int32	Terminating value (=`-1`)

Open Database Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCmdOpenDatabaseReply (=104)`
Transaction ID	Int32	PDU transaction identifier

Fetch All Contacts Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchAllContacts (=111)`
Transaction ID	Int32	PDU transaction identifier
Fetch all fields flag	Int8	If non-zero then return all fields and labels

Fetch More Contacts Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchMoreContacts (=113)`
Transaction ID	Int32	PDU transaction identifier
Fetch all fields flag	Int8	If non-zero then return all fields and labels

Find Contacts Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFindContacts (=112)`
Transaction ID	Int32	PDU transaction identifier
Search String Length	Int16	Length in characters of search string
Search String	Unicode data	Search string
Fetch all fields flag	Int8	If non-zero then return all fields and labels

Fetch Contacts Set Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchContactSet (=117)`
Transaction ID	Int32	PDU transaction identifier
Fetch all fields flag	Int8	If non-zero then return all fields and labels
Contact Identifier This field may be repeated	Int32	Index identifier of a required Contact Card

. . .

Terminator	Int32	Terminating value (=`-1`)

Contacts Reply

This message contains information about zero or more contact cards and has three degrees of nesting. The data for contact cards is repeated until it is terminated by data with a contact card identifier of −1. The data for each contact card contains data on zero or more fields. The data for fields is repeated until it is terminated by data with a field identifier of −1. Each field has one or more field types which are terminated by a field type of −1. The actual data for a field may be text or date/time data and is indicated by a field storage type.

Field	Type	Meaning
Opcode	Int32	`ERCntCmdContactsReply (=114)`
Transaction ID	Int32	PDU transaction identifier

. . .

Contact Card ID	Int32	The Contact Card Identifier

Field ID	Int32	Field identifier

Field Type	Int32	Field type identifier

. . .

Terminating Field Type	Int32	Terminating field type of −1
Mapping ID	Int32	Mapping identifier for the field

. . .

Field Storage Type	Int32	`KStorageTypeText` for a text field
Label Length	Int16	The number of characters in the label (may be zero if labels are not being returned)
Label Text	Unicode data	Label Text
Text Length	Int16	The number of characters in the field text
Field Text	Unicode data	Field Text

. . .

Terminating Field ID	Int32	Terminating field identifier of −1

. . .

Terminating Card ID	Int32	Identifier of −1 to indicate no more card data

Fetch Contacts Count Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchContactsCount (=115)`
Transaction ID	Int32	PDU transaction identifier

Contacts Count Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCmdContactsCountReply (=116)`
Transaction ID	Int32	PDU transaction identifier
Contacts Count	Int32	Count of Contact Cards in the database

Fetch All Groups Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchAllGroups (=121)`
Transaction ID	Int32	PDU transaction identifier

Fetch More Groups Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchMoreGroups (=122)`
Transaction ID	Int32	PDU transaction identifier

Groups Reply

This reply includes zero or more groups. Each group has its identifier and its label. The list of contacts in the group is not included in the reply. The list of groups is terminated by a group identifier of −1.

Field	Type	Meaning
Opcode	Int32	`ERCntCmdGroupsReply (=123)`
Transaction ID	Int32	PDU transaction identifier

. . .

Group ID	Int32	Group identifier
Label Length	Int16	Number of characters in the group label
Label Text	Unicode data	Group label text

. . .

Terminating Group ID	Int32	−1 indicating no more groups

Fetch Contacts in Group Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchContactsInGroup (=124)`
Transaction ID	Int32	PDU transaction identifier
Group ID	Int32	Group identifier

Fetch Contacts in Group Reply
This reply includes a list of Contact Items (not necessarily Contact Cards) in a group. the list of contact items is terminated by a value of −1.

Field	Type	Meaning
Opcode	Int32	`ERCntCmdContactsInGroupReply (=125)`
Transaction ID	Int32	PDU transaction identifier
Group ID	Int32	Identifier of the group concerned
...		
Contact ID	Int32	Identifier of a contact item in the group
...		
Terminating ID	Int32	−1 indicating no more contact items in the group

Delete Contact Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdDeleteContact (=131)`
Transaction ID	Int32	PDU transaction identifier
Contact ID	Int32	Identifier of the contact to delete

Delete Group Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdDeleteGroup (=132)`
Transaction ID	Int32	PDU transaction identifier
Group ID	Int32	Identifier of the group to delete

Delete Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCmdDeleteReply (=133)`
Transaction ID	Int32	PDU transaction identifier

Fetch Template Field Info Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchTemplateFieldInfo (=140)`
Transaction ID	Int32	PDU transaction identifier

Template Field Info Reply

This command contains the field information for the system template. Zero or more fields are listed and the list is terminated with a field identifier of −1. For each field the field types are listed and the list of field types is terminated with a value of −1. The list of field types is followed by the label text for that field.

Field	Type	Meaning
Opcode	Int32	`ERCntCmdTemplateFieldInfoReply (=141)`
Transaction ID	Int32	PDU transaction identifier
Field ID	Int32	Field identifier

. . .

Field Type	Int32	Field Type

. . .

Terminating Field Type	Int32	−1 indicating no more field types
Mapping ID	Int32	Mapping identifier for the field
Label Length	Int16	Number of characters in the field label
Label Text	Unicode data	Field label text

. . .

Terminating Field ID	Int32	−1 to indicate no more fields

Fetch Own Card ID Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdFetchOwncardId (=142)`
Transaction ID	Int32	PDU transaction identifier

Own Card ID Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCmdOwnCardIdReply (=143)`
Transaction ID	Int32	PDU transaction identifier
Own Card ID	Int32	Identifier of Own card

Set Own Card ID Command

Field	Type	Meaning
Opcode	Int32	`ERCntCmdSetOwnCardIdReply (=144)`
Transaction ID	Int32	PDU transaction identifier
Own Card ID	Int32	Identifier of Own card

Set Own Card ID Reply

Field	Type	Meaning
Opcode	Int32	`ERCntSetCmdOwnCardIdReply (=145)`
Transaction ID	Int32	PDU transaction identifier

Create Contact Command

This command contains a list of field types and text sets. Because each field is identified uniquely only by the full set of field types and the mapping, each field includes all of this data. The list is terminated by an initial field type of −1.

Field	Type	Meaning
Opcode	Int32	`ERCntCmdCreateContact (=150)`
Transaction ID	Int32	PDU transaction identifier

. . .

Field	Type	Meaning
Field Type	Int32	Field type to have text set

. . .

Field	Type	Meaning
Terminating Field Type	Int32	−1 indicating no more field types for this field
Mapping ID	Int32	Mapping identifier for the field
Field Text Length	Int16	Length in characters of field text
Field Text	Unicode data	Text to set for field

. . .

Field	Type	Meaning
Terminating Field Type	Int32	−1 indicating no more fields

Create Contact Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCreateContactReply (=151)`
Transaction ID	Int32	PDU transaction identifier
Contact ID	Int32	Identifier of newly created contact

Create Group Command
This command includes a list of contact items to include in the group. The list is terminated by a contact item identifier of −1.

Field	Type	Meaning
Opcode	Int32	`ERCntCreateGroup (=152)`
Transaction ID	Int32	PDU transaction identifier
Label Length	Int16	Length in characters of label text
Label Text	Unicode data	Text to set for label

. . .

Contact ID	Int32	Contact ID to include in the group

. . .

Terminating Contact ID	Int32	−1 indicating no more contact items

Create Group Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCreateGroupReply (=153)`
Transaction ID	Int32	PDU transaction identifier
Group ID	Int32	Identifier of newly created group

Edit Contact Command
This command contains a list of field type and text pairs. Because each field is identified uniquely only by the full set of field types and the mapping, each field includes all of this data. The list is terminated by an initial field type of −1.

Field	Type	Meaning
Opcode	Int32	`ERCntCmdEditContact (=160)`
Transaction ID	Int32	PDU transaction identifier
Contact ID	Int32	Identifier of contact to edit

. . .

Field Type	Int32	Field type to have text set

. . .

Terminating Field Type	Int32	−1 indicating no more field types for this field
Mapping ID	Int32	Mapping identifier for the field
Field Text Length	Int16	Length in characters of field text
Field Text	Unicode data	Text to set for field

. . .

Terminating Field Type	Int32	−1 indicating no more fields

Edit Contact Reply

Field	Type	Meaning
Opcode	Int32	`ERCntEditContactReply (=161)`
Transaction ID	Int32	PDU transaction identifier

Edit Group Command

Field	Type	Meaning
Opcode	Int32	`ERCntEditGroup (=162)`
Transaction ID	Int32	PDU transaction identifier
Group ID	Int32	Identifier of group to edit
Label Length	Int16	Length in characters of label text
Label Text	Unicode data	Text to set for label

Edit Group Reply

Field	Type	Meaning
Opcode	Int32	`ERCntEditGroupReply (=163)`
Transaction ID	Int32	PDU transaction identifier

Edit Contact in Group Command
This command has a list of contact items to be included in or removed from a group. The list is terminated by a value of −1.

Field	Type	Meaning
Opcode	Int32	`ERCntEditContactInGroup (=164)`
Transaction ID	Int32	PDU transaction identifier
Add Flag	Int32	If zero the contacts are removed, otherwise they are added
Group ID	Int32	Identifier of group to edit

. . .

Contact ID	Int32	Contact ID to include in or remove from the group

. . .

Terminating Contact ID	Int32	−1 indicating no more contact items

Edit Contact in Group Reply

Field	Type	Meaning
Opcode	Int32	`ERCntEditContactInGroupReply (=165)`
Transaction ID	Int32	PDU transaction identifier

Error Reply

Field	Type	Meaning
Opcode	Int32	`ERCntCmdError (=170)`
Transaction ID	Int32	PDU transaction identifier
Error Code	Int32	Symbian OS error code

11.7.2 Opening the Contacts Database

The code to open the Contacts database is very simple, but we will also read a sequence of field types to create a view through which we will access the Contact Model. We will also use the same set of fields when we are asked for only a subset of fields.

```
void CRCntUtil::ConstructL(TConnBuff& aBuffer)
  {
  iCdb = CContactDatabase::OpenL();

...
  iViewObs = new(ELeave) CRCntViewObs();

  // Set up a view definition
  // Start by reading field types - we will also
  // use them for return fields
  TInt fieldId = CConnPack::ReadInt32L(aBuffer);
  TFieldType fieldType;
  while(fieldId >= 0)
    {
    fieldType.iUid = fieldId;
    iSortOrder.AppendL(fieldType);
    fieldId = CConnPack::ReadInt32L(aBuffer);
    }

  iView = CContactLocalView::NewL(*iViewObs, *iCdb, iSortOrder,
          (TContactViewPreferences)(EContactsOnly | EUnSortedAtEnd) );
...
  }
```

11.7.3 Retrieving Contact Card Data

The first function we will implement is to retrieve data from Contact cards. If we want to only return a subset of fields then we use the stored list of field types.

Here is a basic pair of methods to return a card's identifier and all text fields; other types of fields we ignore. Time fields would be simple to return, and if we wanted to return store-based fields then we could simply embed the data with a preceding length. For each field we return the field identifier, field types, mapping, label and content. One optimization

here is that we will not return text fields that are empty or contain only a single space (because I found a lot of those).

If we are trying to save space by returning only some fields then we check for fields containing one of the field types that we set in the view sort order (this was a choice – we could have used a separate set of field types). For convenience, we return field labels only if we are returning all fields.

Note that we use the `ReadMinimalContact()` method to read a Contact card here, since it is the most efficient method and we do not need the additional data that it does not supply.

```
void CRCntUtil::GetOneCardL(TContactItemId aItemId,
                            TBool aFetchAllFields, TDes8 &aTempBuffer)
  {
    CContactItem* cItem = iCdb->ReadMinimalContactL(aItemId);
    CleanupStack::PushL(cItem);
    if(cItem->Type() != KUidContactCard)
      {
      CleanupStack::PopAndDestroy(cItem);
      return;
      }

    CConnPack::WriteInt32L(cItem->Id(), aTempBuffer);

    CContactItemFieldSet& cardFields = cItem->CardFields();
    for(TInt fi = 0 ; fi < cardFields.Count() ; ++fi)
      {
        TBool returnField = EFalse;
        if(aFetchAllFields)
          {
          returnField = ETrue;
          }
        else
          {
          for(TInt ti = 0 ; ti < iSortOrder.Count() ; ++ti)
            {
            if(cardFields[fi].ContentType().
               ContainsFieldType(iSortOrder[ti]))
              {
              returnField = ETrue;
              break;
              }
            }//endfor
          }//endif
        if(!returnField)
          {
          continue;
          }

      switch(cardFields[fi].StorageType())
        {
        case(KStorageTypeText):
          {
          TPtrC text = cardFields[fi].TextStorage()->Text();
          if((text.Length() > 1) || (text[0] != ' '))
```

```
          {
          GetFieldInfo(cardFields[fi], aFetchAllFields, aTempBuffer);
          CConnPack::WriteUNCDataL(text, aTempBuffer);
          }
        }
        break;

      default:
      // Unknown type or one that we choose not to handle
        break;
      }//endswitch
    }//endfor
  CConnPack::WriteInt32L(-1, aTempBuffer); // terminating field ID

  CleanupStack::PopAndDestroy(cItem);
  }

void CRCntUtil::GetFieldInfo(CContactItemField &aField,
                             TBool aFetchAllFields, TDes8 &aTempBuffer)
  {
  CConnPack::WriteInt32L(aField.Id(), aTempBuffer);

  TInt fieldTypeCount = aField.ContentType().FieldTypeCount();
  for(TInt fi = 0 ; fi < fieldTypeCount ; ++fi)
    {
    TInt typeUid = aField.ContentType().FieldType(fi).iUid;
    CConnPack::WriteInt32L(typeUid, aTempBuffer);
    }
  CConnPack::WriteInt32L(-1, aTempBuffer);
  // Append the content type mapping
  CConnPack::WriteInt32L(aField.ContentType().Mapping().iUid,
                         aTempBuffer);

  TStorageType fieldStorageType = aField.StorageType();
  CConnPack::WriteInt32L(fieldStorageType, aTempBuffer);
  if(aFetchAllFields)
    {
    TPtrC label = aField.Label();
    CConnPack::WriteUNCDataL(label, aTempBuffer);
    }
  else
    {
    CConnPack::WriteInt16L(0, aTempBuffer); // Zero length text
    }
  }
```

Having routines to return data for one card at a time, we can then provide the routines to return all cards or a subset. We may have more data than will fit in one buffer, so we need a separate command to retrieve remaining data and we use a temporary buffer (because we cannot predict in advance how large data for one Contact card will be). Because we have several methods of returning Contacts (all cards, find cards and a specific set), we use a common member array of card identifiers. Some methods allow require us to add elements to this array individually, but the `FindLC` method returns us a whole new array.

```
void CRCntUtil::GetAllCardsL(TDes8 &aBuffer, TBool aFetchAllFields,
                             TDes8 &aTempBuffer)
  {
  iCardIdArray->Reset();

  // Populate the card ID array from the view.
  // We could use the view directly but it might change between calls
  // from the client and, this way, we can use the same array for being
  // set up in other routines
  // Bear in mind that the view may not be ready, so check
  if(iViewObs->IsReady())
  {
    TInt count = iView->CountL();
    for(TInt i = 0 ; i < count ; ++i)
      {
      iCardIdArray->AddL(iView->AtL(i));
      }
  }

  iArrayIndex = 0;
  GetMoreCardsL(aBuffer, aFetchAllFields, aTempBuffer);
  }

void CRCntUtil::FindCardsL(TDes &aFindText, TBool aFetchAllFields,
                           TDes8 &aBuffer, TDes8 &aTempBuffer)
  {
  if(iCardIdArray != NULL)
    {
    delete iCardIdArray;
    iCardIdArray = NULL;
    }

  iCardIdArray = iCdb->FindLC(aFindText, iSearchFieldDef);
  CleanupStack::Pop(iCardIdArray); // member variable

  iArrayIndex = 0;
  GetMoreCardsL(aBuffer, aFetchAllFields, aTempBuffer);

  if(iArrayIndex == iCardIdArray->Count())
    {iCardIdArray->Reset();}
  }

void CRCntUtil::GetCardSetL(TConnBuff &aBuffer)
  {
  iCardIdArray->Reset();

  TInt cardId = CConnPack::ReadInt32L(aBuffer);
  while(cardId >= 0)
    {
    iCardIdArray->AddL(cardId);
    cardId = CConnPack::ReadInt32L(aBuffer);
    }

  iArrayIndex = 0;
  }

void CRCntUtil::GetMoreCardsL(TDes8 &aBuffer, TBool aFetchAllFields,
                              TDes8 &aTempBuffer)
```

```
 {
 TInt cardCount = iCardIdArray->Count();

 // Go through the set of cards until we finish or run out
 // of buffer space.  We may come around again if we run out
 // of buffer space
 while(iArrayIndex < cardCount)
   {
   TContactItemId cItemId = (*iCardIdArray)[iArrayIndex];
   aTempBuffer.Zero();
   GetOneCardL(cItemId, aFetchAllFields, aTempBuffer);
   if( aTempBuffer.Length() > 0)
     {
     // If the message will fit then append it, otherwise break out
     // Need to leave room for a terminating null-id as well
     if(aTempBuffer.Length()+aBuffer.Length()+4 < aBuffer.MaxLength())
       {
       CConnPack::WriteBufferL(aTempBuffer, aBuffer);

       ++iArrayIndex;
       }
     else
       {
       break;
       }
     }
   else
     {
     ++iArrayIndex;
     }//endif
   }//endwhile
 CConnPack::WriteInt32L(-1, aBuffer);
 }
```

11.7.4 Accessing the System Template

It is possible to use templates in a sophisticated way, with different templates for different purposes. However, most of the time this is unlikely to be necessary. Therefore, we will content ourselves with obtaining information about the system template. It is likely that most Contact cards will have been created based on the system template, so we can store its labels once and use them for all cards (thus, we can avoid retrieving the labels for each field of each card).

```
void CRCntUtil::GetTemplateInfoL( TDes8 &aBuffer)
  {
  CContactItemFieldSet &cardFields = iTemplate->CardFields();
  TInt fieldCount = cardFields.Count();
  for(TInt j = 0 ; j < fieldCount ; ++j)
    {
    CConnPack::WriteInt32L(cardFields[j].Id() , aBuffer);
    TInt fieldTypeCount = cardFields[j].ContentType().FieldTypeCount();
    for(TInt fi = 0 ; fi < fieldTypeCount ; ++fi)
      {
      CConnPack::WriteInt32L(cardFields[j].ContentType().
```

```
                              FieldType(fi).iUid , aBuffer);
      }
    CConnPack::WriteInt32L(-1, aBuffer);
    // Append the content type mapping
    CConnPack::WriteInt32L(cardFields[j].ContentType().Mapping().iUid,
                            aBuffer);
    TPtrC label = cardFields[j].Label();
    CConnPack::WriteUNCDataL(label , aBuffer);
    }
  CConnPack::WriteInt32L(-1, aBuffer);
  }
```

11.7.5 Getting and Setting Own Card

The own card is a useful concept as a means of identifying a phone and for ease of transferring to another phone. The own card is just another Contact card and the methods to set and retrieve it are straightforward:

```
void CRCntUtil::FetchOwnCardIdL(TDes8 &aBuffer)
  {
  TInt ownCardId = iCdb->OwnCardId();
  CConnPack::WriteInt32L(ownCardId, aBuffer);
  }

void CRCntUtil::SetOwnCardIdL(TDes8 &aBuffer)
  {
  TInt ownCardId = CConnPack::ReadInt32L(aBuffer);

  CContactItem *item = iCdb->ReadContactLC(ownCardId);
  iCdb->SetOwnCardL(*item);
  CleanupStack::PopAndDestroy(item);
  }
```

11.7.6 Deleting, Adding and Editing Cards

Deleting a Contact card is straightforward, given the card's identifier:

```
void CRCntUtil::DeleteCardL(TInt aCardId)
  {
  iCdb->DeleteContactL(aCardId);
  }
```

Creating a new Contact card or editing an existing one requires a means of setting or resetting the field data; this means that the majority of the code is common but the edited card needs to have the changes committed:

```
TInt CRCntUtil::AddNewContactL( TDes8 &aBuffer)
  {
  CContactCard *newCard = CContactCard::NewLC(iTemplate);

  DoEditsL( newCard, aBuffer);
```

```
  TContactItemId newId = iCdb->AddNewContactL(*newCard);
  CleanupStack::PopAndDestroy(newCard);
  return newId;
  }

void CRCntUtil::EditCardL(TDes8 &aBuffer)
  {
  // Find the selected card
  TInt cardId = CConnPack::ReadInt32L(aBuffer);
  CContactItem* editCard = (CContactCard*)iCdb->OpenContactL(cardId);
  CleanupStack::PushL(editCard);

  DoEditsL( editCard, aBuffer);

  // Commit changes
  iCdb->CommitContactL(*editCard);
  CleanupStack::PopAndDestroy(editCard);
  }
```

The actual setting or changing of the field data involves reading a set of field types and a mapping from the message to uniquely identify a field and then setting or changing its text data. If we wanted to extend this to support other storage types then all we would need to do would be to extend this routine. Finding the correct field to change is slightly complex. It is possible for more than one field to share one or more field types – it is only the combination that is supposed to be unique. We can search in the set of fields for a field that contains a single field type, but it may not match the complete set of field types, in which case we need to search for other matching fields.

```
void CRCntUtil::DoEditsL(CContactItem* aCard, TDes8 &aBuffer)
  {
  CContactItemFieldSet &cardFields = aCard->CardFields();
  TInt fieldsCount = cardFields.Count();

  // For each set of field types set the text.
  // Each set of types is terminated by -1 and is followed by the
  // mapping value (or -1 if unset). If the first fields type value
  // is -1 then we have reached the end of the fields.
  TUid fieldType;
  fieldType.iUid = CConnPack::ReadInt32L(aBuffer);
  while(fieldType.iUid >= 0)
    {
    // Make a CContentType and set the field types and the mapping
    CContentType *editContent = CContentType::NewL(fieldType);
    CleanupStack::PushL(editContent);
    while(fieldType.iUid >= 0)
      {
      fieldType.iUid = CConnPack::ReadInt32L(aBuffer);
      if(fieldType.iUid >= 0)
        {
        editContent->AddFieldTypeL(fieldType);
        }
      }
```

```
    TUid mapping;
    mapping.iUid = CConnPack::ReadInt32L(aBuffer);
    editContent->SetMapping(mapping);

    TInt fieldLength = CConnPack::PeekInt16L(aBuffer);
    HBufC* fieldBuff = HBufC::NewLC(fieldLength);
    TPtr tempPtr(fieldBuff->Des());
    CConnPack::ReadUNCDataL(tempPtr, aBuffer);

    // Find the field that matches all the field types
    // Find the one that matches the first field type and then check all
    TBool foundField = false;
    TInt fieldIndex = cardFields.Find(editContent->FieldType(0));
    while((fieldIndex >= 0) && (fieldIndex < fieldsCount) && !foundField)
      {
      if(cardFields[fieldIndex].ContentType() == *editContent)
        {
        foundField = true;
        }
      else
        {
        fieldIndex++;
        if(fieldIndex < fieldsCount)
          {
          fieldIndex = cardFields.FindNext(editContent->FieldType(0),
                                           fieldIndex);
          }
        }
      }//endwhile searching for a matching field
    if(foundField &&
       (cardFields[fieldIndex].StorageType() == KStorageTypeText))
      {
      // The field takes ownership of the buffer
      CleanupStack::Pop(fieldBuff);
      cardFields[fieldIndex].TextStorage()->SetText(fieldBuff);
      }
    else
      {
      CleanupStack::PopAndDestroy(fieldBuff);
      }
    CleanupStack::PopAndDestroy(editContent);
    fieldType.iUid = CConnPack::ReadInt32L(aBuffer);
    }// endwhile
  }
```

11.7.7 Retrieving Groups and Contents of Groups

Groups are relatively simple. When retrieving all groups, we will just return the group identifier and group label. We will also provide a separate method to return the list of contacts in a group.

```
void CRCntUtil::GetAllGroupsL(TDes8 &aBuffer)
  {
  if(iGroupIdArray)
    {
    delete iGroupIdArray;
```

```
    iGroupIdArray = NULL;
    }

  iGroupArrayIndex = 0;
  iGroupIdArray = iCdb->GetGroupIdListL();
  GetMoreGroupsL(aBuffer);
  }

void CRCntUtil::GetMoreGroupsL(TDes8 &aBuffer)
  {
  TBool gotRoom = ETrue;
  while((iGroupIdArray != NULL) &&
        (iGroupArrayIndex < iGroupIdArray->Count()) &&
        gotRoom)
    {
    TInt groupId = (*iGroupIdArray)[iGroupArrayIndex];
    CContactItem* cItem = iCdb->ReadContactL(groupId);
    CleanupStack::PushL(cItem);
    if(cItem->Type() == KUidContactGroup)
      {
      CContactGroup *group = (CContactGroup*)cItem;
      TPtrC label = group->GetGroupLabelL();

      // Check for length for group ID  and label plus terminator
      if(aBuffer.MaxLength() - aBuffer.Length() >= 8 + label.Length())
        {
        CConnPack::WriteInt32L(groupId, aBuffer);
        CConnPack::WriteUNCDataL(label, aBuffer);
        ++iGroupArrayIndex;
        }
      else
        {
        gotRoom = EFalse;
        }
      }
      CleanupStack::PopAndDestroy(cItem);
    }// endwhile
  CConnPack::WriteInt32L(-1, aBuffer);
  if((iGroupIdArray != NULL) && (iGroupArrayIndex ==
       iGroupIdArray->Count()))
    {
    delete iGroupIdArray;
    iGroupIdArray = NULL;
    iGroupArrayIndex = 0;
    }
  }

void CRCntUtil::GetCardsInGroupL(TInt aGroupId, TDes8 &aBuffer)
  {
  CContactItem* cItem = iCdb->ReadContactL(groupId);
  CleanupStack::PushL(cItem);
  if(cItem->Type() == KUidContactGroup)
    {
    CContactGroup *group = (CContactGroup*)cItem;

    CConnPack::WriteInt32L(aGroupId, aBuffer);

    TInt itemsCount = group->ItemsContained()->Count();
```

```
    for(TInt ii = 0 ; ii < itemsCount ; ++ii)
      {
      CConnPack::WriteInt32L((*(group->ItemsContained()))[ii], aBuffer);
      }
    }// endif a group  CConnPack::WriteInt32L(-1, aBuffer);
  CConnPack::WriteInt32L(-1, aBuffer);
  CleanupStack::PopAndDestroy(cItem);
  }
```

11.7.8 Deleting, Adding and Editing Groups

Just as with Contact cards, deleting a group is straightforward:

```
void CRCntUtil::DeleteGroupL(TInt aGroupId)
  {
  iCdb->DeleteContactL(aGroupId);
  }
```

Editing a group label is also straightforward:

```
void CRCntUtil::EditGroupL(TDes8 &aBuffer)
  {
  TInt groupId = CConnPack::ReadInt32L(aBuffer);
  TInt labelLength = CConnPack::PeekInt16L(aBuffer);
  HBufC* labelBuff = HBufC::NewLC(labelLength);
  TPtr tempPtr(labelBuff->Des());
  CConnPack::ReadUNCDataL(tempPtr, aBuffer);

  CContactGroup *group = (CContactGroup*)iCdb->OpenContactL(groupId);
  CleanupStack::PushL(group);

  group->SetGroupLabelL(labelBuff->Des());
  iCdb->CommitContactL(*group);

  CleanupStack::PopAndDestroy(group);
  CleanupStack::PopAndDestroy(labelBuff);
  }
```

Editing the contents of a group is a matter of adding contact items to a group or removing them from it:

```
void CRCntUtil::EditCardInGroupL(TDes8 & aBuffer)
  {
  TInt addFlag = CConnPack::ReadInt32L(aBuffer);
  TInt groupId = CConnPack::ReadInt32L(aBuffer);

  TInt cardId = CConnPack::ReadInt32L(aBuffer);
  while(cardId != -1)
    {
    if(addFlag)
      {iCdb->AddContactToGroupL(cardId, groupId);}
    else
```

```
    {iCdb->RemoveContactFromGroupL(cardId, groupId);}
   cardId = CConnPack::ReadInt32L(aBuffer);
   }
  }
```

If we create a new group then we really want to set the label and the contained contact items in one operation; we do not really want to require two commands – one to create an empty group and another to populate it:

```
TInt CRCntUtil::AddNewGroupL(TDes8 &aBuffer)
  {
  TInt labelLength = CConnPack::PeekInt32L(aBuffer);
  HBufC* labelBuff = HBufC::NewLC(labelLength);
  TPtr tempPtr(labelBuff->Des());
  CConnPack::ReadUNCDataL(tempPtr, aBuffer);

  CContactGroup* newGroup =
    (CContactGroup*)iCdb->CreateContactGroupLC(labelBuff->Des());
  TContactItemId groupId = newGroup->Id();

  CleanupStack::PopAndDestroy(newGroup);
  CleanupStack::PopAndDestroy(labelBuff);

  TInt cardId = CConnPack::ReadInt32L(aBuffer);
  while(cardId != -1)
    {
    iCdb->AddContactToGroupL(cardId, groupId);
    cardId = CConnPack::ReadInt32L(aBuffer);
    }
  return groupId;
  }
```

11.7.9 Calling the Contacts Connectivity Service

While developing this service, I created a command-line application to load it and call it, but there would not be much to learn by including the source in the text here. Instead, we will see how to create a more useful GUI application in Chapter 13.

12

Using the Agenda Model

In this chapter we are going to cover the Symbian OS Agenda Model. Alongside Contacts, the Agenda Model is key to most users, as it represents the user's diary or agenda and lets the user maintain appointments, events and anniversaries and to set alarms. As in the previous chapter, we are not going to see how to create synchronization software, but how to use the Agenda Model directly.

12.1 The Various Agenda Models

Just as with the Contacts model, the Agenda Model is built on top of a database. However, there is not just one Agenda Model but three:

- The entry model (`CAgnEntryModel`) is the base model and provides access to an Agenda database. This model is not intended for user interface applications, but for more 'batch' oriented tasks. Repeating entries (of which more below) are represented by a single entry object.
- The indexed model (`CAgnIndexedModel`) extends the entry model. It includes indices of the data in the database which support filtering and so it is easier to access a subset of the data than when using the entry model.
- The instance model (`CAgnModel`) extends the indexed model and provides a separate object for each instance of a repeating entry. This makes it suitable for use by a user interface (which is more concerned with showing all instances for a specific date than with repeating patterns).

The terms entry and instance used here are defined in the next section.
The Agenda Models support observers to allow an application to be informed when entries are added, altered or deleted. The Agenda Models support filtering to allow access to a subset of entries. The Agenda Models

support progress monitoring on long-running operations. The only such operation that we will be concerned with is opening a database with the indexed model.

Alongside the Agenda Models, the Alarm Server keeps track of all pending alarms and notifies the user when any alarms are due. We will not deal with the Alarm Server directly, but we will see how to manipulate alarmed entries.

12.2 Types of Agenda Entries

The Agenda database (Figure 12.1) contains a number of types of entry:

- appointments (`CAgnAppt`)
- to-dos (`CAgnTodo`)
- events (`CAgnEvent`)
- anniversaries (`CAgnAnniv`).

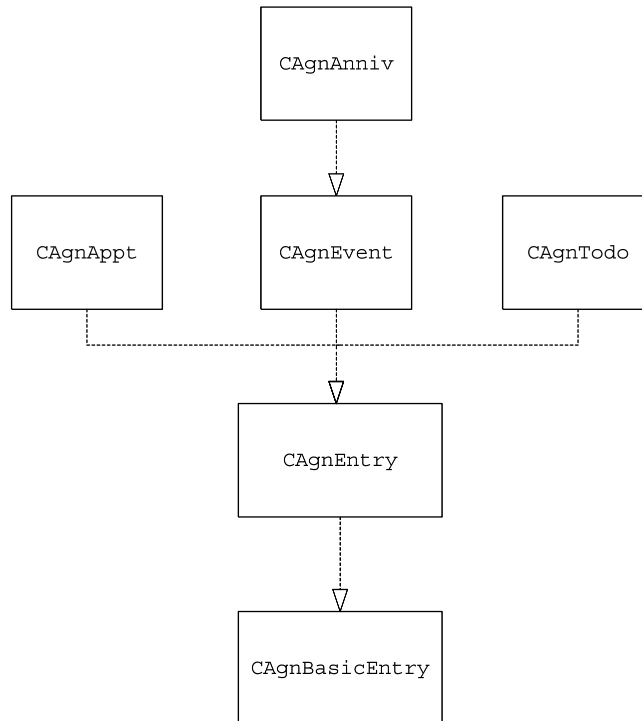


Figure 12.1 The Agenda database

An appointment has a start date and time, an end date and time, and a range of properties. A to-do entry is a task and has a due date and priority, among other attributes. An event is like an appointment but lasts for one or more whole days. An anniversary is an event that repeats on an annual basis.

12.3 Repeating Entries

One of the features supported by the Agenda Model is repeating entries. An appointment or event may be a one-off, but it may also be a repeating entry – a weekly meeting, a monthly event, etc. The Agenda Model allows such an entry to be defined and the repetition configured. This is very efficient for storage and also allows a change to be applied to all instances of an appointment or event, but it is not ideal for displaying a day at a time or for changing one instance of a series.

Therefore, the Agenda Model differentiates between entries and instances. For a repeating entry, the entry defines the whole series, whereas an instance applies to only one date. When using the entry model or the indexed model, we are only presented with entries; if we want to access instances then we need to use the instance model, which is why it is aimed at user interfaces. In the instance model each repeating entry is converted into an instance for each date that applies. Non-repeating entries are also converted into instances, though only one instance each, of course.

Because an instance is an entry on a date, an instance identifier contains an entry identifier and a date. This is worth remembering because most identifiers in Symbian OS are simply four-byte integers.

One point to bear in mind is that the data of a repeating entry must be consistent across the whole of the series. Therefore, changing instance data will mean either altering all instances to have the same data or splitting the series into more than one series. If the data is changed for one instance only or if one instance is deleted, then the series is split to leave a separate series before the changed/deleted entry and a separate series after the change. If an instance is changed along with all entries before or after, the series is split into two. The Agenda Model will take care of this for us, but it means that changing or deleting an instance may affect instances for other dates. This is reflected in the return codes for some operations.

12.4 Alarms

One of the useful features of Agenda entries is that they can have alarms associated with them. A user may want an alarm to go off in advance of

an appointment, so the time for an alarm is defined in terms of a number of days in advance, and a time in terms of a number of minutes after midnight.

12.5 List and Filter Classes

The Agenda Model includes a number of specialized list classes to manage lists of various objects, and filters to control which types of entries or instances should be included in the lists.

12.6 Agenda Model API

As with other APIs covered in this book, we will not cover all aspects of the Agenda Model. It is fully documented in Symbian OS SDKs, though the sheer volume of information (particularly for the three different Agenda Models) can make it more difficult to get started than you might expect.

12.6.1 The Agenda Server and Models

Class `RAgendaServ` Defined in `agclient.h`
`RAgendaServ` provides access to the Agenda Server and is used to set up the Agenda Model. Most server methods are not used directly; one of the Agenda Models is used instead. One `RAgendaServ` object can be used to access only one Agenda database at a time.
Creation Method
`static RAgendaServ* NewL()` This method creates a new `RAgendaServ` object. It must be connected to the Agenda Server using the `Connect()` method.
Member Methods
`TInt Connect()` This method connects to the Agenda Server. returns – an error code, or `KErrNone` for success.
`void Close()` This method severs the connection to the Agenda Server and should be called before the `RAgendaServ` object is deleted.

Class `CAgnEntryModel`
Defined in `agmmodel.h`

The `CAgnEntryModel` model is the base Agenda Model and provides access only to the direct entries in an Agenda database.

Creation Methods

```
static CAgnEntryModel* NewL(
    MAgnModelStateCallBack* aStateCallBack=NULL)
```

This method creates a new `CAgnEntryModel` object, optionally with a state callback observer. If one of the other models is used then this method will not be used.

`aStateCallBack` – a pointer to a state callback observer, if required.
returns – a `CAgnEntryModel` object.

```
void OpenL(const TDesC& aFileName,
           TTimeIntervalMinutes aDefaultEventDisplayTime = 0,
           TTimeIntervalMinutes aDefaultAnnivDisplayTime = 0,
           TTimeIntervalMinutes aDefaultDayNoteDisplayTime = 0)
```

This method opens an Agenda database file in the model. Note that there is no default Agenda database – the name must be specified. If using one of the other models, this method will not be used.
The three `TTimeIntervalMinutes` arguments specify the default display times for the different entry types. The display time values are numbers of minutes from midnight – between zero and 1439 inclusive. These values can be changed using one of the model's `SetDefaultXxxDisplayTime()` functions. The parameters specifying the default display times have default values of zero.

Entry Operations

```
TAgnEntryId AddEntryL(CAgnEntry* aEntry,
  TAgnEntryId aTodoPositionReferenceId = AgnModel::NullId())
```

This method adds a new entry to the database. If the entry is a to-do entry then the entry must specify the to-do list to which it will be added and the position within the list must be specified in the call.

`aEntry` – a pointer to the entry to be added.
`aToDoPositionReferenceId` – if the new entry is a to-do, this argument is used to specify the position in the to-do list at which `aEntry` is added. `aEntry` is inserted at the position immediately before the entry with the identifier specified. If this identifier is null (the default), `aEntry` is added at the end of the to-do list. The value is ignored if the entry is not a to-do.
returns – the entry identifier for the newly added entry.

```
void DeleteEntryL(CAgnEntry* aEntry)
```

This method deletes the entry from the database.

`aEntry` – the entry to be deleted.

`virtual void DeleteEntryL(TAgnEntryId aEntryId)` This method deletes the entry from the database. `aEntryId` – the entry to be deleted.

`void UpdateEntryL(CAgnEntry* aEntry,`
`TAgnEntryId aTodoPositionReferenceId = AgnModel::NullId())`
This method updates an entry in the database.
`aEntry` – pointer to an entry with the details to be updated.
`aTodoPositionReferenceId` – if the entry is a to-do, and the to-do is being moved from one to-do list to another, this specifies the position in the to-do list at which `aEntry`is added. `aEntry` is inserted at the position immediately before the entry with the identifier specified. If this identifier is null (the default), `aEntry` is appended to the to-do list. The value is ignored if the entry is not a to-do, or if it is not being moved to another to-do list.

`CAgnEntry* FetchEntryL(TAgnEntryId aId) const` This method retrieves an entry based on its entry identifier. Note that, if an instance is required rather than an entry, the `CAgnModel::FetchEntryL()` method should be used. `aId` – the identifier of the entry to be retrieved. returns – a pointer to entry data. The caller takes ownership of the object.
To-do Operations
`TInt TodoListCount() const` This method returns the number of to-do lists in the database.
`inline const CAgnTodoListList* PeekAtTodoListList() const` This method returns a read-only list of to-do list identifiers.
`CAgnTodoList* FetchTodoListL(TAgnTodoListId aTodoListId) const` This method fetches a to-do list based on its identifier. `aTodoListId` – the identifier of the to-do list. returns – a pointer to the list. The caller takes ownership of the object.

`virtual TAgnTodoListId AddTodoListL(CAgnTodoList* aTodoList,`
`TInt aPosition=EAddTodoListAtEnd)`
This method adds a new to-do list to the database. If the position is not specified, the to-do list is added at the end of the list of to-do lists. Specifying a position of zero indicates it is the first to-do list. If the position specified is equal to the number of to-do lists in the model, the new to-do list is added at the end. If the position specified is greater than the number of to-do lists in the model, or is negative, the function raises a panic.
`aTodoList` – a pointer to the to-do list data to be added.
`aPosition` – position within the model's list of to-do lists at which to insert the to-do list.

`virtual void DeleteTodoListL(TAgnTodoListId aTodoListId)`
`virtual void UpdateTodoListL(CAgnTodoList* aTodoList)`
These methods delete a to-do list from the database. The to-do list can be specified either by its identifier or by providing a pointer to its data.
`aTodoListId` – the identifier of the to-do list to be deleted.
`aTodoList` – a pointer to the data of the to-do list to be deleted.

```
void ChangeTodoListOrderL(TInt aOldPosition,TInt aNewPosition)
```
Changes the position of a to-do list in the model's list of to-do lists. The two positions specified are indexes into the model's list of to-do lists, so should be specified as relative to zero, i.e. zero indicates the first position in the list. Specify `EAddTodoListAtEnd` to indicate the position at the end of the list. Both positions must be valid, or a panic is raised.

`aOldPosition` – the position of the to-do list to be moved.
`aNewPosition` – the position to which the to-do list is to be moved.

```
void ChangeTodoOrderL(TAgnTodoListId aTodoListId,
  TAgnEntryId aTodoId,
  TAgnEntryId aTodoPositionReferenceId= AgnModel::NullId())
void ChangeTodoOrderL(CAgnTodoList* aTodoList,
  TAgnEntryId aTodoId,
  TAgnEntryId aTodoPositionReferenceId= AgnModel::NullId())
```
These methods move the to-do entry with the entry identifier `aTodoId` to the position immediately before the position occupied by `aTodoPositionReferenceId`. The to-do list affected is identified by `aTodoListId` or `aTodoList`. If `aTodoPositionReferenceId` is null (the default), or if it cannot be found in the list, then the to-do entry is moved to the end of the list.

Note that a to-do instance list (`CAgnTodoInstanceList`) should be used to determine the position to which to move the to-do entry. This is because the user sees the order of the to-do entries as they appear in a to-do instance list. A to-do list (`CAgnTodoList`), on the other hand, may contain to-do entries which are not visible to the user (for instance if they are crossed out, and crossed out entries are not being displayed).

`aTodoListId` – identifies the to-do list to which `aTodoId` belongs.
`aTodoList` – pointer to the to-do list to which `aTodoId` belongs.
`aTodoId` – identifies the to-do entry to move.
`aTodoPosotionReferenceId` – specifies a position in the to-do list. The entry identified by `aTodoId` is moved to the position immediately before this. If this is null (the default) then `aTodoId` is moved to the end of the list.

Miscellaneous Methods

```
inline const CParaFormatLayer* ParaFormatLayer() const
```
This method returns a pointer to the paragraph format layer used by the model's entries.

```
inline const CCharFormatLayer* CharFormatLayer() const
```
This method returns a pointer to the character format layer used by the model's entries.

Class `MAgnProgressCallBack`
Defined in `cagmcallb.h`

This mixin provides information about the progress of prolonged Agenda Model operations, and about how the operation terminated.

As the model carries out the operation it calls `Progress()` with `aPercentageCompleted` set to the amount of the operation that has been carried out so far. Note that when the operation is complete, `Progress()` is not called with `aPercentageCompleted` set to

100. Instead the `Completed()` method is called. Note also that if the operation completes quickly, for instance when opening a small file, then `Progress()` will probably not be called. If, on return, `aError` is anything other than `KErrNone` then something went wrong during the operation.

Member Methods

`virtual void Progress(TInt aPercentageCompleted) = 0`
This method is called by the agenda model during prolonged operations such as opening an indexed database, to indicate how much of the operation has been completed. Note that this function is not called when the operation is complete, with `aPercentageCompleted` set to 100; instead the `Completed()` method is called. The frequency with which this function is called is indicated by a callback frequency parameter.

`virtual void Completed(TInt aError = KErrNone) = 0`
This method is called when the function completes.
`aError` – indicates how the function completed: `KErrNone` if successful, otherwise one of the system error codes.

`enum TOpenCallBackFrequency`

- `EOpenCallBackHigh = 1` – high callback frequency
- `EOpenCallBackMedium = 2` – medium callback frequency
- `EOpenCallBackLow = 4` – low callback frequency

Class `CAgnIndexedModel` – public `CAgnEntryModel`
Defined in `agmmodel.h`

The indexed model (`CAgnIndexedModel`) is opened as an intermediate stage in opening the instance model (`CAgnModel`). It can also be used in its own right for greater efficiency by applications which have no user interface and which need to access only a subset of the data in the file.
When an indexed model is opened, it builds indices of the data in an Agenda file, which allows the information to be filtered before entries are read. As well as being used as an intermediate stage for the instance model, it supports merging and tidying, but these methods are not described here.

Creation and Connection Methods

```
static CAgnIndexedModel* NewL(MAgnModelStateCallBack*
                              aStateCallBack)
```
This method creates an indexed model. It creates a new `CAgnIndexedModel` object, optionally with a state callback observer.
`aStateCallBack` – a pointer to a state callback observer, if required.
returns – a `CAgnIndexedModel` object.

```
TBool OpenL(const TDesC& aFileName,
  TTimeIntervalMinutes aDefaultEventDisplayTime,
  TTimeIntervalMinutes aDefaultAnnivDisplayTime,
  TTimeIntervalMinutes aDefaultDayNoteDisplayTime,
  MAgnProgressCallBack* aProgressCallBack,
  TBool aOpenSynchronously = EFalse,
  TOpenCallBackFrequency aCallBackFrequency = EOpenCallBackMedium)
void OpenL(const TDesC& aFileName,
  TTimeIntervalMinutes aDefaultEventDisplayTime,
  TTimeIntervalMinutes aDefaultAnnivDisplayTime,
  TTimeIntervalMinutes aDefaultDayNoteDisplayTime)
```

These methods open a named Agenda file using the Agenda Server. The three `TTimeIntervalMinutes` arguments specify the default display times for the different entry types. The display time values are numbers of minutes from midnight – between zero and 1439 inclusive.

One version of this method supplies a progress observer. This version returns true or false depending on whether or not an active object was scheduled to build the indices.

- `aFileName` – the name of the Agenda Database file to open.
- `aDefaultEventDisplayTime` – the default display time for events.
- `aDefaultAnnivDisplayTime` – the default display time for anniversaries.
- `aDefaultDayNoteDisplayTime` – the default display time for day notes.
- `aProgressCallBack` – a pointer to a progress callback observer.
- `aOpenSynchronously` – set to `ETrue` if the database is to be opened synchronously.
- `aCallBackFrequency` – the desired callback frequency.
- returns – the version with a progress callback returns `ETrue` if indices are built asynchronously.

Entry methods

`TInt EntryCount() const`

This method returns the number of entries in the database.

`enum TAgnWhichInstances`

This enum specifies which instances of an entry are affected by an update or delete operation. If an update or delete operation does not affect all instances then the entry will be split into two.

- `ECurrentInstance` – only the current instance
- `EAllInstances` – all instances
- `ECurrentAndFutureInstances` – the current and all future instances
- `ECurrentAndPastInstances` – the current and all past instances

`enum TAgnFilter::Ttype`

- `EAgnFilter` – default filter type
- `EDayFilter` – indicates the filter is a `TAgnDayFilter`

- `ESymbolFilter` – indicates the filter is a `TAgnSymbolFilter`
- `EFindFilter` – indicates the filter is a `TAgnsrvFindFilter`
- `ETidyFilter` – indicates the filter is a `TAgnsrvTidyFilter`

Class `TagnFilter`
This class identifies which entry types should be involved when searching or populating an instance list. It is usually used in combination with a date or date range.
Constructor
`TAgnFilter()`
Filter Setting Methods
`virtual TBool IsValid(const CAgnSortEntry* aElement) const` This method returns whether the type of the specified entry is included in or excluded from the filter. `aElement` – the element to be checked. returns – `ETrue` if the entry is included in the filter.
`void SetIncludeTimedAppts(TBool aIncludeTimedAppts)` This method sets the filter to include or exclude timed appointments. `aIncludeTimedAppts` – `ETrue` if timed appointments are to be included.
`inline TBool AreTimedApptsIncluded() const` This method returns `ETrue` if timed appointments are included in the filter.
`void SetIncludeUnTimedAppts(TBool aIncludeUntimedAppts)` This method sets the filter to include or exclude untimed appointments (i.e. day notes). `aIncludeTimedAppts` – `ETrue` if untimed appointments are to be included.
`inline TBool AreUnTimedApptsIncluded() const` This method returns `ETrue` if untimed appointments are included in the filter.
`void SetIncludeEvents(TBool aIncludeEvents)` This method sets the filter to include or exclude events. `aIncludeEvents` – `ETrue` if events are to be included.
`inline TBool AreEventsIncluded() const` This method returns `ETrue` if events are included in the filter.
`void SetIncludeAnnivs(TBool aIncludeAnnivs)` This method sets the filter to include or exclude anniversaries. `aIncludeAnnivs` – `ETrue` if anniversaries are to be included.
`inline TBool AreAnnivsIncluded() const` This method returns `ETrue` if anniversaries are included in the filter.

`void SetIncludeTodos(TBool aIncludeTodos)` This method sets the filter to include or exclude to-do list entries. `aIncludeTodos` – `ETrue` if to-do list entries are to be included.
`inline TBool AreTodosIncluded() const` This method returns `ETrue` if to-do list entries are included in the filter.
`void SetIncludeRpts(TBool aIncludeRpts)` This method sets the filter to include or exclude repeated entries. `aIncludeRpts` – `ETrue` if repeated entries are to be included.
`inline TBool AreRptsIncluded() const` This method returns `ETrue` if repeating entries are included in the filter.
`void SetIncludeNonRpts(TBool aIncludeNonRpts)` This method sets the filter to include or exclude non-repeated entries. `aIncludeNonRpts` – `ETrue` if non-repeated entries are to be included.
`inline TBool AreNonRptsIncluded() const` This method returns `ETrue` if non-repeated entries are included in the filter.

`void SetIncludeRptsNextInstanceOnly(`
`TBool aIncludeRptsNextInstanceOnly)`
This method sets the filter to include/exclude the next instance only of repeated entries.
`aIncludeRptsNextInstanceOnly` – `ETrue` if the next instance only of repeated entries are to be included.

`inline TBool RptNextInstanceOnly() const` This method returns `ETrue` if the next instance only of repeated entries are included in the filter.
`void SetIncludeCrossedOut(TBool aIncludeCrossedOut)` This method sets the filter to include or exclude crossed out entries. `aIncludeCrossedOut` – `ETrue` if crossed out entries are to be included.
`inline TBool AreCrossedOutIncluded() const` This method returns `ETrue` if crossed out entries are included in the filter.
`void SetIncludeAlarmedOnly(TBool aIncludeAlarmedOnly)` This method sets the filter to include, or not, only alarmed entries. `aIncludeAlarmedOnly` – `ETrue` if only alarmed entries are to be included.
`inline TBool AreAlarmedOnlyIncluded() const` This method returns `ETrue` if only alarmed entries are included in the filter.
`void SetIncludeCrossedOutOnly(TBool aIncludeCrossedOutOnly)` This method sets the filter to include, or not, only crossed out entries. `aIncludeCrossedOutOnly` – `ETrue` if only crossed out entries are to be included.
`inline TBool AreCrossedOutOnlyIncluded() const` This method returns `ETrue` if only crossed out entries are included in the filter.

Class `CAgnModel` – public `CAgnIndexedModel`
Defined in `agmmodel.h`

The instance agenda model is used by applications which have a user interface. In the entry model, a repeating entry is represented as a single entry with a set of repeat details. However, for a user interface, a repeating entry needs to be presented to the user as multiple instances, one on each date on which a repeat occurs.
An application with a user interface should generally deal with instances, not entries. It should never call `UpdateEntryL()` or `DeleteEntryL()` and will very rarely (if at all) call `FetchEntryL()`. Instead, it should use the instance model and call `UpdateInstanceL()`, `DeleteInstanceL()` or `FetchInstanceL()`.
An exception to this is when a new entry is added to the database model. Because it does not have an instance date at this point, the function is called `AddEntryL()`.

Creation Methods

```
static CAgnModel* NewL(MAgnModelStateCallBack* aStateCallBack)
```
This method creates an instance model. It creates a new `CAgnModel` object, optionally with a state callback observer.
`aStateCallBack` – a pointer to a state callback observer, if required.
returns – a `CAgnModel` object.

Entry and Instance Methods

```
CAgnEntry* FetchInstanceL(const TAgnInstanceId& aInstanceId) const
CAgnEntry* FetchInstanceLC(const TAgnInstanceId& aInstanceId) const
```
These methods fetch an instance of an entry from the database.
`aInstanceId` – the instance identifier of the instance to be retrieved.
returns – the specified instance.

```
TAgnEntryId AddEntryL(CAgnEntry* aEntry,
  TAgnEntryId aTodoPositionReferenceId = AgnModel::NullId())
```
This method adds an entry to the database. If the entry is a to-do list entry then it must include the identifier of the to-do list to which it belongs.
`aEntry` – the entry data to be added to the database.
`aTodoPositionReferenceId` – If the new entry is a to-do, this argument is used to specify the position in the to-do list at which `aEntry` is added. `aEntry` is inserted at the position immediately before the entry with the identifier specified. If this identifier is null (the default), `aEntry` is added at the end of the to-do list. The value is ignored if the entry is not a to-do.

```
TAgnEntryId UpdateInstanceL(CAgnEntry* aInstance,
  TAgnWhichInstances aWhichInstances=EAllInstances,
  TAgnEntryId aTodoPositionReferenceId = AgnModel::NullId())
```
This method updates the data for an instance in the database.
`aInstance` – the entry data to be updated.
`aWhichInstances` – which instances of a repeating entry are to be updated.
`aTodoPositionReferenceId` – if `aInstance` is a to-do which is being moved from

one to-do list to another, then this specifies its position in the new list. `aInstance` is inserted at the position immediately before the entry with the identifier specified. If this identifier is NULL (the default), `aInstance` is appended to the to-do list. The value is ignored if `aInstance` is not a to-do, or if it is not being moved to another to-do list.
returns – the new entry identifier of `aInstance` if it was changed during the update. Otherwise, a null entry identifier.

```
TAgnEntryId DeleteInstanceL(CAgnEntry* aInstance,
  TAgnWhichInstances aWhichInstances=EAllInstances)
TAgnEntryId DeleteInstanceL(const TAgnInstanceId& aInstanceId,
  TAgnWhichInstances aWhichInstances=EAllInstances)
```

These methods delete an instance from the database.
`aInstance` – the instance to be deleted.
`aInstanceId` – the identifier of the instance to be deleted.
`aWhichInstances` – which instances of a repeating entry are to be deleted.
returns – the new entry identifier of `aInstance` if it was changed during the delete. Otherwise, a null entry identifier.

To-do List Methods

```
virtual TAgnTodoListId AddTodoListL(CAgnTodoList* aTodoList,
  TInt aPosition=EAddTodoListAtEnd)
```

This method adds a to-do list to the database and sets the position of the new list in the database's list of to-do lists.
`aTodoList` – the to-do list to be added.
`aPosition` – position within the database's list of to-do lists at which to insert the to-do list.

```
virtual void DeleteTodoListL(CAgnTodoList* aTodoList)
virtual void DeleteTodoListL(TAgnTodoListId aTodoListId)
```

These methods delete a to-do list.
`aTodoList` – the to-do list to be deleted.
`aTodoListId` – the identifier of the to-do list to be deleted.

```
virtual void UpdateTodoListL(CAgnTodoList* aTodoList)
```

This method updates a to-do list in the database.
`aTodoList` – the to-do list to be updated.

Populating Methods
These methods fill lists of instances. The contained instance identifiers can then be used to fetch the instance data.

```
void PopulateDayInstanceListL(CAgnDayList<TAgnInstanceId>* aList,
  const TAgnFilter& aFilter,
  const TTime& aTodaysDate) const
```

Populates a list with instance IDs for a particular day. The entry types of interest may be specified using a filter. Instances which start on the previous day may be included in the day list if they span midnight.
If the date specified in the day list is invalid, i.e. outside the model's valid range, the function returns with an empty list.

Notes

Instances are sorted by display time. Instances with equal display times are sorted in the following order:

1. To-do entries. Multiple to-do entries with the same display time are ordered by due date, priority then start date.
2. Anniversaries.
3. Day notes.
4. Appointments. Appointments are ordered by start time, then end time.

Finally, if two instances are still the same, then crossed out ones appear after uncrossed out ones.

`aList` – an empty list. Should specify a valid date. On return, contains a list of instance IDs and start and end dates/times for instances which occur on the specified date.

`aFilter` – filters the entry types of interest.

`aTodaysDate` – today's date as a `TTime`. Should be specified to include undated to-do entries in the list.

```
void PopulateDayDateTimeInstanceListL(
  CAgnDayDateTimeInstanceList* aList,
  const TAgnFilter& aFilter,
  const TTime& aTodaysDate) const
```

Populates a list with instance IDs for a particular day.

The list also includes the start and end date/time for each instance. The entry types of interest may be specified using a filter.

Instances which start on the previous day may be included in the day list if they span midnight.

If the date specified in the day list is invalid, i.e. outside the model's valid range, the function returns with an empty list.

The instances are sorted using the same criteria as for `PopulateDayInstanceListL()`.

`aList` – an empty list. Should specify a valid date. On return, contains a list of instance IDs and start and end dates/times for instances which occur on the specified date.

`aFilter` – filters the entry types of interest.

`aTodaysDate` – today's date as a `TTime`. Should be specified to include undated to-do entries in the list.

```
void PopulateMonthInstanceListL(CAgnMonthInstanceList* aList,
  const TAgnFilter& aFilter,
  const TTime& aTodaysDate) const
```

Populates a list with instance IDs for a particular month. The entry types of interest may be specified using a filter.

If the year specified in the month list is invalid, i.e. outside the model's valid range, the function returns with an empty list.

The instances are sorted in chronological order.

`aList` – an empty list. Should specify a valid month and year. On return, contains a list of instance IDs for instances which occur in the specified month.

`aFilter` – filters the entry types of interest.

`aTodaysDate` – today's date as a `TTime`. Should be specified to include undated to-do entries in the list.

```
void PopulateTodoInstanceListL(CAgnTodoInstanceList* aList,
  const TTime& aTodaysDate) const
```

Populates a list with instance IDs for a given to-do list.

The to-do list of interest is identified by a to-do list identifier specified in `aList`. If `aList` does not specify a to-do list identifier, the function returns with an empty list.

The sort order and whether or not crossed out entries should be included in the list are both specified in the to-do list identified in `aList`.

- `aList` – a to-do instance list. Specifies a to-do list identifier. On return, contains a sorted list of to-do instance IDs that match the filter criteria specified in the to-do list. It therefore holds the to-dos that are to be displayed from a particular to-do list at any given time.
- `aTodaysDate` – today's date as a `TTime`. Should be specified to include undated to-do entries in the list.

Find methods

```
void FindNextInstanceL(
  CAgnDayList<TAgnInstanceId>* aMatchedInstanceList,
  const TDesC& aSearchText,
  const TTime& aStartDate,
  const TTime& aEndDate,
  const TAgnFilter& aFilter,
  const TTime& aToday) const
```

Populates a day list with instance IDs for instances whose rich text matches a search string. A match is made if the instance's rich text is an exact match for the search string, or is a subset of it.

After a match is made, other instances on that day are searched. When there are no more instances on that day to search, the function returns.

Only dates within the date range specified are searched, and the entry types to be used in the search are specified using a filter.

- `aMatchedInstanceList` – pointer to a day list. On return, contains the IDs for all instances matching the search string for a single day. If no matching instances were found, the list returns empty.
- `aSearchText` – the search string. Has a maximum of 32 characters.
- `aStartDate` – the date at which to begin the search.
- `aEndDate` – the date at which to end the search.
- `aFilter` – filters the entry types of interest.
- `aToday` – today's date as a `TTime`. Used to identify whether or not undated to-dos should be included.

```
void FindPreviousInstanceL(
  CAgnDayList<TAgnInstanceId>* aMatchedInstanceList,
  const TDesC& aSearchText,
  const TTime& aStartDate,
  const TTime& aEndDate,
  const TAgnFilter& aFilter,
  const TTime& aToday) const
```

Populates a day list with instance IDs for instances whose rich text matches a search string. A match is made if the instance's rich text is an exact match for the search string, or is a subset of it.

After a match is made, other instances on that day are searched. When there are no more instances on that day to search, the function returns.
Only dates within the date range specified are searched, and the entry types to be used in the search are specified using a filter.
The search starts at the start date and works backwards, so the start date should be after the end date.

`aMatchedInstanceList` – pointer to a day list. On return contains the IDs for all instances matching the search string for a single day. If no matching instances were found, the list returns empty.
`aSearchText` – the search string. Has a maximum of 32 characters.
`aStartDate` – the date at which to begin the search.
`aEndDate` – the date at which to end the search.
`aFilter` – filters the entry types of interest.
`aToday` – today's date as a `TTime`. Used to identify whether or not undated to-dos should be included.

12.6.2 Repeat APIs

Class `TAgnRpt`
Defined in `agmrptd.h`

This abstract base class for the Agenda Model repeat types stores the repeat details common to all Agenda Model repeat types. The common repeat details are the start and end dates of the repeat, the interval between repeats, and two flags which indicate whether views should display the next repeat only, and whether it should repeat forever.
An instance of a `TAgnRpt` -derived class is owned by a repeat definition (`CAgnRptDef`), which also contains the repeat exceptions list. It is set using `CAgnRptDef::SetDaily()`, `SetWeekly()`, etc.

Member Methods

```
void SetStartDate(const TTime& aStartdate)
inline void SetStartDate(TAgnDate aStartdate)
```
These methods set the start date for the repeat.
`aStartDate` – the start date to set.

```
TTime StartDate() const
inline TAgnDate StartDateAsAgnDate() const
```
These methods return the repeat start date.

```
void SetEndDate(const TTime& aEndDate)
inline void SetEndDate(TAgnDate aEndDate)
```
These methods set the repeat end date.
`aEndDate` – the end date to set.

`TTime EndDate() const`
`inline TAgnDate EndDateAsAgnDate() const`
These methods return the repeat end date.

`inline void SetInterval(TUint aInterval)`
This method sets the repeat interval.

`inline TInt Interval() const`
This method returns the repeat interval.

`inline void SetDisplayNextOnly(TBool aDisplayNextOnly)`
This method sets whether or not the repeated entry should display only the next entry.
`aDisplayNextOnly` – `ETrue` if only the next entry should be displayed.

`inline TBool DisplayNextOnly() const`
This method returns `ETrue` if only the next entry is set to be displayed.

`void SetRepeatForever(TBool aRepeatForever)`
This method sets whether the entry should repeat forever.
`aRepeatForever` – `ETrue` if the entry should repeat forever.

`inline TBool RepeatForever() const`
This method returns `ETrue` if the entry should repeat forever.

`void ClearAll()`
This method clears all the repeat details. The start and end dates are set to NULL values and the repeat interval to zero. The display next only and repeat forever flags are set to `EFalse`.

`virtual TUint InstanceCount() const`
This method returns the number of instances generated by the repeat algorithm.

`virtual TTime FindRptEndDate(TUint aCount) const`
This method calculates the repeat end date from the number of instances.

Class `TAgnDailyRpt` – public `TAgnRpt`
Defined in `agmrptd.h`

This class stores daily repeat details. It stores the number of days between repeats. The repeat details are used by the `CAgnRptDef` class, which also stores a list of any exceptions to the repeat.

Constructors

`TAgnDailyRpt()`
`TAgnDailyRpt(const TAgnRpt& aRpt)`
`TAgnDailyRpt(const TAgnDailyRpt& aRpt)`
The constructors can take an existing repeat definition.
`aRpt` – an existing repeat definition to copy.

Class `TAgnWeeklyRpt` – public `TAgnRpt`
Defined in `agmrptd.h`

This class stores weekly repeat details. In addition to the common repeat information inherited from `TAgnRpt`, this class stores the days in the week on which the repeat occurs (using a set of flags), and the day which is the start of the week. The repeat details are used by the `CAgnRptDef` class, which also stores a list of any exceptions to the repeat.

Constructors

`TAgnWeeklyRpt()`
`TAgnWeeklyRpt(const TAgnRpt& aRpt)`
`TAgnWeeklyRpt(const TAgnWeeklyRpt& aRpt)`
The constructors can take an existing repeat definition.
  `aRpt` – an existing repeat definition to copy.

Member Methods

`void SetDay(TDay aDay)`
This method adds a day to the repeat. More than one day can be set.
  `aDay` – the day to be set.

`void UnsetDay(TDay aDay)`
This method unsets a day from the repeat.
  `aDay` – the day to be unset.

`inline void ClearDays()`
This method unsets all days from the repeat.

`TBool IsDaySet(TDay aDay) const`
This method returns `ETrue` if a specific day is set in the repeat.
  `aDay` – the day being enquired about.

`TDay FirstDayOfWeek() const`
This method returns which day is being used as the first day of the week. This is set during construction from the operating system's locale settings.

`TUint NumDaysSet() const`
This method returns the number of days which are set in the repeat.

Class `TAgnMonthlyRpt` – public `TAgnRpt`
Defined in `agmrptd.h`

This abstract base class is used by the monthly repeat types `AgnMonthlyByDaysRpt` and `TAgnMonthlyByDatesRpt`.

Class `TAgnMonthlyByDaysRpt` – public `TAgnMonthlyRpt`
Defined in `agmrptd.h`

This class stores the days in a month on which a repeat can occur, for instance Tuesday in the first week, Sunday in the last week.

Constructors

```
TAgnMonthlyByDaysRpt()
TAgnMonthlyByDaysRpt(const TAgnRpt& aRpt)
TAgnMonthlyByDaysRpt(const TAgnMonthlyByDaysRpt& aRpt)
```
The constructors can take an existing repeat definition.
`aRpt` – an existing repeat definition to copy.

Member Methods

```
void SetDay(TDay aDay, TWeekInMonth aWeek)
```
This method sets a repeat for a day in a week in the month.
`aDay` – the day of the week to be set.
`aWeek` – the week in which the day lies.

```
void ClearAllDays()
```
This method clears all days set in the repeat.

```
void ClearWeek(TWeekInMonth aWeek)
```
This method clears all days in a specific week in the month.
`aWeek` – the week to clear.

```
void UnsetDay(TDay aDay,TWeekInMonth aWeek)
```
This method unsets a repeat for a day in a week in the month.
`aDay` – the day of the week to be unset.
`aWeek` – the week in which the day lies.

```
TInt NumDaysSet() const
```
This method returns the number of days set in the repeat.

```
TBool IsDaySet(TDay aDay, TWeekInMonth aWeek) const
```
This method checks whether a specific day in a week is set.
`aDay` – the day of the week to be checked.
`aWeek` – the week in which the day lies.
returns – `ETrue` if the day is set.

```
TBool EventOnDate (const TTime& aTime) const
```
This method checks whether a specific date has a repeat set.
`aTime` – the date to check.
returns – `ETrue` if the date is set in the repeat.

Class `TAgnMonthlyByDatesRpt` – public `TAgnMonthlyRpt`
Defined in `agmrptd.h`

This class stores the dates in a month on which a repeat can occur.

Constructors

`TAgnMonthlyByDatesRpt()`
`TAgnMonthlyByDatesRpt(const TAgnRpt& aRpt)`
`TAgnMonthlyByDatesRpt(const TAgnMonthlyByDatesRpt& aRpt)`
The constructors can take an existing repeat definition.
`aRpt` – an existing repeat definition to copy.

Member Methods

`void SetDate(TUint aDateInMonth)`
This method sets a specific date in a month.
`aDateInMonth` – the date to be set.

`void UnsetDate(TUint aDateInMonth)`
This method unsets a specific date in a month.
`aDateInMonth` – the date to be unset.

`inline void ClearAllDates()`
This method unsets all dates for the repeat.

`TInt NumDatesSet() const`
This method returns the number of dates set in the repeat.

`TBool IsDateSet(TUint aDateInMonth) const`
This method checks whether a date is set.
`aDateInMonth` – the date to check.
returns – `ETrue` if the date is set.

Class `TAgnYearlyByDateRpt` – public `TAgnRpt`
Defined in `agmrptd.h`

This class is a yearly by date repeat, for example 3rd October each year. The base `TAgnRpt` class includes sufficient data to fully define it.

Constructors

`TAgnYearlyByDateRpt()`
`TAgnYearlyByDateRpt(const TAgnRpt& aRpt)`
`TAgnYearlyByDateRpt(const TAgnYearlyByDteRpt& aRpt);`
The constructors can take an existing repeat definition.
`aRpt` – an existing repeat definition to copy.

Class `TAgnYearlyByDayRpt` – public `TAgnRpt`
Defined in `agmrptd.h`

This class is a yearly by day repeat. In addition to a start date, this class stores the day in the week and the week in the month on which the repeat should occur, e.g. Wednesday of the third week of November every year.

Constructors

```
TAgnYearlyByDayRpt()
TAgnYearlyByDayRpt(const TAgnRpt& aRpt)
TAgnYearlyByDayRpt(const TAgnYearlyByDayRpt& aRpt)
```

The constructors can take an existing repeat definition.
`aRpt` – an existing repeat definition to copy.

Member Methods

```
void SetStartDay(TDay aDay, TWeekInMonth aWeek,
                 TMonth aMonth, TInt aYear)
```

This method sets the start day.
`aDay` – the day of the week to be set.
`aWeek` – the week of the month in which the day lies.
`aMonth` – the month in which the day lies.
`aYear` – the year on which the repeat starts.

```
void GetStartDay(TDay& aDay, TWeekInMonth& aWeek,
                 TMonth& aMonth, TInt& aYear) const
```

This method returns the start day.
`aDay` – the day of the week set.
`aWeek` – the week of the month in which the day lies.
`aMonth` – the month in which the day lies.
`aYear` – the year on which the repeat starts.

`enum CAgnRptDef::TType`

- `EDaily` – repeat every x days
- `EWeekly` – repeat every x weeks on days to be set
- `EMonthlyByDates` – repeat every x months on dates to be set
- `EMonthlyByDays` – repeat every x months on specific days of specific weeks
- `EYearlyByDate` – repeat every x years on a date to be set
- `EYearlyByDay` – repeat every x years on a specific day of a specific week of a month

Class `CAgnRptDef` – public `CBase`
Defined in `agmrptd.h`

This class stores a repeat definition, including exceptions, which is owned by an agenda entry. The repeat definition has a type (daily, weekly, etc.) and stores a set of repeat details (e.g. start

<table>
<tr><td>and end date, interval between repeats). The repeat details are stored using an instance of a <code>TAgnRpt</code> -derived class. They are set using <code>SetDaily()</code>, <code>SetWeekly()</code>, etc. These functions also set the type.
The class may optionally store an exception list, which is a list of the dates on which the repeat should not occur.
Note that when the repeat definition is set, the entry's start date (or due date for a to-do entry) may be changed to the repeat definition's start date.</td></tr>
<tr><td>Construction Method</td></tr>
<tr><td><code>static CAgnRptDef* NewL()</code>
This method creates a new repeat definition.</td></tr>
<tr><td>Repeat Type Methods</td></tr>
<tr><td><code>inline TType Type() const</code>
This method returns the repeat type of the repeat.</td></tr>
<tr><td><code>void SetDaily(const TAgnDailyRpt& aRpt)</code>
This method sets the repeat type to be daily and sets the repeat details.
  <code>aRpt</code> – the details to be set.</td></tr>
<tr><td><code>TAgnDailyRpt Daily() const</code>
If the repeat definition is of type <code>EDaily</code>, then this method returns the repeat details. Otherwise, it returns the repeat details as a daily repeat.</td></tr>
<tr><td><code>void SetWeekly(const TAgnWeeklyRpt& aRpt)</code>
This method sets the repeat type to be weekly and sets the repeat details.
  <code>aRpt</code> – the details to be set.</td></tr>
<tr><td><code>TAgnWeeklyRpt Weekly() const</code>
If the repeat definition is of type <code>EWeekly</code>, then this method returns the repeat details. Otherwise, it returns the repeat details as a weekly repeat.</td></tr>
<tr><td><code>void SetMonthlyByDates(const TAgnMonthlyByDatesRpt& aRpt)</code>
This method sets the repeat type to be monthly by dates and sets the repeat details.
  <code>aRpt</code> – the details to be set.</td></tr>
<tr><td><code>TAgnMonthlyByDaysRpt MonthlyByDays() const</code>
If the repeat definition is of type <code>EMonthlyByDays</code>, then this method returns the repeat details. Otherwise, it returns the repeat details as a monthly repeat by days.</td></tr>
<tr><td><code>void SetMonthlyByDays(const TAgnMonthlyByDaysRpt& aRpt)</code>
This method sets the repeat type to be monthly by days and sets the repeat details.
  <code>aRpt</code> – the details to be set.</td></tr>
<tr><td><code>TAgnMonthlyByDatesRpt MonthlyByDates() const</code>
If the repeat definition is of type <code>EMonthlyByDates</code>, then this method returns the repeat details. Otherwise, it returns the repeat details as a monthly repeat by dates.</td></tr>
</table>

`void SetYearlyByDate(const TAgnYearlyByDateRpt& aRpt)` This method sets the repeat type to be yearly by date and sets the repeat details. `aRpt` – the details to be set.
`TAgnYearlyByDateRpt YearlyByDate() const` If the repeat definition is of type `EYearlyByDate`, then this method returns the repeat details. Otherwise, it returns the repeat details as a yearly repeat by date.
`void SetYearlyByDay(const TAgnYearlyByDayRpt& aRpt)` This method sets the repeat type to be yearly by day and sets the repeat details. `aRpt` – the details to be set.
`TAgnYearlyByDayRpt YearlyByDay() const` If the repeat definition is of type `EYearlyByDay`, then this method returns the repeat details. Otherwise, it returns the repeat details as a yearly repeat by day.
Repeat Data Methods
`inline const TAgnRpt* RptDef() const` This method provides direct access to the repeat definition.

`void SetStartDate(const TTime& aStartdate)`
`inline void SetStartDate(TAgnDate aStartdate)`
These methods set the start date for the repeat.
`aStartDate` – the start date to set.

`TTime StartDate() const`
`inline TAgnDate StartDateAsAgnDate() const`
These methods return the repeat start date.

`void SetEndDate(const TTime& aEndDate)`
`inline void SetEndDate(TAgnDate aEndDate)`
These methods set the repeat end date.
`aEndDate` – the end date to set.

`TTime EndDate() const`
`inline TAgnDate EndDateAsAgnDate() const`
These methods return the repeat end date.

`inline void SetInterval(TUint aInterval)` This method sets the repeat interval.
`inline TInt Interval() const` This method returns the repeat interval.
`inline void SetDisplayNextOnly(TBool aDisplayNextOnly)` This method sets whether or not the repeated entry should display only the next entry. `aDisplayNextOnly` – `ETrue` if only the next entry should be displayed.

inline TBool DisplayNextOnly() const This method returns ETrue if only the next entry is set to be displayed.
void SetRepeatForever(TBool aRepeatForever) This method sets whether the entry should repeat forever. aRepeatForever – ETrue if the entry should repeat forever.
inline TBool RepeatForever() const This method returns ETrue if the entry should repeat forever.
TUint InstanceCount() const This method returns the number of instances generated by the repeat algorithm.
TTime FindRptEndDate(TUint aCount) const This method calculates the repeat end date from the number of instances.

Class `TAgnException` Defined in agmexcpt.h
This class stores the date of an exception to a repeating entry. Exceptions can be added, removed, pruned and searched for using functions defined in class CAgnBasicEntry; the methods in the CAgnRptDef class should not be used directly.
Constructors
TAgnException() TAgnException(TAgnDate aDate) The default constructor creates an exception with a null date. aDate – the date to be set for the exception.
Member Methods
inline void SetDate(TAgnDate aDate) This method sets the date for an exception. aDate – the date to be set.
inline TAgnDate Date() const This method returns the date set for the exception.

12.6.3 Agenda Model Entries and Instances

Class `TAgnId` Defined in agmids.h
The base class for Agenda entry and to-do list identifier types. Internally, the identifier is stored as a 32-bit unsigned integer. Although it is not an abstract type, it is not normally used directly.

ID Methods

`inline void SetId(TAgnId aId)`
This method sets the ID.
`aId` – the identifier value to set.

`inline TAgnId Id() const`
This method returns the identifier.

`void SetNullId()`
This method sets the identifier to a null identifier value.

`TBool IsNullId() const`
This method returns `ETrue` if the identifier is set to a null value.

Class `TAgnEntryId` – public `TAgnId`
Defined in `agmids.h`

Uniquely identifies an Agenda entry in the database. It is assigned when the entry is added to the file, but can change if the entry is updated. Instances are uniquely identified by instance IDs (`TAgnInstanceId`) which consist of an entry identifier and a date.

Constructors

`TAgnEntryId()`
This default constructor sets the ID to a null value.

`TAgnEntryId(TAgnId aId)`
This constructor sets the ID.
`aId` – the identifier value to be used.

`typedef TUint16 TAgnDate`

An agenda model date. This is a number of days from the agenda model's minimum valid date (1 January 1980). It is used by the `TAgnDateTime` class, which represents both a date and a time in the Agenda Model, and by the `TAgnInstanceId`.

Class `TAgnInstanceId` – public `TAgnEntryId`
Defined in `agmids.h`

This class identifies an instance of an entry in the database using an entry identifier (`TAgnEntryId`) and an agenda date (`TAgnDate`).

Constructors

`TAgnInstanceId()`
This constructor sets the ID and date to null values.

```
TAgnInstanceId(TAgnEntryId aId,
               AgnDate aDate =AgnDateTime::NullDate())
```
This constructor sets the ID and date to provided values.
`aId` – the identifier value to be set.
`aDate` – the date value to be set.

```
TAgnInstanceId(TAgnEntryId aId,TTime aDate)
```
This constructor sets the ID and date to provided values.
`aId` – the identifier value to be set.
`aDate` – the date value to be set.

Member Methods

```
void SetNullInstanceId()
```
This method sets the ID to a null value.

```
inline TBool IsNullInstanceId() const
```
This method returns `ETrue` if the ID is a null value.

```
inline void SetDate(TAgnDate aDate)
```
This method sets the date for the instance.
`aDate` – the date value to be set.

```
inline TAgnDate Date() const
```
This method returns the date for the instance.

```
inline void SetIdAndDate(TAgnEntryId aId,TAgnDate aDate)
```
This method sets both the ID and date.
`aId` – the identifier value to be set.
`aDate` – the date value to be set.

Class `TAgnTodoListId` – public `TAgnId`
Defined in `agmids.h`

Unique identifier for a to-do list. The identifier is assigned to the to-do list when the list is added to the database, but it can change if the list is updated.

Constructors

```
TAgnTodoListId()
```
This default constructor sets a null ID value.

```
TAgnTodoListId(TAgnId aId)
```
This constructor sets an ID value.
`aId` – the ID value to be set.

enum TAgnStatus

- `EAccepted` – the entry has been accepted
- `ENeedsAction` – the entry needs action
- `ESent` – the entry has been sent
- `ETentative` – the entry is tentative
- `EConfirmed` – the entry has been confirmed
- `EDeclined` – the entry has been declined
- `ECompleted` – the entry has been completed
- `EDelegated` – the entry has been delegated

Class `CAgnBasicEntry`
Defined in `agmbasic.h`

Abstract base class for agenda entry types.
This class provides an interface to the alarm and repeat information and to the entry's attributes, for instance whether it is tentative or crossed out. The repeat information includes a list of repeat exceptions, which are dates that are excluded from an entry's normal repeat dates. For instance, if an entry is to repeat on the 4th of every month, but not on the 4th January, then the 4th January is an exception. All exceptions for a repeat are stored in an exception list, which is part of the entry's repeat details. Entries also have a last changed date/time, which is used for synchronization, and is updated each time the entry is changed.

Alarm Methods

`void SetHasAlarm(TBool aHasAlarm)`
This method sets the entry's alarm property.
`aHasAlarm` – `ETrue` if the entry is to have an alarm.

`TBool HasAlarm() const`
This method returns `ETrue` if the entry has an alarm set.

```
void SetAlarm(TTimeIntervalDays aDaysWarning,
              TTimeIntervalMinutes aTime)
```
This method set the alarm's due date/time. Two intervals may be set: the time interval in days between the entry's start date and the date on which the alarm will sound; and the time of day (as an interval in minutes from midnight) the alarm is set to sound.
Note that to-dos have their alarm warning period set relative to their due date by default. The period may also be set relative to the start date of the to-do, if this is required. `CAgnTodo::SetAlarmFromStartDate` should be called after calling this function.
`aDaysWarning` – the time interval in days between the entry's start date (for appointments, events and anniversaries) and the date on which the alarm will sound.
`aTime` – the time of day, as an interval in minutes from midnight, at which the alarm will sound.

`TTime AlarmInstanceDateTime() const`
This method returns the due date/time of the alarm for the current instance if the entry is repeating, or just its normal due date/time if it is non-repeating.

`TTimeIntervalDays AlarmDaysWarning() const`
This method returns the number of days' warning for the entry. It returns zero if the entry is not alarmed.

`TTimeIntervalMinutes AlarmTime() const`
This method returns the time of the alarm as a time interval in minutes from midnight. This time applies to all instances of the alarm if it is repeating, because they all occur at the same time on different dates.

Attribute Methods

`void SetIsCrossedOut(TBool aIsCrosedOut)`
This method sets the crossed-out attribute for the entry. Note that if the entry is a to-do list entry then call `CAgnTodo::CrossOut/UnCrossOut` instead.
`aIsCrossedOut` – `ETrue` if the entry is to be crossed out.

`inline TBool IsCrossedOut() const`
This method returns `ETrue` if the entry is crossed out.

`void SetIsADayNote(TBool aIsADayNote)`
This method sets the entry's day note attribute.
`aIsADayNote` – `ETrue` if the entry is to be a day note.

`inline TBool IsADayNote() const`
This method returns `ETrue` if the entry is a day note.

`void SetIsTentative(TBool aIsTentative)`
This method sets the tentative attribute.
`aIsTentative` – `ETrue` if the entry is tentative.

`inline TBool IsTentative() const`
This method returns `ETrue` if the entry is tentative.

Repeat Methods

`void SetIsRepeating(TBool aIsRepeating)`
This method sets the entry's repeating attribute.
`aIsRepeating` – `ETrue` if the entry is to be a repeating entry.

`inline TBool IsRepeating() const`
This method returns `ETrue` if the entry is a repeating entry.

`void ClearRepeat()`
This method unsets the entry's repeating attribute and clears the repeating details, including exceptions.

`void SetRptStartDate(TAgnDate aRptStartDate)`
This method sets the start date for the entry's repetition.
`aRptStartDate` – the start date for the repetition.

`void SetRptEndDate(TAgnDate aRptEndDate)`
This method sets the end date for the entry's repetition.
  `aRptEndDate` – the end date for the repetition.

`inline CAgnRptDef* RptDef()`
`inline const CAgnRptDef* RptDef() const`
These methods provide access to the entry's repeat definition.

`void SetHasExceptions(TBool aHasExceptions)`
This method sets the entry's 'has exceptions' attribute.
  `aHasExceptions` – `ETrue` if the entry has exceptions.

`inline TBool HasExceptions() const`
This method returns `ETrue` if the entry has exceptions.

`void AddExceptionL(const TAgnException& aException)`
This method adds an exception to the entry.
  `aException` – the exception to be added to the entry.

`TBool RemoveException(const TAgnException& aException)`
This method removes an exception from the entry. If there are no exceptions after this one is removed then the entry's 'has exceptions' attribute is unset.
  `aException` – the exception to be removed from the entry.

`TBool FindException(TAgnException& aException) const`
This method returns `ETrue` if the exception is found in the entry's list of exceptions.
  `aException` – the exception to find.

`const CAgnExceptionList* Exceptions() const`
This method provides access to the entry's list of exceptions.

`void RemoveAllExceptions()`
This method removes all exceptions from the entry and unsets its 'has exceptions' attribute.

`void PruneExceptions()`
This method compares the entry's exceptions with the repeat start and end dates. It removes any exceptions which occur before the start or after the end of the repeat period and exceptions which do not fall on repeat dates. If, after pruning, there are no exceptions left, the entry's 'has exceptions' property is unset.

`enum CAgnEntry::TType`

- `EAppt` – appointment
- `ETodo` – to-do
- `EEvent` – event
- `EAnniv` – anniversary

Class `CAgnEntry` – public `CAgnBasicEntry`
Defined in `agmentry.h`

This is the abstract base class for the agenda entry classes. `CAgnEntry` adds common entry information to `CAgnBasicEntry`.
Agenda entries are identified by an entry ID. Instances are identified by an instance ID, which consists of an entry ID and a date. IDs are used so that lists of entries can be efficiently processed and passed as function arguments and return values.
This class also lets you cast an entry or instance (which is retrieved into a `CAgnEntry` because its type is not yet known) to a pointer to the appropriate class. To find the type of an entry or instance, use `CAgnEntry::Type()`, which is implemented by each concrete entry class.

Type Methods

`virtual TType Type() const =0`
This pure virtual method returns the type of a derived object. When the derived type is known, the correct casting method can be called.

`const CAgnAppt* CastToAppt() const`
`CAgnAppt* CastToAppt()`
These methods cast an entry to an appointment entry.

`const CAgnTodo* CastToTodo() const`
`CAgnTodo* CastToTodo()`
These methods cast an entry to a to-do list entry.

`const CAgnEvent* CastToEvent() const`
`CAgnEvent* CastToEvent()`
These methods cast an entry to an event entry.

`const CAgnAnniv* CastToAnniv() const`
`CAgnAnniv* CastToAnniv()`
These methods cast an entry to an anniversary entry.

Identifier Methods

`inline void SetId(TAgnEntryId aId)`
This method sets the ID of an entry.
`aId` – the ID to set.

`inline TAgnEntryId EntryId() const`
This method returns the entry ID.

`inline void SetInstanceDate(TAgnDate aDate)`
This method sets the date for an instance.
`aDate` – the date to be set.

`inline TAgnDate InstanceDate() const`
This method returns the date of an instance.

`inline void SetInstanceId(const TAgnInstanceId& aInstanceId)`
This method sets the ID for an instance.
`aInstanceId` – the ID to set.

`inline TAgnInstanceId InstanceId() const`
This method returns the ID for an instance.

`inline void SetIdAndInstanceDate(TAgnEntryId aId,TAgnDate aDate)`
This method sets the date and ID for an instance.
`aId` – the ID to set.
`aDate` – the date to be set.

`inline void SetIdAndInstanceDate(const TAgnInstanceId& aInstanceId)`
This method sets the ID and date for an instance.
`aInstanceId` – the instance identifier to be set.

Methods Common to Multiple Types

`void SetRptDefL(const CAgnRptDef* aRptDef)`
This method sets the entry's repeat definition.
`aRptDef` – the repeat definition to be set.

`void MakeInstanceNonRepeating()`
This method converts an instance of a repeating entry into a non-repeating entry. The entry's start and end date/times (or due date for a to-do entry) are set according to the instance's start and end dates. If the entry is a day note, its display date/time is set according to the instance start date/time. If the entry is a to-do, its due date is set according to the instance end date.

`CRichText* RichTextL()`
This method provides access to the entry's text.

`void SetNotesTextL(HBufC* aNotes)`
This method sets the entry's notes text.
`aNotes` – the notes text to be set.

`const TDesC& NotesTextL() const`
This method provides access to the entry's notes text. If none is set then an empty descriptor is returned.

`virtual TTime InstanceStartDate() const = 0`
This method returns the start date and time for an instance of the entry.

`virtual TTime InstanceEndDate() const = 0`
This method returns the end date and time for an instance of the entry.

`virtual TTimeIntervalMinutes DisplayTime() const = 0`
This method returns the entry's display time.

```
void SetEventPriority(TUint aPriority)
```
This method sets the priority for an entry. It should not be called for a to-do list entry.
`aPriority` – entry priority: should be a value between 0 and 255.

```
TUint EventPriority() const
```
This method returns the priority for an entry.

Class `CAgnAppt` – public `CAgnEntry`
Defined in `agmentry.h`

This class represents an agenda appointment.
While other entry types have a start and end date (or due/crossed out date for to-dos), and a separate display time, appointments have a start date/time and an end date/time – the display time is inherent in the start date/time.
An appointment whose end date/time is the same as its start date/time is known as a day note. A day note has a date (the date component of its date/time) and a display time (the time component of the date/time). The display time can be used to tell the user interface where to display the day note in a view that contains times of day, such as in day or week views.

Construction Methods

```
static CAgnAppt* NewL(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
static CAgnAppt* NewLC(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
```
These methods create a new appointment entry.
`aParaFormatLayer` – pointer to the format layer on which the entry's rich text paragraph formatting will be based.
`aCharFormatLayer` – pointer to the format layer on which the entry's rich text character formatting will be based.
`aCreateHow` – this parameter is for testing purposes and should not be used; the default value is sufficient.

Date and Time Methods

```
void SetStartAndEndDateTime(
  const TTime& aStartDateTime,
  const TTime& aEndDateTime=Time::NullTTime())
```
This method sets the start and end date/time for the appointment. A null value for the end date/time means that the entry is a day note.
`aStartDateTime` – the start date and time.
`aEndDateTime` – the end date and time.

```
TTime StartDateTime() const
```
This method returns the start date and time for the appointment.

```
TTime EndDateTime() const
```
This method returns the end date and time for the appointment.

```
void SetDisplayTime(TTimeIntervalMinutes aDisplayTime)
```
This method sets the display time for day notes. This function should be called only for a day note. It also unsets the entry's 'has default display time' property.

`aDisplayTime` – the display time for the day note (minutes after midnight).

```
virtual TTimeIntervalMinutes DisplayTime() const
```
This method returns the display time for a day note. When called for an appointment that is not a day note, it returns the appointments start time.

Class `CAgnEvent` – public `CAgnEntry`
Defined in `agmentry.h`

This class represents an agenda event.
An event has a start date, an optional end date, and a display time. Events differ from appointments in that events have a start and end date, whereas appointments have a start and end date and time. Whereas an appointment can have a duration of anything from zero to 23 hours and 59 minutes, an event is always an all-day entry with no start or end times.

Construction Methods

```
static CAgnEvent* NewL(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
static CAgnEvent* NewLC(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
```
These methods create a new appointment entry.

`aParaFormatLayer` – pointer to the format layer on which the entry's rich text paragraph formatting will be based.

`aCharFormatLayer` – pointer to the format layer on which the entry's rich text character formatting will be based.

`aCreateHow` – this parameter is for testing purposes and should not be used; the default value is sufficient.

Date and Time Methods

```
void SetStartAndEndDate(
  const TTime& aStartDate,
  const TTime& aEndDate=Time::NullTTime())
```

This method sets the start and end date for the event. A null value for the end date means that the event lasts for only one day.
`aStartDate` – the start date.
`aEndDate` – the end date.

```
TTime StartDate() const
```
This method returns the event's start date.

```
TTime EndDate() const
```
This method returns the event's end date.

```
void SetDisplayTime(TTimeIntervalMinutes aDisplayTime)
```
This method sets the display time for the event.
`aDisplayTime` – the display time for the day note (minutes after midnight).

```
virtual TTimeIntervalMinutes DisplayTime() const
```
This method returns the display time for an event.

enum CAgnAnniv::TDisplayAs

- `ENone` – none of the following
- `EBaseYear` – displays the base year
- `EElapsedYears` – displays the years elapsed since the base year
- `EBaseAndElapsed` – displays both base and elapsed years

Class `CAgnAnniv` – public `CAgnEvent`
Defined in `agmentry.h`

This class represents an agenda anniversary.
An anniversary is an event which occurs once a year. It may be repeating, and if so, the repeat type must be annual. Its details include an optional base year, so that an anniversary can be displayed showing the number of years since that year (for example, how old someone is on this birthday) and/or showing the base year itself.

Construction Methods

```
static CAgnAnniv* NewL(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
static CAgnAnniv* NewLC(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
```

These methods create a new anniversary entry.

- `aParaFormatLayer` – pointer to the format layer on which the entry's rich text paragraph formatting will be based.
- `aCharFormatLayer` – pointer to the format layer on which the entry's rich text character formatting will be based.
- `aCreateHow` – this parameter is for testing purposes and should not be used; the default value is sufficient.

Year Methods

`inline void SetHasBaseYear(TBool aHasBaseYear)`
This method sets the anniversary to have a base year or not.
- `aHasBaseYear` – `ETrue` if the anniversary is to have a base year.

`inline TBool HasBaseYear() const`
This method returns `ETrue` if the anniversary has a base year.

`void SetBaseYear(TTimeIntervalYears aBaseYear)`
This method sets the base year for an anniversary. It also sets the 'has base year' attribute.
- `aBaseYear` – the base year to be set for the anniversary.

`inline TTimeIntervalYears BaseYear() const`
This method returns the base year for an anniversary.

`void SetDisplayAs(TDisplayAs aDisplayAs)`
This method sets how the anniversary is to be displayed.
- `aDisplayAs` – how the anniversary is to be displayed.

`inline TDisplayAs DisplayAs() const`
This method returns how the anniversary is to be displayed.

`enum CAgnTodo::TDisplayDueDateAs`

- `EAutomatic` – displays the due date as a date. When the current date is within one week of the due date, then the due date is displayed using the format 'Next Wed'.
- `EDate` – always displays the due date as a date.
- `EDays` – displays the due date as the number of days between the current date and the due date.
- `EDontDisplay` – never displays the due date.

Class `CAgnTodo` – public `CagnEntry`
Defined in `agmentry.h`

This class represents a to-do entry or instance.
A to-do entry represents a task or an action to be carried out. It has a display time, which allows it to be displayed in views that contain times of day, such as day or week views. Also, it may optionally have a start date (the first date on which it is displayed in day or week views)

<table>
<tr><td>and either a due date (to indicate the date by which the action should be carried out) or a crossed out date (the date the action was carried out). An undated to-do entry does not have a due date, a crossed out date or a start date. To-do details include the ID of the to-do list to which the to-do belongs and a priority.</td></tr>
<tr><td>Construction Methods</td></tr>
<tr><td><code>static CAgnTodo* NewL(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)
static CAgnTodo* NewLC(
  const CParaFormatLayer* aParaFormatLayer,
  const CCharFormatLayer* aCharFormatLayer,
  TCreateHow aCreateHow = ECreateRichText)</code></td></tr>
<tr><td>Member Methods</td></tr>
<tr><td><code>void SetTodoListId(TAgnTodoListId aTodoListId)</code>
This method sets the identifier of the to-do list that the entry belongs to.
  <code>aTodoListId</code> – the list identifier.</td></tr>
<tr><td><code>inline TAgnTodoListId TodoListId() const</code>
This method returns the identifier of the to-do list that the entry belongs to.</td></tr>
<tr><td><code>void SetPriority(TUint aPriority)</code>
This method sets the priority for the to-do list entry.
  <code>aPriority</code> – the priority to be set.</td></tr>
<tr><td><code>inline TUint Priority() const</code>
This method returns the entry priority.</td></tr>
<tr><td><code>void SetDisplayDueDateAs(TDisplayDueDateAs aDisplayAs)</code>
This method sets the display mode of the entry's due date.
  <code>aDisplayAs</code> – the display mode to be set.</td></tr>
<tr><td><code>inline TDisplayDueDateAs DisplayDueDateAs() const</code>
This method returns the display mode of the entry's due date.</td></tr>
<tr><td><code>void SetDisplayTime(TTimeIntervalMinutes aDisplayTime)</code>
This method sets the display time for the entry.
  <code>aDisplayTime</code> – the display time for the entry in minutes after midnight.</td></tr>
<tr><td><code>virtual TTimeIntervalMinutes DisplayTime() const</code>
This method returns the display time for the entry.</td></tr>
<tr><td><code>void SetDueDate(const TTime& aDueDate)</code>
This method sets the due date for the entry.
  <code>aDueDate</code> – the due date to be set.</td></tr>
</table>

`TTime DueDate() const` This method returns the due date for the entry.
`void CrossOut(const TTime& aDate)` This method sets the crossed out date for the entry, i.e. the date the action required for the entry was completed. `aDate` – the date the action was completed.
`void UnCrossOut()` This method sets the entry as uncrossed out.
`TTime CrossedOutDate() const` This method returns the crossed out date for the entry. If the entry is not crossed out then a NULL value is returned.
`void SetDuration(TTimeIntervalDays aDuration)` This method sets the entry's duration, i.e. the number of days between the start date and the due date. This function can be called on an entry if it is not repeating or on an individual instance if the entry is repeating `aDuration` – the duration to be set.
`inline TTimeIntervalDays Duration() const` This method returns the entry's duration.
`void SetAlarmFromDueDate()` Sets the alarm date to be a number of days from the entry's due date rather than its start date. This function should be called after calling `SetAlarm()`.
`void SetAlarmFromStartDate()` Sets the alarm date to be a number of days from the to-do entry's start date rather than its due date. This function should be called after calling `SetAlarm()`.
`inline TBool IsAlarmSetFromDueDate() const` This method returns `ETrue` if the alarm is set from the entry's due date.
`inline TBool IsAlarmSetFromStartDate() const` This method returns `ETrue` if the alarm is set from the entry's start date.
`TBool IsDated() const` This method returns `ETrue` if the entry has its due date set.
`void MakeUndated()` This method makes the entry undated, thus clearing most of its fields. This function sets the due date and crossed out date to NULL, sets the duration to zero and clears any alarm and repeat details.

12.6.4 List Classes

Class `CAgnList` – public `CBase`
template<class InstanceItem>
Defined in `agmlists.h`

This class represents a list of Agenda items. The type of item is specified by the user of the class.

Constructor

```
CAgnList()
```
This method creates a new, empty list.

Member Methods

```
inline TInt Count() const
```
This method returns the number of items in the list.

```
inline const InstanceItem& operator[](TInt aIndex) const
```
This operator returns an item from the list.
`aIndex` – the index of the item to be returned.

```
inline void Reset()
```
This method empties the list.

```
inline void Delete(TInt aIndex)
```
This method deletes an item from the list. A panic occurs if the index is out of range.
`aIndex` – the index of the item to be deleted.

```
inline void AppendL(const InstanceItem& aElement)
```
This method adds an item to the end of the list.
`aElement` – the item to be added to the end of the list.

```
inline void InsertL(TInt aIndex, const InstanceItem& aElement)
```
This method inserts an item into the list.
`aIndex` – the position at which to insert the item.
`aElement` – the item to be inserted.

```
inline TInt Find(const InstanceItem& aItem,
                 TKeyArrayFix& aKey,
                 TInt& aPos) const
```
This method returns the index of the first item that matches the key.
`aItem` – a reference to the item to be searched for.
`aKey` – a reference to a set of fields to be searched on.
`aPos` – if the element is found, the reference is set to the position of that element within the array. The position is relative to zero, i.e. the first element in the array is at position 0. If the element is not found and the array is not empty, then the value of the reference is set to the number of elements in the array. If the element is not found and the array is empty, then the reference is set to zero.
returns – zero if a match is found, otherwise a non-zero value.

Class CAgnDayList – public CAgnList<InstanceItem>
Defined in agmlists.h

This class represents a list of Agenda items for a single day.
Note that the instance date of an instance does not have to be the same as the date of the day list if the entry spans midnight.
The type of item is specified by the user of the class.

Construction Methods

```
inline static CAgnDayList* NewL(const TTime& aDay)
```
This method creates a new, empty list.
  aDay – the day for the day list.

Member Methods

```
inline void SetDay(const TTime& aDay)
```
This method sets the day for the day list.
  aDay – the day to be set.

```
inline TTime Day() const
```
This method returns the day for the day list.

Class CAgnDayDateTimeInstanceList –
public CAgnDayList<TAgnInstanceDateTimeId>
Defined in agmlists.h

This class represents a list of TAgnInstanceDateTimeId for one day.

Construction Method

```
static CAgnDayDateTimeInstanceList* NewL(const TTime& aDay)
```
This method creates a new, empty list.
  aDay – the day for the day list.

Class CAgnMonthInstanceList – public CAgnList<TAgnInstanceId>
Defined in agmlists.h

This class represents a list of instance IDs for a month.

Construction Method

```
static CAgnMonthInstanceList* NewL(TTimeIntervalYears aYear,
                                   TMonth aMonth)
```
This method creates a new, empty list.
  aYear – the year for the list.
  aMonth – the month for the list.

Member Methods

`inline void SetYear(TTimeIntervalYears aYear)`
This method sets the year for the list.
`aYear` – the year to be set.

`inline TTimeIntervalYears Year() const`
This method returns the year for the list.

`inline void SetMonth(TMonth aMonth)`
This method sets the month for the list.
`aMonth` – the month to be set.

`inline TMonth Month() const`
This method returns the month for the list.

`struct CAgnTodoListList::TListItem`
Defined in `agmtodos.h`

This structure represents an item in a list of to-do lists. Not all members are covered here.

Members

`TAgnTodoListId iTodoListId` – the identifier of the to-do list.

Class `CAgnTodoListList`
Defined in `agmtodos.h`

This class is a list of to-do list identifiers, each of which uniquely identifies a to-do list.

Construction Method

`static CAgnTodoListList* NewL()`
This method creates a new, empty list.

Member Methods

`const CAgnTodoListList::TListItem& operator[](TInt aIndex) const`
`CAgnTodoListList::TListItem& operator[](TInt aIndex)`
These operators provide access to an item from the list.
`aIndex` – index of the item required.

`TAgnTodoListId TodoListId(TInt aIndex) const`
This method returns the to-do list identifier from the item at the specified position.
`aIndex` – index of the item required.
returns – the to-do list identifier.

`inline TInt Count() const`
This method returns the number of items in the list.

`void Reset()`
This method empties the list.

`inline void AppendL(const TListItem* aItem)`
This method adds an item to the end of the list.
`aItem` – the item to be added.

`inline void InsertL(TInt aPosition, const TListItem* aItem)`
This method inserts a new item into the list.
`aPosition` – the position at which the item is to be inserted.
`aItem` – the item to be inserted.

`void Delete(TAgnTodoListId aTodoListId)`
`inline void Delete(TInt aPosition)`
These methods delete an item from the list, either by identifier or by position.
`aTodoListId` – the identifier of the list to be deleted.
`aPosition` – the position of the item to be deleted.

`TInt Find(TAgnTodoListId aTodoListId, TInt& aPos) const`
This method searches for an item with the specified to-do list identifier.
`aTodoListId` – the identifier of the list to be found.
`aPos` – on return, the position in the list of the matching item, if found.
returns – 0 if the item is found, 1 if it cannot be found, and – 1 if the list is empty.

`enum CAgnTodoList::TSortOrder`

- `EManual` – sort manually
- `EByDate` – sort by date
- `EByPriority` – sort by priority

Class `CagnTodoList`
Defined in `agmtodos.h`

This class represents a to-do list. This is a list of to-do entries, with a name, an identifier and display settings. The display settings include the sort order, whether to-do entries should be displayed in other views, and whether crossed out to-do entries should be displayed.
It should not be confused with a to-do instance list (`CAgnTodoInstanceList`), which is a filtered list of to-do instance IDs, representing a subset of a to-do list's entries that are to be displayed at a given time.

Construction Methods

`static CAgnTodoList* NewL()`
This method creates a new, empty list.

Member Methods

`inline TAgnTodoListId Id() const`
This method returns the list identifier.

`void SetName(const TDesC& aName)`
This method sets the name for the list.
`aName` – the name to be set.

`inline TPtrC Name() const`
This method returns the name for the list.

`void SetSortOrder(TSortOrder aSortOrder)`
This method sets the sort order for the list.
`aSortOrder` – the sort order to be set.

`inline TSortOrder SortOrder() const`
This method returns the sort order for the list.

`void SetDisplayCrossedOutEntries(TBool aDisplay)`
This method sets whether or not the list is to display crossed-out entries.
`aDisplay` – `ETrue` if crossed-out entries are to be displayed.

`inline TBool DisplayCrossedOutEntries() const`
This method returns `ETrue` if crossed-out entries are to be displayed.

Class `CAgnTodoInstanceList` – public `CAgnList<TAgnInstanceId>`
Defined in `agmlists.h`

This class represents a list of instance identifiers of to-do entries that belong to a particular to-do list. It stores the identifier of the to-do list and a sort order, which is used to sort the instance list.

Construction Method

```
static CAgnTodoInstanceList* NewL(
  TAgnTodoListId aTodoListId=AgnModel::NullTodoListId())
```
This method creates a new, empty list.
`aTodoListId` – the identifier for the list containing the entries.

Member Methods

`inline void SetTodoListId(TAgnTodoListId aTodoListId)`
This method sets the identifier for the to-do list containing the entries.
`aTodoListId` – the identifier for the list containing the entries.

`inline TAgnTodoListId TodoListId() const`
This method returns the identifier for the to-do list containing the entries.

`inline void SetSortOrder(CAgnTodoList::TSortOrder aSortOrder)` This method sets the sort order for the list. `aSortOrder` – the sort order to be used.
`inline CAgnTodoList::TSortOrder SortOrder() const` This method returns the sort order for the list.
`TInt Find(TAgnInstanceId aId,TInt& aPos)` This method returns the position in the list of an instance identifier. `aId` – the instance identifier to search for. `aPos` – on return, this is set to the position of the instance identifier, if found. returns – zero if the identifier was found, a positive number if the identifier was not found, and a negative number if the list is empty.
`inline const TAgnInstanceId& Id(TInt aIndex) const` This method returns the instance identifier at the specified position in the list. `aIndex` – the position in the list of the desired instance identifier. returns – the instance identifier at the desired position.

12.7 An Agenda Connectivity Service

We now need to plan the functions that we want our Agenda connectivity service to support. We are not trying to compete with PIM synchronization software – what we want to do is to provide a window on the Agenda data from the PC. This allows the user to keep one copy of their data on the Symbian OS mobile phone and to access it by the most convenient means at the time.

For reasons of practicality, we will confine our service to listing appointments, events and anniversaries, and to-do entries, and to creating and editing appointments and to-do entries. We will not try to handle the full range of repeating entries, simply for space reasons. If you want to create a more fully-featured service then add more functions.

12.7.1 Protocol Description

The functions that we are going to create are planned around a user interface, not a batch approach. Therefore, we will fetch all appointments and events for a specific day. We will not (at least for now) create functions to fetch entries for a monthly or yearly view, but we will retrieve to-do entries. We will not try to create new to-do lists, but we will provide functions to create new appointments, events or to-do entries.

This then allows us to propose a set of commands for our protocol and a set of responses. The following defines the command and response operation codes.

```
enum TRAgnCmdCode
  {
  ERAgnCmdNone = 100,      // Reserved for internal usage
  ERAgnCmdQueryVersion = 101, // Query version
  ERAgnCmdVersionReply = 102, // Version reply
  ERAgnCmdError = 103, // An error has occurred

  ERAgnCmdOpenAgenda = 110, // Open an agenda database
  ERAgnCmdOpenAgendaReply = 111, // An agenda database has been opened

  ERAgnCmdFetchApptsByDay = 120, // Fetch all appointments for a day
  ERAgnCmdFetchMoreAppts = 121, // Fetch more appointments
  ERAgnCmdFetchApptsReply = 122, // Reply including appointments
  ERAgnCmdCreateAppt = 123, // Create a new appointment or event
  ERAgnCmdCreateApptReply = 125, // Reply after creating an appointment
  ERAgnCmdEditAppt = 126, // Edit an appointment
  ERAgnCmdEditApptReply = 128, // Reply after editing an appointment

  ERAgnCmdFetchTodoLists = 130, // Fetch all To-do lists
  ERAgnCmdFetchTodoListsReply = 131, // Reply including To-do lists
  ERAgnCmdFetchOneTodoList = 132, // Fetch entries from one To-do list
  ERAgnCmdFetchMoreTodoListEntries = 133, // Fetch more To-dos
  ERAgnCmdFetchTodoListEntriesReply = 134, // Reply containing
                                           // To-do list entries
  ERAgnCmdCreateTodoListEntry = 135, // Create a new To-do list entry
  ERAgnCmdCreateTodoListEntryReply = 136, // Reply after creating a
                                          // To-do list entry
  ERAgnCmdEditTodoListEntry = 137, // Edit a To-do list entry
  ERAgnCmdEditTodoListEntryReply = 138, // Reply after editing a
                                        // To-do list entry

  ERAgnCmdDeleteInstance = 140, // Delete instance
  ERAgnCmdDeleteReply = 141 // Reply after deletion
  };
```

Some commands need no additional parameters or just one or more identifiers. Opening an Agenda database requires the database file name, and fetching appointments for a date requires the date. Creating or editing appointments, events or to-do entries requires additional data.

Just as with the other protocols we have defined, some of the responses include just a reply code while others include more data – notably the appointments, events and to-do entries replies.

Query Version Command

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdQueryVersion (=101)`
Transaction ID	Int32	PDU transaction identifier

Version Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdVersionReply (=102)`
Transaction ID	Int32	PDU transaction identifier
Major Version	Int32	Major version number
Minor Version	Int32	Minor version number
Build Number	Int32	Build number

Error Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdError (=103)`
Transaction ID	Int32	PDU transaction identifier
Error Code	Int32	Symbian OS error code

Open Agenda Database Command
This command includes the name of the agenda database to open.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdOpenAgenda (=110)`
Transaction ID	Int32	PDU transaction identifier
Name Length	Int16	Length in characters of file name
Name Text	Unicode data	File name

Open Agenda Database Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdOpenAgendaReply (=111)`
Transaction ID	Int32	PDU transaction identifier

Fetch Appointments By Day Command

This command includes a text string for which to retrieve appointments, events and anniversaries.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchApptsByDay (=120)`
Transaction ID	Int32	PDU transaction identifier
Day Text Length	Int16	Length in characters of date text
Day Text	ASCII data	Date text of the form YYYYMMDD, where MM and DD are zero based

Fetch More Appointments Command

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchMoreAppts (=121)`
Transaction ID	Int32	PDU transaction identifier

Fetch Appointments Reply

The returned appointments, events and anniversaries are instances rather than entries, so they are identified by a date and instance identifier. They include appropriate dates and times, a set of Boolean properties and the entry text. We are not going to include repeat details or notes text. Because multiple instances can be included, we take a value of −1 as the end of the list. Alarm details are included only if the alarm is set. Appointments have a start and end date and time, while events and anniversaries have a start and end date and a display time.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchApptsReply (=122)`
Transaction ID	Int32	PDU transaction identifier

. . .

Entry Type	Int32	The entry type (`EAppt`, `EEvent` or `EAnniv`)
Instance ID	Int32	Instance identifier
Instance Date	Date/Time	Instance date
Crossed Out	Int8	Boolean ‘crossed out’ flag
Tentative	Int8	Boolean tentative flag
Day Note	Int8	Boolean day-note flag

Repeat	Int8	Boolean repeat flag
Has Alarm	Int8	Boolean 'has alarm' flag

. . .

Alarm Days	Int16	Days before for alarm
Alarm Minutes	Int16	Minutes after midnight for alarm

. . .

Text Length	Int16	Length of text
Text	Unicode data	Text

. . . An appointment has a start and end date and time.

Start Date/Time	Date/Time	Start date and time
End Date/Time	Date/Time	End date and time

. . . An event or anniversary has a start and end date and a display time.

Start Date	Date/Time	Start date
End Date	Date/Time	End date
Display Time	Int16	Display time in minutes after midnight

. . .

−1	Int32	Terminating instance identifier

Create Appointment or Event

The appointment or event details include alarm details, but not repeat details. This allows the user to create a one-off appointment or event, but not a repetitive series. Alarm details are included only if they are required. Because appointments and events are very similar, they can be handled by one command type with an entry type and a display time that is included only for events.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdCreateAppt (=123)`
Transaction ID	Int32	PDU transaction identifier

Entry Type	Int32	Entry type – `EAppt` or `EEvent`
Start Date/Time	Date/Time	Start date and time for appointments or start date for events
End Date/Time	Date/Time	End date and time for appointments or start date for events

. . .

Display Time	Int16	Display time for events in minutes after midnight

. . .

Has Alarm	Int8	Boolean 'has alarm' flag

. . .

Alarm Days	Int16	Days before for alarm
Alarm Minutes	Int16	Minutes after midnight for alarm

. . .

Crossed Out	Int8	Boolean 'crossed out' flag
Tentative	Int8	Boolean tentative flag
Day Note	Int8	Boolean day-note flag
Text Length	Int16	Length in characters of text
Text	Unicode data	Text

Create Appointments Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdCreateApptReply (=125)`
Transaction ID	Int32	PDU transaction identifier
Instance ID	Int32	Identifier for newly created appointment

Edit Appointment or Event

The appointment or event details include alarm details but not repeat details. However, the instance being edited may already be part of a series. Therefore, we include the ability to edit one instance or more of a series. We do not want to set all members if not necessary, so we include flags for setting or not the times and alarms and the start and end times, and alarm details are included only if they are being set. The instance type is not contained in the command – it is found from the instance itself. Events have a display time and start and end dates, while appointments have start and end dates and times, otherwise their data is common.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdEditAppt (=126)`
Transaction ID	Int32	PDU transaction identifier
Instance ID	Int32	Instance identifier
Instance Date	Date/Time	Instance date
Which Instances	Int32	'Which instances' enum value
Edit Times	Int8	Boolean 'edit times' flag

. . .

Start Date/Time	Date/Time	Start date and time for appointments or start date for events
End Date/Time	Date/Time	End date and time for appointments or start date for events

. . .

Display Time	Int16	Display time in minutes after midnight – only for events

. . .

Edit Alarm	Int8	Boolean 'edit alarm' flag

. . .

Has Alarm	Int8	Boolean 'has alarm' flag

. . .

Alarm Days	Int16	Days before for alarm
Alarm Minutes	Int16	Minutes after midnight for alarm

. . .

Crossed Out	Int8	Boolean 'crossed out' flag
Tentative	Int8	Boolean tentative flag
Day Note	Int8	Boolean day-note flag
Edit Text	Int8	Boolean 'edit text' flag

. . .

Text Length	Int16	Length in characters of text
Text	Unicode data	Text

Edit Appointments Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdEditApptReply (=128)`
Transaction ID	Int32	PDU transaction identifier
Instance ID	Int32	Identifier for edited appointment (may be zero if it has not been necessary to create a new sequence of appointments)

Fetch All To-do Lists Command

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchTodoLists (=130)`
Transaction ID	Int32	PDU transaction identifier

Fetch To-do Lists Reply

The returned lists include their names, whether or not they display crossed-out entries, and sort order. The actual entries are not included. Because multiple lists may be included, the list is terminated with a list identifier of −1.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchTodoListsReply (=131)`
Transaction ID	Int32	PDU transaction identifier

. . .

List ID	Int32	List identifier
Display Crossed Out	Int8	Boolean 'display crossed out' flag
Sort Order	Int32	Sort order
Name Length	Int16	Length in characters of list name text
Name Text	Unicode data	Name text

. . .

−1	Int32	Terminating instance identifier

Fetch To-do Entries Command

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchOneTodoList (=132)`
Transaction ID	Int32	PDU transaction identifier
List ID	Int32	Identifier of the list

Fetch More To-do Entries Command

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchMoreTodoListEntries (=133)`
Transaction ID	Int32	PDU transaction identifier

Fetch To-do Entries Reply
The returned to-do entries include the owning list identifier, instance date and identifier, priority, due date and entry text. Because multiple instances can be included, we take a list identifier of – 1 as the end of the list.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdFetchTodoListEntriesReply (=134)`
Transaction ID	Int32	PDU transaction identifier

. . .

List ID	Int32	Owning list identifier
Instance ID	Int32	Instance identifier
Instance Date	Date/Time	Instance date
Priority	Int32	Entry priority
Due Date	Date/Time	Due date and time
Text Length	Int16	Length in characters of text
Text	Unicode data	Text

. . .

−1	Int32	Terminating list identifier

Create To-do Entry

The appointment details include alarm details, but not repeat details. This allows the user to create a one-off appointment, but not a repetitive series. Alarm details are included only if they are required.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdCreateTodoListEntry (=135)`
Transaction ID	Int32	PDU transaction identifier
Due Date/Time	Date/Time	Due date and time
Priority	Int32	Priority
Text Length	Int16	Length in characters of text
Text	Unicode data	Text

Create To-do Entry Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdCreateTodoListEntryReply (=136)`
Transaction ID	Int32	PDU transaction identifier
Instance ID	Int32	Identifier for newly created entry

Edit To-do Entry

We do not want to set all members if not necessary, so we include flags for setting or not the times and alarms, and the start and end times and alarm details are included only if they are being set.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdEditTodoListEntry (=137)`
Transaction ID	Int32	PDU transaction identifier
Edit Due Date	Int8	Boolean 'edit due date' flag

. . .

Due Date/Time	Date/Time	Due date and time

. . .

Priority	Int32	Priority
Edit Text	Int8	Boolean 'edit text' flag

. . .

Text Length	Int16	Length in characters of text
Text	Unicode data	Text

Edit To-do Entry Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdEditTodoListEntryReply (=138)`
Transaction ID	Int32	PDU transaction identifier
Instance ID	Int32	Identifier for edited appointment (may be zero if it has not been necessary to create a new sequence of appointments)

Delete Instance Command

The same command can delete any type of instance.

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdDeleteInstance (=140)`

Transaction ID	Int32	PDU transaction identifier
Instance ID	Int32	Instance identifier
Instance Date	Date/Time	Instance date

Delete Instance Reply

Field	Type	Meaning
Opcode	Int32	`ERAgnCmdDeleteReply (=141)`
Transaction ID	Int32	PDU transaction identifier
New Entry Flag	Int32	Boolean flag indicating whether a new entry has been created due to a repeating entry being split.

12.7.2 Opening the Agenda Database

In this section we see some code to access the Agenda model. This code builds on custom servers or socket servers and the routines to pack or unpack data used in previous chapters.

We create an instance of the Agenda Server and the Agenda Model, link the two and open an Agenda Database.

```
class CRAgnUtil : public CBase, public MAgnProgressCallBack
{
public:
  static CRAgnUtil *NewL(const TDesC& aFileName);
  CRAgnUtil();
  void ConstructL(const TDesC& aFileName);
  virtual ~CRAgnUtil();

// For MAgnProgressCallBack
public:
  virtual void Progress(TInt aPercentageCompleted);
  virtual void Completed(TInt aError = KErrNone);

// Service Routines
public:
...

// Member Variables
private:
  RAgendaServ* iAgnSrv;
  CAgnModel* iModel;

  CAgnDayList<TAgnInstanceId>* iApptsList;
  TAgnFilter iApptFilter;
  TInt iApptArrayIndex;
```

```
  CAgnTodoInstanceList* iTodoListArray;

  HBufC* iPlainTextBuff;
};
```

```
void CRAgnUtil::ConstructL(const TDesC& aFileName)
  {
  iAgnSrv = RAgendaServ::NewL(); // allocate and construct server
  iAgnSrv->Connect(); // connect to the server
  iModel = CAgnModel::NewL(NULL); // allocate and construct model
  iModel->SetServer(iAgnSrv); // set server pointer for model
  TTimeIntervalMinutes zeroTime = TTimeIntervalMinutes();
  iModel->OpenL(aFileName, zeroTime, zeroTime, zeroTime,
    this, ETrue); // Open file using server

  // Set up a list and filter to return appointments, events
  // and anniversaries
  TTime nullDate(0);
  iApptsList = CAgnDayList<TAgnInstanceId>::NewL(nullDate);
  iApptFilter.SetIncludeTodos(EFalse);
  iPlainTextBuff = HBufC::NewL(KMaxTextLen);

  iTodoListArray = CAgnTodoInstanceList::NewL();
  iTodoListArray->SetSortOrder(CAgnTodoList::EByDate);
  }

// Routines for MAgnProgressCallBack
void CRAgnUtil::Progress(TInt aPercentageCompleted)
  {
  }

void CRAgnUtil::Completed(TInt aError)
  {
  }
```

12.7.3 Retrieving Entry Data

Assuming that we want to display appointments, events and anniversaries for one day, we set up a `TTime` object and use that to populate a list of appointments. We use the filter set up in the constructor to not receive to-do entries. We could set up alternate filters for different types of entries, but it is easier to just accept appointments, events and anniversaries and handle them based on their types. As in previous chapters, we assume that we may have more entries than will fit in a buffer and allow them to be retrieved in more than one iteration.

```
// Fetch all entries for one day
void CRAgnUtil::FetchApptsByDayL(TTime aDay, TDes8 &aBuffer,
                                 TDes8 &aTempBuffer)
  {
  iApptsList->Reset();
  iApptsList->SetDay(aDay);
  iModel->PopulateDayInstanceListL(iApptsList, iApptFilter, aDay);
```

```
  iApptArrayIndex = 0;
  FetchMoreApptsL(aBuffer, aTempBuffer);
  }

// Fetch more pending appointments
void CRAgnUtil::FetchMoreApptsL(TDes8 &aBuffer, TDes8 &aTempBuffer)
  {
  while((iApptArrayIndex >= 0) && (iApptArrayIndex <
                                   iApptsList->Count()))
    {
    aTempBuffer.Zero();
    GetOneApptL((*iApptsList)[iApptArrayIndex], aTempBuffer);

    // The entry may not have been useful so check that data has been
    // added
    if( aTempBuffer.Length() > 0)
    { // If the data will fit then append it, otherwise break out
      if(aTempBuffer.Length()+aBuffer.Length()+4 < aBuffer.MaxLength())
        {
        CConnPack::WriteBufferL(aTempBuffer, aBuffer);
        ++iApptArrayIndex;
        }
      else
        { // Run out of room - leave it till next PDU
        break;
        }// endif will fit
      }//endif got non-zero data
    }//endwhile
    CConnPack::WriteInt32L(-1, aBuffer);
  }
```

The code to fetch the details of one appointment, event or anniversary instance is straightforward and largely consists of writing out member data. We check the type before casting. Most of the member data is common, but the times are slightly different based on type. We also check in case we have an unexpected type. If the instance is not an expected type then no action is taken – in a debug implementation we could raise a panic, because this would indicate a coding error. Because we are working with instances rather than entries, we need to output the instance date as well as the instance identifier.

```
void CRAgnUtil::GetOneApptL(const TAgnInstanceId &aId, TDes8 &aBuffer)
  {
  TAgnEntryId entryId(aId.Id());
  CAgnEntry* entry = iModel->FetchEntryL(entryId);
  CleanupStack::PushL(entry);

  if((entry->Type() != CAgnEntry::EAppt) &&
     (entry->Type() != CAgnEntry::EEvent) &&
     (entry->Type() != CAgnEntry::EAnniv))
    {

    CleanupStack::PopAndDestroy(entry);
```

```
    return;
    }

  // Write general instance details for all supported types
  CConnPack::WriteInt32L((TInt32)entry->Type(), aBuffer);
  CConnPack::WriteInt32L((TInt32)aId.Value(), aBuffer);
  CConnPack::WriteTTimeL(AgnDateTime::AgnDateToTTime(aId.Date()),
                         aBuffer);

  // Write general properties
  CConnPack::WriteInt8L((TInt8)entry->IsCrossedOut(), aBuffer);
  CConnPack::WriteInt8L((TInt8)entry->IsTentative(), aBuffer);
  CConnPack::WriteInt8L((TInt8)entry->IsADayNote(), aBuffer);
  // Write Repeat properties - we only indicate if it is - no
  // more detail
  CConnPack::WriteInt8L((TInt8)entry->IsRepeating(), aBuffer);

  CConnPack::WriteInt8L((TInt8)entry->HasAlarm(), aBuffer);
  if(entry->HasAlarm())
    {
    CConnPack::WriteInt16L((TInt16)entry->AlarmDaysWarning().Int(),
                           aBuffer);
    CConnPack::WriteInt16L((TInt16)entry->AlarmTime().Int(), aBuffer);
    }

  // Write the entry rich text
  TPtr tempPtr(iPlainTextBuff->Des());
  entry->RichTextL()->Extract(tempPtr);
  CConnPack::WriteUNCDataL(tempPtr, aBuffer);

  if(entry->Type() == CAgnEntry::EAppt)
    {
    CAgnAppt* appt = entry->CastToAppt() ;
    // Write the start and end date and time
    CConnPack::WriteTTimeL(appt->StartDateTime(), aBuffer);
    CConnPack::WriteTTimeL(appt->EndDateTime(), aBuffer);
    }
  else
    {// Anniversaries derive from events
    CAgnEvent* event = entry->CastToEvent() ;
    // Write the start and end date and the display time
    CConnPack::WriteTTimeL(event->StartDate(), aBuffer);
    CConnPack::WriteTTimeL(event->EndDate(), aBuffer);
    CConnPack::WriteInt16L((TInt16)event->DisplayTime().Int(), aBuffer);
    }

  CleanupStack::PopAndDestroy(entry);
  }
```

12.7.4 Retrieving To-do Data

In order to provide the full set of to-do lists, we can simply use the `CAgnEntryModel::PeekAtTodoList` method. All we want to provide is the list identifier, name and a couple of attributes. In fact, because most users will not have many to-do lists and are unlikely to have hundreds of entries in the lists (unless they are way behind with their work), we could have just returned all to-do list entries as well, but this split

is neater and retrieving one extra packet should not slow things down too much.

```
void CRAgnUtil::FetchTodoListsL(TDes8 &aBuffer)
  {
  Tint listCount = iModel->PeekAtTodoListList()->Count()
  for(TInt listIndex = 0 ; listIndex < listCount ; ++listIndex)
    {
    Tint listId = iModel->PeekAtTodoListList()->TodoListId(listIndex);
    CAgnTodoList *list = iModel->FetchTodoListL(listId);
    CleanupStack::PushL(list);

    CConnPack::WriteInt32L(list->Id().Value(), aBuffer);
    CConnPack::WriteInt8L((TInt8)list->DisplayCrossedOutEntries(),
                          aBuffer);
    CConnPack::WriteInt32L(list->SortOrder(), aBuffer);
    TPtrC listName = list->Name();
    CConnPack::WriteUNCDataL(listName, aBuffer);

    CleanupStack::PopAndDestroy(list);
    }
  CConnPack::WriteInt32L(-1, aBuffer);
  }
```

Given a to-do list, retrieving the entries uses the `CAgnModel::PopulateTodoInstanceList` method.

```
void CRAgnUtil::FetchOneTodoListL(TInt aListId, TDes8 &aBuffer,
                                  TDes8 &aTempBuffer)
  {
  TAgnTodoListId listId(aListId);
  TTime todoDate;
  todoDate.HomeTime();

  iTodoListArray->Reset();
  iTodoListArray->SetTodoListId(listId);
  iModel->PopulateTodoInstanceListL(iTodoListArray, todoDate);

  FetchMoreTodoListEntriesL(aBuffer, aTempBuffer);
  }

// Fetch more Todos
void CRAgnUtil::FetchMoreTodoListEntriesL(TDes8 &aBuffer,
                                          TDes8 &aTempBuffer)
  {
  while(iTodoListArray->Count() > 0)
    {
    aTempBuffer.Zero();
    GetOneTodoListEntryL(iTodoListArray->TodoListId().Value(),
                         (*iTodoListArray)[0], aTempBuffer);

    // The entry may not have been useful so check that data has been
    // added
    if( aTempBuffer.Length() > 0)
      {
```

```
     // If the data will fit then append it, otherwise break out
     if(aTempBuffer.Length()+aBuffer.Length()+4 < aBuffer.MaxLength())
       {
       CConnPack::WriteBufferL(aTempBuffer, aBuffer);
       iTodoListArray->Delete(0);
       }
     else
       { // Run out of room - leave it till next PDU
       break;
       }// endif will fit
     }//endif got non-zero data

   }//endwhile
   CConnPack::WriteInt32L(-1, aBuffer);
  }
```

Once again, we just write member data to the buffer, remembering to check the type of the instance before casting it.

```
void CRAgnUtil::GetOneTodoListEntryL(const TInt aTodoListId,
                                     const TAgnInstanceId &aId,
                                     TDes8 &aBuffer)
  {
  CAgnEntry* item = iModel->FetchInstanceL(aId);
  CleanupStack::PushL(item);

  if(item->Type() == CAgnEntry::ETodo)
    {
    CConnPack::WriteInt32L(aTodoListId, aBuffer);
    CAgnTodo* todo = item->CastToTodo() ;
    CConnPack::WriteInt32L(aId.Value(), aBuffer);
    CConnPack::WriteTTime(AgnDateTime::AgnDateToTTime(aId.Date()),
                          aBuffer);

    CConnPack::WriteInt32L(todo->Priority(), aBuffer);
    TTime dueDate = todo->DueDate();
    CConnPack::WriteTTime(dueDate, aBuffer);

    TPtr tempPtr(iPlainTextBuff->Des());
    todo->RichTextL()->Extract(tempPtr);
    CConnPack::WriteUNCDataL(tempPtr, aBuffer);
    }//endif a Todo entry

  CleanupStack::PopAndDestroy(item);
  }
```

12.7.5 Creating, Editing and Deleting Entries

Creating a new appointment or event requires us to create a new object and set its attributes before adding it to the database. Agenda model entries do not need anything unusual to create. Appointments and events differ in their times, so we can create them in one routine with the differences handled.

```
TInt CRAgnUtil::CreateApptL(TDes8 &aBuffer)
  {
  CAgnEntry::TType entryType =
                   (CAgnEntry::TType)CConnPack::ReadInt32L(aBuffer);
  // We need a base entry
  CAgnEntry* newEntry = NULL;

  if((entryType != CAgnEntry::EAppt) &&
                   (entryType != CAgnEntry::EEvent))
    {
    User::Leave(KErrArgument);
    }

  if(entryType == CAgnEntry::EAppt)
    {
    newEntry = CAgnAppt::NewL(iModel->ParaFormatLayer(),
                              iModel->CharFormatLayer());
    CleanupStack::PushL(newEntry);
    CAgnAppt*appt = newEntry->CastToAppt();
    // Read and set start end end times
    TTime startTime = CConnPack::ReadTTimeL(aBuffer);
    TTime endTime = CConnPack::ReadTTimeL(aBuffer);
    appt->SetStartAndEndDateTime( startTime, endTime);
    }
  else if(entryType == CAgnEntry::EEvent)
    {
    newEntry = CAgnEvent::NewL(iModel->ParaFormatLayer(),
                               iModel->CharFormatLayer());
    CleanupStack::PushL(newEntry);
    CAgnEvent* event = newEntry->CastToEvent();
    // Read and set start and end date
    TTime startTime = CConnPack::ReadTTimeL(aBuffer);
    TTime endTime = CConnPack::ReadTTimeL(aBuffer);
    event->SetStartAndEndDate( startTime, endTime);
    TInt displayTime = (TInt)CConnPack::ReadInt16L(aBuffer);
    TTimeIntervalMinutes displayInterval(displayTime);
    event->SetDisplayTime(displayInterval);
    }

  // Read and set alarm details
  TInt8 hasAlarm = CConnPack::ReadInt8L(aBuffer);
  if(hasAlarm != 0)
    {
    newEntry->SetHasAlarm(ETrue);
    TInt alarmDays = (TInt16)CConnPack::ReadInt16L(aBuffer);
    TTimeIntervalDays alarmIntervalDays(alarmDays);
    TInt alarmMinutes = (TInt16)CConnPack::ReadInt16L(aBuffer);
    TTimeIntervalMinutes alarmIntervalMinutes(alarmMinutes);
    newEntry->SetAlarm(alarmIntervalDays, alarmIntervalMinutes);
    }
  else
    {
    newEntry->SetHasAlarm(EFalse);
    }

  // Set other properties
  TInt8 isCrossedOut = CConnPack::ReadInt8L(aBuffer);
  TBool bIsCrossedOut = (isCrossedOut!=0);
```

```
  newEntry->SetIsCrossedOut(bIsCrossedOut);
  TInt8 isTentative = CConnPack::ReadInt8L(aBuffer);
  TBool bIsTentative = (isTentative!=0);
  newEntry->SetIsTentative(bIsTentative);
  Tin8t isDayNote = CConnPack::ReadInt8L(aBuffer);
  TBool bIsDayNote = (isDayNote!=0);
  newEntry->SetIsADayNote(bIsDayNote);

  TInt textLength = CConnPack::PeekUint16L(aBuffer);
  HBufC* textBuff = HBufC::NewLC(textLength);
  TPtr tempPtr(textBuff->Des());
  CConnPack::ReadUNCDataL(tempPtr, aBuffer);
  newEntry->RichTextL()->InsertL(0,textBuff->Des());
  CleanupStack::PopAndDestroy(textBuff);

  TAgnEntryId newId = iModel->AddEntryL(newEntry, AgnModel::NullId());

  CleanupStack::PopAndDestroy(newEntry);
  return newId.Value();
  }
```

Editing an appointment uses the `FetchInstanceL()` and `Update-InstanceL()` methods, but most of its code is again concerned with unpacking data from the buffer. As with creating appointments and events, most of the behavior is common, so one routine can handle both types.

```
TInt CRAgnUtil::EditApptL(TDes8 &aBuffer)
  {
  TInt editId = CConnPack::ReadInt32L(aBuffer);
  TTime editTime = CConnPack::ReadTTimeL(aBuffer);

  TAgnWhichInstances whichInstances =
    (TAgnWhichInstances)CConnPack::ReadInt32L(aBuffer);
  TAgnEntryId editEntryId(editId);
  TAgnInstanceId editInstance(editEntryId,
                              AgnDateTime::TTimeToAgnDate(editTime));
  CAgnEntry* editEntry = iModel->FetchInstanceLC(editInstance);
  CAgnEntry::TType instanceType = editEntry->Type();

  TAgnEntryId newId;
  newId.SetNullId();
  if(instanceType == CAgnEntry::EAppt)
    {
    CAgnAppt* appt = editEntry->CastToAppt();

    TInt8 editTime = CConnPack::ReadInt8L(aBuffer);
    if(editTime != 0)
      { // Read and set start end end date and time
      TTime startTime = CConnPack::ReadTTimeL(aBuffer);
      TTime endTime = CConnPack::ReadTTimeL(aBuffer);
      appt->SetStartAndEndDateTime( startTime, endTime);
      }
    }
  else if((instanceType == CAgnEntry::EEvent) ||
          (instanceType == CAgnEntry::EAnniv))
```

```
  {
  CAgnEvent* event = editEntry->CastToEvent();

  TInt8 editTime = CConnPack::ReadInt8L(aBuffer);
  if(editTime != 0)
    {
    // Read and set start and end date
    TTime startTime = CConnPack::ReadTTimeL(aBuffer);
    TTime endTime = CConnPack::ReadTTimeL(aBuffer);
    event->SetStartAndEndDate( startTime, endTime);
    TInt displayTime = (TInt)CConnPack::ReadInt16L(aBuffer);
    TTimeIntervalMinutes displayInterval(displayTime);
    event->SetDisplayTime(displayInterval);
    }
  }

TInt8 editAlarm = CConnPack::ReadInt8L(aBuffer);
if(editAlarm != 0)
  { // Read and set alarm details
  TInt8 hasAlarm = CConnPack::ReadInt8L(aBuffer);
  if(hasAlarm != 0)
    {
    editEntry->SetHasAlarm(ETrue);
    TInt alarmDays = (TInt)CConnPack::ReadInt16L(aBuffer);
    TTimeIntervalDays alarmIntervalDays(alarmDays);
    TInt alarmMinutes = (TInt)CConnPack::ReadInt16L(aBuffer);
    TTimeIntervalMinutes alarmIntervalMinutes(alarmMinutes);
    editEntry->SetAlarm(alarmIntervalDays, alarmIntervalMinutes);
    }
  else
    {
    editEntry->SetHasAlarm(EFalse);
    }
  }//endif editing alarm details

// Set other properties
TInt8 isCrossedOut = CConnPack::ReadInt8L(aBuffer);
TBool bIsCrossedOut = (isCrossedOut!=0);
editEntry->SetIsCrossedOut(bIsCrossedOut);
TInt8 isTentative = CConnPack::ReadInt8L(aBuffer);
TBool bIsTentative = (isTentative!=0);
editEntry->SetIsTentative(bIsTentative);
TInt8 isDayNote = CConnPack::ReadInt8L(aBuffer);
TBool bIsDayNote = (isDayNote!=0);
editEntry->SetIsADayNote(bIsDayNote);

TInt8 editText = CConnPack::ReadInt8L(aBuffer);
if(editText)
  {
  TInt textLength = CConnPack::PeekUint16L(aBuffer);
  HBufC* textBuff = HBufC::NewLC(textLength);
  TPtr tempPtr(textBuff->Des());
  CConnPack::ReadUNCDataL(tempPtr, aBuffer);
  editEntry->RichTextL()->InsertL(0,textBuff->Des());
  CleanupStack::PopAndDestroy(textBuff);
  }

newId = iModel->UpdateInstanceL(editEntry, whichInstances);
```

```
   CleanupStack::PopAndDestroy(editEntry);

  return newId.Value();
  }
```

To-do entries are created and edited in the same way as appointments.

```
TInt CRAgnUtil::CreateTodoListEntryL(TDes8 &aBuffer)
  {
  CAgnEntry* newEntry = CAgnTodo::NewL(iModel->ParaFormatLayer(),
                                       iModel->CharFormatLayer());
  CleanupStack::PushL(newEntry);
  CAgnTodo* todo = newEntry->CastToTodo();

  // Read and set the owning To-do list
  TInt listIdValue = CConnPack::ReadInt32L(aBuffer);
  TAgnTodoListId listId(listIdValue);
  todo->SetTodoListId(listId);

  // Read and set due date
  TTime dueDate = CConnPack::ReadTTime(aBuffer);
  todo->SetDueDate(dueDate);

  // Read and set priority
  TUint priority = CConnPack::ReadInt32L(aBuffer);
  todo->SetPriority(priority);

  TInt textLength = CConnPack::PeekUint16L(aBuffer);
  HBufC* textBuff = HBufC::NewLC(textLength);
  TPtr tempPtr(textBuff->Des());
  CConnPack::ReadUNCDataL(tempPtr, aBuffer);
  newEntry->RichTextL()->InsertL(0,textBuff->Des());
  CleanupStack::PopAndDestroy(textBuff);

  TAgnEntryId newId = iModel->AddEntryL(newEntry, AgnModel::NullId());

  CleanupStack::PopAndDestroy(newEntry);
  return newId.Value();
  }

TInt CRAgnUtil::EditTodoListEntryL(TDes8 &aBuffer)
  {
  TInt editId = CConnPack::ReadInt32L(aBuffer);
  TTime editTime = CConnPack::ReadTTime(aBuffer);

  TAgnWhichInstances whichInstances =
    (TAgnWhichInstances)CConnPack::ReadInt32L(aBuffer);
  TAgnEntryId editEntryId(editId);
  TAgnInstanceId editInstance(editEntryId,
                              AgnDateTime::TTimeToAgnDate(editTime));
  CAgnEntry* editEntry = iModel->FetchInstanceLC(editInstance);

  TAgnEntryId newId;
  newId.SetNullId();
  if(editEntry->Type() == CAgnEntry::ETodo)
```

```
    {
    CAgnTodo* todo = editEntry->CastToTodo();

    TInt8 editTime = CConnPack::ReadInt8L(aBuffer);
    if(editTime != 0)
      { // Read and set due date
      TTime dueDate = CConnPack::ReadTTime(aBuffer);
      todo->SetDueDate(dueDate);
      }

    // Read and set priority
    TUint priority = CConnPack::ReadInt32L(aBuffer);
    todo->SetPriority(priority);

    TInt8 editText = CConnPack::ReadInt8L(aBuffer);
    if(editText != 0)
      {
      TInt textLength = CConnPack::PeekUint16L(aBuffer);
      HBufC* textBuff = HBufC::NewLC(textLength);
      TPtr tempPtr(textBuff->Des());
      CConnPack::ReadUNCDataL(tempPtr, aBuffer);
      editEntry->RichTextL()->InsertL(0,textBuff->Des());
      CleanupStack::PopAndDestroy(textBuff);
      }

    newId = iModel->UpdateInstanceL(editEntry, whichInstances);
    }

  CleanupStack::PopAndDestroy(editEntry);

  return newId.Value();
  }
```

An instance can be deleted without caring what type it is – all we need is the instance date and identifier. If the instance being deleted is part of a repeating series then deleting it may split the repeat into two separate entries.

```
TBool CRAgnUtil::DeleteInstanceL(TDes8 &aBuffer)
  {
  TInt deleteId = CConnPack::ReadInt32L(aBuffer);
  TTime deleteTime = CConnPack::ReadTTime(aBuffer);

  TAgnWhichInstances whichInstances =
    (TAgnWhichInstances)CConnPack::ReadInt32L(aBuffer);
  TAgnEntryId delId(deleteId);
  TAgnInstanceId deleteEntry(delId,
                             AgnDateTime::TTimeToAgnDate(deleteTime));
  TAgnEntryId newId = iModel->DeleteInstanceL(deleteEntry,
                                              whichInstances);
  TBool hasNewId = newId.IsNullId();
  return hasNewId;
  }
```

13

Developing a Specialized Connectivity GUI Application

13.1 What is Special About a GUI Application?

In this chapter, we will investigate the issues involved in developing a GUI application to use specialized PC Connectivity services. We will use as examples applications to interface to Contacts, Agenda and SMS Messaging services, but we will also try to cover design aspects that could apply to other applications.

Before we look at the issues and design approaches relevant to GUI applications, it is worth touching briefly on the differences between a GUI application and other types of application.

The major difference is that a GUI application is designed to be used directly by a user; the sequence of actions is unpredictable and the user expects a prompt response. In contrast, a batch-type synchronization application could have a much more predictable sequence of actions and would be less sensitive to delays. For these reasons, a GUI application will tend to be event driven and will place a premium on minimizing perceived delays.

13.2 Managing Connections to Phones

Just as with the file browser that we created earlier, we need to start by managing the connection to the Symbian OS smartphone. This time we will use BAL rather than SCOM. We do not plan to use any of the file access methods, just the manufacturer, model and identification and the ability to start a service. We could get these from SCOM, but the extra functionality that we do not need will incur an overhead. If we wanted any of the functions from SCOM then we could just use the `ISCDevice.OpenDeviceService` to access the services on the Symbian OS smartphone instead of using BAL directly.

The first thing we will need is an interface that we can call whenever a phone is connected or disconnected:

```
/// <summary>
/// BALForm contains the interface methods required for event handling
/// so a form can interact with a BALApp.
/// </summary>
public interface BALForm
{
  // Called when phones are connected or disconnected
  void UpdatePhoneList();
}
```

This will be implemented by our application-specific classes.

Then we need a class that will encapsulate a connected phone. This is not complicated and is similar to the one we created in Chapter 5.

```
// Class to hold information about a connected phone
public class BALConnectedPhone
{
  public SymbianConnectBAL.ISCBALDevice mDevice;

  public BALConnectedPhone(SymbianConnectBAL.ISCBALDevice aDevice)
  {
    // Device identification string is retrieved from the device
    mDevice = aDevice;
    mPhoneId = mDevice.Id;
    mPhoneManufacturer = mDevice.Manufacturer;
    mPhoneModel = mDevice.Model;

    // The user-specified device name is stored in the registry
    string nameKeyName = @"SOFTWARE\Symbian\Symbian Connect QI\Devices\"
                         + mPhoneId;
    Microsoft.Win32.RegistryKey aKey =
       Microsoft.Win32.Registry.CurrentUser.OpenSubKey(nameKeyName);
    if(aKey != null)
    {
      mPhoneName = aKey.GetValue("Name").ToString();
      aKey.Close();
    }
    else
    {
      mPhoneName = "My " + mPhoneManufacturer + " " + mPhoneModel;
    }
  }

  private string mPhoneId;
  public string Id
  {
    get
    {return mPhoneId;}
  }

  private string mPhoneManufacturer;
  public string Manufacturer
```

```
  {
    get
    {return mPhoneManufacturer;}
  }

  private string mPhoneModel;
  public string Model
  {
    get
    {return mPhoneModel;}
  }

  private string mPhoneName;
  public string Name
  {
    get
    {return mPhoneName;}
  }
}// End of class BALConnectedPhone
```

Now we can create a class that owns a BAL application and manages phone connection and disconnection events. It will own a public array of connected phones and will implement an event handler, so it gets informed when phones connect or disconnect. In turn, it will call its owner whenever this happens.

```
public class BALApp
{
  private BALForm mBALForm;

  public BALApp(BALForm aBALForm)
  {
    mBALForm = aBALForm;
    try
    {
      // Get access to BAL via the Application member
      mBALApp = new SymbianConnectBAL.BALApplication();
      // Set up the event handlers before initializing the collection
      mPhoneArray = new System.Collections.ArrayList();
      mBALApp.OnDeviceListChanged += new
        SymbianConnectBAL.ISCBALEvents_OnDeviceListChangedEventHandler
                                  (OnDeviceListChanged);
      UpdatePhoneList(mBALApp.ConnectedDevices);
    }
    catch(System.Exception exc)
    {
      System.Console.WriteLine("Exception when accessing BAL
                         application: {0}", exc.Message);
    }
  }

  public void UpdatePhoneList(SymbianConnectBAL.ISCBALDeviceCollection
                        aDevices)
  {
    lock(mPhoneArray)
```

```
    {
      // Check for adding new devices
      foreach(SymbianConnectBAL.ISCBALDevice device in aDevices)
      {
        bool alreadyPresent = false;
        foreach(BALConnectedPhone phone in mPhoneArray)
        {
          if(phone.Id == device.Id)
          {
            alreadyPresent = true;
          }
        }
        if(!alreadyPresent)
        {
          mPhoneArray.Add(new BALConnectedPhone(device));
        }
      } //end foreach device in collection

      // Check for removing devices
      if(aDevices.Count != mPhoneArray.Count)
      {
        for(int i = 0 ; i < mPhoneArray.Count ; i ++)
        {
          bool found = false;
          foreach(SymbianConnectBAL.ISCBALDevice device in aDevices)
          {
            if(((BALConnectedPhone)(mPhoneArray[i])).Id == device.Id)
            {
              found = true;
              break;
            }
          }
          if(!found)
          {
            mPhoneArray.RemoveAt(i);
          }
        } // end foreach phone in array
      }
    }// endlock
  }

  // Event handlers
  public void OnDeviceListChanged()
  {
    UpdatePhoneList(mBALApp.ConnectedDevices);
    mBALForm.UpdatePhoneList();
  }

  // SCOM Application handle
  SymbianConnectBAL.BALApplication mBALApp;

  // We will hold the details of connected phones in an
  // array.  We could use other collection types (e.g. a hashtable)
  // but the small number of connected devices makes this overkill.
  public System.Collections.ArrayList mPhoneArray;
}//End class
```

We could have embedded this logic in a class that was directly part of the GUI, but by separating it out in this way we are able to reuse the

same code for a number of applications and our maintenance burden is reduced.

Given the `BALApp` class, we can create a form class that is the root of our GUI application.

13.3 Starting a PC Connectivity Service

Once we have a connected phone, we will need to load and start a service on it. We saw how to do this for a custom server in Chapter 7 and for a socket server in Chapter 8. You will need to ensure that your special services are present on the phone and you will need to know which type of service you want to start. However, you can try to load a socket server or `ectcpadapter` and if that fails then try the other one, as long as you handle exceptions for the loading failures. Remember that you must handle the case where you cannot load your service because the user may not have installed it.

A full example of this code is contained in the example applications that accompany this book.

In the example applications that we will create later in this chapter, we will use this method to start a service and obtain a stream to communicate with it whenever a connected phone is selected.

13.4 Communicating and Managing Delays

In contrast to the SCOM application that we created in Chapter 5, we need to explicitly manage read and write operations and delays. For our SCOM application, we just changed the cursor to an hourglass whenever we had an operation that we expected to take some time. In this application, I will show an alternative approach that provides better feedback and is more flexible in some ways.

Whenever we send a command to the phone, we will expect a response (this is true for all of the commands that we implemented in the previous chapters, with the exception of the Message Server event handling). Therefore, we will start a read operation and set a flag to indicate that there is a read operation pending and we refuse to send any more PDUs. When a PDU is received, we unset the flag and allow more PDUs to be sent. At the same time we can use a status text box to display progress information to the user.

Putting these pieces together, Figure 13.1 shows a Windows Form with controls to show connected phones. We use a list box for the phones and text boxes for some of the phone details. We will also provide a status box for progress information.

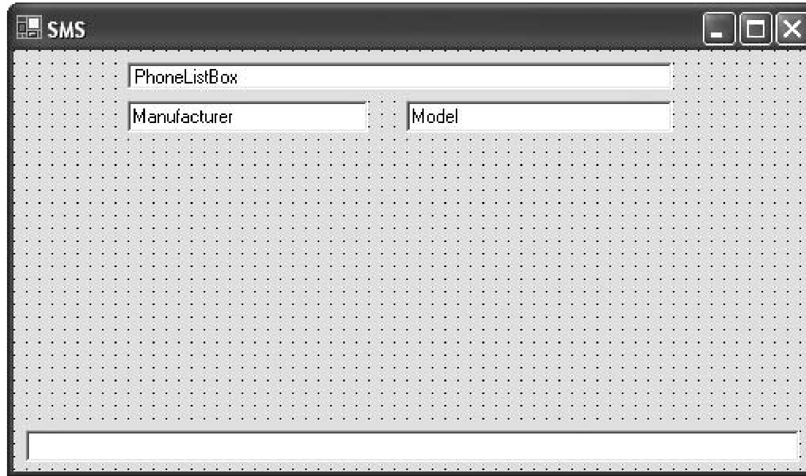


Figure 13.1

The form class must implement the `BALForm` interface that we defined for use with the `BALApp` class we defined earlier and include the controls that we have created:

```
public class ExampleForm : System.Windows.Forms.Form, BALForm
{

  /// <summary>
  /// BAL Application giving access to phones
  /// </summary>
  public BALApp mBALApp;

  private System.Windows.Forms.ListBox PhoneListBox;
  private System.Windows.Forms.TextBox ManufacturerTextBox;
  private System.Windows.Forms.TextBox ModelTextBox;
  private System.Windows.Forms.TextBox StatusBox;
```

We will need a range of members for the current phone and the state of the phone list.

```
  /// <summary>
  /// Reference to the currently selected phone
  /// </summary>
  public BALConnectedPhone mCurrentPhone;
  /// <summary>
  /// ID of the currently selected phone
  /// </summary>
  public string mCurrentPhoneId;
  /// <summary>
  /// Is our list of connected devices empty
  /// </summary>
  public bool mIsEmptyPhoneList;
```

```
// The following members represent states that enable or disable
// certain functionality
/// <summary>
/// Set when initializing the form - do not redraw anything and
/// ignore some events
/// </summary>
public bool mInitializing;
/// <summary>
/// Set when adding or removing from the phone list - ignore index
/// changes
/// </summary>
public bool mUpdatingPhoneList;
/// <summary>
/// Set when a phone has connected or disconnected - an OnPaint
/// event will cause the phone list to be refreshed
/// </summary>
public bool mPendingPhoneViewUpdate;
```

We will need some members for the stream that we use to talk to the service on the phone, the state of a read operation and the next transaction identifier.

```
/// <summary>
/// Is a read pending from the device - do not issue any commands
/// to the device
/// </summary>
private bool mPendingRead;
/// <summary>
/// BAL service to the device
/// </summary>
private BALApplicationAsyncStream mStream;
/// <summary>
/// Are we reading the length part of a PDU
/// </summary>
private bool mReadingLength;
/// <summary>
/// PDU transaction identifier
/// </summary>
private int mNextPDUId;
```

The constructor creates a new `BALApp` and passes itself in for event callbacks.

```
public ExampleForm()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();

  mInitializing = true;
  // Create a BALApp for access to phones
  mBALApp = new BALApp(this);
  mCurrentPhone = null;
  mCurrentPhoneId = "";
```

```
    mInitializing = false;
    mNextPDUId = 1;
    mUpdatingPhoneList = false;
    mPendingPhoneViewUpdate = true;

    ResetPhoneList();
    SetButtonStates();
  }
```

The `SetButtonStates` method will be seen later – it enables or disables buttons or controls based on state variables.

We then have a set of methods that maintain the phone list initially and when events are received from the `BALApp`. We handle the event from BAL and trigger another event to actually update our list of connected smartphones. This prevents us from interfering with BAL by carrying out too much work directly in its event handler.

```
    /// <summary>
    /// Update display of list of connected phones.
    /// </summary>
    public void UpdatePhoneList()
    {
      // This method can be called from an event when a phone
      // connects or disconnects.
      // Don't rewrite other views here as we don't
      // want to do too much in an event so just prime it.
      mPendingPhoneViewUpdate = true;
      PhoneUpdate();
    }

    /// <summary>
    /// A phone has connected or disconnected so we need to reset
    /// the list
    /// </summary>
    public delegate void PhoneUpdateEventHandler();
    public event PhoneUpdateEventHandler PhoneUpdate;

    protected virtual void OnPhoneUpdate()
    {
      ResetPhoneList();
      mPendingPhoneViewUpdate = false;
    }

    /// <summary>
    /// Called (often via an event) when a phone is connected or
    /// disconnected
    /// </summary>
    private void ResetPhoneList()
    {
      bool wasCurrentPhone = false;
      bool currentPhonePresent = false;
      if( !mInitializing)
      {
        wasCurrentPhone = (mCurrentPhoneId != "");
```

```
      mUpdatingPhoneList = true; // Prevent selection events
      PhoneListBox.BeginUpdate();
      PhoneListBox.Items.Clear();
      foreach(BALConnectedPhone phone in mBALApp.mPhoneArray)
      {
        PhoneListBox.Items.Add(phone);
        if(phone.Id == mCurrentPhoneId)
        {
          currentPhonePresent = true;
          PhoneListBox.SelectedIndex = PhoneListBox.Items.Count-1;
        }
      }
      if(PhoneListBox.Items.Count > 0)
      {
        mIsEmptyPhoneList = false;
        if(!currentPhonePresent)
        {
          PhoneListBox.SelectedIndex = 0;
          mCurrentPhone = (BALConnectedPhone)mBALApp.mPhoneArray[0];
          mCurrentPhoneId = mCurrentPhone.Id;
          ManufacturerTextBox.Text = mCurrentPhone.Manufacturer;
          ModelTextBox.Text = mCurrentPhone.Model;
          InitializePhoneService();
        }
      }
      else
      {
        mIsEmptyPhoneList = true;
        mCurrentPhone = null;
        mCurrentPhoneId = "";
        ManufacturerTextBox.Text = "";
        ModelTextBox.Text = "";
      }
      PhoneListBox.EndUpdate();
    }
    mUpdatingPhoneList = false;

    // If we have lost the current phone then we
    // may have to take some action
    if(wasCurrentPhone && !currentPhonePresent)
    {
      mLostPhone = true;
      StatusBox.Text = "";
      SetButtonStates();
    }
  }
```

Whenever we connect to a new phone we want to load and start our service(s) on it. This means that we just keep a member variable for the stream. If we wanted to handle multiple connected phones or multiple services (and therefore streams) then we would want to store the stream(s) with the phone objects. When we check for an error we use the Symbian OS constant `KErrNone`. In this case I have included this constant manually – we are not really including Symbian OS header files for their constants.

```
/// <summary>
/// Initializes the service we need on the phone.
/// </summary>
private void InitializePhoneService()
{
  int errorCode = mCurrentPhone.StartService("dummy service",
                                              out mStream);
  if(errorCode != BALConnectedPhone.KErrNone)
  {
    MessageBox.Show("The service plug-in was not found",
                    "Service Missing Error",
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
  else
  {
    mStream.OnRead += new
            ISCBALSequentialStreamSink_OnReadEventHandler(OnRead);
    mStream.OnWrite += new
            ISCBALSequentialStreamSink_OnWriteEventHandler(OnWrite);

    // Clear data related to this phone
    StatusBox.Text = "";
    SetButtonStates();
  }
}
```

Then we need to reset members when the user changes phone by associating an event with the user changing the selected entry in the phone list.

```
  /// <summary>
  /// Called when the selected phone has changed.
  /// </summary>
  private void PhoneListBox_SelectedIndexChanged(object sender,
                                                 System.EventArgs e)
  {
    if(!mUpdatingPhoneList)
    {
      int newIndex = PhoneListBox.SelectedIndex;
      if(newIndex >= 0)
      {
        BALConnectedPhone newPhone =
          (BALConnectedPhone)PhoneListBox.Items[newIndex];
        mCurrentPhone = newPhone;
        mCurrentPhoneId = newPhone.Id;
        ManufacturerTextBox.Text = newPhone.Manufacturer;
        ModelTextBox.Text = newPhone.Model;
        InitializePhoneService();
        ClearMainList();
      }
    }
  }
```

We are going to implement the reads from and writes to the device asynchronously. Actually, we don't really mind about the writes, because

we have not put together any protocols that would benefit from asynchronous writes, but we do want asynchronous reads. When we started the service we associated two event handlers and here they are. The `OnRead` event handler performs the read in two stages. First it reads the four-byte integer that is the number of bytes in the rest of the PDU and then it reads the rest of the PDU. Internally, the whole PDU will almost certainly have been buffered, so we will expect the second `OnRead` call to be returned immediately.

```
/// <summary>
/// Event handler for writing to the BAL service.
/// </summary>
public void OnWrite( int aError )
{
}

/// <summary>
/// Event handler for reading from the BAL service.
/// </summary>
public void OnRead( object aBuffer, int aError )
{
  if(aError == 0)
  {
    byte[] buff = (byte[])aBuffer;
    if(mReadingLength && buff.Length >= 4)
    { // Just read the length of the rest of the buffer
      int readPos = 0;
      int msgLen = ConnPack.ReadInt32(ref buff, ref readPos);
      if(msgLen > 0)
      {
        mReadingLength = false;
        mStream.Read(msgLen);
      }
    }
    else if(!mReadingLength && buff.Length >= 4)
    { // Read the 4-byte op-code and respond accordingly
      int readPos = 0;
      int opCode = ConnPack.ReadInt32(ref buff, ref readPos);
      int PDUId = ConnPack.ReadInt32(ref buff, ref readPos);
      ServiceRead(opCode, PDUId, ref buff, ref readPos);
    }
  }
  else
  {
    StatusBox.Text = "OnRead error" + aError.ToString();
  }
}
```

If we were using synchronous I/O then whenever we read from the stream, we would know exactly what we were expecting. At first sight, this seems a good idea, but we would still have to handle errors or unexpected responses. Because we are using asynchronous I/O, we have one point where we receive all responses. Therefore, we switch on the opcode. In the example here, we have not yet put in any of the actual

logic for specific responses, but we can see the framework – note that when we have completed a read operation we reset flags accordingly. The logic behind the setting of the button states will become apparent when we implement the real applications.

```
  /// <summary>
  /// Handle a PDU read from the device
  /// </summary>
  private void ServiceRead(int aOpCode, int aPDUId, ref byte[] aBuff,
                           ref int aReadPos)
  {
    int errno = 0;
    switch(aOpCode)
    {
      default:
        // Unrecognized operation - raise an error in debug or ignore
        break;
    }

    ClearPendingReads();
    if(errno != 0)
    {
      StatusBox.Text = "Error " + errno.ToString();
    }
    SetButtonStates();
  }

  /// <summary>
  /// Called when pending reads are complete - clears status text and
  /// enables buttons
  /// </summary>
  private void ClearPendingReads()
  {
    mPendingRead = false;
    StatusBox.Text = "";
  }

  /// <summary>
  /// Enables or disables buttons depending on state flags
  /// </summary>
  private void SetButtonStates()
  {
  // Disable all buttons if a read is pending
  }
}
```

13.5 A GUI SMS Application

Now we have the basic structure for the application, we can add the controls and functionality for the specific service we want. In order to manage SMS, we will have a main form (Figure 13.2) with a list of SMS

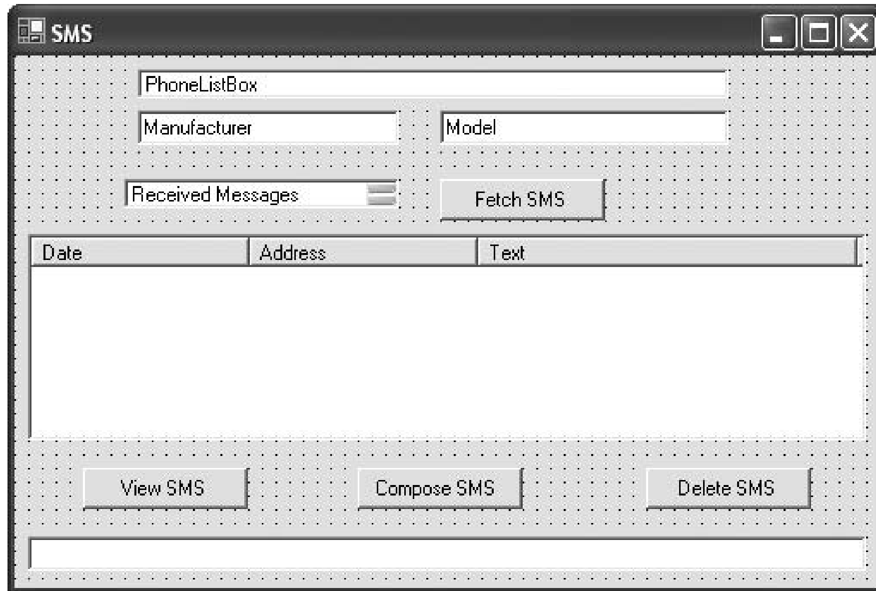


Figure 13.2

message summaries and buttons to allow us to view the detail of a message, delete a message or compose and send a message.

This is deliberately a very simple messaging application; we are not including the ability to reply or forward a message, and we will show only one message folder at a time (the Inbox or Sent Messages). A more sophisticated application might implement a tabbed display and hold details of multiple folders. The aim of this book is to show how Connectivity applications can be created, not to spend too much time fine-tuning the user interface.

One of the obvious functions that we will omit from this application is integration with Contacts. A good messaging application (such as the one supplied on Symbian OS smartphones) will display contact names rather than just telephone numbers when displaying messages and will allow recipients to be selected from the user's Address Book.

Our first step will be to retrieve and display message summaries, including the date of sending or receiving, the sending or receiving phone number, and the body of the message. The user chooses which message folder to display and the application then sends a command to the Symbian OS smartphone when the 'View SMS' button is pressed.

As part of developing this application I created an `RSMSUtil` class to hold the constants and a set of static methods to create commands and decode responses. The same pattern is used in the subsequent

applications. You could just embed the code or make the methods part of the form class, but I was planning for some reuse.

```
/// <summary>
/// Start to read Sms from the selected folder
/// </summary>
private void RetrieveButton_Click(object sender, System.EventArgs e)
{
  // The folder chooser list box has hard-wired values, Inbox
  // first and then the Sent Messages folder.  We could populate
  // the list programmatically and include IDs but this is adequate
  int i = FolderChooseListBox.SelectedIndex;
  int folderId = RSMSUtil.KMsvGlobalInBoxIndexEntryId;
  if(i != 0)
  { folderId = RSMSUtil.KMsvSentEntryId; }
  RSMSUtil.WriteFetchAllSms(mStream, ref mNextPDUId, folderId);
  ClearMainList();
  StatusBox.Text = "Fetching SMS - please wait";
  mPendingRead = true;
  mReadingLength = true;
  mStream.Read(4);
  mReadingSms = true;
  SetButtonStates();
}
```

We will see that the `WriteFetchAllSms` method adds the command code, the PDU identifier and the folder identifier into a PDU and then sends it to the smartphone using the methods that we developed in Chapter 10.

```
// Standard folder ID constants
public const int KMsvGlobalInBoxIndexEntryId = 0x1002;
public const int KMsvSentEntryId = 0x1005;

  /// Fetch all SMS in a folder
  public static void WriteFetchAllSms
                    (BALApplicationAsyncStream aStream,
                     ref int aNextPDUId, int aFolderId )
  {
    ArrayList message = new ArrayList();
    ConnPack.WriteInt32(ERSmsCmdGetAllSms, ref message);
    ConnPack.WriteInt32(aNextPDUId, ref message);
    ConnPack.WriteInt32(aFolderId, ref message);
    aNextPDUId++;
    aStream.Write(ConnPack.AsByteArr(ref message));
  }
```

Once we have sent the message, we need to extend our reading method to handle the reply. We will read SMS details from the buffer until we run out. At the same time, we set flags to indicate that we now have some messages in our list and so the ‘Delete’ and ‘View’ buttons can be enabled. Because we may have more messages in a folder than can be returned in one packet, we request more messages until we receive no more.

```
/// <summary>
/// Handle a PDU read from the device
/// </summary>
private void ServiceRead(int aOpCode, int aPDUId, ref byte[] aBuff,
                         ref int aReadPos)
{
  int errno = 0;
  switch(aOpCode)
  {
    case ( RSMSUtil.ERSmsCmdReceiveSms ):
      bool readSome = false;
      bool gotOne = true;
      while(gotOne)
      {
        RSMS newSms = new RSMS();
        gotOne = RSMSUtil.ReadSms(ref aBuff, ref aReadPos,
                                  ref newSms);
        if(gotOne)
        {
          AddSms(ref newSms);
          readSome = true;
        }
      }
      if(!readSome)
      { mReadingSms = false; }
      break;
    case ( RSMSUtil.ERSmsCmdReceiveNoMoreSms ):
      mReadingSms = false;
      break;

...

  }
  ClearPendingReads();
  if(errno != 0)
  {
    StatusBox.Text = "Error " + errno.ToString();
  }
  else if(mReadingSms)
  { // Read more SMS from the selected folder
    RSMSUtil.WriteFetchMoreSms(mStream, ref mNextPDUId);
    StatusBox.Text = "Fetching more SMS - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
  }
  if(SmsView.Items.Count > 0)
  { mGotSms = true; }
  SetButtonStates();
}
```

The actual method to read an SMS constructs an object as a convenient container for the data.

```
/// Class to hold details of an SMS address
/// Member variables are public - really just a struct
public class RSMSAddress
```

```
{
  public string mName;
  public string mAddress;
}

/// Class to hold details of an SMS message
/// Member variables are public - really just a struct
public class RSMS
{
  public int mId; // Message ID
  public int mParentId; // Parent folder
  public string mFromAddress; // Sender address
  public string mBodyText; // Message body
  public string mDescription; // Message description
  public string mDetail; // Message detail
  public ArrayList mDestinationAddress; // Destination addresses
  public DateTime mDate; // Message date and time stamp

  public RSMS()
  {
    mDestinationAddress = new ArrayList();
  }
}

  /// Read an SMS - terminated by a zero or negative message ID
  static public bool ReadSms(ref byte[] aBuff, ref int aBuffOffset,
                             ref RSMS aSms)
  {
    int smsId = ConnPack.ReadInt32( ref aBuff, ref aBuffOffset);
    if(smsId <= 0)
    {return false;}

    aSms.mId = smsId;
    aSms.mParentId = ConnPack.ReadInt32( ref aBuff, ref aBuffOffset);
    bool nullDate; // can't be null for an SMS
    ConnPack.ReadDateTime( ref aBuff, ref aBuffOffset,
                           out nullDate, out aSms.mDate);

    System.Text.StringBuilder tempText;
    ConnPack.ReadUNCData( ref aBuff, ref aBuffOffset, out tempText);
    aSms.mDescription = tempText.ToString();
    ConnPack.ReadUNCData( ref aBuff, ref aBuffOffset, out tempText);
    aSms.mDetail = tempText.ToString();
    ConnPack.ReadUNCData( ref aBuff, ref aBuffOffset, out tempText);
    aSms.mFromAddress = tempText.ToString();

    int recipientCount = ConnPack.ReadInt16(ref aBuff, ref aBuffOffset);
    for(int i = 0 ; i < recipientCount ; i++)
    {
      System.Text.StringBuilder recName, recAddress;
      ConnPack.ReadUNCData( ref aBuff, ref aBuffOffset, out recName);
      ConnPack.ReadUNCData( ref aBuff, ref aBuffOffset, out recAddress);
      RSMSAddress address = new RSMSAddress();
      address.mName = recName.ToString();
      address.mAddress = recAddress.ToString();
      aSms.mDestinationAddress.Add(address);
    }//endfor
```

```
  System.Text.StringBuilder bodyText;
  ConnPack.ReadASCIIData( ref aBuff, ref aBuffOffset, out bodyText);
  aSms.mBodyText = bodyText.ToString();

  return true;
}
```

Having read in messages, we will display a summary and store the object in the main list.

```
/// <summary>
/// Add a read SMS to the main list
/// </summary>
private void AddSms(ref RSMS aSms)
{
  System.Windows.Forms.ListViewItem newItem = new ListViewItem();
  newItem.Tag = aSms;

  newItem.Text = aSms.mDate.ToString();
  newItem.SubItems.Add(aSms.mDetail);
  newItem.SubItems.Add(aSms.mBodyText);
  SmsView.Items.Add(newItem);
}
```

In the SMS service, we implemented event handling to allow the Symbian OS smartphone to inform the PC when a new message arrived (or other events occurred). However, in this application we will not make use of event handling; we simply rely on the user to refresh the list when necessary. If we did want to use event handling then we would create a second stream, associate it with the service and send a PDU to initiate event handling. We cannot use one stream for both purposes in parallel.

Once we have our list of messages, deleting or displaying details is straightforward. Deleting involves retrieving the message identifier from the stored object and sending a delete command to the phone.

```
/// <summary>
/// Delete an SMS
/// </summary>
private void DeleteButton_Click(object sender, System.EventArgs e)
{
  if(SmsView.SelectedIndices.Count <= 0)
  {
    return;
  }
  int editIndex = SmsView.SelectedIndices[0];
  RSMS delSms = (RSMS)SmsView.Items[editIndex].Tag;
  if (MessageBox.Show
    "Are you sure you want to delete this message ?",
    "Confirm Delete",
    MessageBoxButtons.YesNo, MessageBoxIcon.Question,
    MessageBoxDefaultButton.Button2 )
    == DialogResult.Yes)
```

```
  {
    RSMSUtil.WriteDeleteSms(mStream, ref mNextPDUId, delSms.mId);
    StatusBox.Text = "Deleting message - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    SmsView.Items.RemoveAt(editIndex);
    if(SmsView.Items.Count == 0)
    { mGotSms = false; }
  }
  SetButtonStates();
}
```

```
/// Delete one specific SMS
public static void WriteDeleteSms
                  (BALApplicationAsyncStream aStream,
                   ref int aNextPDUId, int aSmsId )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERSmsCmdDeleteSms, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteInt32(aSmsId, ref message);
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

We will also extend our `ServiceRead()` method for the delete reply, but we do not need to take any action on it.

Actually, we have a design choice to make here. The user has chosen to delete an SMS message and we send a command to the smartphone to achieve this. We then wait for a response (which could be an error). For the sake of simplicity, I have chosen to update the visible message list as soon as we send the command, but I could have waited until we receive the response. For these examples I have chosen the simpler approach, but for a more complex service you might choose to delay updating the local data or display until the response is received.

Similarly, when a message is sent I have not chosen to update the displayed list of sent messages, but the new message could simply be appended.

Viewing the whole of a message, including all recipients, requires a form with the necessary controls and just a Cancel button (because all data is read-only), as shown in Figure 13.3.

To compose and send a message, we use a form with controls for the body text and recipient, as shown in Figure 13.4. As was mentioned earlier, a commercial application would allow the user to select recipients from their Address Book and would also consider providing a count of the number of characters typed, so the user would know when the message would require multiple SMS.

The code behind the dialog is simply concerned with storing the data entered for access. We pass in a reference to an object that the composing

Figure 13.3

Figure 13.4

form uses to pass back the data that the user has entered. We will use the same technique repeatedly in the other applications where we are editing or creating objects.

```
public class ComposeForm : System.Windows.Forms.Form
{

  public RSMS mSms;

  public ComposeForm(ref RSMS aSms)
  {
  //
```

```
  // Required for Windows Form Designer support
  //
  InitializeComponent();
  mSms = aSms;
  }

  /// <summary>
  /// Save required data before returning
  /// </summary>
  private void SendButton_Click(object sender, System.EventArgs e)
  {
    mSms.mDestinationAddress.Add(AddressBox.Text);
    mSms.mBodyText = TextBox.Text;
  }
}
```

The main form contains the code to send the message.

```
  /// <summary>
  /// Compose a new SMS
  /// </summary>
  private void ComposeButton_Click(object sender, System.EventArgs e)
  {
    RSMS newSms = new RSMS();
    ComposeForm composeForm = new ComposeForm(ref newSms);
    if(composeForm.ShowDialog() == DialogResult.OK)
    {
      RSMSUtil.WriteSendSms(mStream, ref mNextPDUId, ref newSms);
      StatusBox.Text = "Sending SMS - please wait";
      mPendingRead = true;
      mReadingLength = true;
      mStream.Read(4);
    }
    SetButtonStates();
  }
```

```
    /// Create and send an SMS
    public static void WriteSendSms( BALApplicationAsyncStream aStream,
                                     ref int aNextPDUId,
                                     ref RSMS aSms )
    {
      ArrayList message = new ArrayList();
      ConnPack.WriteInt32(ERSmsCmdSendSms, ref message);
      ConnPack.WriteInt32(aNextPDUId, ref message);
      ConnPack.WriteASCIIData(aSms.mBodyText, ref message);
      ConnPack.WriteUint16((ushort)aSms.mDestinationAddress.Count,
                           ref message);
      for(int i = 0 ; i < aSms.mDestinationAddress.Count ; i++)
      {
        string addressText = (string)aSms.mDestinationAddress[i];
        ConnPack.WriteUNCData(addressText, ref message);
      }
      aNextPDUId++;
      aStream.Write(ConnPack.AsByteArr(ref message));
    }
```

As with the deletion of messages, we will need to extend our `ServiceRead()` method to handle the send reply, but we will not do anything with the data. We could choose to update the message object with the actual message identifier to allow deletion.

One aspect that we have not handled with the deletion and sending functions is what happens if the smartphone disconnects while the form is displayed. In all cases when writing or reading, we have the possibility that the smartphone will disconnect and the write or read may fail. However, if we display a dialog to ask the user to confirm a deletion or to compose a SMS for sending, the phone may disconnect but the user may still choose the operation. If we do not handle this properly then the main form will display status text that is incorrect (we are not deleting the message, for example, because the smartphone connection has gone). To handle this, we use a flag to indicate when the smartphone has disconnected and we check it when reacting to the OK button from the delete or compose forms. If the smartphone has disconnected then we ignore the action.

13.6 A Contacts GUI Application

The Contacts application that we will create has a similar structure to the SMS application. We will have a main list that will hold some information on all contacts and we will allow the user to view full details of selected contacts, delete contacts or create new ones.

For the sake of simplicity, we will not deal with groups of contacts, although this could be a useful avenue for a commercial application, as PIM synchronization of contacts sometimes omits group, and we will make only very limited use of templates.

The main aspect that we will have to deal with in the Contacts application is the handling of fields. As we saw in Chapter 11, a contact contains a set of fields and each field is identified by the full set of associated field types. If a user creates all of their contacts in one way (manually using the phone, for example, or on a PC PIM) then they are likely to have a very similar set of fields on all of their contacts. However, if a user has contacts created in a range of ways, the fields may not all have the same types. Therefore, it is not possible to impose a standard list of fields in a dialog. Instead, we need to use more flexible methods of handling fields. Some of these methods will be explained below.

To begin with, we define classes to hold details of contacts and fields.

```
/// Class to hold details of a contact field - text value only
/// Member variables are public - really just a struct
public class RCNTField
{
  public int mFieldId; // field ID
```

```
  public ArrayList mFieldType; // array of field type IDs
  public int mMapping; // Field type additional mapping
  public string mFieldLabel; // field label
  public string mFieldText; // field contents (text only)
  public bool mChanged; // has the field data changed
}

/// Class to hold a contact details
/// Member variables are public - really just a struct
public class RCNTContact
{
  public int mId; // contact ID
  public ArrayList mFields; // array of fields
}
```

For our overview list on the main form (see Figure 13.5), I have chosen to just display the names of contacts – we will provide access to other details via a more detailed view.

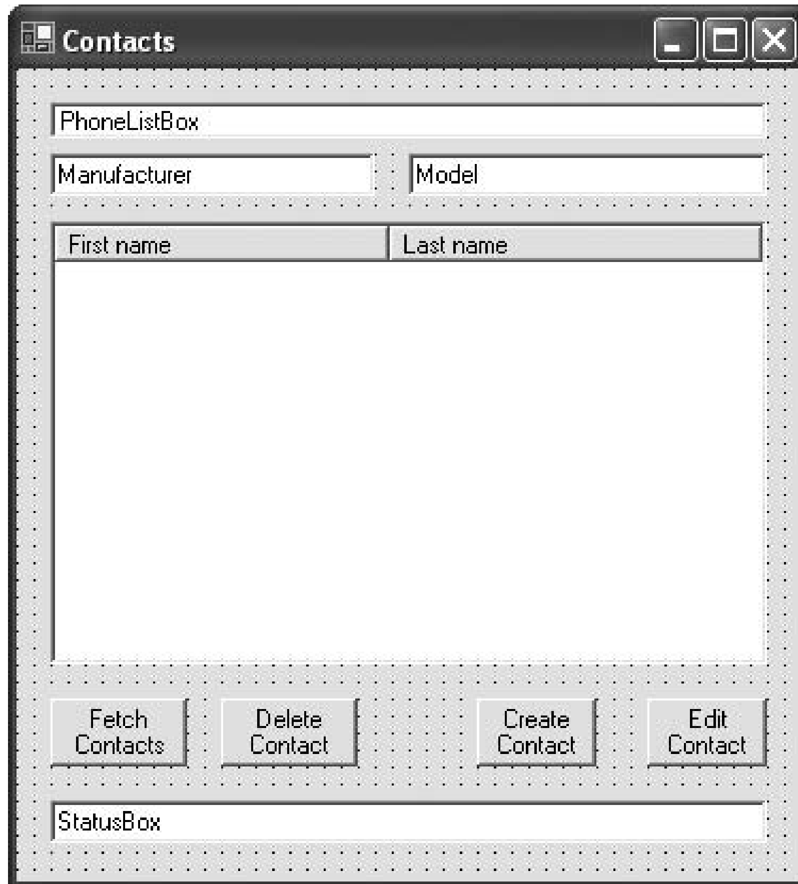


Figure 13.5

We are going to request these names only when bulk contacts are read and we will need the system template to create new contacts. Therefore, as soon as we get a connection to a phone, we will open the Contacts database with a view that includes only the name fields and then we will retrieve the template information.

As with the previous application, we use a utility class (`RCNTUtils` in this case) to hold structures, command writing and response reading methods.

```
/// <summary>
/// Initializes the service we need on the phone.
/// </summary>
private void InitializePhoneService()
{
  int errorCode = mCurrentPhone.StartService("RCNTCS", out mStream);
  if(errorCode != BALConnectedPhone.KErrNone)
  {
    MessageBox.Show("The Contacts plug-in was not found",
                    "Service Missing Error",
                    MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
  else
  {
    mStream.OnRead += new
      ISCBALSequentialStreamSink_OnReadEventHandler(OnRead);
    mStream.OnWrite += new
      ISCBALSequentialStreamSink_OnWriteEventHandler(OnWrite);

    // Clear data related to this phone
    mTemplateFields = new ArrayList();
    // Open the database
    RCNTUtils.WriteOpenDatabase(mStream, ref mNextMessageId);
    StatusBox.Text = "Opening Contacts database - please wait";
    mWantToFetchTemplateFields = true;
    mWantToFetchContacts = false;
    mGotContacts = false;
    mPendingEditContact = 0;

    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    SetButtonStates();
  }
}
```

```
/// Open the Contacts database and set the view
public static void WriteOpenDatabase
                  (BALApplicationAsyncStream aStream,
                   ref int aNextPDUId )
{
  ArrayList message = new ArrayList();;
  ConnPack.WriteInt32(ERCntCmdOpenDatabase, ref message);
  ConnPack.WriteInt32(aNextPDUId , ref message);
```

```
      ConnPack.WriteInt32(KLastNameType, ref message);
      ConnPack.WriteInt32(KFirstNameType, ref message);
      ConnPack.WriteInt32(-1, ref message);

      aNextPDUId ++;
      aStream.Write(ConnPack.AsByteArr(ref message));
    }

    /// Fetch details of the template fields
    public static void WriteFetchTemplateFieldInfo(
      BALApplicationAsyncStream aStream, ref int aNextPDUId )
    {
      ArrayList message = new ArrayList();;
      ConnPack.WriteInt32(ERCntCmdFetchTemplateFieldInfo, ref message);
      ConnPack.WriteInt32(aNextPDUId , ref message);
      aNextPDUId ++;
      aStream.Write(ConnPack.AsByteArr(ref message));
    }
```

The `ServiceRead()` method is then extended to store the template field information. This illustrates a technique for queuing PDUs and servicing the queue by means of the`ServiceRead()` method.

```
/// <summary>
/// Handle a PDU read from the device
/// </summary>
private void ServiceRead(int aOpCode, int aPDUId, ref byte[] aBuff,
                         ref int aReadPos)
{
  switch(aOpCode)
  {

    case(RCNTUtils.ERCntCmdTemplateFieldInfoReply):
      RCNTUtils.ReadTemplateInfo(ref aBuff, ref aReadPos,
                                 ref mTemplateFields);
      break;
...
    default:
      break;
  }

  // If we have operations that we want to carry out then trigger them.
  // Ask for template fields to get labels
  if(mWantToFetchTemplateFields)
  {
    RCNTUtils.WriteFetchTemplateFieldInfo(mStream, ref mNextMessageId);
    StatusBox.Text = "Fetching labels - please wait";
    mWantToFetchTemplateFields = false;
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
  }
  else
  {
    ClearPendingReads();
    StatusBox.Text = "";
```

```
  }
  SetButtonStates();
}
```

```
/// Read template field info
static public void ReadTemplateInfo(ref byte[] aBuff,
                   ref int aBuffOffset, ref ArrayList aFields)
{
  int fieldId = ConnPack.ReadInt32( ref aBuff, ref aBuffOffset);
  while(fieldId >= 0)
  {
    RCNTField fieldData = new RCNTField();
    fieldData.mFieldId = fieldId;
    fieldData.mFieldType = new ArrayList();
    int fieldTypeId = ConnPack.ReadInt32( ref aBuff, ref aBuffOffset);
    while(fieldTypeId >= 0)
    {
      fieldData.mFieldType.Add(fieldTypeId);
      fieldTypeId = ConnPack.ReadInt32( ref aBuff, ref aBuffOffset);
    }
    fieldData.mMapping = ConnPack.ReadInt32( ref aBuff,
                                             ref aBuffOffset);

    System.Text.StringBuilder fieldLabel;
    ConnPack.ReadData( ref aBuff, ref aBuffOffset, out fieldLabel);
    fieldData.mFieldLabel = fieldLabel.ToString();

    aFields.Add(fieldData);

    fieldId = ConnPack.ReadInt32( ref aBuff, ref aBuffOffset);
  }//endwhile
}
```

Having set the view and cached the template fields, we can allow the user to retrieve contacts. We could choose to retrieve the contacts data and populate the list as soon as a new phone is connected or selected, but I have chosen to use a button to allow the user to retrieve contacts. This make it easier to see the behavior involved in retrieving contacts, and it will be apparent that it can take several packets to retrieve all contacts.

```
  private void FetchButton_Click(object sender, System.EventArgs e)
  {
    ClearContactList();

    RCNTUtils.WriteFetchAllContacts(mStream, ref mNextPDUId);
    StatusBox.Text = "Fetching Contacts - please wait";
    mPendingRead = true;
    mWantToFetchContacts = true;
    mReadingLength = true;
    mStream.Read(4);
    SetButtonStates();
  }
```

```
    /// Fetch all contacts
    public static void WriteFetchAllContacts
           ( BALApplicationAsyncStream aStream, ref int aNextPDUId )
    {
      ArrayList message = new ArrayList();
      ConnPack.WriteInt32(ERCntCmdFetchAllContacts, ref message);
      ConnPack.WriteInt32(aNextPDUId, ref message);
      ConnPack.WriteInt8(0, ref message); // Do not fetch all fields
      aNextPDUId++;
      aStream.Write(ConnPack.AsByteArr(ref message));
    }

    /// Fetch more contacts
    public static void WriteFetchMoreContacts
           ( BALApplicationAsyncStream aStream, ref int aNextPDUId)
    {
      ArrayList message = new ArrayList();
      ConnPack.WriteInt32(ERCntCmdFetchMoreContacts, ref message);
      ConnPack.WriteInt32(aNextPDUId, ref message);
      ConnPack.WriteInt8(0, ref message); // Do not fetch all fields
      aNextPDUId++;
      aStream.Write(ConnPack.AsByteArr(ref message));
    }
```

As with the SMS application, we need to extend the `ServiceRead()` method to handle the incoming contact details, and we will automatically fetch more contacts until we run out. In the example applications, I have just extended the `ServiceRead()` method for each new purpose, but this is unwieldy. If I had more response codes, I would put in place a more maintainable structure.

```
/// <summary>
/// Handle a PDU read from the device
/// </summary>
private void ServiceRead(int aOpCode, int aPDUId, ref byte[] aBuff,
                         ref int aReadPos)
{
  switch(aOpCode)
  {
...
    case(RCNTUtils.ERCntCmdContactsReply):
      StatusBox.Text = "Reading Contacts -please wait";
      bool carryOn = true;
      bool gotAtLeastOne = false;
      while(carryOn)
      {
        RCNTContact contact = new RCNTContact();
        carryOn = RCNTUtils.ReadContact( ref aBuff, ref aReadPos,
                                ref mTemplateFields, ref contact);
        if(carryOn)
        {
          gotAtLeastOne = true;
          AddCard(ref contact, -1);
        }
      }
```

```
      if(!gotAtLeastOne)
      { mWantToFetchContacts = false;}
      if(gotAtLeastOne)
      { mGotContacts = true;}
      break;
...
  }

  // If we have operations that we want to carry out then trigger them.
  if(mWantToFetchTemplateFields)
  {
...
  }
  else if(mWantToFetchContacts)
  { // Ask for more contacts
    RCNTUtils.WriteFetchMoreContacts(mStream, ref mNextPDUId);
    StatusBox.Text = "Fetching more contacts - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
  }
  else
  {
    ClearPendingReads();
    StatusBox.Text = "";
  }
  SetButtonStates();
}
```

```
/// Read details of a contact - terminated by a negative card ID
static public bool ReadContact(ref byte[] aBuff, ref int buffOffset,
                               ref ArrayList aTemplateFields,
                               ref RCNTContact aContact)
{
  int cardId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
  if(cardId < 0)
  {return false;}

  aContact.mId = cardId;
  aContact.mFields = new ArrayList();

  int fieldId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
  while(fieldId != -1)
  {
    RCNTField fieldData = new RCNTField();
    fieldData.mFieldId = fieldId;
    fieldData.mChanged = false;
    fieldData.mFieldType = new ArrayList();
    int typeId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
    while(typeId >= 0)
    {
      fieldData.mFieldType.Add(typeId);
      typeId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
    }
    fieldData.mMapping = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
```

```
    int fieldType = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
    System.Text.StringBuilder label;
    bool recognizedField = false;
    switch(fieldType)
    {
      case(KStorageTypeText):
        ConnPack.ReadData( ref aBuff, ref buffOffset, out label);
        System.Text.StringBuilder fieldText;
        ConnPack.ReadData( ref aBuff, ref buffOffset, out fieldText);
        fieldData.mFieldLabel = label.ToString();
        fieldData.mFieldText = fieldText.ToString();
        recognizedField = true;
        break;
      default: // unrecognized field type
        break;
    }

    // If a recognized field type then add it to the contact
    if(recognizedField)
    {aContact.mFields.Add(fieldData) ;}

    fieldId = ConnPack.ReadInt32( ref aBuff, ref buffOffset);
  }//endwhile

  // Check that labels are set and use the template if necessary
  for(int i = 0 ; i < aContact.mFields.Count ; i++)
  {
    RCNTField field = (RCNTField)aContact.mFields[i];
    if(field.mFieldLabel.Length < 1)
    {
      field.mFieldLabel = FindFieldLabelByType(ref aTemplateFields,
                                               ref field);
      aContact.mFields[i] = field;
    }
  }

  return true;
}
```

At this point we need to use the template data to supply labels.

```
// Find the label of a field of a set of types - or return "" for
// not found
static public string FindFieldLabelByType
  ( ref ArrayList aTemplateFields, ref RCNTField aContactField)
{
  string fieldLabel = "";
  for(int tFieldi = 0 ; tFieldi < aTemplateFields.Count ; tFieldi++)
  {
    RCNTField tempField = (RCNTField)aTemplateFields[tFieldi];
    if(DoFieldTypesMatch(ref tempField, ref aContactField))
    {
      fieldLabel = tempField.mFieldLabel;
      break;
    }

    if(fieldLabel.Length > 0)
```

```
    {
      break;
    }
  }// endfor each field in the template
  return fieldLabel;
}

/// As contact fields are identified uniquely by the combination of
/// field types and mapping, we need to compare two fields by
/// comparing all the field types and the mapping.
/// If any field type is not found in both then there is a mismatch.
static public bool DoFieldTypesMatch( ref RCNTField field1,
                                      ref RCNTField field2)
{
  bool retVal = true;
  if(field1.mFieldType.Count != field2.mFieldType.Count)
  {
    retVal = false;
  }
  if(field1.mMapping != field2.mMapping)
  {
    retVal = false;
  }

  for(int i1 = 0 ; (i1 < field1.mFieldType.Count) && retVal ; i1++)
  {
    int type1 = (int)field1.mFieldType[i1];
    bool foundMatch = false;
    for(int i2 = 0 ; i2 < field2.mFieldType.Count ; i2++)
    {
      if(type1 == (int)field2.mFieldType[i2])
      {
        foundMatch = true;
        break;
      }
    }
    if(!foundMatch)
    {
      retVal = false;
    }
  }//endfor each field type in first set

  return retVal;
}
```

We can then add each contact to the list of contacts. In our summary list we want to show the first and last names. These are not necessarily fixed fields and so we have to search for the field text based on a field type (note – not all field types in this case).

```
/// <summary>
/// Add a card to the contacts list view
/// if aWhere is negative then append the item, otherwise insert it
/// </summary>
private void AddCard(ref RCNTContact aContact, int aWhere)
{
  System.Windows.Forms.ListViewItem newItem = new ListViewItem();
```

```
  newItem.Tag = aContact;
  newItem.Text = RCNTUtils.FindFieldTextByType(ref aContact,
               RCNTUtils.KFirstNameType); // first name
  newItem.SubItems.Add(RCNTUtils.FindFieldTextByType(ref aContact,
               RCNTUtils.KLastNameType)); // last name
  if((aWhere >= 0) && (aWhere < ContactList.Items.Count))
  {
    ContactList.Items.Insert(aWhere,newItem);
  }
  else
  {
    ContactList.Items.Add(newItem);
  }
}
```

```
// Find the text of a field of a type - or return "" for not found
static public string FindFieldTextByType( ref RCNTContact aContact,
                                          int aFieldType)
{
  string fieldValue = "";
  for(int fieldi = 0 ; fieldi < aContact.mFields.Count ; fieldi++)
  {
    RCNTField testField = (RCNTField)aContact.mFields[fieldi];
    for(int typei = 0 ; typei < testField.mFieldType.Count ; typei++)
    {
      if(aFieldType == (int)testField.mFieldType[typei])
      {
        fieldValue = testField.mFieldText;
        break;
      }
    }
  }
  return fieldValue;
}
```

If you run this code (the complete application is available with the rest of the source code for this book) then you will see that there is a delay when the phone is first asked for contacts and then batches of contacts come in. For a ‘consumer’ type of user this behavior is satisfactory – I can load this application and retrieve all my contacts in less time than Microsoft Outlook takes to load and be ready for me to access contact data. However, an ‘Enterprise’ user with hundreds of contacts would find this unacceptable: the application would be unresponsive for too long.

There are a number of possible ways of handling a phone with a very large number of contacts:

- The application could choose never to retrieve all contacts, but only to work with the ‘find’ method, though this makes it difficult to browse the contacts set.

- The application could start to retrieve names in the same way as above, but could allow other commands to interrupt the flow of PDUs and so avoid making the application totally unresponsive.
- The application could retrieve all details, but in a separate thread and with a second communications stream. The two threads would share a cache of contacts data. This would be the most sophisticated method, but it is not presented here as it would detract from the core illustration – how to communicate with the contacts service developed in Chapter 11.

The best solution will depend on details of the environment.

At this point we have allowed the user only to see the names in their Address Book. When the user selects a contact to view or edit, we will hastily retrieve the whole of the contact data from the smartphone.

We can display or edit a full contact and all of its fields by using a form that has a list box (as illustrated in Figure 13.6) to which each field can be added in turn. This is very flexible, as it will work for any combination of fields.

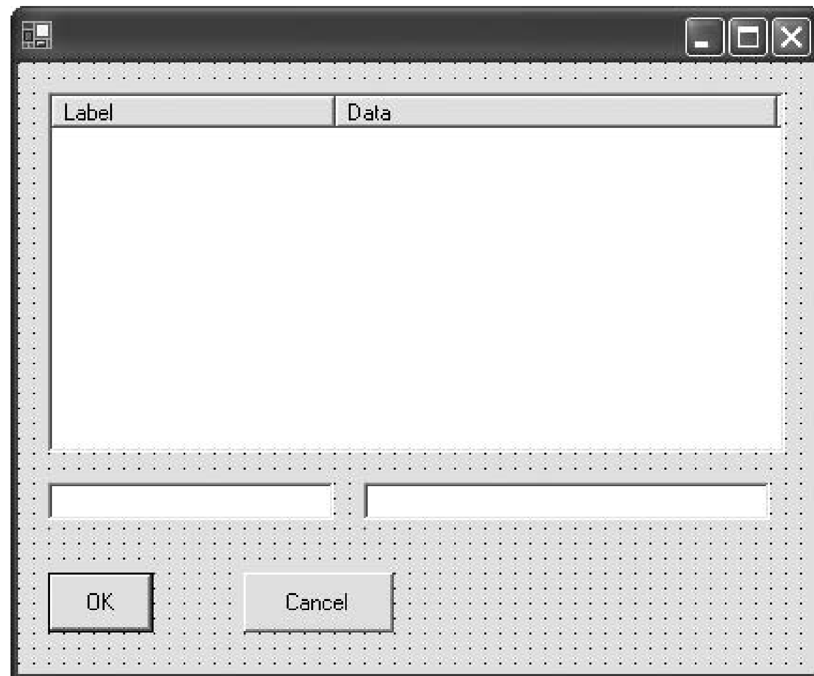


Figure 13.6

When we use the form to display or edit an existing contact, we load the list with label-value pairs without knowing (or caring) which field is really which.

```
public RCNTContact mContact;

public ContactEditForm(ref RCNTContact aContact)
{
  mContact = aContact;
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();

  // Populate the fields list
  for(int fi = 0 ; fi < aContact.mFields.Count ; fi++)
  {
    RCNTField newField = (RCNTField)aContact.mFields[fi];
    System.Windows.Forms.ListViewItem newItem = new ListViewItem();
    newItem.Tag = newField;
    newItem.Text = newField.mFieldLabel;
    newItem.SubItems.Add(newField.mFieldText);

    ContactDetailsList.Items.Add(newItem);
  }
}
```

However, by default we have not stored all fields for our contacts. Therefore, this form is going to show just the name fields. When the user chooses to view or edit a contact, we need to fetch all the fields for the contact. We could have chosen to retrieve the contact details and then put up the dialog, but instead, for reasons that will become apparent, I have chosen to put up the dialog and then fetch the contact details. When the contact details are received, we need to be able to overwrite the previously stored data.

```
public void ResetContact(ref RCNTContact aContact)
{
  mContact = aContact;
  // Repopulate the fields list
  ContactDetailsList.Items.Clear();
  for(int fi = 0 ; fi < aContact.mFields.Count ; fi++)
  {
    RCNTField newField = (RCNTField)aContact.mFields[fi];
    System.Windows.Forms.ListViewItem newItem = new ListViewItem();
    newItem.Tag = newField;
    newItem.Text = newField.mFieldLabel;
    newItem.SubItems.Add(newField.mFieldText);

    ContactDetailsList.Items.Add(newItem);
  }
}
```

Within the edit form we will not try to allow the user to edit the fields in place, but whenever a field is selected we will copy the label and value into text boxes and then reflect any changes to the value back to the value in the list.

```
private void ContactDetailList_SelectedIndexChanged(object sender,
                                                    System.EventArgs e)
{
  FieldLabelBox.Clear();
  FieldTextBox.Clear();
  ListView.SelectedIndexCollection indices =
                     ContactDetailsList.SelectedIndices;
  if( indices.Count >= 1)
  {
    int selIndex = indices[0];
    RCNTField selField = (RCNTField)ContactDetailsList.
                                    Items[selIndex].Tag;
    FieldLabelBox.Text = selField.mFieldLabel;
    FieldTextBox.Text = selField.mFieldText;
  }//endif got a selection
}

private void FieldTextBox_TextChanged(object sender, System.EventArgs e)
{
  ListView.SelectedIndexCollection indices =
                      ContactDetailsList.SelectedIndices;
  if( indices.Count >= 1)
  { // Set the field text, mark it changed and update the list
    int selIndex = indices[0];
    RCNTField selField = (RCNTField)ContactDetailsList.
                                    Items[selIndex].Tag;
    selField.mFieldText = FieldTextBox.Text;
    selField.mChanged = true;
    ContactDetailsList.Items[selIndex].SubItems[1].Text =
                                       FieldTextBox.Text;
  } //endif got a selection
}
```

This means that the button-click routine for the edit button has two logical parts to it. The first part raises the dialog and writes a command to retrieve the contact data. The second part takes the data from the form and writes the changed details to the smartphone.

This method also shows the use of the `mLostPhone` member. This is set to false before the dialog is raised and is checked afterwards. It will be set only if the currently connected smartphone disconnects while the dialog is raised. We use the same technique for all the button-click event handlers.

```
private void EditButton_Click(object sender, System.EventArgs e)
{
  if(ContactList.SelectedIndices.Count <= 0)
  {
    return;
  }
```

```
  mLostPhone = false;
  int editIndex = ContactList.SelectedIndices[0];
  RCNTContact editContact = (RCNTContact)ContactList.
                                       Items[editIndex].Tag;
  mPendingEditContact = editContact.mId;
  ArrayList contactSet = new ArrayList();
  contactSet.Add(mPendingEditContact);
  RCNTUtils.WriteFetchContactSet(mStream, ref mNextMessageId,
                                        ref contactSet );
  StatusBox.Text = "Fetching contact details - please wait";
  mPendingRead = true;
  mReadingLength = true;
  mStream.Read(4);

  mEditForm = new ContactEditForm(ref editContact);
  if(mEditForm.ShowDialog()== DialogResult.OK)
  { // Check for some fields changed
    bool newData = false;
    editContact = mEditForm.mContact;
    for(int fi = 0 ; fi < editContact.mFields.Count ; fi++)
    {
      RCNTField field = (RCNTField)mEditForm.mContact.mFields[fi];
      if(field.mChanged)
      {
        newData = true;
        break;
      }
    }
    if(newData && !mLostPhone)
    { // Save the new contact
      RCNTUtils.WriteEditContact(mStream, ref mNextMessageId,
                                        ref editContact);
      StatusBox.Text = "Saving changes to contact - please wait";
      mPendingRead = true;
      mReadingLength = true;
      mStream.Read(4);
      ContactList.Items.RemoveAt(editIndex);
      AddCard(ref editContact, editIndex);
    }// endif got new data
  }// endif OK from edit form
  SetButtonStates();
}
```

The contact being viewed or edited is fetched using the command to retrieve a set of contacts, although in this case the set always contains only one entry.

```
/// Fetch a set of contacts
public static void WriteFetchContactSet
                   ( BALApplicationAsyncStream aStream,
                     ref int aNextMessageId, ref ArrayList aIds )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERCntCmdFetchContactSet, ref message);
  ConnPack.WriteInt32(aNextMessageId, ref message);
  ConnPack.WriteInt8(1, ref message); // Do fetch all fields
```

```
  for(int i = 0 ; i < aIds.Count ; i++)
  {
    int ID = (int)aIds[i];
    ConnPack.WriteInt32(ID, ref message);
  }
  ConnPack.WriteInt32(-1, ref message);
  aNextMessageId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

When the contact details are received, we need to recognize that they are destined for the edit form. In order to access the edit form, a reference to it has been stored in `EditButton_Click()` and so the `ServiceRead()` method can overwrite the details. If this was slow then we would have an unfriendly dialog, but, in practice, the delay is not visible.

If this model of raising the dialog and then correcting its data was not acceptable (because of delays) then we could have the `EditButton_Click()` method send the command to retrieve the contact details and have a method to raise the dialog called when the response is received.

When we write an edited contact back to the phone, we can choose to send only those fields that have changed.

```
/// Edit a contact
public static void WriteEditContact( BALApplicationAsyncStream aStream,
                                     ref int aNextPDUId,
                                     ref RCNTContact aContact )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERCntCmdEditContact, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteInt32(aContact.mId, ref message);

  for(int i = 0 ; i < aContact.mFields.Count ; i++)
  {
    RCNTField field = (RCNTField)aContact.mFields[i];
    if(field.mChanged)
    {
      for(int fi = 0 ; fi < field.mFieldType.Count ; fi++)
      {
        ConnPack.WriteInt32((int)field.mFieldType[fi], ref message);
      }
      ConnPack.WriteInt32(-1, ref message);
      ConnPack.WriteInt32(field.mMapping, ref message);
      ConnPack.WriteUNCData(field.mFieldText, ref message);
    }
  }
  ConnPack.WriteInt32(-1, ref message);

  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

Deleting a contact is simply a matter of finding the contact identifier and constructing the appropriate PDU. Again, we check for the smartphone having disconnected while the dialog is raised using the `mLostPhone` member.

```
private void DeleteButton_Click(object sender, System.EventArgs e)
{
  if(ContactList.SelectedIndices.Count <= 0)
  {
    return;
  }
  mLostPhone = false;
  int delIndex = ContactList.SelectedIndices[0];
  RCNTContact delContact = (RCNTContact)ContactList.
                                        Items[delIndex].Tag;
  if ((MessageBox.Show ("Are you sure you want to delete this
                         contact ?",
    "Confirm Delete",
    MessageBoxButtons.YesNo, MessageBoxIcon.Question,
    MessageBoxDefaultButton.Button2 )
    == DialogResult.Yes) && !mLostPhone)
  {
    RCNTUtils.WriteDeleteContact(mStream, ref mNextMessageId,
                                          delContact.mId);
    StatusBox.Text = "Deleting contact - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    ContactList.Items.RemoveAt(delIndex);
  }
  SetButtonStates();
}
```

As with some of the earlier actions, we will need to extend `ServiceRead()` for the reply when we edit or delete contacts, but we are not concerned with the contents of the PDU.

We can create a new contact using the edit form.

```
private void CreateButton_Click(object sender, System.EventArgs e)
{
  mLostPhone = false;
  RCNTUtils.CreateNewContact(ref mTemplateFields,
                             out mPendingCreateContact);
  ContactEditForm editForm = new ContactEditForm
                            (ref mPendingCreateContact);

  if(editForm.ShowDialog()== DialogResult.OK)
  { // Check for some fields changed
    bool newData = false;
    for(int fi = 0 ; fi < mPendingCreateContact.mFields.Count ; fi++)
    {
      RCNTField field = (RCNTField)mPendingCreateContact.mFields[fi];
      if(field.mChanged)
      {
        newData = true;
```

```
        break;
      }
    }
    if(newData && !mLostPhone)
    { // Save the new contact
      RCNTUtils.WriteCreateContact(mStream, ref mNextMessageId,
                                   ref mPendingCreateContact);
      StatusBox.Text = "Creating new contact - please wait";
      mPendingRead = true;
      mReadingLength = true;
      mStream.Read(4);
    }// endif got new data
  }// endif OK from create form
  SetButtonStates();
}
```

We can create a new contact object using the template fields.

```
// Creates a new contact based on template fields
static public void CreateNewContact(ref ArrayList aTemplateFields,
                                    out RCNTContact aContact)
{
  aContact = new RCNTContact();
  aContact.mId = 0;
  aContact.mFields = new ArrayList(aTemplateFields);
}
```

Once again, we choose to send data only for fields containing some data.

```
/// Create a contact
public static void WriteCreateContact
                   ( BALApplicationAsyncStream aStream,
                     ref int aNextPDUId, ref RCNTContact aContact )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERCntCmdCreateContact, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);

  for(int i = 0 ; i < aContact.mFields.Count ; i++)
  {
    RCNTField field = (RCNTField)aContact.mFields[i];
    if(field.mChanged)
    {
      for(int fi = 0 ; fi < field.mFieldType.Count ; fi++)
      {
        ConnPack.WriteInt32((int)field.mFieldType[fi], ref message);
      }
      ConnPack.WriteInt32(-1, ref message);
      ConnPack.WriteInt32(field.mMapping, ref message);
      ConnPack.WriteUNCData(field.mFieldText, ref message);
    }
  }
  ConnPack.WriteInt32(-1, ref message);
```

```
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

At this point we have a contacts application that allows the user to browse through their contacts and to edit or delete existing contacts or to create new ones. It does not make any use of the find methods and it has no explicit integration with any other application. If you were building a combined application then you would almost certainly choose to integrate more.

13.7 An Agenda GUI Application

Our final application to consider at this point is an Agenda application. There is a lot of scope for good or clever GUI design (the two are not always the same) to present the Agenda information in ways that are helpful to the user. For example, Agenda applications may have year, month, week or day views, and they may choose various ways to show free time or days when appointments are booked. In this application, I have chosen to display only one day of information at a time, and the Agenda Connectivity service was written with this in mind. If you want to display other views then you should consider extending the service on the Symbian OS smartphone to retrieve more than one day's data at a time; creating a year view one day at a time would have serious performance problems.

When the Contacts application obtained a connection, it sent some commands to set the view and retrieve template information. In a similar way, we will open the calendar file for our Agenda application. We saw that the Agenda Model, unlike the Calendar Model, requires us to explicitly specify the name of the calendar file. Unfortunately, different models of Symbian OS smartphone use different locations and names for their calendar file. Therefore, we need to choose the file name based on the phone manufacturer and model. In my example I have handled only two manufacturers and I have specified the models that I have tested this software with. It would be possible to be more relaxed and try different locations or names – as the service responds with an error number if the calendar file is not found, this would be manageable. I could have put the code to select the file name in the service on the phone, but it is easier to upgrade PC software. By devolving the choice of name to the PC application, I provide the ability to support new devices at a later date simply by releasing an upgrade to the PC side. If I wanted to be really clever, I could put the data about which manufacturers and models have which named calendar file into the registry, and then supporting new phones would require simply releasing a registry upgrade file.

```
/// <summary>
/// Initializes the service we need on the phone.
/// </summary>
private void InitializePhoneService()
{
  // Check for a phone with a known calendar file
  string manuf = mCurrentPhone.Manufacturer;
  string model = mCurrentPhone.Model;
  string calendarFile = "unknown";
  if((manuf == "Nokia") &&
     ((model=="7650") || (model=="3650") || (model=="6600")))
  { calendarFile = "C:\\System\\Data\\Calendar"; }
  else if((manuf == "SonyEricsson") &&
     ((model=="P800") || (model=="P900")))
  { calendarFile = "C:\\Documents\\Agenda\\Agenda"; }
  else
  {
    MessageBox.Show("The connected phone is not a recognized type",
      "Unrecognized Model Error",
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    return;
  }
  int errorCode = mCurrentPhone.StartService("RAGNCS", out mStream);
  if(errorCode != BALConnectedPhone.KErrNone)
  {
    MessageBox.Show("The Agenda plug-in was not found",
      "Service Missing Error",
      MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
  else
  {
    mStream.OnRead += new
            ISCBALSequentialStreamSink_OnReadEventHandler(OnRead);
    mStream.OnWrite += new
            ISCBALSequentialStreamSink_OnWriteEventHandler(OnWrite);

    // Clear data related to this phone
    mAgendaOpen = false;
    mGotAppts = false;

    // try to open an agenda file - the name depends on the phone
    // connected
    RAGNUtils.WriteOpenAgenda(mStream, ref mNextPDUId, calendarFile);
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    StatusBox.Text = "Opening Agenda File - please wait";
    SetButtonStates();
  }
}
```

```
/// Open agenda file by name
public static void WriteOpenAgenda( BALApplicationAsyncStream aStream,
                            ref int aNextPDUId, string aAgendaName )
{
  ArrayList message = new ArrayList();
```

```
  ConnPack.WriteInt32(ERAgnCmdOpenAgenda, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteUNCData(aAgendaName, ref message);
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

The main form for this application has a list display that allows us to display summary information on all appointments or events for a day, and controls to select the day concerned and a range of actions, as shown in Figure 13.7.

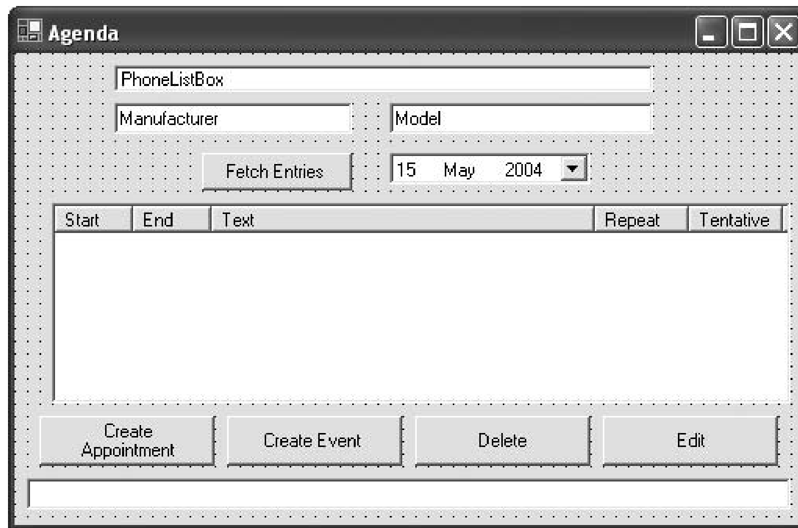


Figure 13.7

Microsoft Windows provides a very convenient control to allow the user to select a date and so retrieving a day's worth of data can be easily initiated. In my experience, it is unlikely that a single day will have more entries than can be retrieved in one go, so we do not include support for this. If your entries are likely to include extensive notes or other data then you might need to take a different approach.

```
private void FetchButton_Click(object sender, System.EventArgs e)
{
  ClearApptList();
  // Symbian OS wants a date string of the form YYYYMMDD:
  // Where MM and DD are the month and day numbers but zero-based.
  int year = DatePicker.Value.Year;
  int month = DatePicker.Value.Month-1;
  int day = DatePicker.Value.Day-1;
  string dateString = year.ToString() + month.ToString("D2")
                                      + day.ToString("D2") + ":";
```

```
  RAGNUtils.WriteFetchApptsByDay(mStream, ref mNextPDUId, dateString);
  mPendingRead = true;
  mReadingLength = true;
  mStream.Read(4);
  StatusBox.Text = "Fetching entries - please wait";
  SetButtonStates();
}

/// Fetch appointments by day - based on a date string of the form
/// YYYYMMDD, where MM and DD are zero based
public static void WriteFetchApptsByDay
  (BALApplicationAsyncStream aStream, ref int aNextPDUId, string aDate)
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERAgnCmdFetchApptsByDay, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteASCIIData(aDate, ref message);
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

Reading entries back follows the same pattern as for the previous applications. Note that we can expect a combination of appointments, events and anniversaries. Luckily, the differences that we care about are relatively small and so we can store all types of entry in one class.

```
public class RAGNDateTime
{
  public bool mIsNull; // Is the date  time null - i.e. unset
  public DateTime mDate; // date and time (if not null)

  public RAGNDateTime()
  {
    mDate = new DateTime();
  }
}

public class RAGNInstance
{
  public int mInstanceId; // instance identifier
  public RAGNDateTime mDate; // instance date

  public RAGNInstance()
  {
    mDate = new RAGNDateTime();
  }
}

public class RAGNEntry
{
  public int mEntryType; // type - appointment, event or anniversary
  public RAGNInstance mInstance; // instance information
  public RAGNDateTime mStartDate; // start date and time
  public RAGNDateTime mEndDate; // end date and time
  public int mDisplayTime; // display time in minutes after midnight
```

```
  public int mAlarmFlag; // true if the appointment has an alarm set
  public int mAlarmDays; // days before start for the alarm
  public int mAlarmMinutes; // mins after midnight for alarm to start
  public int mCrossedOut; // appointment crossed out?
  public int mTentative; // appointment tentative?
  public int mDayNote; // appointment a day note?
  public int mRepeating; // appointment part of repeating sequence?
  public string mText; // appointment text

  public RAGNEntry()
  {
    mInstance = new RAGNInstance();
    mStartDate = new RAGNDateTime();
    mEndDate = new RAGNDateTime();
    mDisplayTime = 0;
    mAlarmFlag = 0;
    mTentative = 0;
    mCrossedOut = 0;
    mDayNote = 0;
    mText = "";
  }
}
```

```
/// <summary>
/// Handle a PDU read from the device
/// </summary>
private void ServiceRead(int aOpCode, int aPDUId, ref byte[] aBuff,
                         ref int aReadPos)
{
  int errno = 0;
  switch(aOpCode)
  {
    case(RAGNUtils.ERAgnCmdFetchApptsReply):
      StatusBox.Text = "Reading Entries - please wait";
      bool carryOn = true;
      bool gotAtLeastOne = false;
      while(carryOn)
      {
        RAGNEntry appt = new RAGNEntry();
        carryOn = RAGNUtils.ReadAppt( ref aBuff, ref aReadPos,
                                      ref appt);
        if(carryOn)
        {
          gotAtLeastOne = true;
          AddEntry(ref appt, -1);
        }
      }
      if(gotAtLeastOne)
      { mGotAppts = true;}
      break;
...
```

```
/// Read an appointment from the PDU - terminated by a -1 value for type
static public bool ReadAppt(ref byte[] aBuff, ref int aBuffOffset,
                            ref RAGNEntry aAppt)
```

```
{
  int apptType = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  if(apptType == -1)
  {
    return false;
  }
  aAppt.mEntryType = apptType;
  int instanceId = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  aAppt.mInstance.mInstanceId = instanceId;
  ConnPack.ReadDateTime(ref aBuff, ref aBuffOffset,
                        out aAppt.mInstance.mDate.mIsNull,
                        out aAppt.mInstance.mDate.mDate);

  aAppt.mCrossedOut = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  aAppt.mTentative = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  aAppt.mDayNote = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  aAppt.mRepeating = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  aAppt.mAlarmFlag = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  if(aAppt.mAlarmFlag != 0)
  {
    aAppt.mAlarmDays = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
    aAppt.mAlarmMinutes = ConnPack.ReadInt32(ref aBuff,
                                             ref aBuffOffset);
  }
  System.Text.StringBuilder apptName;
  ConnPack.ReadData(ref aBuff, ref aBuffOffset, out apptName);
  aAppt.mText = apptName.ToString();

  ConnPack.ReadDateTime(ref aBuff, ref aBuffOffset,
           out aAppt.mStartDate.mIsNull, out aAppt.mStartDate.mDate);
  ConnPack.ReadDateTime(ref aBuff, ref aBuffOffset,
           out aAppt.mEndDate.mIsNull, out aAppt.mEndDate.mDate);
  if(apptType != RAGNUtils.EAppt)
  {
    aAppt.mDisplayTime = ConnPack.ReadInt32(ref aBuff, ref aBuffOffset);
  }
  else
  {
    aAppt.mDisplayTime = 0;
  }
  return true;
}
```

When we add an entry to the day view, we handle appointments differently from events and anniversaries because the times work slightly differently.

```
/// <summary>
/// Add an appointment or event to the view
/// if aWhere is negative then append the item, otherwise insert it
/// </summary>
private void AddEntry(ref RAGNEntry aAppt, int aWhere)
{
  System.Windows.Forms.ListViewItem newItem = new ListViewItem();
  newItem.Tag = aAppt;
  // Set the start time and end time for an appointment
```

```
  if(aAppt.mEntryType == RAGNUtils.EAppt)
  {
    newItem.Text = aAppt.mStartDate.mDate.ToShortTimeString();
    newItem.SubItems.Add(aAppt.mEndDate.mDate.ToShortTimeString());
  }
  // Set the display time and a blank end for an event or anniversary
  else
  {
    int hours = aAppt.mDisplayTime / 60;
    int mins = aAppt.mDisplayTime % 60;
    newItem.Text = hours.ToString() + ":" + mins.ToString("d2");
    newItem.SubItems.Add("");
  }
  newItem.SubItems.Add(aAppt.mText);
  string repeatText = "";
  if(aAppt.mRepeating != 0)
  {repeatText = "R";}
  newItem.SubItems.Add(repeatText);
  string tentativeText = "";
  if(aAppt.mTentative != 0)
  {tentativeText = "T";}
  newItem.SubItems.Add(tentativeText);

  if((aWhere >= 0) && (aWhere < ApptList.Items.Count))
  {
    ApptList.Items.Insert(aWhere,newItem);
  }
  else
  {
    ApptList.Items.Add(newItem);
  }
}
```

At this point, we have a useful application that allows the user to select a day and display entries in summary form. If we want to display all the data for an entry, or to create or edit an entry, then we need a form with controls for all the data that we are concerned with. We will need two forms, one for appointments and one for events. The appointments form has the start and end date and time, text and properties, as shown in Figure 13.8.

We will use a very similar form for events, the only difference being that a display time is added. Anniversaries can be treated as events, and this application can display them and allow some editing.

The code behind this form is concerned with initializing the controls from a received object and updating the object with the control contents on exit.

```
public RAGNEntry mEntry;

public ApptForm(ref RAGNEntry aAppt)
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();
```

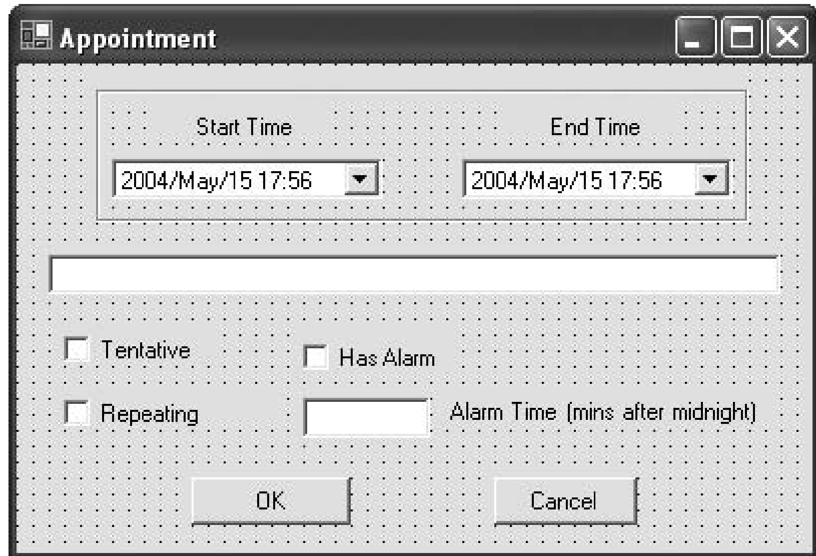


Figure 13.8

```
  mEntry = aAppt;
  StartTimePicker.Value = new DateTime(aAppt.mStartDate.mDate.Year,
    aAppt.mStartDate.mDate.Month, aAppt.mStartDate.mDate.Day,
    aAppt.mStartDate.mDate.Hour, aAppt.mStartDate.mDate.Minute,
    aAppt.mStartDate.mDate.Second);
  EndTimePicker.Value = new DateTime(aAppt.mEndDate.mDate.Year,
    aAppt.mEndDate.mDate.Month, aAppt.mEndDate.mDate.Day,
    aAppt.mEndDate.mDate.Hour, aAppt.mEndDate.mDate.Minute,
    aAppt.mEndDate.mDate.Second);
  ApptText.Text = aAppt.mText;
  RepeatCheckBox.Checked = (aAppt.mRepeating != 0);
  TentativeCheckBox.Checked = (aAppt.mTentative != 0);
  AlarmCheckBox.Checked = (aAppt.mAlarmFlag != 0);
  if(aAppt.mAlarmFlag != 0)
  {
    AlarmTimeBox.Text = aAppt.mAlarmMinutes.ToString();
  }
}

/// <summary>
///  Put current values into entry
/// </summary>
private void OKButton_Click(object sender, System.EventArgs e)
{
  mEntry.mStartDate.mDate = StartTimePicker.Value;
  mEntry.mEndDate.mDate = EndTimePicker.Value;
  mEntry.mText = ApptText.Text;
  mEntry.mTentative = 0;
  if(TentativeCheckBox.Checked)
  {
    mEntry.mTentative = 1;
  }
```

```
  mEntry.mAlarmFlag = 0;
  if(AlarmCheckBox.Checked)
  {
    mEntry.mAlarmFlag = 1;
    mEntry.mAlarmMinutes = system.Convert.ToInt32(AlarmTimeBox.Text);
  }
}
```

Given this code, the code to edit an appointment follows the pattern that we saw in the contacts application, except that it has to choose which type of edit form to bring up depending on the entry type.

```
private void EditButton_Click(object sender, System.EventArgs e)
{
  mLostPhone = false;
  if(ApptList.SelectedIndices.Count <= 0)
  {
    return;
  }
  int editIndex = ApptList.SelectedIndices[0];
  RAGNEntry editEntry = (RAGNEntry)ApptList.Items[editIndex].Tag;
  bool saveData = false;
  if(editEntry.mEntryType == RAGNUtils.EAppt )
  {
    ApptForm editForm = new ApptForm(ref editEntry);
    if(editForm.ShowDialog()== DialogResult.OK)
    {saveData = true;}
  }
  else
  {
    EventForm editForm = new EventForm(ref editEntry);
    if(editForm.ShowDialog()== DialogResult.OK)
    {saveData = true;}
  }
  if(saveData && !mLostPhone)
  { // Save the changed entry
    RAGNUtils.WriteEditEntry(mStream, ref mNextMessageId,
                             ref editEntry);
    StatusBox.Text = "Saving changes to entry - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    ApptList.Items.RemoveAt(editIndex);
    AddEntry(ref editEntry, editIndex);
  }// endif OK from edit form
  SetButtonStates();
}
```

There is one glaring omission from this code – any form of validity checking. In a commercial application, it would be necessary to check that the end date and time are no earlier than the start time and to query the user if the times are in the past.

```
public static void WriteEditEntry( BALApplicationAsyncStream aStream,
                            ref int aNextPDUId, ref RAGNEntry aAppt )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERAgnCmdEditAppt, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteInt32(aAppt.mInstance.mInstanceId, ref message);
  ConnPack.WriteDateTime(false, aAppt.mInstance.mDate.mDate,
                         ref message);
  ConnPack.WriteInt32(ECurrentInstance, ref message);
  // Set to edit times - different ones based on type
  ConnPack.WriteInt8(1, ref message);
  ConnPack.WriteDateTime(aAppt.mStartDate.mIsNull,
                         aAppt.mStartDate.mDate, ref message);
  ConnPack.WriteDateTime(aAppt.mEndDate.mIsNull,
                         aAppt.mEndDate.mDate, ref message);
  if( aAppt.mEntryType != EAppt)
  {
    ConnPack.WriteInt16(aAppt.mDisplayTime, ref message);
  }
  // Set to edit alarm details
  ConnPack.WriteInt8(1, ref message);
  ConnPack.WriteInt8(aAppt.mAlarmFlag, ref message);
  if( aAppt.mAlarmFlag != 0 )
  {
    ConnPack.WriteInt16(aAppt.mAlarmDays, ref message);
    ConnPack.WriteInt16(aAppt.mAlarmMinutes, ref message);
  }
  // Set other properties
  ConnPack.WriteInt8(aAppt.mCrossedOut, ref message);
  ConnPack.WriteInt8(aAppt.mTentative, ref message);
  ConnPack.WriteInt8(aAppt.mDayNote, ref message);
  ConnPack.WriteInt8(1, ref message);
  ConnPack.WriteUNCData(aAppt.mText, ref message);
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

As with other edit and create PDUs, we will need to extend the `ServiceRead()` method, but we do not need to pay any attention to the response.

The real reason for showing alarm details is to allow the user to edit them or to create a new appointment with an alarm. We have made no real attempt to make it easy to set an alarm time in this form; a more friendly approach would be to provide another date and time picker.

Another area that has been avoided here is any detail on repeating entries. Chapter 12 on the Agenda Model showed how complex Symbian OS allows repeating entries to be, and I prefer to avoid the issue. If you want to allow the user to create and edit repeating entries then you will probably choose to expose only a subset of the possible options. You will also have to extend the Agenda service to support whatever repeat functionality you have chosen.

Creating an appointment or an event involves creating a new entry and then using the same forms as we used earlier for edits. Because we have different forms for appointments and events, we need separate buttons and so separate (but very similar) methods.

```
private void CreateApptButton_Click(object sender, System.EventArgs e)
{
  mLostPhone = false;
  RAGNEntry newEntry = new RAGNEntry();
  newEntry.mEntryType = RAGNUtils.EAppt;
  newEntry.mStartDate.mDate = DateTime.Now;
  newEntry.mEndDate.mDate = DateTime.Now;
  ApptForm editForm = new ApptForm(ref newEntry);
  if(editForm.ShowDialog()== DialogResult.OK && !mLostPhone)
  { // Save the new entry
    RAGNUtils.WriteCreateEntry(mStream, ref mNextMessageId,
                                        ref newEntry);
    StatusBox.Text = "Saving new entry - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    AddEntry(ref newEntry, -1);
    mGotAppts = true;
  }// endif OK from edit form
  SetButtonStates();
}

private void CreateEventButton_Click(object sender, System.EventArgs e)
{
  mLostPhone = false;
  RAGNEntry newEntry = new RAGNEntry();
  newEntry.mEntryType = RAGNUtils.EEvent;
  newEntry.mStartDate.mDate = DateTime.Now;
  newEntry.mEndDate.mDate = DateTime.Now;
  EventForm editForm = new EventForm(ref newEntry);
  if(editForm.ShowDialog()== DialogResult.OK && !mLostPhone)
  { // Save the new entry
    RAGNUtils.WriteCreateEntry(mStream, ref mNextMessageId,
                                        ref newEntry);
    StatusBox.Text = "Saving new entry - please wait";
    mPendingRead = true;
    mReadingLength = true;
    mStream.Read(4);
    AddEntry(ref newEntry, -1);
    mGotAppts = true;
  }// endif OK from edit form
  SetButtonStates();
}
```

```
/// Create a new appointment or event
public static void WriteCreateEntry( BALApplicationAsyncStream aStream,
                   ref int aNextPDUId, ref RAGNEntry aEntry )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERAgnCmdCreateAppt, ref message);
```

```
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteInt32(aEntry.mEntryType, ref message);
  ConnPack.WriteDateTime(aEntry.mStartDate.mIsNull,
                         aEntry.mStartDate.mDate, ref message);
  ConnPack.WriteDateTime(aEntry.mEndDate.mIsNull,
                         aEntry.mEndDate.mDate, ref message);
  if(aEntry.mEntryType == RAGNUtils.EEvent)
  {
    ConnPack.WriteInt32(aEntry.mDisplayTime, ref message);
  }
  ConnPack.WriteInt8(aEntry.mAlarmFlag, ref message);
  if( aEntry.mAlarmFlag != 0 )
  {
    ConnPack.WriteInt16(aEntry.mAlarmDays, ref message);
    ConnPack.WriteInt16(aEntry.mAlarmMinutes, ref message);
  }
  ConnPack.WriteInt8(aEntry.mCrossedOut, ref message);
  ConnPack.WriteInt8(aEntry.mTentative, ref message);
  ConnPack.WriteInt8(aEntry.mDayNote, ref message);
  ConnPack.WriteUNCData(aEntry.mText, ref message);
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

If we wanted to maintain a good cache of device data then we would want to respond to the entry created response PDU by storing the instance identifier, but, for the purposes of this application, we can just ignore the response.

Because we are avoiding handling repeating entries in any detail, we will not allow the user to create an anniversary. It would be possible to create one using the event form, but without repeat information it is pointless.

Deletion simply requires obtaining the instance date and identifier and composing the appropriate PDU.

```
private void DeleteButton_Click(object sender, System.EventArgs e)
{
  mLostPhone = false;
  if(ApptList.SelectedIndices.Count <= 0)
  {
    return;
  }
  int editIndex = ApptList.SelectedIndices[0];
  RAGNEntry delEntry = (RAGNEntry)ApptList.Items[editIndex].Tag;
  if (MessageBox.Show ("Are you sure you want to delete this entry ?",
    "Confirm Delete",
    MessageBoxButtons.YesNo, MessageBoxIcon.Question,
    MessageBoxDefaultButton.Button2 )
    == DialogResult.Yes && !mLostPhone)
  {
    RAGNUtils.WriteDeleteInstance(mStream, ref mNextMessageId,
                                  ref delEntry.mInstance);
    StatusBox.Text = "Deleting entry - please wait";
    mPendingRead = true;
```

```
    mReadingLength = true;
    mStream.Read(4);
    ApptList.Items.RemoveAt(editIndex);
    if(ApptList.Items.Count == 0)
    { mGotAppts = false; }
  }
  SetButtonStates();
}
```

```
/// Delete an instance
public static void WriteDeleteInstance
                   ( BALApplicationAsyncStream aStream,
                     ref int aNextPDUId, ref RAGNInstance aInstance )
{
  ArrayList message = new ArrayList();
  ConnPack.WriteInt32(ERAgnCmdDeleteInstance, ref message);
  ConnPack.WriteInt32(aNextPDUId, ref message);
  ConnPack.WriteInt32(aInstance.mInstanceId, ref message);
  ConnPack.WriteDateTime(aInstance.mDate.mIsNull, aInstance.mDate.mDate,
                         ref message);
  ConnPack.WriteInt32(ECurrentInstance, ref message);
  aNextPDUId++;
  aStream.Write(ConnPack.AsByteArr(ref message));
}
```

13.8 Conclusion and Ideas for Further Development

That concludes the applications associated with the SMS, Contacts and Agenda services. These were an almost arbitrary selection chosen to illustrate some of the more common and more generally useful services. Not all the features included in the services created in earlier chapters were used (for example, to-do lists were omitted for reasons of space and because they did not demonstrate any new principles), but the intention was to show what can be achieved by creating a specialized service for a Symbian OS smartphone.

The most obvious direction in which GUI applications could be extended is that of integration. The SMS application, or a similar MMS or email application, is crying out for integration with contacts data, and a specialized meetings application could be created by selecting contacts entries and sending them messages while associating them with an appointment.

Although I have chosen to demonstrate GUI applications, the services will work at least as well with other applications such as databases or corporate systems (these are difficult to use as illustrations in a book). I used command-line applications to create and debug all of the device services. This kind of enterprise-level integration would allow Symbian OS smartphones to form a part of a much larger system.

14

Starting General Socket Servers

Earlier chapters have described how to create specialized services based on the Symbian OS Connectivity frameworks – either custom servers or socket servers. These are pretty good and are adequate for most purposes, but some developers may still want to use services that are general-purpose TCP/IP socket servers.

There are a number of reasons why you might want to use such a server:

- You may have a server that has not been designed for use with Connect and you may want to use it without converting it.
- You may need to get around the assumptions imposed by the custom server framework (being forced to operate within the `RunL()` of a process that you cannot debug has its limitations).
- You may want to create a service that can be used with a wide range of Symbian OS smartphones. I have designed the services described in this book so as to separate out the specialized functionality for reuse, but this still requires some different code for pre v8.0 and post v8.0 Symbian OS smartphones.

Having decided that you want to use a general-purpose TCP/IP server, there are several obstacles to be overcome:

1. You need to either use a fixed port number or have some way of publishing a port number.
2. You need a method of connecting a client on the PC to the listening port on the phone.
3. You need a method of starting the socket server when required.

I do not have any good solution to problem 1 – you will probably have to use a fixed port number. This is frowned upon in theory, as fixed TCP/IP port numbers are supposed to be allocated by the IANA, but, in practice,

you may risk choosing a number that you do not think will clash with anybody else's process. I don't recommend it, but I do recognize that it happens. The Symbian OS Connectivity framework for v8.0 and later smartphones solves this problem by using one fixed port for the Service Broker (and this fixed port number has been allocated by the IANA) and defining a message protocol that allows other servers to pass dynamically allocated port numbers back to the PC.

Problems 2 and 3 are addressed below.

14.1 Communicating with a Socket Server

A socket server works by listening on a port number. The following code is extracted from several different methods of a test socket server that was created to test this chapter.

```
class CTestSocketServer : public CActive
  {
  public:
  static CTestSocketServer* NewL(TUint16 aPortNumber,
                                 TUint16 aMaxConnections);

  CTestSocketServer();
  void ConstructL(TUint16 aPortNumber, TUint16 aMaxConnections);
  ~CTestSocketServer();
  void StartL();

  public: //from CActive
    void RunL();
    void DoCancel();
    TInt RunError(TInt aErr);

  private:
    RSocketServ iServer; // Session to the EPOC socket server (ESOCK)
    RSocket iSocket; // The server socket
    RSocket iClientSocket;
    TUint16 iPortNumber; // Port number to listen on
    TUint16 iMaxConnectionQueue; // Max connections

    CRx* iRx;
    CTx* iTx;
  };

void DoMainL()
{
  CActiveScheduler* p = new (ELeave) CActiveScheduler;
  CleanupStack::PushL(p);
  CActiveScheduler::Install(p);

  CTestSocketServer* server = CTestSocketServer::NewL(3050, 1);
  CleanupStack::PushL(server);
  server->StartL();
  p->Start();

  CleanupStack::PopAndDestroy(server);
  CleanupStack::PopAndDestroy(p);
```

```
}

CTestSocketServer* CTestSocketServer::NewL(TUint16 aPortNumber,
                                           TUint16 aMaxConnections)
  {
  CTestSocketServer* self = new (ELeave) CTestSocketServer();
  CleanupStack::PushL(self);
  self->ConstructL(aPortNumber, aMaxConnections);
  CleanupStack::Pop(self);
  return self;
  }

CTestSocketServer::CTestSocketServer():
  CActive(CActive::EPriorityStandard)
  {
  }

void CTestSocketServer::ConstructL(TUint16 aPortNumber,
                                   TUint16 aMaxConnections)
  {
  iPortNumber = aPortNumber;
  iMaxConnectionQueue = aMaxConnections;
  CActiveScheduler::Add(this);
  }

CTestSocketServer::~CTestSocketServer()
  {
  delete iRx;
  delete iTx;
  iSocket.Close();
  iServer.Close();
  }

void CTestSocketServer::StartL()
  {
  if (!IsActive())
    {
    User::LeaveIfError(iServer.Connect());
    User::LeaveIfError(iSocket.Open(iServer, KAfInet, KSockStream,
                       KProtocolInetTcp));

    TInetAddr address(KInetAddrAny,iPortNumber);
    User::LeaveIfError(iSocket.Bind(address));
    User::LeaveIfError(iSocket.Listen(iMaxConnectionQueue));

    User::LeaveIfError(iClientSocket.Open(iServer));
    iSocket.Accept(iClientSocket, iStatus);
    SetActive();
    }
  }
```

I will not go into too much detail on the APIs that have been used – they are the standard Symbian OS socket APIs and more details can be found in the SDKs. The result of the above code is to listen on socket 3050. If and when a client attempts to connect on this port number, the `CTestSocketServer::RunL()` method will be called and the socket server will use additional Active Objects to start communications. Commonly, it

will create an Active Object to call `RSocket::RecvOneOrMore()` to wait for commands.

Given a socket server on the phone listening on a known port, BAL provides a method of connection. You can connect to a listening port by using the `StartGeneralService()` method with a service name that is the port number in decimal, prefixed by a '#' character:

```
int serviceError = mPhone.StartGeneralService("#3050", out mAStream);
```

If a server is listening on the specified port then this will connect to it and return a stream that can be used in the same way as a stream that was used to connect to a named service.

14.2 Starting a Server

The final problem, that of starting the socket server when required, is less easily solved. You could put the socket server in a GUI application on the phone and require the user to start it manually, but that will not be regarded as friendly by the users (it will also get closed down if a backup is triggered which may be a problem for a Connectivity service). You could arrange for your service to be started whenever the Symbian OS smartphone boots up, but this is frowned upon. The available methods involve misusing other APIs; in any case, any running server takes up resources and a smartphone has only a limited amount of resources.

What we really want is to start the service from the PC only when we need to use it. This is exactly what the Service Broker provides for registered services from Symbian OS v8.0 onwards, so we need to implement our own service that provides a similar service.

We can create a PC Connectivity service (either a custom server or a specialized socket server) with a message protocol that allows us to start another server on the phone. It may sound as though we are having to create a specialized service which we were trying to avoid in the first place, but, in fact, we are creating a service that can be used to start any number of other servers and that allows us to keep our servers separate from any other PC Connectivity issues.

I will not include the custom server code, but the key method is as follows:

```
void CStartCSSession::StartServiceL( TInt aMsgId)
  {
  TInt nameLength = CConnPack::PeekInt32L(iReadPtr);
  HBufC* nameBuff = HBufC::NewLC(nameLength);
  TPtr namePtr(nameBuff->Des());
  CConnPack::ReadUNCDataL(namePtr, iReadPtr);
```

```
RLibrary lib;
RThread processHandle;

TInt ret = lib.Load(namePtr);
if (ret == KErrNone)
  {
  TLibraryFunction ordinal1 = lib.Lookup(1);
  TThreadFunction processFunc = REINTERPRET_CAST(TThreadFunction,
                                                 ordinal1());

  ret = processHandle.Create(namePtr, processFunc, KServerStackSize,
                             NULL, &lib, NULL, KServerInitHeapSize,
                             KServerMaxHeapSize, EOwnerProcess);
  lib.Close();
  }
CleanupStack::PopAndDestroy(nameBuff);

if (ret == KErrNone)
  {
  processHandle.Resume();

  iWritePtr.Zero();
  CConnPack::WriteInt32L(EStartResponse, iWritePtr);
  CConnPack::WriteInt32L(aMsgId, iWritePtr);
  Write(&iWritePtr);
  }
else
  {
  WriteErrorL(ret, aMsgId);
  }
}
```

We use `RLibrary::Load` to load the named DLL and then find the first ordinal for it. We will always want the first ordinal for a real server. Once we have a reference to the server main function, we can create a thread by calling `RThread::Create` and then start the thread by calling `RThread::Resume`.

If we wanted to be notified when the new thread exits, we could use the `RThread::Logon` method, but in this case we are content to start the thread and let it run.

The similar code for an EKA2 Symbian OS smartphone is simpler, as we do not need to load the library and obtain the ordinal directly. Instead of using an `RThread` we use an `RProcess` and we call `RProcess::Create()` using the name of the executable to be loaded, and then `RProcess::Resume()`.

If we are running on Symbian OS v8.0 or later then we will not use a custom server but a socket server, and the above code can be used in an `ExecuteLD()` call.

Using a service in this way requires that we install both the real server and the starter service on the Symbian OS smartphone, connect to the starter service by name, use it to start the real server, and then use a separate BAL or SCOM stream to connect to the waiting server.

15

Connectivity Dos and Don'ts

This chapter consists of pieces of advice for developers creating PC Connectivity services and applications. They could be presented as design patterns, but they are not all formally presented. They could be labeled 'design principles', but they are not all that grand. They are all useful, however, and based on real experience of PC Connectivity for Symbian OS smartphones.

15.1 Protocol Design

A PC Connectivity application is a true client-server application. You will need to design and implement a service on a Symbian OS smartphone and then implement a client on the PC to use the service. The protocol between the service and the client is a crucial part of your design.

15.1.1 Document Your Protocol

It can be tempting to leave your protocol undocumented if you maintain both the device-side and PC-side code, but resist the temptation. It does not require much effort to document the protocol and there are both short- and long-term advantages.

Plan your protocol as part of your design. By writing down the messages and responses that you expect, you will be forced to think about some of the details at an early stage.

It is quite likely that you may assign different developers to the device-side and PC-side development. A documented protocol will ease their interaction and allow them to proceed independently to some degree.

Expect your protocol to change during implementation. Unless you are very skilled or very lucky, your first version of the protocol will need to be adjusted. Some prototyping can reduce the scale of the changes, but some changes are almost inevitable. When the protocol does change, take the time to maintain the documentation, otherwise the difference between the documentation and reality will grow and may put you off maintenance later on.

15.1.2 Plan for a Maximum Packet Size

Plan for a fixed maximum size of Protocol Data Unit (PDU) in your protocol. The size can be large, but should be fixed. If you need to transfer an indeterminate amount of data then you need to design the protocol in such a way as to spread the data over multiple PDUs. If you avoid documenting the protocol then this issue of a maximum packet size may be overlooked until late in the development, at which time your developers may be reluctant to make changes.

The reason for a fixed maximum packet size is that the Symbian OS smartphone has limited resources and cannot support indeterminate sized buffers. There are APIs for buffers that can be extended but, if used to excess, these can crash your service or the phone.

When setting the size of a PDU, bear in mind that the data may be copied in an IP stack and so multiple copies may exist.

When designing a protocol to handle multi-PDU messages, do not just add all the incoming PDUs to a buffer – that defeats the purpose of splitting the message up in the first place. This implies that you will probably need to build on a system API that supports multi-part operations.

15.1.3 Plan for Expansion and Extension

If your protocol and service are any good then you may find that you discover new uses for them over time. If users like your application then they will suggest ways to extend it.

One key to a protocol that may be extended is some version information. Look forward to a time when many copies of your software are in circulation on PCs and Symbian OS smartphones and it may be possible for mismatched versions of client and service software to come together. If you have not planned for this then the results will be unfortunate as the two sides misunderstand each other. At best they will fail to communicate; at worst the client will cause unexpected damage to data on the phone.

If you include version information in your protocol (a simple fetch version command should suffice) then your client can check the version of a service after loading it. If your client is able to handle multiple versions of a protocol then it can take appropriate measures; if the client does not recognize the version then it can fail gracefully and inform the user of the problem.

One way to ease the handling of multiple versions is to be reluctant to change existing messages or responses. Instead, add new versions and maintain the old ones. If maintaining the old ones causes a maintenance burden then cause the old ones to return an error value of `KErrNotSupported` to the client. However, if a significant proportion of the protocol is changed then it may be better to create a service with a new name and a new protocol.

15.1.4 Plan for Debugging

In an ideal world you will always have your debugging tools available, but in real life you may have to make do with less than perfect information. In some cases it is possible to obtain dumps of TCP/IP traffic and that may be your only information. In these cases, a protocol that includes extra information (such as transaction identifiers) and some repetitive or redundant data (such as PDU counts, for example PDU 1 of 5) may be considerably easier to understand than a protocol that relies heavily on context for meaning. In addition, such protocols may reveal defects earlier, as integrity checks can be built in.

15.1.5 Consider Compression

One of the limitations of PC Connectivity performance is the bandwidth of the connection, so compressing data or tuning protocols to minimize data volumes may improve performance. However, there are always costs and trade-offs to consider.

Actually compressing data requires processing power at both ends and requires that data be copied between buffers. This may offset the performance gain of transferring less data. Also, there is a fixed overhead to sending and receiving a message and so compressing an already small message may have little effect. If a large amount of data, such as multi-megabyte files, is to be transferred then compression will provide a more relevant gain, although modern file formats, such as image and audio file formats, tend to be compressed in the first place and so further compression is likely to be ineffective.

When considering tuning protocols rather than just compressing data, for example using two-byte integers rather than four-byte integers, consider whether an increase in complexity is justified by the gain.

15.1.6 Consider Localization

Localization of applications should be second nature to any developer, but, while their GUI may be correctly localized, bear in mind that the communications protocol also requires consideration. If any text strings are transferred then consider whether they need to be localized, and bear in mind that the PC may be running a different language and locale from the Symbian OS smartphone. If you send text localized on the device and display it on the PC then it may not match the rest of the GUI.

Wherever possible, avoid transferring fixed-text strings and simply transfer identifiers that can be translated on the PC or the phone. As a bonus, these identifiers will also be smaller than the text and so will take less time to transfer.

15.2 Robustness and Defensive Design

Robustness and defensive programming have been promoted in software engineering for some time, but they are even more relevant to embedded devices than other types of hardware, and they are more relevant to communications-based applications than to other types of software.

Almost in contradiction, consumers expect higher levels of robustness from smartphones than from other computing devices that they are exposed to. Jokes about Microsoft Windows crashing may be unfair in view of the size and complexity of the software that users run and expect to coexist, but they are part of the PC culture; there is no such tolerance of crashes in smartphones – users expect them to just work. This means that any shortcomings in robustness will be highlighted.

15.2.1 Consider Running Out of Memory

Running low on memory is a standard issue for an embedded device. PC Connectivity applications are no more prone to it than any other type of Symbian OS software; nonetheless the issue is mentioned here because it bears repeating. Implement all Symbian OS software based on the approaches that have been developed and tested over a number of versions of Symbian OS.

15.2.2 Consider Loss of Power or Connection

Any computing device may be subject to loss of power, but a smartphone is more vulnerable than many others. The battery may run low because the user has forgotten to charge the device, or the user may remove the battery while tinkering with the phone.

More common (thankfully) than a complete loss of power is loss of a connection. If you are implementing an Over The Air (OTA) service then the smartphone may lose its signal at any time. PC Connectivity applications are less subject to loss of connection for this reason, but wireless connections can still be vulnerable and it is surprising how often users break the connection, whether accidentally or deliberately.

Loss of power gives very little time to save any data or recover, and loss of connection can be little better. Depending on the implementation, you may find your service simply unloaded, or you may get a chance to call a cleanup routine.

Therefore, you need to consider what happens if your service is terminated at any point. This is most relevant where you are writing data to the device, either writing directly to a file or writing to a database by means of some API. Where you are using a system API, you may be able to rely on it to manage transactions and recover from failure. If you are writing directly to the filing system then consider the consequences of a

partially written file, particularly if you have to write it across multiple PDUs because of the size of the file. In some cases it can be sensible to write to a temporary file and rename it for use when it is complete, but this has been known to backfire if the file is very large – in some cases the drive is not large enough for the original file and the new copy.

Although I have emphasized the need to protect the state of the smartphone, you should also consider the effect, on any data on the PC, of the connection to the device being lost. The first level is to include exception handling to catch a loss of connection wherever you communicate with the smartphone. The second level is to consider how to protect data from being partially written, for which similar approaches can be used to those on the smartphone.

15.2.3 Allow for a Missing Service

Fortunately, this issue is simpler to handle than some of the others. If you are using a specialized PC Connectivity service that you have developed, then you need to handle the possibility that the user may attempt to work with a smartphone that lacks the service, either because they have just installed the PC software and have not realized that it needs extra software installed on the phone, or because they are trying to use a phone other than their customary one. In either case, you can provide a simple and clear indication of the problem and the user will know how to correct it. If you do not respond helpfully, the user may simply dismiss your application as broken and never again try to use it.

15.2.4 Allow for Mismatched Versions

This issue was mentioned earlier in this chapter as a protocol design issue, but it is also another form of missing service. Allow for a phone with the wrong version of your service – check for it when the service is loaded and respond accordingly.

15.3 Device and Service Management

When implementing a PC Connectivity application, it is possible to focus on the headline functionality and test an application extensively using one PC and one Symbian OS smartphone and thus miss issues that will appear only with multiple phones. Similarly, a developer may well use a service in a different way from a real user and so miss situations that can cause problems.

15.3.1 Multiple Models of Symbian OS Smartphone

Symbian OS provides a good range of standard services that are common across multiple smartphones and multiple versions of Symbian

OS. However, there are often variations, some caused by upgrades (even improvements may have undesirable side effects) between versions of Symbian OS or just by smartphone manufacturers having different opinions on the best location for files or the best way to solve a problem.

In earlier chapters we have encountered differences in the locations and names of common types of file and this is a good example to consider. Unless Symbian OS mandates a location and a file name, do not assume that all Symbian smartphones will use the same values. Therefore, your software needs to allow for differences. It may be possible to detect differences on the device and so hide them from the PC, or it may be better to put the responsibility for detecting and handling differences on the PC because PC software is easier to extend.

However you handle the differences, be very wary of making assumptions about smartphones that you have not seen. You may think that a certain manufacturer always puts image files or their calendar database in the same location, but you may be unpleasantly surprised by their newest model of phone. If you must make assumptions then design your software so it can check the assumptions and respond robustly. For example, in the Agenda service implemented in Chapter 12 the command to load a calendar database fails with a specific error number if the database is not found, so the PC client can detect the problem and inform the user.

Along with designing your application to handle multiple models of smartphone, also test it with as many models as possible. A developer who does all his development on one phone is asking for embarrassment when it fails totally on another phone. As a case in point, I developed the services for this book on one phone and the same services crashed another model of phone immediately when loaded. The services worked perfectly well when using the emulator and the problem turned out to be caused by a flaw in system logging software. I knew of all the API changes between the phones, but I needed to carry out real tests on real phones to be sure that they would work.

It may be a good idea to distribute your application with a list of phones that it is known to work well with. This way users will not be disappointed by applications that claim to work with 'any Symbian OS smartphone' but do not work with *their* Symbian OS smartphone. The PC application could warn the user when it encounters an unknown phone and either allow or disallow the user to proceed. There are benefits to allowing users to proceed after issuing a warning – the chances may be good that things will work anyway, thus eliminating the need for you to update and re-release your PC software just to add another phone to the supported list. Also, you could provide a means for the users to contact you, letting you know that the PC application works fine with a new model of smartphone; you could then add this smartphone to the list of potentially supported ones for the next release of the PC software.

15.3.2 Why Device Identification is Important

As well as considering multiple models of phone, you need to consider multiple physical phones, whether of the same manufacturer or not. In a consumer environment you may not think that a user will ever connect more than one Symbian OS smartphone to a PC, but increasing use of mobile phones by all members of the family and increasing penetration of smartphones in the mobile phone market make that an unwise assumption. Smartphones are also being used more and more in the enterprise and corporate environment, and that may well be the trigger for developing PC Connectivity software in the first place.

Multiple connected phones raise two issues. Firstly, more than one smartphone may be connected at a time, particularly with bearers such as Bluetooth, and so PC Connectivity software must provide some way of deciding which one to communicate with. While developing the services for this book, I used command-line PC clients because they were very simple and they allowed me to focus on the device side, so that I was then able to develop the PC software on the assumption that I had relatively reliable services on the phone. These command-line clients could use just the first connected smartphone, and that was acceptable in a test application. The same behavior is not acceptable in a production application, and all of my GUI applications allow the user to select the phone to work with.

Early PC Connect software was developed on the assumption that there would only ever be one connected phone; subsequent work-arounds for this looked clumsy and were not intuitive to users. In contrast, SCOM and BAL have been designed to handle multiple connected phones from the start, and there is no excuse for not doing the same.

Even if your user has only one Symbian OS smartphone connected at a time, they may use multiple smartphones in series, i.e. different smartphones at different times. This does not cause any problems when directly working with the phone, but it does require some thought to associate data with the correct smartphone. As a good example, the backup software supplied by most Symbian OS smartphone manufacturers stores archives in different directories for each smartphone. This is achieved by using the device identifier that SCOM makes available (normally the IMEI number). You should try to do the same thing with any data that you store. Alternatively, you may associate data with a specific user, particularly in an enterprise environment, and you need to take care how a specific smartphone is associated with a specific user.

15.3.3 Loading and Unloading Services

Although not obvious, the choice of how and when to load and unload a service is as much a design decision as details of protocol. Loading a

service takes time – a message has to be sent to the Symbian OS smartphone, the service itself has to start up and will probably need to connect to one or more system servers, and then a response has to be sent back to the PC. It has been known for a PC client to load and unload a service for each transaction, but this is clumsy and reduces the responsiveness of the application.

However, it is also wasteful of scarce resources to load a service when it is not required. Setting a service to load at device boot time guarantees that it will always be available, but it also guarantees that it will absorb scarce resources even when the smartphone is not connected to a PC.

One aspect of a PC Connectivity service that is not obvious at first is that a service may have multiple simultaneous connections from the same PC. At the lowest level, a TCP/IP socket server can support multiple connections and Symbian OS PC Connectivity services are built on TCP/IP connections. In Symbian OS v8.0 and later, the socket servers spawn a separate client socket for each connection. When using custom servers, each connection uses a separate session. Unless you are certain that your service will only ever use one connection at a time, check your design for resources that are not thread-safe.

15.4 General Development and Debugging Skills

15.4.1 Learn to Love Logging

Modern debuggers and other development aids are very welcome and should be used wherever possible. However, there can be times when they cannot be used. The Symbian OS emulator works with the debuggers that are part of the IDEs, but PC connectivity does not always work well with a debugger; the connection can timeout when a break-point is encountered. In addition, some functions, such as drive formatting, camera access and audio playback, do not work well on the emulator; and software that performs well on the emulator does not always work properly on a real smartphone (sometimes due to the single-process design of the emulator).

If the debugger cannot be used then you will need to revert to older and more primitive methods of investigating behavior. The most common one is logging. Inserting logging statements makes for a slower debugging cycle and requires slightly more planning to be productive. You need to consider possible causes of a problem and insert a number of logging statements rather than putting in one at a time and running the service to the next stage. However, if you put in too many logging statements, particularly in loops, then the logging statements can cause the service to grind to a halt.

Rather than waiting until you encounter a problem that requires logging, make a habit of putting in logging from the start. Create a logging

class and use macros that compile to nothing in a release build, so that you do not need to remove it for a production build. In creating the services for this book I made extensive use of logging. I have removed the logging statements from the code in the text of the book for reasons of clarity – the logging statements can obscure the structure of the code – but they were essential during development.

15.4.2 Become Multi-skilled

Programming for Microsoft Windows requires considerable skill and experience and a wide range of specialist knowledge to do a good job. The same is true of programming for Symbian OS, but some of the skills and knowledge are different (some skills are common, of course). Some development teams address this difference by deciding that some developers are PC development specialists while others are phone development specialists. This works fine when all the software works correctly, but as soon as a defect is found that is not obvious, it will become clear that you may need to make changes to both components to trace the problem.

It is very useful if, at a minimum, any developer can read and understand the code that they communicate with and can insert logging statements or use the debugger to understand what is going on. If individual developers do not have these abilities then pair-programming becomes necessary.

15.4.3 Consider Simple Test Harnesses

When developing a PC Connectivity application, you will have to decide which part to develop first. I have found that I normally need to develop the service on the Symbian OS smartphone first, as the PC client depends too much on the service to be developed in isolation. However, you will need some form of client to load and drive the service before you can test it.

It is possible to develop both the service and the client simultaneously and then test them together, but this makes it harder to trace faults. If something goes wrong then it may not be obvious whether the fault is in the phone service or in the PC client. A documented and clear protocol can be valuable at this point, particularly if you have a means of dumping commands and responses to a log file.

However, by investing some time in a simplified PC test harness to drive the phone service you can make debugging easier. For all the services described in this book, I created a simple command-line client and I did not try to create the final PC client application until I had a relatively robust phone service. The command-line clients were so simple that they did not hide any obscure defects and so my attention was focused on the phone service at first.

Then, when I was implementing the PC client I assumed that the phone service was correct and that any defect was in the PC client. This was not always true, as the PC client sometimes used commands in a different way and so exposed some new defects in the phone service, but it was certainly true more often than not.

Finally, if you are really keen on good development practices then you could convert the command-line client into a scripted or automated test harness for future regression testing.

15.4.4 Don't Forget General Good Practices

This book has deliberately tried to highlight the aspects of programming that are specific to developing a PC Connectivity application for Symbian OS smartphones. However, do not think that this means that you can ignore more general good habits.

On the Symbian OS smartphone, all the normal behaviors associated with memory handling, robustness and efficient design apply to a Connectivity service as much as any other application or server.

On the PC, all the principles of good UI design apply. The PC client applications developed for this book are definitely not optimized UI designs; they were written to illustrate specific points, and there are a number of obvious ways in which they could be made more friendly if they were part of a commercial package.

As a final point on PC development, pay sufficient attention to the PC installer. Most developers build and test their software directly and so may neglect to create or test the installer until a late stage in the process. Installers need to be tested on a range of PCs, and uninstallation and upgrade scenarios need to be tested particularly carefully.

Appendix 1
Developer Resources

A1.1 Symbian OS Software Development Kits

SDKs are built based on a particular reference platform (sometimes known as a 'reference design') for Symbian OS. A reference platform provides a distinct UI and an associated set of system applications for such tasks as messaging, browsing, telephony, multimedia and contact/calendar management. These applications typically make use of generic application engines provided by Symbian OS. Reference platforms intended to support the installation of third-party applications written in native C++ have to be supported by an SDK which defines that reference platform, or at least a particular version of it. Since Symbian OS v6.0, four such reference platforms have been introduced, resulting in four flavors of SDK which can be found at the websites listed here:

- **UIQ** (***www.symbian.com/developer***)
- **Nokia Series 90** (***www.forum.nokia.com***)
- **Nokia Series 60** (***www.forum.nokia.com***)
- **Nokia Series 80** (***www.forum.nokia.com***)

Prior to this, SDKs were targeted at specific devices, such as the Psion netPad. Symbian no longer supports these legacy SDKs, but they are still available from Psion Teklogix at ***www.psionteklogix.com***.

For the independent software developer, the most important thing to know in targeting a particular phone is its associated reference platform. Then you need to know the Symbian OS version the phone is based on. This knowledge defines to a large degree the target phone as a platform for independent software development. You can then decide which SDK you need to obtain. In most cases, with a single version of your application, you will be able to target all phones based on the same reference platform and Symbian OS version working with this SDK. The Symbian OS System Definition papers give further details of possible differences between phones based on a single SDK.

- Symbian OS System Definition
  www.symbian.com/developer/techlib/papers/SymbOS_def/symbian_os_sysdef.pdf
- Symbian OS System Definition (in detail, including Symbian OS v8.0)
  www.symbian.com/developer/techlib/papers/SymbOS_cat/SymbianOS_cat.html

A1.2 Getting a UID for Your Application

A UID is a 32-bit number, which you get as you need from Symbian.

Every Uikon application should have its own UID. This allows Symbian OS to distinguish files associated with that application from files associated with other applications. UIDs are also used in other circumstances, such as to identify streams within a store, and to identify one or more of an application's views.

Getting a UID is simple enough. Just send an email to ***uid@symbiandevnet.com***, titled 'UID request' and requesting clearly how many UIDs you want – 10 is a reasonable first request. Assuming your email includes your name and return email address, that's all the information Symbian needs. Within 24 hours, you'll have your UIDs.

If you're impatient, or you want to do some experimentation before using real UIDs, you can allocate your own UIDs from a range that Symbian has reserved for this purpose: `0×01000000 to 0×0fffffff`. However, you should never release any programs with UIDs in this range.

Don't build different Symbian OS applications with the same application UID – even the same test UID – on your emulator or Symbian OS machine. If you do, the system will recognize only one of them, and you won't be able to launch any of the others.

A1.3 Symbian OS Developer Tools

As well as the following tools offerings from Symbian DevNet partners, Symbian DevNet provides a number of free and open source tools:

www.symbian.com/developer/downloads/tools.html

AppForge
Develop Symbian Applications Using Visual Basic and AppForge. AppForge development software integrates directly into Microsoft Visual Basic, enabling you to immediately begin writing multi-platform

applications using the Visual Basic development language, debugging tools and interface you already know.

www.appforge.com

Borland

Borland offers C++BuilderX Mobile Edition and JBuilder Mobile Edition as well as the more recent Borland Mobile Studio for developers that want to develop rapidly on Symbian OS using C++, Java or both. These multi-platform IDEs offer on-target debugging, GUI RAD and a unifying IDE for Symbian OS SDKs and compilers.

www.borland.com

Forum Nokia

In addition to a wide range of SDKs, Forum Nokia also offers various development tools to download, including the Nokia Developer Suite for J2ME, which plugs in to Borland's JBuilder MobileSet or Sun's Sun One Studio integrated development environment.

www.forum.nokia.com

Metrowerks

Metrowerks offers the following products supporting Symbian OS development:

- CodeWarrior Development Tools for Symbian OS Professional Edition
- CodeWarrior Development Tools for Symbian OS Personal Edition
- CodeWarrior Wireless Developer Kits for Symbian OS

www.metrowerks.com

Sun Microsystems

Sun provides a range of tools for developing Java 2 Micro Edition applications, including the J2ME Wireless Toolkit and Sun One Studio Mobile Edition.

http://java.sun.com

Texas Instruments

Development Tools for the OMAP Platform Easy-to-use software development environments are available today for OMAP application developers, OMAP Media Engine developers and device manufacturers. Tool suites that include familiar third-party tools and TI's own industry-leading eXpressDSP DSP tools are available, allowing developers to easily develop software across the entire family of OMAP processors.

http://focus.ti.com

Symbian DevNet Tools

Symbian DevNet offers the following tools as an unsupported resource to all developers:

- Symbian OS SDK add-ons
  www.symbian.com/developer/downloads/tools.html
- Symbian OS v5 SDK patches and tools archive
  www.symbian.com/developer/downloads/archive.html

A1.4 Support Forums

Symbian DevNet offers two types of support forum:

- Support newsgroups
  www.symbian.com/developer/public/index.html
- Support forum archive
  www.symbian.com/developer/prof/index.html

Symbian DevNet partners also offer support for developers:

Sony Ericsson Developer World

As well as tools and SDKs, Sony Ericsson Developer World provides a range of services, including newsletters and support packages for developers working with the latest Sony Ericsson products.

http://developer.sonyericsson.com

Forum Nokia

As well as tools and SDKs, Forum Nokia provides newsletters, the Knowledge Network, fee-based case-solving, a Knowledge Base of resolved support cases, discussion archives, and a wide range of C++ and Java-based technical papers of relevance to developers targeting Symbian OS.

http://forum.nokia.com/main.html

Sun Microsystems Developer Services

In addition to providing a range of tools and SDKs, Sun also provides a wide variety of developer support services including free forums, newsletters, and a choice of fee-based support programs.

- Forums
  http://forum.java.sun.com

- Support and newsletters
  http://developer.java.sun.com/subscription

A1.5 Symbian OS Developer Training

Symbian's Technical Training team and Training Partners offer public and on-site developer courses around the globe.

- Course dates and availability
  www.symbian.com/developer/training

Early bird discount: Symbian normally offers a 20% discount on all bookings confirmed up to 1 month before the start of any course. This discount cannot be used in conjunction with any other discounts.

Course	Level	Language
Symbian OS essentials	Introductory	C++
Java on Symbian OS	Introductory	Java
Symbian OS: Application engine development	Intermediate	C++
Symbian OS: Application UI development	Intermediate	C++
Symbian OS: Internals	Advanced	C++
Symbian OS: UI system creation	Advanced	C++

Please note
Intermediate and advanced courses require previous attendance of OS Essentials. UI system creation course also requires previous attendance of Application UI course.

A1.6 Developer Community Links

These community websites offer news, reviews, features and forums, and represent a rich alternative source of information that complements the Symbian Development Network and the development tools publishers. They are good places to keep abreast of new software and, of course, to announce the latest releases of your own applications.

My-Symbian
My-Symbian is a Poland-based website dedicated to news and information about Symbian OS phones. This site presents descriptions of new software for Symbian OS classified by UI. It also features discussion forums and an online shop.

http://my-symbian.com

All About Symbian

All About Symbian is a UK-based website dedicated to news and information about Symbian OS phones. The site features news, reviews, software directories and discussion forums. It has strong OPL coverage.

www.allaboutsymbian.com

SymbianOne

SymbianOne features news, in-depth articles, case studies, employment opportunities and event information all focused on Symbian OS. A weekly newsletter provides up-to-date coverage of developments affecting the Symbian OS ecosystem. This initiative is a joint venture with offices in Canada and New Zealand.

www.symbianone.com

NewLC

NewLC is a France-based collaborative website dedicated to Symbian OS C++ development. It aims to be initially valuable to developers just starting to write C++ applications for Symbian OS, and in time will cover more advanced topics.

www.newlc.com

infoSync World

infoSync World is a Norway-based site providing features, news, reviews, comments and a wealth of other content related to mobile information devices. It features a section dedicated to Symbian OS covering new phones, software and services, mixed with strong opinions that infoSync is not afraid to share.

http://symbian.infosyncworld.com

Your Symbian

Your Symbian (YS) is a fortnightly magazine distributed exclusively by email. YS takes a lighthearted look at the Symbian OS world. Major news is covered in its editorial and it includes a software round-up. To sign up, browse the archives, or get in touch with the editorial team.

www.yoursymbian.com

TodoSymbian (Spanish)

TodoSymbian is a Spain-based website for everyone wanting to read in Spanish about Symbian OS. It provides news, reviews, software directories, discussion forums, tutorials and a developers' section.

www.todosymbian.com

A1.7 Symbian OS Books

Symbian OS C++ for Mobile Phones, Vol. 2
Richard Harrison *et al*.
John Wiley and Sons, ISBN 0470871083

Symbian OS C++ for Mobile Phones, Vol. 1
Richard Harrison *et al*.
John Wiley and Sons, ISBN 0470856114

Symbian OS Explained
Jo Stichbury
John Wiley and Sons, ISBN 0470021306

Programming Java 2 Micro Edition on Symbian OS
Martin de Jode *et al*.
John Wiley and Sons, ISBN 0470092238

Wireless Java for Symbian Devices
Jonathan Allin *et al*.
John Wiley and Sons, ISBN 0471486841

Symbian OS Communications Programming
Michael J Jipping
John Wiley and Sons, ISBN 0470844302

Programming for the Series 60 Platform and Symbian OS
Digia, Inc.
John Wiley and Sons, ISBN 0470849487

Developing Series 60 Applications
Edwards, Barker
Addison Wesley, ISBN 032126875X

A1.8 Open Source Projects

Many open source projects are happening on Symbian OS. They are a rich source of partially or fully functional code which should prove useful to learn about use of APIs you're not yet familiar with. Please also consider contributing to any project that you have an interest in.

Repository websites

SymbianOS.org
http://symbianos.org

Community website dedicated to the development of open source programs for Symbian OS. Hosted projects include Vim, Rijndael encryption algorithm, MakeSis package for Debian GNU/Linux, etc.

Symbian open source
www.symbianopensource.com

Repository for Symbian OS open source software development. It provides free services to developers who wish to create, or have created, open source projects.

Open Source for EPOC32
www.edmund.roland.org/osfe.html

Website of Alfred Heggestad where he maintains a list of open source projects for Symbian OS.

Appendix 2

Specifications of Symbian OS Phones

Additional technical information on a limited number of phones can be found at ***www.symbian.com/phones***.

Please note that this is a quick guide to Symbian OS phones, some of which are not yet commercially available. The information contained within this appendix was correct at time of going to press.

For full, up-to-date information, refer to the manufacturer's website. C++ developers may retrieve extended information using HAL APIs.

Nokia 9210i

OS Version	Symbian OS v6.0
UI/Category	Series 80
Memory available to user	40 MB
Storage media	Yes; MMC
Screen	640 × 200, 4096 colors
Pointing device	No
Camera	No
Network Protocol(s)	GSM 900/1800 HSCSD
Java APIs	CLDC 1.0 MIDP 1.0 PersonalJava 1.1.1 JavaPhone
Connectivity	Infrared Serial

This Symbian OS smartphone uses PLP rather than m-Router and so cannot be accessed using the methods described in this book.

Browsing	WAP 1.1 XHTML (MP)

Nokia 7650

OS Version	Symbian OS v6.1
UI/Category	Series 60
Memory available to user	4 MB NOR flash user data storage
Storage media	No
Screen	176 × 208, 4096 colors
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800 HSCSD GPRS (2 + 2, 3 + 1, class B and C)
Java APIs	MIDP 1.0 CLDC 1.0 Nokia UI API
Connectivity	Infrared Bluetooth

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	WAP 1.2.1

Nokia 3600/3650

OS Version	Symbian OS v6.1
UI/Category	Series 60
Memory available to user	3.4 MB
Storage media	Yes; MMC
Screen	176 × 208, 4096/65536 colors
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800/1900
	HSCSD
	GPRS (2 + 2, 3 + 1, class B and C)
Java APIs	MIDP 1.0
	CLDC 1.0
	Nokia UI
	Wireless Messaging
	Mobile Media
Connectivity	Infrared
	Bluetooth

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	WAP 1.2.1
	XHTML (MP)

Nokia 3620/3660

OS Version	Symbian OS v6.1
UI/Category	Series 60
Memory available to user	4 MB
Storage media	Yes; MMC
Screen	176 × 208, 4096/65536 colors
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 850/1900 HSCSD GPRS
Java APIs	MIDP 1.0 CLDC 1.0 Nokia UI Mobile Media Wireless Messaging
Connectivity	Infrared Bluetooth

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	WAP 1.2.1 XHTML (MP)

Siemens SX1

OS Version	Symbian OS v6.1
UI/Category	Series 60
Memory available to user	16 MB
Storage media	Yes; MMC
Screen	176 × 220, 65536 colors TFT
Pointing device	No
Camera	Yes; 640 × 480 and 160 × 120 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (class 10, B (2Tx, 4Rx)
Java APIs	CLDC 1.0 MIDP 1.0 Wireless Messaging Mobile Media
Connectivity	Infrared Bluetooth USB

This Symbian OS smartphone ships with a PC suite based on SCOM and is fully compatible with the methods described in this book.

Browsing	WAP 2.0 XHTML (MP)

Nokia N-Gage

OS Version	Symbian OS v6.1
UI/Category	Series 60
Memory available to user	4 MB NOR flash user data storage
Storage media	Yes; MMC
Screen	176 × 208, 4096 colors
Pointing device	No
Camera	No
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (2 + 2, 3 + 1, class B and C)
Java APIs	MIDP 1.0 CLDC 1.0 Nokia UI Wireless Messaging Mobile Media
Connectivity	Bluetooth USB

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed. However, the USB connection is not compatible with SCOM.

Browsing	WAP 1.2.1 XHTML (MP)

Sendo X

OS Version	Symbian OS v6.1
UI/Category	Series 60
Memory available to user	12 MB
Storage media	Yes; MMC and SD
Screen	176 × 220, 65536 colors
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800/1900 GPRS (Class 8 (4 + 1))
Java APIs	MIDP1.0 Nokia UI Bluetooth Wireless Messaging Mobile Media
Connectivity	Infrared Bluetooth USB Serial

This Symbian OS smartphone ships with a PC suite based on SCOM and is fully compatible with the methods described in this book.

Browsing	WAP 2.0 XHTML (MP)

BenQ P30

OS Version	Symbian OS v7.0
UI/Category	UIQ 2.1
Memory available to user	32 MB
Storage media	Yes; MMC and SD
Screen	208 × 320, 65536 colors TFT
Pointing device	Yes
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800/1900
	HSCSD
	GPRS (4 + 2, class 10)
Java APIs	MIDP 2.0
	PersonalJava 1.1.1
	Wireless Messaging
Connectivity	Infrared
	Bluetooth
	USB

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	WAP 2.0
	XHTML (MP)

Sony Ericsson P800

OS Version	Symbian OS v7.0
UI/Category	UIQ
Memory available to user	12 MB
Storage media	Yes; Sony MS Duo
Screen	208 × 320 (Flip Open); 208 × 144 (Flip Closed), 4096 colors
Pointing device	Yes
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (4 + 1)
Java APIs	CLDC 1.0 MIDP 1.0 PersonalJava 1.1.1
Connectivity	Infrared Bluetooth USB (high speed serial connector with a USB->Serial adapter built into the desk stand) Serial

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	WAP 2.0 XHTML (MP)

Motorola A920/ A925

OS Version	Symbian OS v7.0
UI/Category	UIQ
Memory available to user	8 MB
Storage media	Yes; MMC and SD
Screen	208 × 320, 65536 colors TFT
Pointing device	Yes
Camera	Yes
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS 3G
Java APIs	MIDP 1.03 PersonalJava 1.1.1a
Connectivity	Infrared Bluetooth (A920 No/A925 Yes) USB Serial

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	
WAP	No
XHTML (MP)	Yes

Sony Ericsson P900

OS Version	Symbian OS v7.0 (+ security updates and MIDP2.0)
UI/Category	UIQ 2.1
Memory available to user	16 MB
Storage media	Yes; Sony MS Duo
Screen	208 × 320 (Flip Open); 208 × 208 (Flip Closed), 65536 colors TFT
Pointing device	Yes
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS
Java APIs	MIDP 2.0 PersonalJava 1.1.1 Bluetooth Wireless Messaging
Connectivity	Infrared Bluetooth USB (high speed serial connector with a USB->Serial adapter built into the desk stand)

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	WAP 2.0 XHTML (MP)

Nokia 6600

OS Version	Symbian OS v7.0s
UI/Category	Series 60
Memory available to user	6 MB NOR flash user data storage
Storage media	Yes; MMC
Screen	176 × 208; 65536 colors TFT
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 900/1800/1900
	HSCSD
	GPRS (2 + 2, 3 + 1, class B and C)
Java APIs	MIDP 2.0
	CLDC 1.0
	Nokia UI
	Mobile Media
	Wireless Messaging
	Bluetooth
Connectivity	Infrared
	Bluetooth

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and with the methods described in this book if SCOM is separately installed.

Browsing	
WAP	WAP 2.0
XHTML (MP)	Yes

Nokia 6620

OS Version	Symbian OS v7.0s
UI/Category	Series 60
Memory available to user	6 MB NOR flash user data storage
Storage media	Yes; MMC
Screen	176 × 220; 65536 colors TFT
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	GSM 850/1800/1900 EDGE
Java APIs	MIDP 2.0 CLDC 1.0 Nokia UI Mobile Media Wireless Messaging Bluetooth
Connectivity	Infrared Bluetooth USB

This Symbian OS smartphone uses a version of PC Connectivity based on OBEX and so cannot be accessed using the methods described in this book.

Browsing	
WAP	WAP 2.0
XHTML (MP)	Yes

Nokia 9500

OS Version	Symbian OS v7.0s
UI/Category	Series 80
Memory available to user	80 MB
Storage media	MMC card
Screen	Two displays, both 65536 colors *main screen*: 200 × 640 pixels *secondary screen*: 128 × 128 pixels
Pointing device	No
Camera	Yes; 640 × 480 resolution
Network Protocol(s)	EGSM 850/900 GSM 1800/1900 HSCSD GPRS EDGE
Java APIs	MIDP 2.0 Personal profile
Connectivity	Infrared Bluetooth USB

This Symbian OS smartphone uses a version of PC Connectivity based on OBEX and so cannot be accessed using the methods described in this book.

Browsing	HTML XHTML

Motorola A1000

OS Version	Symbian OS v7.0
UI/Category	UIQ 2.1
Memory available to user	24 MB
Storage media	Triflash-R
Screen	208 × 320; 65536 colors TFT
Pointing device	No
Camera	Yes; 1.2 megapixel, VGA camera
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS 3G EDGE
Java APIs	MIDP 2.0
Connectivity	Bluetooth USB

This Symbian OS smartphone ships with a PC suite based on SCOM and is fully compatible with the methods described in this book.

Browsing	
WAP	
XHTML (MP)	
Browser available	Yes

Panasonic X700

OS Version	Symbian OS v7.0s
UI/Category	Series 60
Memory available to user	–
Storage media	miniSD
Screen	176 × 280; 65536 colors TFT
Pointing device	No
Camera	Yes; VGA camera
Network Protocol(s)	GSM 900/1800/1900 GPRS (class 10 (4 + 1/3 + 2))
Java APIs	MIDP 2.0
Connectivity	Infrared Bluetooth USB

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and the methods described in this book if SCOM is separately installed.

Browsing	WAP 2.0 XHTML (MP)

Nokia 7610

OS Version	Symbian OS v7.0s
UI/Category	Series 60 v2.1
Memory available to user	8 MB
Storage media	MMC
Screen	176 × 208; 65536 colors TFT
Pointing device	No
Camera	Yes; 1152 × 864 resolution, 4× digital zoom
Network Protocol(s)	GSM 850/900/1800/1900 GPRS
Java APIs	CLDC 1.0 MIDP 2.0 Nokia UI Wireless Messaging Mobile Media Bluetooth API (no OBEX)
Connectivity	Bluetooth USB

This Symbian OS smartphone uses a version of PC Connectivity based on OBEX and so cannot be accessed using the methods described in this book.

Browsing	WAP 2.0 XHTML

Nokia N-Gage QD

OS Version	Symbian OS v6.1
UI/Category	Series 60 (v1)
Memory available to user	3.4 MB
Storage media	MMC
Screen	176 × 208; 4096 colors
Pointing device	No
Camera	No
Network Protocol(s)	GSM 850/900/1800/1900 HSCSD (CSD only in America) GPRS (2 + 2, 3 + 1, class B)
Java APIs	CLDC 1.0 MIDP 1.0 Nokia UI Wireless Messaging Mobile Media
Connectivity	Infrared Bluetooth USB

This Symbian OS smartphone ships with a PC suite that is not based on SCOM but is compatible with SCOM and the methods described in this book if SCOM is separately installed. However, the USB connection is not compatible with SCOM.

Browsing	WAP 1.2.1 XHTML (MP)

Index

'+' icon 66
_ASSERT_ALWAYS macro 106
_ASSERT_DEBUG macro 106, 155–61
.cs files 128
.def files 94–5, 129
.dll files 90–1, 162
.exe files 90–1, 162
#import device 79
.ini files 41
_LIT macro 108, 160–1
.mmp files 90–6, 128–9
.oby files 93–4
.pkg files 95–6, 162–3
.sis files 95–6, 129–30, 162–3
abld build command 97
abld freeze command 95
abld makefile command 92–3
abld.bat files 92–3
abstract classes
 see also base...
 concepts 107–8
AccessCount 238–81
Active 34
active objects
 see also event-driven programs
 anti-patterns 113
 cancellations 113
 concepts 110–13, 144, 160–1, 212–25, 399–401
 errors 113
 non-preemptive multitasking 110
 RunL 110–13, 399–401
Add 111–12, 219–25, 242–81, 350–8, 362–7, 371–84
AddAddresseeL 187–90
AddContactToGroupL 234–81
AddEntryL 287–346, 388–96
AddExceptionL 311–46
AddFieldL 238–81
AddFieldTypeL 244–81
AddNewContactL 234–81
AddNewGroupL 281
AddPhone 51–2, 81–6, 350–8
Address 190, 211–25
Address Book 364–7, 377–84
addresses
 cards 228–81
 messaging 170–90, 196–225
AddTodoListL 288–346
agclient.h 286
agenda model 1, 16, 283–346, 384–96
 alarms 284–6, 309–46, 387–96
 anniversaries 284–346, 390–6
 API 286–346, 384–96
 appointments 284–346, 386–96
 classes 284–325, 384–96
 concepts 283–346, 384–96
 connection processes 290–346
 connectivity service 325–46, 384–96
 creation procedures 290–346, 384–96
 databases 283–346
 deletions 341–6, 384–96
 edits 326–46, 384–96
 entries 284–6, 306–46
 events 284–346, 386–96
 filtering 283–4, 286, 336–46
 GUI applications 347, 384–96
 lists 283–4, 286, 320–46, 384–96

agenda model (*continued*)
  operations code 326–46, 384–96
  protocols 325–46, 388–96
  repeat APIs 298–306
  repeating entries 285, 298–306, 387–96
  retrieval code 337–46, 384–96
  sorting methods 295–346
  to-dos 284–346, 396
  types 283–4
agenda servers 286–346
`agmbasic.h` 309–46
`agmentry.h` 312, 314–17
`agmexcpt.h` 306
`agmids.h` 306–7
`agmlists.h` 320–1
`agmmodel.h` 287, 290, 294
`agmrptd.h` 298–9, 302–4
`agmtodos.h` 322–5
`Alarm` 309–46, 387–96
Alarm Server 284
`AlarmDaysWarning` 310–46
`AlarmInstanceDateTime` 309–46
alarms, agenda model 284–6, 309–46
`AlarmTime` 310–46
`Alloc` 107–8
`AllocL` 119–23, 127–9
anniversaries
  agenda model 284–346, 390–6
  sorting methods 296
APIs see Application Programming Interfaces
`Append` 149, 155–61, 201–4, 252–81
`AppendL` 252–81, 320–46
Application Programming Interfaces (APIs) 1–3, 13, 16–17, 109–10, 117–36, 167–90, 227–81
  agenda model 286–346, 384–96
  backwards compatibility 114–15
  client-server processes 109–10, 403–12
  concepts 1–3, 13, 16–17, 109–10, 118, 167–90, 227–81
  contacts 227, 231–81, 367–84
  custom servers 117–36, 397
  databases 227–346, 368–84, 396
  messaging 167–90
  Service Brokers 141–2, 398
  socket servers 140–66
applications
  BAL 33–7, 48–52, 134–6, 347–96
  class diagram 19–20
  SCOM 15, 16–38, 48–52, 58–66, 78–86
appointments
  agenda model 284–346, 386–96
  sorting methods 296
architectural overview, PC Connectivity applications 11–13
`AreAlarmedOnlyIncluded` 293–346
`AreAnnivsIncluded` 292–346
`AreCrossedOutIncluded` 293–346
`AreCrossedOutOnlyIncluded` 293–346
`AreEventsIncluded` 292–346
`AreNonRptsIncluded` 293–346
`AreRptsIncluded` 293–346
`AreTimedApptsIncluded` 292–346
`AreTodosIncluded` 293–346
`AreUnTimedApptsIncluded` 292–346
ARM targets 90–2, 97, 163
`armi` 93
`ArrayList` 49–52, 219–25, 349–58, 362–84, 385–96
arrays
  concepts 106–9, 133, 185–90, 219–25, 231–3, 336–46, 349–58, 362–96
  contacts 231–81
  errors 106–7
ASCII data 196, 201–4, 206–25, 328–46, 363–7
assert macros, concepts 106
asynchronous service 21, 25, 27, 32–3, 36–7, 53–7, 71–8, 133–6, 165, 169, 173–90, 353–8, 369–96
  concepts 21, 25, 27, 32–3, 36–7, 53–7, 133–6, 169, 173–5, 211–25, 353–8
  copy functions 53–7, 71–8, 87–8
  custom servers 133–6, 211–25
  disconnection operations 78, 87–8
  errors 78, 218–19
  event handlers 218–19
  messaging 169, 173–5, 180–90, 211–25
  MTMs 169, 174–5
  SMS 174–5, 211–25
  socket servers 165, 211–25
attachments, messaging 170, 172–5
`Attributes` 23, 27
audio 1–2, 16, 53, 117, 410
  see also multimedia

backup functions 2, 8, 15–16, 21, 29, 400, 409
backwards compatibility, Symbian OS 113–15
BAL see Bearer Abstraction Layer
`BALApplication` 33–4, 133–6, 222–5, 348–96

BALApplicationAsyncStream 36–7, 133–6, 222–5, 353–67, 369–96
BALConnectedPhone 348–96
BalForm 348–96
base classes, concepts 106–8, 119–23, 228–30
batteries 22–4
BatteryState 24
Bearer Abstraction Layer (BAL)
applications and devices 33–7
classes 33–7, 347–96
concepts 9, 16–18, 33–8, 133–6, 163–5, 347–96, 401, 409
custom servers 131–2, 133–6
efficiency issues 17
errors 18, 34, 37
event handling 34, 37, 349–96
services 34–7, 357–8
socket servers 163–5, 401
streams 35–7, 347–58, 401
type library 48–52
bearers, concepts 11–12
BenQ 429
Big Endian 145
BIO Messaging 168
bitmaps 98
bld.inf files 90–6
Bluetooth 8–9, 11–12, 40, 42, 46–7, 60, 97, 136, 168
concepts 8–9, 11–12, 40, 42, 46–7, 60, 97
errors 47
time considerations 60
Body 187–90, 197–225
BodyLength 147, 152–61, 197–225
book list 419
BSTR variables 38
bt.esk 42, 47
buffer descriptors
see also descriptors; TBuf...
concepts 106–8
buffers 8, 32–3, 36–7, 106–8, 123, 127–36, 145–8, 193–225, 274–81, 336–46, 357–8, 389–96, 404
build processes, concepts 90–6
buttons, disabling operations 67–8, 354–8

C# 3, 5, 18, 39, 48–52, 72, 78, 80, 86, 133, 191
concepts 3, 5, 18, 39, 48–52, 72, 78, 80, 86, 133, 191
SCOM access 48–52, 78, 80, 86, 133
C++
concepts 3, 5, 18, 37–8, 39, 78–88, 129–30
custom servers 129–30
destructors 101–2, 105–6
SCOM 37–8, 39, 78–88
Symbian OS 3, 5, 37–8, 39, 78–88
c: drive, conventions 54–5, 58, 94, 161–2
C classes, concepts 101, 105–6
caching operations
messaging 171–90
SCOM 60, 81–2
CActive 110–13, 212–25, 398–401
CActiveScheduler 111–13, 160–1, 398–401
cagmcallb.h 289
CAgnAnniv 284–346
CAgnAppt 284–346
CAgnBasicEntry 284–346
CAgnDayDateTimeInstanceList 321–46
CAgnDayList 321–46
CAgnEntry 283–346
CAgnEntryModel 283–346
CAgnEvent 284–346
CAgnExceptionList 311–46
CAgnIndexedModel 283–346
CAgnList 320–46
CAgnModel 283–346
CAgnMonthInstanceList 321–46
CAgnRptDef 298–346
CAgnTodo 284–346
CAgnTodoInstanceList 324–46
CAgnTodoList 323–46
CAgnTodoListList 288–346
calendar data 16, 384–96
see also agenda model
cameras 7, 114, 410
Cancel 110–13
Cancel buttons 264–7
Capacity 24, 59–60
cards
contacts models 228–81, 367–84
editing code 276–8, 367–84
retrieval code 271–81, 371–84
templates 228–30, 235–81, 367, 370–84
CArrayFix 232–81
CArrayPtrFlat 189–90
CastToAnniv 312–46
CastToAppt 312–46
CastToEvent 312–46
CastToTodo 312–46

CBase class
  cleanup stack 105–6
  concepts 101–2, 105–6, 110–11, 147–8, 152–61, 175, 205, 241–5, 303–4, 320
  messaging 172–90
  virtual destructors 105–6
  zero initialization 101
CBaseMtm 168–9, 172–90, 210–25
CBaseMtmUi 168–9
CBaseMtmUiData 168–9
CBaseServerMtm 168–9
CCharFormatLayer 205–25, 289–346
CClientMtmRegistry 185–90, 205–25
CClientSocket 139–65
CCmdTarget 84–6
CCommand 140, 150–1, 154–61
CConnPack 201–25, 271–81, 337–46, 360–7, 370–84, 386–96, 400–1
CContactCard 228–81
CContactCardTemplate 228–81
CContactDatabase 231–81
CContactDataField 228–81
CContactDateField 244–81
CContactFieldStorage 228–81
CContactGroup 228–81
CContactIdArray 232–81
CContactItem 228–81
CContactItemField 228–81
CContactItemFieldDef 247–81
CContactItemPlusGroup 228–81
CContactItemViewDef 249–81
CContactLocalView 255–81
CContactOwnCard 239–81
CContactStoreField 228–81
CContactTextField 228–81
CContactViewBase 254–81
CContentType 228–81
CCustomServer 119–27
CCustomServerSession 119–27
CD 5–6
CDesCArray 187–90
CEchoCSServer 125–9
ChangeL 175–90
ChangeTodoListOrderL 289–346
ChangeTodoOrderL 289–346
CharFormatLayer 289–346
ChildDataL 173–90
ChildDirectories 25, 54–7, 58–60, 86–7
ChildEntryL 173–90
ChildFiles 25–6, 58–60, 86–7
Children 173–90, 207–25
ChildrenL 179–90
ChildrenWithMtmL 173–90
ChildrenWithTypeL 173–90, 207–25
classes
  see also C...; M...; R...; T...; individual classes
  agenda model 284–325, 384–96
  BAL 33–7, 134–6, 347–96
  concepts 18–19, 33–9, 58–60, 61–6, 100–2, 105–8, 119–23
  contacts 228–56, 367–84
  custom servers 119–29
  descriptors 106–8
  extended classes 18–19
  logging classes 410–11
  messaging 167–225, 359–67
  naming conventions 100–2
  SCOM 18–33, 39, 58–60, 61–6, 79–88
  SMS 185–90, 359–67
  socket servers 138–51
  types 100–2, 105–8
cleanup stack 101–2, 103–6, 125–9, 159–61, 206–25, 272–81, 338–46, 398–401
CleanupClosePushL 105–6
CleanupStack 104–6, 125–9, 159–61, 206–25, 272–81, 338–46, 398–401
CleanupStack::PushL 105–6, 159, 209–25, 280–1, 338–46, 398–401
ClearAll 299–346
ClearAllDays 301–46
ClearAllSessions 119–23
ClearApptList 386–96
ClearContactList 371–84
ClearDays 300–46
ClearPendingReads 358, 361–7, 373–84
ClearRepeat 310–46
ClearWeek 301–46
client-server processes, PC Connectivity applications 12–13, 109–10, 403–12
client-side MTMs, concepts 168–90
clients, concepts 12–13, 109–10, 139–40, 151–61
Close 102, 105–6, 109, 252–81, 286–346
CloseContactL 233–6
CMessage 139–65
CMsgWait 212–25
CMsvEntry 170–90, 209–25
CMsvEntrySelection 179–90, 207–25

CMsvOperationWait 205–25
CMsvSession 174–90, 205–25
CMsvStore 178–90
cntdb.h 231–6, 249–50
cntdef.h 229–30, 241, 247–8
cntfield.h 242–7
cntfldst.h 249
cntitem.h 236–40
cntviewbase.h 251–6
code
  see also programming
  conventions 5
color schemes 7–8
COM servers 15–38, 78, 84–6
  see also Bearer Abstraction Layer; Symbian Connect Object Module
command classes, socket servers 140–3, 149–61
command-line applications 219–25, 281, 396, 411–12
  contacts 281
  SMS 219–25
command–response sequence, SMS 192–225
CommDB 44
CommitContactL 234–81
common directories, concepts 93–4
comms
  client-server processes 12–13, 109–10, 403–12
  custom servers 132–6
  socket servers 164–5, 398–401
community links 417–18
Complete 176–90
Completed 290–346
ComposeForm 365–7
compression considerations, PC Connectivity applications 405
concrete descriptor objects
  see also Hbuf...; TBuf...; TPtr...
  concepts 107–8, 146, 148–9
configuration files, serial ports 42
conn namespace 141–51, 152–61
Connect 141–2, 149–50, 165–6, 286–346
ConnectedDevices 19–20, 33–4, 48–9, 58–60, 82–6, 349–58
ConnectedPhone 49–52, 60–6
connection processes
  agenda model 290–346
  concepts 8–9, 11–12, 39–52, 60–6, 78–86, 123, 165–6, 247–58, 290–346, 406–10
  interruptions 406–7
  power interruptions 406–7
  Visual C++ 78–86
  visual feedback 41
ConnectionBearer 20, 34
connectivity socket servers see socket servers
ConnPack 201–25, 271–81, 337–46, 360–7, 370–84, 386–96
console programs, emulators 98
ConstructL 104–6, 121–3, 127–9, 144–5, 148, 159–61, 205–25, 271–81, 336–46, 398–401
constructors, two-phase construction 104–6, 143–5, 151, 205–25
ContactDetailsList 378–84
ContactEditForm 378–84
ContactList 375–84
contacts 1, 10, 16, 121–2, 227–81, 359, 367–84
  see also cards; groups
  API 227, 231–81, 367–84
  classes 228–56, 367–84
  command-line applications 281
  concepts 1, 10, 16, 121–2, 227–81, 359, 367–84
  connectivity service 256–81
  creation procedures 231–81, 326–46, 367–84
  databases 227–81, 367–84
  deletions 231–81, 367–84
  edits 231–81, 367–84
  event handlers 253–81
  GUI applications 281, 347, 359, 367–84
  label methods 240–81, 374–84
  mapping methods 230, 246–81, 368–84
  models 227–81, 367–84
  navigation techniques 232–6, 367–84
  observers 230–1, 254–81
  operations code 256–81, 367–84
  own cards 235–6, 276–81
  protocols 256–81, 370–84
  sorting methods 232–6
  system templates 275–81
  templates 228, 235–81, 367, 370–84
  views 230–1, 249–81, 368–84
container classes, concepts 108–9
Contains 251–81
ContainsItem 241–81
content return, SMS messages 206–9
conventions 5
cooperative multitasking, concepts 110–13
copy functions 16, 20, 26, 27, 28–30, 53–7, 71–7, 87–8, 211–25
  asynchronous actions 53–7, 71–8, 87–8

copy functions (*continued*)
  concepts 16, 20, 26, 27, 28–30, 53–7, 71–7, 87–8
  event handlers 55–7, 71–7, 87–8
  PC to phone 54–7, 71–7, 87–8
  phone to PC 71–7, 87–8
  root directories 54–7
  synchronous actions 55, 78, 87–8
  time considerations 55, 72–7
  Visual C++ 87–8
CopyFileFromPC 26–7, 55–7, 72–7, 88
CopyL 252–81
CopyToPC 27, 73–7, 88
corporate systems 396
Count 173–90, 241–81, 320–46
CountL 234–81
CParaFormatLayer 205–25, 289–346
CRAgnCSServer 104
CRAgnUtil 336–46
crashes 406–7
CRCntUtil 271–81
Create 102, 401
CreateApptButton_Click 394–6
CreateApptL 342–6
CreateAttachmentL 172–90
CreateButton_Click 382–4
CreateContactCardTemplate... 235–81
CreateContactGroup... 234–81
CreateDirectory 70–1
CreateEventButton_Click 394–6
CreateInstance 79–80
CreateL 173–90, 232–81, 401
CreateMessageL 188–90, 210–25
CreateOwnCard... 235–81
CreateTodoListEntryL 345–6
creation procedures
  agenda model 290–346, 384–96
  contacts 231–81, 367–84
  custom servers 124–9
  databases 231–81
  directories 70–1
  SMS messages 209–25, 359–67
  socket servers 138, 151–61, 397–401
CRichText 108, 170–90, 205–25
CrossOut 319–46
CRSmsCSServer 204–25
CRSmsMsg 204–25
CServerSocket 139–65
CSmsClientMtm 171–90, 205–25
CSmsHeader 174–90, 210–25
CSmsMessageSettings 189–90
CSmsNumber 174–90
CSmsSettings 189–90
CStartCSSession 400–1
CTestSocketServer 398–401
Ctrl-Q key 41
cursors, hourglass images 70–2, 351
custom servers
  see also plug-ins
  asynchronous service 133–6, 211–25
  classes 119–29
  comms 132–6
  concepts 8–9, 13, 95–6, 117–36, 191–225, 351, 397, 410
  contacts connectivity 256–81
  creation procedures 124–9
  debugging 136
  definition 117–18
  development 117–36
  echo custom server 95–6, 118, 124–36
  errors 119–23, 126–9, 130–2, 136
  installation 129–30
  limitations 118
  log files 136
  overview 117–18
  packets 123–4, 132–3
  protocols 123–4, 191–225, 256–81, 325–46
  SCOM 130–2
  SMS management 191–225
  socket servers 137
  starting procedures 130–2
  synchronous service 125–36, 211–25
  unloading considerations 132
customserver.h 125
customserver.ini 125
customservershared.h 125
CViewContact 256–81

Daily 304–46
Data 146, 149
data, metadata 106–7
data directory 93–4
databases
  agenda model 283–346
  concepts 227–30, 283–346, 367–84, 396
  contacts 227–81, 367–84
  creation procedures 231–81
  open code 271–81

rollbacks/recoveries 231
views 230–1, 249–81, 368–84
DataBuf 148–9
DataPtr 148–9
dates 21, 27, 30, 59–60, 72–7, 176–90, 220–5, 244–81, 285–346, 386–96
DateTime... 27, 30, 59–60, 72–7, 244–81, 387–96
day notes, sorting method 296
debugging 3, 92–4, 97–8, 104–5, 396–7, 405, 410–12
concepts 3, 92–4, 97–8, 104–5, 136, 396–7, 405, 410–12
custom servers 136
design issues 405, 410–12
messaging 171, 191–2, 219–25
socket servers 138, 155–61, 165–6, 397
DefaultSC 189–90
defensive designs, PC Connectivity applications 406–7
DEFFILE 90–1
delays
GUI applications 351–8
hourglass images 70–2, 351
Delete 26–7, 67–9, 88, 180–90, 209–25, 320–46, 363–7, 395–6
DeleteButton_Click 395–6
DeleteCardL 276–81
DeleteContactL 231–81
DeleteContactsL 231–81
DeleteEntryL 287–346
DeleteGroupL 280–1
DeleteInstanceL 295–346
DeleteL 180–90, 209–25
DeleteSmsL 209–25, 363–7
DeleteTodoListL 288–346
deletions 26–7, 67–70, 88, 173–90, 192
agenda model 341–6, 384–96
concepts 26–7, 67–70, 88
contacts 231–81, 367–84
SMS 192–3, 198–225, 359–67
time considerations 70
descriptors
see also buffer...; heap...; pointer...; strings
classes 106–8
concepts 106–8, 124
literals 108
modifiable/non-modifiable contrasts 108
DeSerialiseL 145–6
design issues, PC Connectivity applications 403–12
'destination' addresses, messaging 170, 174
destructors, C++ 101–2, 105–6
development resources 5–6, 413–20
device connection 19–20, 27–32, 33–7, 38, 48–53, 60–6, 78–86, 87–8, 123, 165–6, 347–58, 407–10
BAL 33–7, 48–52, 134–6, 347–96, 409
difference handling 52–3, 80–6, 113–15, 407–10
SCOM 19–20, 27–32, 38, 48–52, 60–6, 78–86, 87–8, 409
Visual C++ 78–86, 87–8
device and service management 52–3, 113–15, 180–6, 407–10
DeviceConnected 19–20, 28, 38, 84–8
DeviceCopyStorageFileComplete 20, 30, 57, 87–8
DeviceCopyStorageFileError 20, 29, 57, 87–8
DeviceCopyStorageFileExistingFileFound 20, 30, 57, 88
DeviceCopyStorageFileProgress 20, 28–9, 57, 87–8
DeviceDisconnected 19–20, 28, 84–6, 87–8
DeviceFormatStorageDriveComplete 31, 87–8
DeviceFormatStorageDriveError 31, 87–8
DeviceFormatStorageDriveProgress 31, 87–8
DeviceListChanged 33–4
difference handling, devices 52–3, 80–6, 113–15, 407–10
directories
see also filing systems
common directories 93–4
concepts 19, 21, 22–7, 53–7, 58–60, 63–77, 90–6, 161–3, 171–2
creation operations 70–1
deletions 26–7, 67–9, 88
message stores 171–2, 178–90, 213–25
naming conventions 69
navigation techniques 53–4, 58–60, 63–6, 86–7
renaming operations 67–8, 69–70, 88
simple actions 66–77
tree views 63–76, 87
disabling operations, buttons 67–8, 354–8
disconnection events 19–20, 27–8, 60–6, 73–8, 80–6, 87–8, 141–2, 348–58, 367
dispatch maps, Visual C++ 84–8

display properties, files 71
DisplayDueDateAs 317–46
DisplayNextOnly 306–46
DisplayTime 313–46
DLLs 90–6, 118–36
DoActivate 113
DoCancel 111–13, 119–23, 212–25, 398–401
documentation needs, protocols 191–2, 403–4
DoEditsL 276–81
dongles 46–7
DoServiceL 103, 215–25
DoStartL 120–3
Drafts folder 172–3
drives, concepts 19, 21, 22–7, 31, 54–7, 58–60, 86–7, 94
DueDate 318–46
Duration 319–46

E32Dll 119–23, 161
E32main 152
e32std.h, trap harness 101
EAgnFilter 291–346
EAllInstances 291–346
EBig 145
echo custom server plug-in 95–6, 118, 124–36
echo socket server 151–61
echocs.cs 128
echocs.mmp file 90–6, 128–9
echoss.pkg 162–4
ectcpadapter 9, 35, 90–1, 117–36, 212–25, 351
ECurrentAndFutureInstances 291–346
ECurrentAndPastInstances 291–346
ECurrentInstance 291–346
EDayFilter 291–346
EditButton_Click 381–4, 392–6
EditCardInGroupL 280–1
EditCardL 277–81
EditGroupL 280–1
edits
  agenda model 326–46, 384–96
  contacts 231–81, 367–84
EditStoreL 171–90
EditTodoListEntryL 345–6
EFBForm 61–3, 74–7
EFindFilter 292–346
EKA2 151–2, 161, 401
ELittle 145
emails 108, 168, 171, 174, 396
  see also messaging
EMsvCloseSession 181–90, 204–25
EMsvCorruptedIndexRebuilding 182–90
EMsvCorruptedIndexRebuilt 181–90
EMsvEntriesChanged 181–90, 199–225
EMsvEntriesCreated 181–90, 199–225
EMsvEntriesDeleted 181–90, 199–225
EMsvEntriesMoved 181–90, 199–225
EMsvGeneralError 181–90, 204–25
EMsvMediaAvailable 181–90
EMsvMediaChanged 181–90
EMsvMediaIncorrect 182–90
EMsvMediaUnavailable 181–90
EMsvMtmGroupDeInstalled 181–90
EMsvMtmGroupInstalled 181–90
EMsvServerFailedToStart 181–90, 204–25
EMsvServerReady 181–90
EMsvServerTerminated 181–90, 204–25
EMsvSort... 183–90
emulators
  concepts 39–52, 90, 92, 94–100, 136, 410–12
  connection techniques 39–52
  console programs 98
  definition 90, 96–7
  driving procedures 98–100
  emulator bearer 48
  screens 98–100
  socket servers 161–2, 165–6
  starting procedures 97–8
  usage details 96–100, 410
encoding data, headers 170
EndDate 299–346
EndDateTime 315–46
Endian 145–9
Entry 178–90
EntryCount 291–346
EntryId 178–90
enumerated object types, concepts 101
EOpenCallBack... 290–346
EPOC 7, 97–8, 151–2, 398–401
epoc32 45, 90–1, 93–4, 97–8, 125–6, 163
epoc.exe 97–8
epoc.ini 98
ER5 7–8, 151–2
ERAgnCmdCreateAppt 326–46, 394–6
ERAgnCmdCreateApptReply 326–46
ERAgnCmdCreateTodoListEntry 326–46
ERAgnCmdCreateTodoListEntryReply 326–46
ERAgnCmdDeleteInstance 326–46

ERAgnCmdDeleteReply 326–46
ERAgnCmdEditAppt 326–46
ERAgnCmdEditApptReply 326–46
ERAgnCmdEditTodoListEntry 326–46
ERAgnCmdEditTodoListEntryReply 326–46
ERAgnCmdError 326–46
ERAgnCmdFetchApptsByDay 326–46, 387–96
ERAgnCmdFetchApptsReply 326–46, 388–96
ERAgnCmdFetchMoreAppts 326–46
ERAgnCmdFetchMoreTodoListEntries 326–46
ERAgnCmdFetchOneTodoList 326–46
ERAgnCmdFetchTodoListEntriesReply 326–46
ERAgnCmdFetchTodoLists 326–46
ERAgnCmdFetchTodoListsReply 326–46
ERAgnCmdNone 326–46
ERAgnCmdOpenAgenda 326–46
ERAgnCmdOpenAgendaReply 326–46
ERAgnCmdQueryVersion 326–46
ERAgnCmdVersionReply 326–46
ERCntCmdContactsCountReply 257–81
ERCntCmdContactsInGroupReply 257–81
ERCntCmdContactsReply 257–81, 372–84
ERCntCmdCreateContact 257–81, 383–4
ERCntCmdCreateContactReply 257–81
ERCntCmdCreateGroup 257–81
ERCntCmdCreateGroupReply 258–81
ERCntCmdDeleteContact 257–81
ERCntCmdDeleteGroup 257–81
ERCntCmdDeleteReply 257–81
ERCntCmdEditContact 258–81, 381–4
ERCntCmdEditContactInGroup 258–81
ERCntCmdEditContactInGroupReply 258–81
ERCntCmdEditContactReply 258–81
ERCntCmdEditGroup 258–81
ERCntCmdEditGroupReply 258–81
ERCntCmdError 258–81
ERCntCmdFetchAllContacts 257–81, 372–84
ERCntCmdFetchAllGroups 257–81
ERCntCmdFetchContactsCount 257–81
ERCntCmdFetchContactSet 257–81, 380–4
ERCntCmdFetchContactsInGroup 257–81
ERCntCmdFetchMoreContacts 257–81, 372–84
ERCntCmdFetchMoreGroups 257–81
ERCntCmdFetchOwnCardId 257–81
ERCntCmdFetchTemplateFieldInfo 257–81
ERCntCmdFindContacts 257–81
ERCntCmdGroupsReply 257–81
ERCntCmdNone 257–81
ERCntCmdOpenDatabase 257–81
ERCntCmdOpenDatabaseReply 257–81
ERCntCmdOwnCardIdReply 257–81
ERCntCmdQueryVersion 257–81
ERCntCmdSetOwnCardId 257–81
ERCntCmdSetOwnCardIdReply 257–81
ERCntCmdTemplateFieldInfoReply 257–81
ERCntCmdVersionReply 257–81
error handling, concepts 102–6
errors
  see also debugging; testing facilities
  arrays 106–7
  BAL 18, 34, 37
  Bluetooth 47
  concepts 18, 20, 29–32, 55–7, 68–78, 81–8, 101, 102–6, 110–13
  copy functions 55–7, 71–7, 87–8
  custom servers 119–23, 126–9, 130–2, 136
  file browsers 78
  infrared connections 44
  messaging 178–90, 193–225
  out-of-memory errors 102–6, 118, 406
  panics 106, 155–61, 188
  RunL 110–13
  SCOM 18, 20, 29–32, 55–7, 68–78, 81–8
  SMS 193–225
  socket servers 141–2, 150, 154–61, 164–6
  types 77–8, 102–3
ERSmsCmdDeleteSms 192–225
ERSmsCmdError 193–225
ERSmsCmdFetch 193–225
ERSmsCmdGetAllSms 192–225
ERSmsCmdGetMoreSms 192–225
ERSmsCmdGetSmsById 192–225
ERSmsCmdMsvEvent 199–225
ERSmsCmdNone 194
ERSmsCmdQueryVersion 194, 215–25
ERSmsCmdReceivedNoMoreSms 223–5, 361–7
ERSmsCmdReceiveSms 361–7
ERSmsCmdReturnEvents 193–225
ERSmsCmdSendSms 192–225
ERSmsCmdVersionReply 193–225
ESOCK 17
ESymbolFilter 292–346
etel 44
ETidyFilter 292–346
event handlers

event handlers (*continued*)
- asynchronous service 218–19
- BAL 34, 37, 134–6, 349–96
- contacts 253–81
- copy functions 55–7, 87–8
- SCOM 19–20, 25, 27–33, 52, 55–7, 61–6, 85–8
- SMS management 193–225

event-driven programs
- see also active objects
- concepts 110–13

EventOnDate 301–46
EventPriority 314–46
events
- agenda model 284–346, 386–96
- messaging 173–4, 193–225

ExecuteLD 112, 140–1, 150, 154–61, 401
exepath 162
EXPORT 91–2, 126–7
exports 91–6
extended classes, concepts 18–19
ExtractDataL 149, 156–61
ExtractL 149
ExtractLC 149
ExtractRawDataL 149
ExtractWithRawLenPrefixLC 149

factory classes, socket servers 139–43, 151–61
FailedToStart 142
faxes 44–5, 168, 170
- see also messaging

FetchApptsByDayL 336–46
FetchEntryL 288–346
FetchInstanceL 294–346
FetchInstanceLC 294–346
FetchMoreApptsL 338–46
FetchMoreTodoListEntriesL 340–6
FetchOneTodoListL 340–6
FetchTodoListL 288–346
Field... 246–81, 367–84
fields, contacts 228–81, 367–84
file browsers 1–2, 39–88, 129–30, 347–8
- custom servers 129–30
- errors 78
- examples 39, 60–88
- graphical programs 60–88, 347–8

file management, concepts 16, 39
file servers 12
FileCopyComplete 72–7
FileCopyError 72–7
FileCopyExistingFileFound 72–7
FileCopyProgress 72–7
FileName 26–7, 58–60
files 1–2, 8, 10, 16, 19, 20–7, 28–30, 53–7, 58–60, 63–77
- copy functions 16, 20, 26, 27, 28–30, 53–7, 71–7, 87–8
- deletions 26–7, 67–9, 88
- display properties 71
- header files 91–2, 355–8
- navigation techniques 53–4, 58–60, 63–6, 86–7
- renaming operations 67–8, 69–70, 88
- simple actions 66–77

filing systems
- see also databases; directories
- concepts 1–2, 8, 10, 21–7, 58–60
- copy functions 16, 20, 26, 27, 28–30, 53–7, 71–7, 87–8
- navigation techniques 53–4, 58–60, 63–6, 86–7

filtering, agenda model 283–4, 286, 336–46
Find 185–90, 242–81, 320–46, 374–84
FindException 311–46
FindLC 232–81
FindNextInstanceL 297–346
FindPreviousInstanceL 297–346
FindRptEndDate 299–346
FirstDayOfWeek 300–46
FM radios 7
folders, messaging 172–3, 359–67
form class, GUI applications 351–96
Format 25
format events 25, 27, 30–1
forwarded messages 188, 359
FreeSpace 24, 59–60
freezing concepts 94–5
'from' addresses, messaging 170, 174, 189–90, 196–225
FromAddress 190, 196–225

gcc directory 93
gencserv 118
general socket servers
- see also TCP/IP
- concepts 397–401

GetAllCards 274–81
GetAllGroupsL 278–81
GetAllSmsL 216–25
GetCardSet 274–81
GetCardsInGroupL 279–81

GetCardTemplateIdListL 235–81
GetEntryL 173–90, 207–25
GetFieldInfo 273–81
GetGroupIdListL 234–81
GetGroupLabelL 241–81
GetMoreCards 274–81
GetMoreCardsL 274–81
GetMoreGroupsL 279–81
GetMoreSmsL 217–25
GetMsvEventL 213–25
GetOneApptL 338–46
GetOneCardL 272–81
GetOneSmsL 207–25
GetOneTodoListEntry 340–6
GetSmsByIdL 216–25
GetStartDay 303–46
GetTemplateInfoL 275–81
GetTemplateLabelL 240–81
GetValue 348–58
good practices, programming 412
graphical programs, file browsers 60–88, 347–88
GroupByMtm 184–90
GroupByPriority 184–90
GroupByType 184–90
GroupCount 234–81
GroupingOn 184–90
groups of cards
  contacts 228–9, 234–81, 367
  editing code 280–1
  retrieval code 278–81
GroupStandardFolders 184–90
GUI applications 53–4, 59, 60–88, 98–100, 168–9, 191, 219, 225, 347–96, 400, 409
  agenda model 347, 384–96
  C# uses 39, 191
  communications 351–8
  concepts 39, 191, 281, 347–96, 400, 409
  contacts 281, 347, 359
  definition 347
  delays 351–8
  development issues 347–96
  file browsers 60–88, 347–8
  form class 351–96
  further developments 396
  SMS 191, 219, 225, 347, 358–67

HAL APIs 421
HandleContactViewEvent 254–81
HandleSessionEventL 182–90, 204–25
hardware 7–8
HasAlarm 309–46
HasBaseYear 317–46
HasContext 187–90
HasExceptions 311–46
HasItemLabelField 241–81
HasStoreL 178–90
HBufC concrete class 107–8, 148–9, 207, 245–9, 337, 343
Header 147–8, 152–61
header files 91–2, 125–9, 355–8
headers, messaging 170–2, 174–90
heap 103–8
heap descriptors
  see also descriptors; HBufC...
  concepts 108
hourglass images, cursors 70–2, 351
HRESULT 18, 30, 33, 37, 80, 87

IANA 138, 397–8
Id 20, 34, 49–52, 57, 80–1, 153, 237–81, 348–58
IDE 90–3, 165–6, 410
IErrorInfo 18
images 1–2, 16, 39, 52–3
  see also bitmaps; multimedia; video
  device difference handling 52–3
  management applications 39, 52–3
IMAP 168
IMEI numbers 20, 49–50, 80, 81, 409
Inbox 172–3, 221–5, 359–67
include directory 90–4, 125–6
IncludeCrossedOut 293–346
indexed agenda models 283–346
indices
  agenda models 283–346
  messaging 172–3, 178–90, 358–67
infrared connections 8, 11–12, 40–1, 42, 43–5, 60, 97, 136, 168
  concepts 8, 11–12, 40–1, 42, 43–5, 60, 97
  errors 44
  time considerations 60
InitializeComponent 353–8, 366–7, 378–84, 390–6
InitializePhones 82–6, 356–8
InitializePhoneService 368–84, 385–96
inner schedulers, concepts 212
InPreparation 176–90
InsertFieldL 238–81
InsertL 242–81, 320–46

installation 95–6, 129–30, 161–3, 397–401, 412
  custom servers 129–30
  installer (.sis)files 95–6, 129–30, 162–3
  socket servers 161–3, 397–401
installer (.sis)files 95–6, 129–30, 162–3
instance agenda models 283–346, 387–96
InstanceCount 299–346
InstanceDate 312–46
InstanceEndDate 313–46
InstanceId 313–46
InstanceStartDate 313–46
Integrated Development Environments 3
Interval 299–346
Intuwave 8
InvokeAsync 186–90
IPAddress 35
irda.esk 42, 45
IsActive 21
IsADayNote 310–46
IsAlarmSetFromDueDate 319–46
IsAlarmSetFromStartDate 319–46
ISCApplication 79–86
ISCBALDevice 34–5, 163–5, 348–58
ISCBALDeviceCollection 33–5, 349–58
ISCBALDeviceService 35–6, 131–2, 163–5
ISCBALDeviceServiceCollection 34, 35, 163–5
ISCBALSequentialStream 35–6, 131–6, 163–5, 221–5, 356–8, 368–96
ISCBALSequentialStreamSink 37, 135–6, 356–8, 368–96
ISCDevice2 , concepts 18–19, 20–1, 32–3, 38, 49–53, 58–60, 80–6, 347–58
ISCDeviceCollection 19–20, 21
ISCDeviceStorage 19, 21, 22, 26, 58–60, 86–7
ISCDeviceStorageDirectory 19, 21, 24, 25–6, 58–60, 65–6, 68–76, 86–7, 88
ISCDeviceStorageDirectoryCollection 25, 26, 59–60, 86–7
ISCDeviceStorageDrive2 19, 23–5, 27, 54–7, 58–60, 86–7
ISCDeviceStorageDriveCollection 21, 25, 54–5, 86–7
ISCDeviceStorageFile 19, 21, 26–7, 66, 68–76, 86–7, 88
ISCDeviceStorageFileCollection 25–6, 27, 86–7
ISCEvents 19–20, 28–31, 52, 84–6
IsCrossedOut 310–46
ISCSequentialStream 21, 32–3
IsDated 319–46
IsDaySet 300–46
IsDeletable 237–81
IsDeleted 237–81
IsFull 247–81
IsHidden 237–81
IsNullId 307–46
IsNullInstance 308–46
IsOperationCanceled 74
IsReadOnly 243–81
IsReceiveComplete 146
IsRepeating 310–46
IsSendCompleteL 146–7
IsTentative 310–46
IsValid 292–346
IsValidLabel 245–81

Java 3, 89, 102
jog dials 7, 99
joysticks 7, 99
jukebox applications 39, 52
'Just in Time' basis, tree views 65–6

kernel-side programming 109, 151–2
KErrAccessDenied 178–90
KErrArgument 158, 200–4, 210, 342–6
KErrGeneral 204–5, 217–25
KErrNone 101, 102–3, 113, 126–7, 141–2, 161, 183–90, 217–25, 286, 290, 355–8, 368, 401
KErrNotFound 131, 185–90, 250–81
KErrNotSupported 120–3, 126–9, 131, 156–61, 404
keyboards 7
KMsvDraftEntryId 173, 175–90
KMsvGlobalInBoxIndexEntryId 173, 175–90, 360–7
KMsvGlobalOutBoxIndexEntryId 173, 175–90
KMsvGroup... 183–90
KMsvMediaIncorrect 182–90
KMsvNoGrouping 183–90
KMsvNullIndexEntryId 172–3, 175–90
KMsvSentEntryId 173, 175–90, 360–7
KStorageTypeText 262–81, 374–84
KUidContact... 232, 237–81
KUidMsgTypeSMS 175–90
KUidMsvMessageEntry 175–90
KUidMsvNullEntry 176–90

Label 244–81
label methods, contacts 240–81, 374–84
LabelUnspecified 244–81
laptops 45
LastModified 27, 237–81
leak problems, memory 103–4
least significant byte (LSB) 145–6
Leaves 102–3
Length 149, 152–61, 164–5, 219–25
length information 133–6, 147–9, 152–61, 164–5, 174–90, 196–225, 278–81, 327–46
libraries 90–6, 128–30, 138, 400–1
LIBRARY 90–1
Linux 4
list boxes 377–84
listeners 44
lists
  agenda model 283–4, 286, 320–46, 384–96
  SMS messages 206–9, 358–67
literals
  see also _LIT macro
  concepts 108
Little Endian 145
loading considerations 132, 409–10
LoadMessageL 171–90, 206–25
localization considerations, PC Connectivity applications 405
log files 136, 410–12
Logon 401
Lotus Notes 2
LPCTSTR variables 38
LSB see least significant byte

M classes, concepts 102
m-Router
  see also TCP/IP
  concepts 8–9, 12, 17–18, 40–3, 45–8, 117
  versions 40, 47
Mac OS 4
MAgnModelStateCallBack 287–346
MAgnProgressCallBack 289–346
Mail directory 171
MakeInstanceNonRepeating 313–46
makekeys 96
MakeUndated 319–46
makmake 90
Managed C++ 78
management connectivity 16
Manufacturer 20, 34, 49–52, 80–2, 347–58, 385
Mapping 246–81, 374–84
mapping methods, contacts 230, 246–81, 368–84
MatchesAll 250–81
MaxBodyLength 147, 152–61, 210–25
MCommand 140–3, 149–51, 154–61
MContactDbObserver 255–81
MContactViewObserver 254–81
MediaAttributes 24
MediaType 24
memory
  leak problems 103–4
  out-of-memory errors 102–6, 118, 406
Memory Sticks 94
message folders 172–3, 359–67
message servers
  concepts 167–90, 204–25
  events 173–4, 213–25
  MTMs 167–72, 204–25
  SMS 169, 204–25
message stores 171–2, 178–90, 213–25
Message Type Modules (MTMs)
  asynchronous service 169, 174–5, 211–25
  client-side MTMs 168–90
  components 168–9
  concepts 117, 167–90, 204–25
  generic classes 185–7
  message servers 167–72, 204–25
  SMS 169, 204–25
  synchronous service 169, 211–25
messaging 7–8, 117–66, 167–90, 191–225, 256–81, 358–67
  see also Short Messaging System
  asynchronous service 169, 173–4, 180–90, 211–25
  attachments 170, 172–4
  bodies 170–2, 197–225
  caching operations 171–90
  classes 167–225, 359–67
  common classes 175–87
  components 168–9
  concepts 167–90, 191–225
  contacts connectivity 256–81, 359
  debugging 171, 191–2, 219–25
  errors 178–90, 193–225
  events 173–4, 193–225
  folders 172–3, 359–67
  generic classes 175–85
  headers 170–2, 174–90
  indices 172–3, 178–90, 358–67

messaging (*continued*)
navigation techniques 172–3
'pull/push' methods 174–5
sessions 173–4
stores 171–2, 178–90, 213–25
structures 170–3
synchronous service 169, 180–90, 211–25
Messaging Application 173–4
metadata, concepts 106–7
Metrowerks CodeWarrior 45, 90–2, 97, 165–6, 415
`MFactory` 139–43, 157–61
MFC 79, 84–6
`MHeader` 147–8, 152–61
Microsoft 2, 3–4, 8–9, 12, 18, 39, 227, 348, 376
Outlook 2, 227, 376
Visual C++ 3, 18, 39
Windows 3–4, 8–9, 12, 386, 406
`MIMEType` 27
missing services 407
Mixin classes
`see also` M classes
concepts 102, 150, 204
`mLostPhone` 379–80
MMC cards 25, 94
`MMessage` 142–3, 146–8
MMS `see` Multimedia Messaging Service
`MMsvSessionObserver` 173–90, 204–25
`MMsvStoreObserver` 177–90
mobile phones
`see also` device...; smartphones
BenQ 429
difference handling 52–3, 80–6, 113–15, 407–10
Motorola 431, 436
multiple models 52–3, 80–6, 113–15, 407–10
NOKIA 6, 8–9, 50, 52–3, 60, 81, 98, 385, 422–5, 427, 433–5, 438–9
Panasonic 437
Sendo 6, 428
Siemens 426
Sony Ericsson 6, 9, 41, 53, 385, 430, 432
specifications 421–39
`Mode` 251–81
`Model` 20, 34, 49–52, 80–2, 347–58
`Month` 322–46
`MonthlyByDates` 304–46
`MonthlyByDays` 304–46
most significant byte (MSB) 145–6
Motorola 431, 436
`MoveL` 180–90, 242–81
MP3 players 2, 117
MSB `see` most significant byte
`MServerSocketObserver` 139–44, 159–61, 165–6
`msvapi.h` 182
`msvids.h` 172–3
`msvstd.h` 175–90
`mtclreg.h` 185–7
MTMs `see` Message Type Modules
multimedia 1–2, 7, 22–7, 52–3, 117, 410
`see also` audio; bitmaps; images; video
Multimedia Messaging Service (MMS) 168, 170–4, 396
multiphone programming, Symbian OS 113–15, 409–10
multiple models 52–3, 80–6, 113–15, 407–10
multiple versions, protocols 404, 407–8

`Name` 35, 49–52, 80–2, 190, 324–46
named servers `see` socket servers
naming conventions, Symbian OS 69, 100–3
navigation techniques
contacts 232–6, 367–84
filing systems 53–4, 58–60, 63–6, 86–7
messaging 172–3
time considerations 60
Visual C++ 86–7
`NewClientL` 142–3
`NewCommandL` 157–61
`NewInfoL` 126–9
`NewL` 95, 105, 125–9, 154–61, 178–90, 205–25, 239–81, 286–346, 398–401
`NewLC` 105–6, 159–61, 212–25, 239–81, 314–46
`NewMessageL` 143, 157–61
`NewMessageReceived` 150–1
`NewMessageReceivedL` 141
`NewMtmL` 186–90
`NewSessionL` 119–23, 125–9
NOKIA 6, 8–9, 50, 52–3, 60, 81, 98, 385, 422–5, 427, 433–5, 438–9
non-modifiable descriptors, concepts 108
non-preemptive multitasking, concepts 110
`NotesTextL` 313–46
`NumDaysSet` 300–46
`NumSCAddresses` 189–90

OBEX `see` Object Exchange Protocol
Object Exchange Protocol (OBEX) 10, 168
object-oriented programming, concepts 3

observers 139–44, 159–61, 165–6, 173–90, 204–25, 230–1, 254–81
  concepts 139–44, 159–61, 165–6, 173–90, 204–25, 230–1, 254–81
  contacts 230–1, 254–81
Ogg formats 117
OleView 85
`OnDeviceConnected` 28
`OnDeviceCopyStorageFileComplete` 30, 56–7, 74–7
`OnDeviceCopyStorageFileError` 29, 56–7, 75–7
`OnDeviceCopyStorageFileExistingFileFound` 30, 56–7, 75–7
`OnDeviceCopyStorageFileProgress` 28, 56–7, 74–7
`OnDeviceDisconnected` 28
`OnDeviceListChanged` 349–58
`OnPaint` 62–6, 75–7
`OnPhoneUpdate` 354–8
`OnRead` 33, 37, 135–6, 165, 356–8, 368–84
`OnWrite` 33, 37, 134–5, 222–5, 356–8
`Open` 102
open source projects, Symbian OS 419–20
`OpenAsyncDeviceService` 21, 32–3
`OpenAsyncL` 181–90
`OpenContactL` 233–81
`OpenContactLX` 233–81
`OpenDeviceService` 21, 32–3, 347–58
`OpenL` 231–81, 287–346
`OpenSyncL` 182–90
`Operator` 179–90
OPL 89
optimization issues 412
OTA *see* Over The Air
out-of-memory errors 102–6, 118, 406
out-of-process COM servers, concepts 15
Outbox 172–3, 210–25
Over The Air (OTA) 138, 406
overview 4
`OwnCard...` 235–81

packets
  custom servers 123–4, 132–3
  design issues 404
  protocol conventions 123–4, 132–3, 404
  size issues 404
packing/unpacking data, SMS 200–4
Panasonic 437

panics, concepts 106, 155–61, 188
`ParaFormatLayer` 289–346
`Parent` 25–6, 58–60, 176–90
`Path` 22, 24–5, 26, 58–60
PC Connectivity applications
  agenda model 283–346, 384–96
  architectural overview 11–13
  client-server processes 12–13, 109–10, 403–12
  compression considerations 405
  concepts 1–4, 7–10, 39–52, 359, 396, 403–12
  connection processes 8–9, 11–12, 39–52, 60–6, 78–86, 123, 247–58, 290–346, 406–10
  contacts 1, 10, 16, 121–2, 227–81, 359, 367–84
  custom servers 8–9, 13, 95–6, 117–36, 191–225, 256–81, 397, 410
  debugging considerations 405, 410–12
  defensive designs 406–7
  definition 2
  design issues 403–12
  device difference handling 52–3, 80–6, 113–15, 407–10
  device and service management 52–3, 113–15, 180–6, 407–10
  disconnection events 19–20, 27–8, 60–6, 73–8, 80–8, 141–2, 348–58, 367
  dos and don’ts 403–12
  examples 39–88
  file-browser example 39–88
  GUI applications 347–96, 409
  historical background 7–10, 117
  Java 3, 89
  loading considerations 132, 409–10
  localization considerations 405
  messaging 7–8, 117–66, 167–90, 191–225, 256–81
  missing services 407
  multiphone programming 113–15, 409–10
  multiple models 52–3, 80–6, 113–15, 407–10
  OBEX 10, 168
  PLP 8–9, 117
  power interruptions 406–7
  protocol designs 191–2, 403–4
  robust designs 405, 406–7
  SMS 1, 10, 44–5, 121–2, 167–90, 191–225, 358–67
  socket servers 13, 35–6, 137–66, 191–225, 256–81, 397–401, 410
  TCP/IP 8–9, 11–13, 35–6, 123–4, 137–66, 397–401, 405, 410

PC Connectivity applications (*continued*)
testing facilities 96–7, 405, 410–12
PC suites
concepts 2, 8–10, 12–13, 15–16
SCOM 15–17
typical functions 2, 15–16
PDAs 8, 41–2, 98
PDU see Protocol Data Unit
PeekAtTodoListList 288–346
PeekInt32L 200–4, 400–1
PeekL 149
Perl 4, 92
Personal Information Manager (PIM) 2, 8, 16, 227–81, 325–6
see also Lotus Notes; Microsoft...
PhoneUpdate 354–8
PIM see Personal Information Manager
pipe processors 117–18, 137
PLP 8–9, 117
plug-ins
see also custom servers
concepts 8–9, 90–1, 95, 103, 106, 112, 114–15, 117–36, 192–225
pointer descriptors
see also descriptors; TPtr...
concepts 107–8
pointers 103–8, 125–9
Pop 104–6, 125–9, 159–61, 206, 213–25
POP3 168, 174
PopAndDestroy 105–6, 160–1, 207–25, 272–81, 338–46, 398–401
PopulateDayInstanceListL 295–346
PopulateMonthInstanceListL 296–346
PopulateTodoInstanceListL 297–346
Port 36
ports 36, 40–8, 138–66, 397–401
m-Router 40–3, 45–8
socket servers 138–66, 397–401
power interruptions, PC Connectivity applications 406–7
PPP 8, 11–12
preemptive multitasking, concepts 110
PrepareToWrite 147, 152–61
Priority 318–46
processes, Symbian OS 109–10
programming
see also software...
code conventions 5
crashes 406–7
good practices 412
multiphone programming 113–15, 409–10
optimization issues 412
panics 106, 155–61, 188
skills 411
Symbian OS background 89–115, 411
programs directory 161–2
Progress 289–346
Protocol Data Unit (PDU) 121, 191–225, 259–81, 326–46, 351–67, 370–84, 388–96, 404–5, 407
protocols
agenda models 325–46, 388–96
contacts connectivity 256–81, 370–84
custom servers 123–9, 132–3, 191–225, 256–81, 325–46
debugging plans 405
design issues 403–5
documentation needs 191–2, 403–4
dos and don'ts 403–5
multiple versions 404, 407–8
PDU 121, 191–225, 259–81, 326–46, 351–67, 370–84, 388–96, 404–5, 407
reverse engineering 12–13
size issues 404
tuning considerations 405
version information 404, 407–8
prototyping issues 403
PruneExceptions 311–46
Psion 7–8, 41, 98
'pull' concepts, messaging 174
'push' concepts, SMS 174
PushL 105–6, 159, 209–25, 280–1, 338–46, 398–401

R classes, concepts 101–2, 105–6, 108–9
RAgendaServ 286–346
RAGNDateTime 387–96
RAGNEntry 388–96
RAGNInstance 387–96
RAGNUtils 385–96
RAM drives 94
RArray 108–9
RCNTField 367–84
RCNTUtils 368–84
RContactViewSortOrder 251–81
Read 32–3, 36–7, 112, 132–6, 152–61, 164–5, 200–4, 210–25, 232–81, 347–96, 400–1
ReadComplete 112
ReadCompleteL 119–29, 215–25

ReadContactL 232–6
ReadContactLC 233–6
reading operations, SMS 192–209, 359–67
ReadL 147, 152–61
ReadMinimalContactL 232–6, 272–81
ReadPtr 147, 152–61
ReadStoreL 171–90
Receive 146–7
ReceiveCompleteL 146–7
Recipients 206–25
recipients, SMS 196–225
RecvOneOrMore 400–1
reference boards 40–7
Refresh 25
RegisterPort 142
registration, socket servers 161–3
registry 50–2, 188–90, 348–58, 384
release directory 93–4, 97–8
remote installation, software 2, 8, 15–16
removable drives 25, 54–5
Remove 242–81, 350–8, 364–7, 392–3
RemoveAddressee 187–90
RemoveAllExceptions 311–46
RemoveCommand 140–1, 144–5
RemoveContactFromGroupL 235–81
RemoveException 311–46
RemoveField 238–81
RemoveFieldType 244–81
RemovePhone 51–2, 83–6
Rename 26–7, 67–8, 69–70
renaming operations 26–7, 67–8, 69–70, 88
  concepts 26–7, 67–8, 69–70, 88
  time considerations 70
repeat APIs, agenda model 298–306
RepeatForever 306–46
repeating agenda entries 285, 298–306, 387–96
RequestId 57
Reset 109, 146–7, 149, 153–61, 242–81, 320–46, 354–8, 378–84
ResetContact 378–84
ResetPhoneList 78, 354–8
ResetStore 244–81
responses
  command–response sequence 192–225
  packing/unpacking data 200–4
restore functions 2, 8, 15–16, 171, 188–90
RestoreBodyTextL 171–90
RestoreL 190
RestoreMessageL 190
reverse engineering, protocols 12–13
ReverseOrder 242–81
RFile class 101–2
Rich Text messages 170–90, 205, 313–46
RichTextL 313–46
RLibrary 401
robust designs, PC Connectivity applications 405, 406–7
robustness needs 102–3
rollbacks/recoveries, databases 231
ROM conventions 54, 58, 93, 129
  z: drive 54, 58, 94
root directories
  concepts 24, 54–7, 86–7
  copy functions 54–7
RootDirectory 24, 54–7, 58–60, 86–7
RPointerArray 108–9
RProcess 401
RptDef 311–46
RptNextInstanceOnly 293–346
RS232 serial connections 8, 11–12, 40, 41–3, 45–6
RServiceBrokerClient 141–66
RSMSAddress 361–7
RSMSUtil 359–67
RSocket 399–401
RThread 401
RunError 111–13, 398–401
RunL 110–13, 119–23, 212–25, 397–401

SaveMessage 171–90
SaveMessageL 186–90
SCAddress 189–90
SCAsyncStreamSink 21, 32–3
SCBAL.exe 48–52
ScDriveAttributes 22–3
ScDriveBatteryState 22–4
ScErrorDescription 29
ScFormatStorageFormatting 30–1
ScFormatStorageProgress 30–1
ScMediaAttributes 23
ScMediaTypes 23–4
SCOM see Symbian Connect Object Module
ScOverwrite 29–30
screens 7–8
  see also multimedia; video
  emulators 98–100
ScStorageType 22, 24, 26

SDKs see software development kits
Send 146–7, 150–61
Send-As API, concepts 169
SendButton 366–7
SendCompleteL 140–1, 146–7, 150–1
SendError 150, 154–61
SendErrorMessage 150, 154–61
SendingState 177–90
SendMessage 155–61
Sendo 6, 428
SendResponse 140–1, 151, 155–61
SendSmsL 210–25
serial connections 8, 11–12, 40, 41–5, 47
  concepts 8, 11–12, 40, 41–5, 47
Serialise 145–6, 153–61, 200–4
server socket classes 142–51
server-side MTMs, concepts 168–9
servers
  see also client-server...
  concepts 12–13, 15–38, 109–10
ServerSocketStoppedDueToErr 142, 159–61, 165–6
Service Brokers, concepts 138–66, 398, 400–1
Service Centers, SMS settings 211–25
ServiceId 188–90
ServiceRead 357–67, 370–84, 393–6
services
  BAL 34–7, 357–8
  SCOM 32–3
ServiceSettings 188–90
sessions
  custom servers 119–36
  messaging 173–4
SetActive 21, 112, 212–25
SetAddress 190
SetAlarm 319–46
SetAlarmFromDueDate 319–46
SetAlarmFromStartDate 319–46
SetAttrib 205–25
SetBaseYear 317–46
SetBodyLength 147, 152–61
SetButtonStates 354–96
SetComplete 176–90
SetCurrentEntry 175–90
SetCurrentEntryL 186–90
SetDaily 304–46
SetDate 302–46
SetDay 300–46
SetDbViewContactType 232–81
SetDeleted 237–81
SetDisplayDueDateAs 318–46
SetDisplayNextOnly 299–346
SetDisplayTime 315–46
SetDueDate 318–46
SetDuration 319–46
SetEndDate 298–346
SetEntryL 173–90, 209–25
SetEventPriority 314–46
SetEventSession 216–25
SetFromAddressL 189–90
SetGroupByMtm 184–90
SetGroupByPriority 184–90
SetGroupByType 184–90
SetGroupLabelL 241–81
SetGroupStandardFolders 184–90
SetHasAlarm 309–46
SetHasBaseYear 317–46
SetHasExceptions 311–46
SetHidden 237–81
SetId 307–46
SetIdAndDate 308–46
SetIdAndInstanceDate 313–46
SetIncludeAlarmedOnly 293–346
SetIncludeAnnivs 292–346
SetIncludeCrossedOutOnly 293–346
SetIncludeEvents 292–346
SetIncludeNonRpts 293–346
SetIncludeRpts 293–346
SetIncludeRptsNextInstanceOnly 293–346
SetIncludeTimedAppts 292–346
SetIncludeTodos 293–346
SetIncludeUnTimedAppts 292–346
SetInPreparation 177–90
SetInstanceDate 312–46
SetInstanceId 313–46
SetInterval 299–346
SetIsADayNote 310–46
SetIsCrossedOut 310–46
SetIsRepeating 310–46
SetIsTentative 310–46
SetLabelL 245–81
SetLastModified 237–81
SetLength 153–61
SetMapping 246–81
SetMode 251–81
SetMonth 322–46
SetMonthlyByDates 304–46
SetMonthlyByDays 304–46

SetName 190, 324–46
SetNameL 190
SetNotesTextL 313–46
SetNullId 307–46
SetNullInstance 308–46
SetOwnCardL 235–81
SetPriority 110–11, 318–46
SetRepeatForever 299–346
SetRptDefL 313–46
SetRptEndDate 310–46
SetRptStartDate 310–46
SetSendingState 177–90
SetServiceCenterAddressL 190
SetShowInvisibleEntries 185–90
SetSmsSettingsL 190
SetSorting 184–90
SetSortOrder 325–46
SetSortTypeL 178–90
SetStandardText... 248–81
SetStartAndEndDateTime 314–46
SetStartDate 298–346
SetTemplateLabelL 240–81
SetTemplateRefId 238–81
SetText 248–81
SetTextArray 248–81
SetTextL 248–81
SetThingL 249–81
SetTime 249–81
SetTodoListId 318–46
SetUnread 177–90
SetUse 251–81
SetVisible 177–90
SetWeekly 304–46
SetYear 322–46
SetYearlyByDate 305–46
SetYearlyByDay 305–46
Short Messaging System (SMS) 1, 10, 44–5, 121–2, 167–90, 191–225, 358–67
  see also messaging
  asynchronous service 174–5, 211–25
  classes 185–90, 359–67
  command-line applications 219–25
  command–response sequence 192–225
  concepts 167–90, 191–225, 358–67
  content return 206–9
  creation procedures 209–25, 359–67
  custom servers 191–225
  deletions 192–3, 198–225, 359–67
  errors 193–225
  event handlers 193–225
  GUI applications 191, 219, 225, 347, 358–67
  length issues 174–5, 196–225
  lists 206–9, 358–67
  management protocols 191–9
  management-connectivity-service development 191–225
  message servers 169, 204–25
  MTMs 169, 204–25
  packing/unpacking data 200–4
  reading operations 192–209, 359–67
  Service Centers 211–25
  socket servers 191–225
  specific classes 187–90
  specific variations 174–5
ShowInvisibleEntries 185–90
Siemens 426
Size 26
skills, programming 411
smartphones
  see also mobile phones
  concepts 1–6, 7–10, 45–6, 52–3, 113–15, 401
  difference handling 52–3, 80–6, 113–15, 407–10
  multiple models 52–3, 80–6, 113–15, 407–10
  USB 45–6
SMS see Short Messaging System
smsclnt.h 188–90
SmsHeader 174–90, 206–25
SMTP 168
smuthdr.h 189–90
smutset.h 189–90
socket servers
  asynchronous service 165, 211–25
  classes 138–51
  command classes 140–3, 149–61
  comms 164–5, 398–401
  concepts 13, 35–6, 137–66, 191–225, 256–81, 351, 397–401, 410
  contacts connectivity 256–81
  creation procedures 138, 151–61
  custom servers 137
  debugging 138, 155–61, 165–6, 397
  definition 137–8
  development 151–61
  echo socket server 151–61
  EKA2 151–2, 161, 401
  emulators 161–2, 165–6
  errors 141–2, 150, 154–61, 164–6
  factory classes 139–43, 151–61

socket servers (*continued*)
general socket servers 397–401
installation 161–3, 397–401
libraries 138
overview 137–8
ports 138, 397–401
registration 161–3
SCOM 163–5, 401
SMS management 191–225
`SocketClosing` 150–1
softkeys 7
software
`see also` programming
remote installation 2, 8, 15–16
software development kits (SDKs) 5–6, 125, 136, 399, 413–14
Sony Ericsson 6, 9, 41, 53, 385, 430, 432
`Sorting` 184–90
sorting methods
agenda model 295–346
contacts 232–6
`SortL` 232–6
`SortType` 179–90
`SOURCE` 90–1
source files, concepts 90–6
`SOURCEPATH` 90–1
specifications, mobile phones 421–39
stacks, protocols 123–4
stand-alone PC Connectivity applications, concepts 1–2
Standard Template Library (STL) 108
standardization benefits 9–10
`StandardTextLC` 248–81
`Start` 35, 113, 159–61, 212–25
`StartDate` 298–346
`StartDateTime` 315–46
`StartGeneralService` 400–1
`StartL` 159–61, 398–401
`StartService` 36, 400–1
`StartServiceOnStream` 36, 131–2, 163–5, 221–5
`State` 148–9
static classes, concepts 106, 145, 148
`StdAfx.h` 79, 86
STL `see` Standard Template Library
`Stop` 113, 161
`Storage` 21, 70–1
storage classes, SCOM 21, 22–31, 58–60, 86–7
`StorageDrives` 21, 58–60, 86–7
`StorageType` 244–81
`StoreBodyTextL` 171–90
`StoreL` 190
stores, messaging 171–2, 178–90, 213–25
`StoreStorage` 249–81
streams
BAL 35–7, 347–96, 401
message stores 171
SCOM 32–3, 130–2, 401
`StringBuilder` 362–3, 371–84
strings 49, 106–8, 358, 361–7, 405
support forums 416–17
`SwitchCurrentEntryL` 175, 186–90, 206–25
Symbian Connect Object Module (SCOM) 9, 13, 15–38, 39–88, 351–2, 401, 409
application, connection and device classes 18–21, 27–32, 38, 58–66
BAL 16–17, 33, 134–6, 347–96
C# access 48–52, 78, 80, 86, 133
C++ 37–8, 39, 78–88
caching operations 60, 81–2
classes 18–33, 39, 58–66, 79–88, 100–2
concepts 9, 15–38, 39–88, 351–2, 401, 409
custom servers 130–2
device connection 19–20, 27–31, 38, 48–52, 60–6, 78–86, 87–8, 409
efficiency issues 17
errors 18, 20, 29–32, 55–7, 68–78, 81–8
event handling 19–20, 25, 27–32, 52, 55–7, 61–6, 85–8
functionality provisions 15–38
PC Suites 15–16
services 32–3
socket servers 163–5, 401
storage classes 21, 22–31, 58–60
streams 32–3, 130–2, 401
type library 48–52, 79–80, 85–6
Visual Basic 37–8
Visual C++ 78–88
Symbian OS
`see also individual topics`
active objects 110–13, 144, 160–1, 212–25, 399–401
agenda model 1, 16, 283–346, 384–96
arrays 106–7, 108–9, 133
backwards compatibility 113–15
book list 419
build processes 90–6
C++ 3, 5, 37–8, 39, 78–88, 89–115

clients 109–10
community links 417–18
concepts 1–6, 7–10, 89–115
contacts 1, 10, 16, 121–2, 227–81, 367–84
databases 227–346, 368–84, 396
descriptors 106–8, 124
design issues 403–12
development resources 5–6, 413–20
dos and don'ts 403–12
EPOC 7, 97–8, 151–2, 398–401
error handling 102–6
historical background 7–10, 117
messaging 7–8, 117–66, 167–90, 191–225, 256–81
multiphone programming 113–15, 409–10
naming conventions 69, 100–3
OBEX 10, 168
open source projects 419–20
PLP 8–9, 117
processes 109–10
programming background 89–115, 411
servers 12–13, 15–38, 109–10
TCP/IP 8–9, 11–13, 35–6, 123–4, 137–66, 405, 410
threads 109–10
v6.0 7–8
v6.1 3, 7–9, 13, 40, 46, 114–15
v7.0 3, 5, 7–9, 40, 114–15
v7.0s 7–9, 13, 40, 114–15
v8.0 3, 13, 35, 41, 114–15, 137–8, 152, 397–8, 400–1, 410
SymbianConnectBAL 48–52, 134–5, 163–5, 348–96
SymbianConnect.dll 48–52
SymbianConnectRunTime 48–52
SynchroniseDateTime 21
synchronous service 32–3, 35, 55, 78, 125–36, 211–25, 357–8
copy functions 55, 78, 87–8
custom servers 125–36, 211–25
disconnection operations 78, 87–8
errors 78
messaging 169, 180–90, 211–25
system templates, contacts 275–81
SYSTEMINCLUDE 90–1

T classes, concepts 101, 105
TAgnDailyRpt 299–346
TAgnDate 306–46
TAgnEntryId 287–346
TAgnException 306–46
TAgnFilter 291–346
TAgnId 306–46
TAgnInstance 307–46
TAgnInstanceDateTimeId 321–46
TAgnInstanceId 307–46
TAgnMonthlyByDaysRpt 301–46
TAgnMonthlyRpt 300–46
TAgnRpt 298–346
TAgnsrvFindFilter 292–346
TAgnsrvTidyFilter 292–346
TAgnStatus 309–46
TAgnSymbolFilter 292–346
TAgnTodoListId 288–346
TAgnWeeklyRpt 300–46
TAgnWhichInstances 291–346
TAgnYearlyByDateRpt 302–46
TAgnYearlyByDayRpt 303–46
TakeOwnershipOfMessage 150–1
TARGET 90–1
TARGETPATH 90–1
TARGETTYPE 90–1
TBuf concrete class type 107–8, 211
TBufC concrete class type 107–8
TBufCBase class type 107–8
TCleanupItem 105–6
TConnBuff 200–4
TContactItemId 254–81
TContactViewEvent 253–81
TContactViewPreferences 252–81
TCP/IP
see also general socket servers
concepts 3, 8–9, 11–13, 35–6, 123–4, 137–66, 397–401, 405, 410
custom servers 123
PC Connectivity applications 8–9, 11–13, 35–6, 123–4, 137–66, 397–401, 405, 410
socket servers 137–66, 397–401
TCreator 157–61
TDes base class, concepts 107–8, 121–2, 143–7, 152, 200–4, 342–6
TDesC base class, concepts 107–8, 127–9, 143–7, 152, 187–90, 232–6, 313
TDisplayAs 316–46
TDisplayDueDateAs 317–46
TDriveNumber 181–90
TechView emulator 98
TechView Technology Kit 5

templates, cards 228, 235–81, 367, 370–84
TEndian 145, 148–9
testing facilities 96–7, 405, 410–12
see also debugging
emulators 96–7, 410–12
log files 410–12
test harnesses 411–12
TEventType 253–81
Text 248–81
text strings, transfer considerations 405
Thing 249–81
threads, concepts 109–10
Time 249–81, 387–96
time considerations 55, 60, 70, 72–7, 121–2, 244–81, 285–346, 351–8, 387–96, 409–10
agenda model 285–346, 387–96
Bluetooth 60
contacts 244–81
copy functions 55, 72–7
deletions 70
GUI applications 351–8, 387–96
infrared connections 60
messaging 170–3, 206–25
navigation techniques 60
renaming operations 70
times 21, 27, 30, 59–60, 72–7, 206–7, 220–5
TInt types 101, 103, 110–11, 122, 126–7, 141–2, 176–90, 200–25, 234–6, 243–6, 254, 299
TLibraryFunction 401
TListItem 322–46
TMsvEntry 170–90, 206–25
TMsvEntrySelection 183–90
TMsvGrouping 183–90
TMsvId 175–90, 208–25
TMsvSelectionOrdering 178–90
TMsvSessionEvent 181–90, 218–25
TMsvSorting 183–90
to-dos
agenda model 284–346, 396
sorting methods 296
TodoListCount 288–346
tools directory 93
TOpenCallBackFrequency 290–346
ToString method 49, 358, 361–7, 374–84, 386–96
touch-sensitive screens 7, 99
TPtr concrete class type 107–8, 176–90
TPtrC concrete class type 107–8, 143–4, 176–90, 240–2, 324
TRAgnCmdCode 326–46
trap harness, concepts 102–3, 111–12, 215–25
TRAP macro 102–3, 111–12, 215–25
TRAPD macro 102–3, 161, 217–25
TRCntCmdCode 257–81
tree views 63–76, 87
TRemsvr 200–4
TRequestStatus 111–13, 180–90
TRSmsCmdCode 192–225
TserverInfo 142–3, 160–1
TSortOrder 323–46
TSortPref 236–81
TState 148–9
TThreadFunction 401
TTime... 202–4, 287–346
TType 291–346
TUid 175–90, 237–81
TUint 143–4, 150, 153–61, 202–4, 307–8, 399–401
tuning considerations, protocols 405
two-phase construction, concepts 104–6, 143–5, 151, 205–25
Type 22, 24–6, 312–46, 373–84
type library
BAL 48–52
SCOM 48–52, 79–80, 85–6

udeb build 92–4, 97–8
UIDs 90–1, 94–5, 128–9, 162–3, 205–6, 247, 251–2, 414
UIQ, emulators 98–100
UnCrossOut 319–46
Unicode 107, 196–225
UniqueId 24
Unix 60
unloading considerations 132, 409–10
Unread 177–90
UnsetDay 300–46
UpdateEntryL 288–346
UpdateFieldSet 238–81
UpdateInstanceL 294–346
UpdatePhoneList 61–3, 348–58
UpdateTodoListL 288–346
urel build 92–4
USB 11–12, 44, 45–6
Use 251–81
User::Leave() 102–3, 126–9, 200–4, 207–25, 342–6, 399–401
User::Panic() 106

user interfaces, concepts 2, 7–8
UserAddField 243–81
USERINCLUDE 90–1

VB see Visual Basic
VC++ see Visual C++
Version 35
version information, protocols 404, 407–8
video 1–2, 53, 117
  see also images
views, contacts 230–1, 249–81, 368–84
virtual destructors 105–6
virtual functions, concepts 120–9
Visible 177–90
Visual Basic (VB) 3, 18, 37–8, 89
Visual C++ (VC++) 3, 18, 39, 78–88
  concepts 3, 18, 39, 78–88
  copy functions 87–8
  device connection 78–86, 87–8
  navigation techniques 86–7
  SCOM 78–88
visual feedback, connection processes 41
VolumeLabel 24

'waiter' objects 205
watchers 44
Weekly 304–46
wins 92
winscw directory 92–4, 97–8
winsock 17
Write 32–3, 36–7, 112, 122–3, 128–36, 164–5, 202–4, 206–25, 272–81, 349–84, 387–96, 401
WriteComplete 112
WriteCompleteL 122–3
WriteErrorL 401
WriteInt8L 339–46, 380–4, 393–6
WriteInt16L 273–81, 339–46, 380–4, 393–6
WriteInt32L 272–81, 339–46, 360–7, 380–4, 387–96, 401
WritePtr 147, 152–61, 164–5

XML 162–3

YearlyByDate 305–46
YearlyByDay 305–46
z: drive 54, 57–60, 94, 129–30, 161–2
  custom servers 129–30
  ROM conventions 54, 58, 94, 129
  socket servers 161–2
zero initialization, CBase class 101