"If you're interested in developing for this burgeoning platform, there is no one better able to get you up-to-speed."

—From the Foreword by Rob Tiffany, mobility architect, Microsoft

Second Edition

Programming .NET Compact Framework 3.5

Paul Yao
David Durant

Download at WoweBook.Com

Praise for *Programming .NET Compact Framework 3.5, Second Edition*

"Each day it seems that mobile devices are becoming more common and critical to our everyday lives. As more developers target this growing market, the need for a trusted guide grows, and this book provides that guidance. The authors have reached a rare balance in which they provide the core knowledge required to start the journey of mobile development as well as the depth to support a seasoned developer. Every developer building applications for Windows Mobile devices should have this book close at hand to guide them through the challenges and rewards of creating their applications."

—Mike Erickson, principal consultant, Neudesic

"This is one of the most complete technical books available on developing smart-device applications with Visual Studio and the Compact Framework 3.5. It covers many necessary topics, from preserving battery life to communicating with the outside world using WCF and everything in between. Whether you have been developing in the Compact Framework for years or are just getting started, this book has something in it for you."

—Jason Estes, senior software engineer, Itron

"This book arms Windows Mobile developers with the core knowledge they need for building rich business and personal applications. It also gives them great insight into what is going on under the covers, how the platform is unique compared to other versions of Windows, and how to build robust Windows Mobile applications productively."

—Brian Noyes, chief architect, IDesign, Inc.

"This is the best resource I've ever seen for programming the Compact Framework. This book should be on the desk of everybody doing mobile development."

—Rebecca M. Riordan, author of Seeing Data

"With valuable information added to this new edition, Paul Yao and David Durant have enhanced their already invaluable guide to programming for the .NET Compact Framework. If you're a serious mobile programmer, this is the one book you need on your shelf—but it won't stay there."

—Josh Trupin, Microsoft Corporation

"Yao and Durant's second edition of their popular .NET CF book is a much-needed update for all developers who want to build apps for the Windows Marketplace for Mobile. The book provides comprehensive coverage of the technology, and it is well suited for both .NET CF beginners and experts."

—Michael Yuan, cofounder, Ringful LLC

Download at WoweBook.Com

This page intentionally left blank

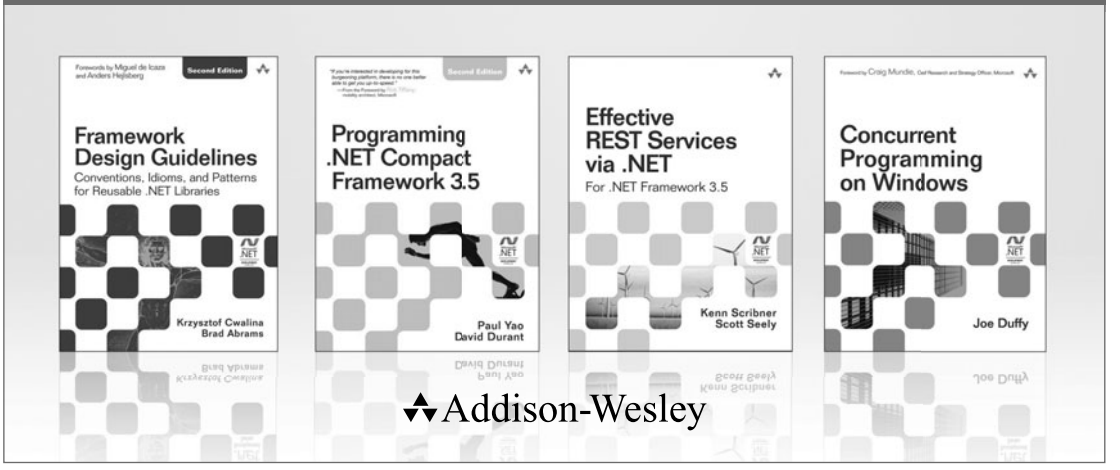
Download at WoweBook.Com

Programming .NET Compact Framework 3.5

Second Edition

Download at WoweBook.Com

Microsoft® .NET Development Series



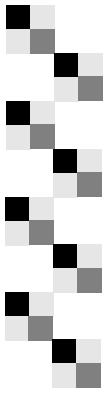
Visit **informit.com/msdotnetseries** for a complete list of available products.

The award-winning **Microsoft .NET Development Series** was established in 2002 to provide professional developers with the most comprehensive, practical coverage of the latest .NET technologies. Authors in this series include Microsoft architects, MVPs, and other experts and leaders in the field of Microsoft development technologies. Each book provides developers with the vital information and critical insight they need to write highly effective applications.

PEARSON

Addison-Wesley | Cisco Press | EXAM/CRAM | IBM Press | QUE | PRENTICE HALL | SAMS | Safari Books Online

Download at WoweBook.Com



Programming .NET Compact Framework 3.5

Second Edition

Paul Yao
David Durant

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Download at WoweBook.Com

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries/regions.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Yao, Paul.
Programming .NET Compact Framework 3.5 / Paul Yao, David Durant. — 2nd ed.
p. cm.
Rev. ed of: .NET Compact Framework programming with C#. 2004.
Includes index.
ISBN 978-0-321-57358-2 (pbk. : alk. paper)
1. C# (Computer program language) 2. Microsoft .NET Framework. I. Durant, David. II. Yao, Paul. .NET Compact Framework programming with C#. III. Title.
QA76.73.C154Y36 2009
006.7'882—dc22

2009022724

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston St., Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-57358-2
ISBN-10: 0-321-57358-7
Text printed in the United States on recycled paper at Edwards Brothers, in Ann Arbor, Michigan.
First printing, September 2009

Download at WoweBook.Com

For my son, Jasper Harold Yao.
—Paul

To my wife, Betty, for her love and support during the writing of this book.
—Dave

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

Contents

Figures xvii
Tables xxi
Foreword xxv
Preface xxvii
Acknowledgments xxxiii
About the Authors xxxvii

1 **Mobile Phone Programming** 1

1.1 **Selecting an Application Programming Interface** 4
1.1.1 *The Win32 API* 8
1.1.2 *The .NET Compact Framework* 14
1.1.3 *The Web Browser* 17
1.1.4 *Rich Internet Applications (RIAs)* 18

1.2 **Memory Management** 19
1.2.1 *Metadata Maps* 19
1.2.2 *The JITted Code Pool* 20
1.2.3 *Garbage Collector Pools* 22
1.2.4 *Garbage Collection and Data* 23
1.2.5 *Automatic Garbage Collection* 25
1.2.6 *Special Handling of Managed Data* 27
1.2.7 *Manual Memory Management of Native Data* 36

1.3 **Conclusion** 37

Download at WoweBook.Com

2 Extending Battery Life 39
2.1 What Is the Problem? 40
2.1.1 *Basic Questions to Ask* 41
2.1.2 *Factors Influencing Battery Life* 41
2.2 Measuring Battery Usage 43
2.2.1 *Talk Time and Standby Time* 44
2.2.2 *Power Measurement Method 1: Software-Only Approach* 45
2.2.3 *Power Measurement Method 2: Dedicated Hardware* 48
2.3 Device Power Study 51
2.3.1 *Standby Power Usage* 53
2.3.2 *Backlight Power Usage* 54
2.3.3 *Power Usage for Communicating* 57
2.3.4 *Multimedia Power Usage* 57
2.4 Conclusion 58

3 Platform Invoke 59
3.1 Overview of P/Invoke 60
3.1.1 *When to Use P/Invoke* 61
3.1.2 *Why We Prefer .NET Compact Framework Classes over Win32 Functions* 63
3.1.3 *Porting Native Code to Managed Code* 64
3.1.4 *Component Object Model (COM) Support* 65
3.2 Creating P/Invoke Declarations 66
3.2.1 *A Simple Function:* `MessageBox` 66
3.2.2 *Native Function Details* 68
3.2.3 *Function Return Values* 72
3.2.4 *Getting Started: C-Style Function Declarations* 72
3.3 Supported P/Invoke Function Parameters 75
3.3.1 *The Limits of Parameter Passing* 76
3.3.2 *Simple Data Types* 76
3.3.3 *Passing Parameters by Value versus by Reference* 79
3.3.4 *Passing String Parameters by Value* 82
3.3.5 *Structures* 82
3.3.6 *Type of Types versus Type of Parameters* 88
3.3.7 *Arrays* 90
3.4 A Sample Program: CallWin32 93
3.5 Writing Win32 Dynamic Link Libraries 96
3.5.1 *Declaring C++ Functions in a DLL* 97

Download at WoweBook.Com

3.6 Manual P/Invoke Parameter Passing 100
3.6.1 *The* `Marshal` *Class* 101
3.6.2 *Copying to Native Memory* 102
3.6.3 *Fine-Tuning Structures with the* `MarshalAs` *Attribute* 105
3.7 Communicating between Native and Managed Code 108
3.7.1 *The* `MessageWindow` *Class* 108
3.7.2 *Other Ways to Communicate between Native and Managed Code* 111
3.8 Comparing P/Invoke Support 112
3.8.1 *Windows CE–Specific Differences* 112
3.8.2 *.NET Compact Framework Implementation Details* 113
3.9 Conclusion 113

4 Data Binding to Controls 115

4.1 Data Binding 115
4.1.1 *Data-Bindable Controls* 117
4.1.2 *Data-Bindable Objects* 119
4.2 Complex Data Binding 124
4.2.1 *Using Complex Data Binding with* `ComboBox` *Controls* 125
4.3 Simple Data Binding 126
4.3.1 *The* `BindingsCollection` *Property* 126
4.3.2 *Formatting and Parsing* 129
4.4 The `DataGrid` Control 132
4.4.1 *Using Complex Data Binding with the* `DataGrid` *Control* 136
4.4.2 *Styling the Display of Data in a* `DataGrid` *Control* 137
4.4.3 *Creating Table and Column Styles* 139
4.4.4 *Creating Styles at Runtime* 142
4.4.5 *Responding to User Input* 144
4.4.6 *Using Simple Data Binding with the* `DataGrid` *Control* 149
4.4.7 *Accessing* `DataGrid` *Information* 149
4.4.8 *Providing Drill-Down Capability* 155
4.4.9 *Providing In-Place Editing Capability* 161
4.4.10 *Providing Automated In-Place Editing Capability* 167
4.5 Conclusion 177

5 Storage 179

5.1 Smart-Device Data Storage 179
5.1.1 *Installable File Systems* 181

Download at WoweBook.Com

5.1.2 *The Windows CE File System* 182
5.1.3 *ROM-Based Files* 182

5.2 File I/O 183
5.2.1 *The* `File` *and* `Directory` *Classes* 184
5.2.2 *Byte-Level I/O* 185
5.2.3 *Higher-Level I/O* 187
5.2.4 *Encoding and Decoding Data* 190
5.2.5 *Using the I/O Classes* 191
5.2.6 *Text File I/O* 193
5.2.7 *Binary File I/O* 196
5.2.8 *Writing Binary Data* 197
5.2.9 *Reading Binary Data* 199
5.2.10 *Binary I/O and Structures* 200
5.2.11 *XML File I/O* 203
5.2.12 *Easier XML Serialization* 210

5.3 Registry Access 213
5.3.1 *Opening and Creating Registry Keys* 214
5.3.2 *Reading and Writing Registry Values* 215
5.3.3 *Updating the* `Storage` *Sample Application to Use the Registry* 217

5.4 Conclusion 221

6 ADO.NET Programming 223

6.1 Examining ADO.NET 226
6.1.1 *A Layered Approach* 227
6.1.2 *The ADO.NET Classes* 229
6.1.3 *ADO.NET Error Handling* 234

6.2 Working with Data Sets 236
6.2.1 *Creating and Accessing* `DataSet`, `DataTable`, *and* `DataView` *Objects* 237
6.2.2 *Data Binding* 241
6.2.3 *Reading and Writing a Data Set as XML* 252

6.3 Microsoft SQL Server CE 254
6.3.1 *SQL Server CE Files* 255
6.3.2 *SQL Server CE Syntax* 256
6.3.3 *SQL Server CE Query Analyzer* 259
6.3.4 *Creating a SQL Server CE Database* 261
6.3.5 *Populating a SQL Server CE Database* 263
6.3.6 *Retrieving and Displaying Data* 266
6.3.7 *Updating a SQL Server CE Database* 275

Download at WoweBook.Com

6.3.8 *The* `SqlCeDataAdapter` *Class* 276
6.3.9 *Querying Schema Information* 282

6.4 Microsoft SQL Server 287
6.4.1 *Connecting to SQL Server* 289
6.4.2 *Creating Command Objects* 297
6.4.3 *Using SQL Server Stored Procedures* 297
6.4.4 *Using Stored Procedures with* `DataSet` *Objects* 303
6.4.5 `DataSet` *Objects and Concurrency* 307
6.4.6 *Using Multi* `SELECT` *Stored Procedures* 307
6.4.7 *Working with Typed Data Sets* 310

6.5 Web Services 318
6.5.1 *XML, XSD, and SOAP* 319
6.5.2 *A Web Services Application* 321
6.5.3 *A Web Services Client Application* 331

6.6 Conclusion 335

7 LINQ 337

7.1 Overview 337
7.1.1 *Set Classes in .NET* 340
7.1.2 *LINQ in the Compact Framework* 341
7.1.3 *Deferred Execution* 343

7.2 The Sample Application 344
7.2.1 *LINQ to Datasets* 348
7.2.2 *LINQ to Objects* 355
7.2.3 *Business Object Properties* 366
7.2.4 *The Hybrid Version* 374

7.3 LINQ to XML 378

7.4 Conclusion 383

8 Synchronizing Mobile Data 385

8.1 Understanding SQL Server CE Synchronization 387
8.1.1 *Three Synchronization Mechanisms* 388
8.1.2 *IIS Connectivity* 389
8.1.3 *Database Connectivity* 390

8.2 Installing Remote Data Connectivity 392
8.2.1 *Creating the Virtual Directory* 392
8.2.2 *Configuring Additional Components* 396

Download at WoweBook.Com

8.3 Using RDA 397
8.3.1 *RDA Capabilities and Overhead 397*
8.3.2 *Programming for RDA 398*
8.4 Using Merge Replication 409
8.4.1 *Using Good Design to Avoid Synchronization Failures 410*
8.4.2 *Configuring Merge Replication 415*
8.4.3 *Programming for Merge Replication 416*
8.5 Choosing between Merge Replication and RDA 424
8.6 Using Data Synchronization Services 424
8.6.1 *Understanding Data Synchronization Service Requirements 425*
8.6.2 *Building a Data Synchronization Service 427*
8.7 Conclusion 448

9 The Remote API 451
9.1 RAPI Fundamentals 451
9.1.1 *Available RAPI Functions 452*
9.1.2 *Building .NET ActiveSync Applications 452*
9.1.3 *RAPI Startup and Shutdown 458*
9.2 Accessing the Object Store 470
9.2.1 *Using RAPI to Access Device Files 471*
9.2.2 *Remote Access to Device Registry Entries 486*
9.2.3 *Remote Access to Device Property Databases 496*
9.3 Detecting Changes in Device Connection State 499
9.3.1 *The Auto-Start Approach 500*
9.3.2 *The Callback Approach 501*
9.4 Loading Programs and DLLs 507
9.4.1 *Running Device-Side Programs 508*
9.4.2 *Loading Device-Side DLLs 509*
9.5 Conclusion 514

10 Windows Communication Foundation 515
10.1 What Is WCF? 515
10.1.1 *WCF Terminology 518*
10.1.2 *WCF in the .NET Compact Framework 522*
10.2 Creating a WCF Service 523
10.2.1 *Generating the Code 523*
10.2.2 *Making a Windows Mobile–Compatible WCF Service 525*

Download at WoweBook.Com

10.2.3 *Setting the Host Address* 526
10.2.4 *SmartMeter: A Sample WCF Service* 530

10.3 Creating a WCF Client in Windows Mobile 538
10.3.1 *WCF Client Namespaces and Assemblies* 538
10.3.2 *Generating the WCF Client Proxy* 539
10.3.3 *Instantiating a WCF Client* 541
10.3.4 *Accessing the WCF Service* 542
10.3.5 *WCF Client Sample:* `ReadSmartMeter` 542

10.4 Conclusion 546

11 Creating Graphical Output 549

11.1 An Introduction to .NET Compact Framework Graphics 549
11.1.1 *Drawing Surfaces* 551
11.1.2 *Drawing Function Families* 553
11.1.3 *.NET Compact Framework Graphics* 555

11.2 Drawing on the Display Screen 561
11.2.1 *Accessing a* `Graphics` *Object* 562
11.2.2 *Drawing in Controls* 564
11.2.3 *The* `Paint` *Event* 566
11.2.4 *Non-*`Paint` *Event Drawing* 568

11.3 Raster Graphics 570
11.3.1 *Specifying Colors* 570
11.3.2 *Creating Brushes* 577
11.3.3 *Creating Bitmaps* 579
11.3.4 *Drawing Bitmaps* 591
11.3.5 *A Sample Program:* `ShowBitmap` 596

11.4 Vector Graphics 602
11.4.1 *Creating Pens* 604
11.4.2 *A Game:* `JaspersDots` 605

11.5 Conclusion 619

12 Text and Fonts 621

12.1 Drawing Text 621
12.1.1 *Text-Drawing Support in the .NET Compact Framework* 622
12.1.2 *The* `DrawString` *Method* 623
12.1.3 *A Sample Program:* `SimpleDrawString` 624

12.2 Font Selection 625
12.2.1 *The* `Font` *Property of Controls* 626
12.2.2 *Enumerating Fonts* 627

Download at WoweBook.Com

12.2.3 *A Sample Program:* `FontPicker` 627
12.2.4 *A Sample Program:* `RotateText` 628
12.3 Placing Text 632
12.3.1 *Text Size and the* `MeasureString` *Method* 633
12.3.2 *A Sample Program:* `MeasureString` 634
12.3.3 *Text Alignment* 635
12.3.4 *A Sample Program:* `TextAlign` 636
12.4 Conclusion 640

A Hungarian Notation for .NET Programs 641
A.1 Goals and Objectives 642
A.2 Guidelines 642
A.3 .NET Naming Guidelines 644
A.4 Hungarian Notation 646
A.4.1 *Our Use of Hungarian Notation* 647
A.4.2 *The* `m_` *Prefix for Private Data* 647
A.4.3 *Hungarian Prefixes for CTS Value Types* 649

B Windows API Allocation and Cleanup Functions 659

Index 669

Download at WoweBook.Com

Figures

Figure 1.1: *The GCInfo Sample Calls Methods in the GC Class 28*
Figure 1.2: *Disposer Demonstrating the Operation of the Dispose and Finalize Methods 34*

Figure 2.1: *PowerMeter Sample Displaying Battery Usage 46*
Figure 2.2: *Power Monitor from Monsoon Solutions, Inc. 49*
Figure 2.3: *Power Monitor Powering an HTC Kaiser 51*
Figure 2.4: *Monsoon Solutions Power Tool Software Displaying Power Usage Data 52*
Figure 2.5: *Power Consumption during Transition from Backlight Level 10 to 1 56*

Figure 3.1: *Example of an Exception Caused by a P/Invoke Error 68*
Figure 3.2: *The P/Invoke Wizard Converting the MessageBox Function to C# 75*
Figure 3.3: *Results of a Call to the GlobalMemoryStatus Function 88*
Figure 3.4: *CallWin32 Showing a String Passed to ShowParam.dll, a Native DLL 93*

Figure 4.1: *Application Tiers 116*
Figure 4.2: *Project Data Displayed in a DataGrid Control 117*
Figure 4.3: *A DataTable Object That Contains Project Information 118*
Figure 4.4: *Different Controls Bound to the Same Data 118*
Figure 4.5: *A DataSet Object That Contains Two DataTable Objects 120*
Figure 4.6: *A DataGrid Control Displaying a DataTable Object 132*
Figure 4.7: *A DataGrid Control with the Projects Table Bound to It 138*
Figure 4.8: *A DataGrid Control with the DataView of the Tasks Table Bound to It 138*
Figure 4.9: *Choosing the TableStyles Collection 140*
Figure 4.10: *Adding a Table Style 141*
Figure 4.11: *Entering DataGridColumnStyle Properties 142*
Figure 4.12: *Displaying Cell Information 150*

Download at WoweBook.Com

Figure 4.13: *Response to the Drill-Down Request 156*
Figure 4.14: *Manual In-Place Editing 162*
Figure 4.15: *A DataGrid Control with the Projects Table Bound to It 163*
Figure 4.16: *Moving to a New Cell 176*

Figure 5.1: *The Sample Application, a .NET Compact Framework Version of Notepad 192*

Figure 6.1: *A Two-Layer (Connected) Approach 228*
Figure 6.2: *A Three-Layer (Disconnected) Approach 228*
Figure 6.3: *The Parent/Child Form 243*
Figure 6.4: *The Single-Item Form 244*
Figure 6.5: *Three Ways to Designate Which Row to Display 247*
Figure 6.6: *The XML Generated by Executing the WriteXml Method 253*
Figure 6.7: *A Database File Viewed in the File Explorer 256*
Figure 6.8: *A DataGrid Control with Aliased Column Names 260*
Figure 6.9: *SQL Server CE Query Analyzer, Objects Tab 261*
Figure 6.10: *SQL Server CE Query Analyzer, SQL Tab 261*
Figure 6.11: *The CreateDatabase Program 262*
Figure 6.12: *The DataReader Program 268*
Figure 6.13: *The Output from a Join Using the Alternative LoadOneProduct Method 274*
Figure 6.14: *The Category Modification Form 279*
Figure 6.15: *The DataReader Program (Repeated) 290*
Figure 6.16: *Dragging and Dropping a Connection Object 295*
Figure 6.17: *Requesting a New Connection String 296*
Figure 6.18: *Specifying Connection Information 296*
Figure 6.19: *Adding a Stored Procedure Command to a Project 300*
Figure 6.20: *Graphic Representation of a Parameters Collection 300*
Figure 6.21: *The Category Modification Form (Repeated) 304*
Figure 6.22: *Adding a New Typed Data Set 311*
Figure 6.23: *Specifying the Data Connection 312*
Figure 6.24: *Coding the Query 312*
Figure 6.25: *The Resulting DataTable and Its TableAdapter 313*
Figure 6.26: *Two DataTables and Their DataRelation 313*
Figure 6.27: *The Five Tables of Application Data 314*
Figure 6.28: *A Display of One Employee's Orders 315*
Figure 6.29: *An XML Document 319*
Figure 6.30: *The XSD Schema for the Document in Figure 6.29 320*
Figure 6.31: *Creating a New Web Service 321*
Figure 6.32: *The Primes Web Services Page in the Browser 324*
Figure 6.33: *Testing the Web Services Page's Primes Class in the Browser 326*
Figure 6.34: *The Structure of a SOAP Message Request 326*
Figure 6.35: *The SOAP Message Returned by the GetFloorPrime Method Call 327*
Figure 6.36: *The SOAP Message Representation of a Data Set 330*
Figure 6.37: *Using the Primes Service 331*

Download at WoweBook.Com

Figure 6.38: *Using the DataSupplier Service 332*
Figure 6.39: *The Add Web Reference Dialog Box 333*
Figure 6.40: *The Renamed Reference Entry 334*

Figure 7.1: *Sample Application Schematic Diagram 346*
Figure 7.2: *The Application's Data 348*
Figure 7.3: *Sample Application Diagram: LINQ to Datasets 348*
Figure 7.4: *Structure of Data Set 350*
Figure 7.5: *Results from LINQ Query Displayed on Smart-Device Screen 352*
Figure 7.6: *Display of Unioned Lists 355*
Figure 7.7: *Sample Application Diagram: LINQ to Objects 356*
Figure 7.8: *The Five Business Objects 360*
Figure 7.9: *Structure of the Application Data Object 362*
Figure 7.10: *Display of LINQ Assembled Data 372*
Figure 7.11: *Sample Application Diagram: Hybrid Version 376*
Figure 7.12: *XML Generated from LINQ to Objects Query 382*

Figure 8.1: *The Five Components of Remote Data Connectivity 390*
Figure 8.2: *After Server-Side Installation of SQL Server CE 394*
Figure 8.3: *Launching the Virtual Directory Creation Wizard 394*
Figure 8.4: *Configuration Test Showing Successful Site Installation 396*
Figure 8.5: *A Table Pulled with Tracking and Indexes 403*
Figure 8.6: *The Categories Table after Updates 404*
Figure 8.7: *Remote Data before the Push 406*
Figure 8.8: *Remote Data after the Push 406*
Figure 8.9: *The Push Method Error Message 407*
Figure 8.10: *The ErrorCategories Table after the Push 407*
Figure 8.11: *Tables to Be Included in the Sample Publication 412*
Figure 8.12: *The Merge Replication Sample Application 417*
Figure 8.13: *Objects after Synchronization 421*
Figure 8.14: *Constraints after Synchronization 422*
Figure 8.15: *Order Detail Row Added for Merge Replication 423*
Figure 8.16: *Order Detail Row Viewed at the Server 423*
Figure 8.17: *Server and Client Projects in the Same Solution 428*
Figure 8.18: *Adding a Local Data Cache Item 429*
Figure 8.19: *Configuring Data Synchronization 430*
Figure 8.20: *Specifying Tables 431*
Figure 8.21: *Providing a Client-Side Data Set 431*
Figure 8.22: *Viewing the Designer-Added Components 432*
Figure 8.23: *The Test Client Program Displayed 435*
Figure 8.24: *Setting the Web Reference 436*
Figure 8.25: *Solution Explorer Showing Generated Proxy Files 437*
Figure 8.26: *The Main Form and Its Controls 438*
Figure 8.27: *The Device Synced with the Server 442*
Figure 8.28: *Pending Client-Side Changes 442*
Figure 8.29: *Pending Server-Side Changes 443*

Download at WoweBook.Com

Figure 8.30: *Viewing the Nonconflicting Results 443*
Figure 8.31: *Modifying the Two Rows 444*
Figure 8.32: *Viewing the Conflicting Results 444*
Figure 8.33: *Displaying the Failed Row 446*
Figure 8.34: *The Smart Device Application 447*
Figure 8.35: *The Data Synchronization Service 448*

Figure 9.1: *FindPrograms, Which Uses Remote File Functions to Locate Executable Programs 473*
Figure 9.2: *Remote Registry Editor Viewing the Contents of a Windows Mobile Registry 487*
Figure 9.3: *RegShowStartup with the Startup List from a Windows Mobile Device 491*
Figure 9.4: *ShowDatabases with Names of Databases Available on a Currently Connected Device 496*

Figure 10.1: *The WCF Test Client Running the Default WCF Service Library 525*
Figure 10.2: *WCF Service Configuration Editor 526*
Figure 10.3: *ReadSmartMeter Displaying Customer Cost and Usage Details 543*

Figure 11.1: *A Stretchable Rubber Rectangle Created in the DrawRectangles Program 569*
Figure 11.2: *The Settings to Turn CUP.BMP into an Embedded Bitmap Resource 587*
Figure 11.3: *Four Calls to the DrawImage Method, Three with Transparency Enabled 594*
Figure 11.4: *ShowBitmap Displaying a JPEG File Scaled to 50% 596*
Figure 11.5: *ShowBitmap Displaying a Bitmap from a Resource 596*
Figure 11.6: *New Game Dialog Box for the JaspersDots Program 606*
Figure 11.7: *JaspersDots with a Game under Way 609*

Figure 12.1: *Output of the SimpleDrawString Program 624*
Figure 12.2: *Selecting Control Fonts in the Visual Studio Forms Designer 626*
Figure 12.3: *Output of FontPicker Program with a Sample of an Available Font 628*
Figure 12.4: *A Single Line of Rotated Text 629*
Figure 12.5: *A Fan Blade of Rotated Text 629*
Figure 12.6: *MeasureString Illustrating Exactly What Is Being Measured 634*
Figure 12.7: *Default Text Alignment 636*
Figure 12.8: *One of Nine Possible Text Alignments 637*

Figure A.1: *Hungarian Notation Helping IntelliSense Remind You of Variable Names 648*

Download at WoweBook.Com

Tables

Table 1.1: *Comparison between a Windows Mobile Phone and a Windows Desktop PC 2*
Table 1.2: *The Universe of Mobile and Embedded APIs 5*
Table 1.3: *Possible Interactions between Code and Data 24*

Table 2.1: *Power Consumption of Four Mobile Phones (Sorted by Battery Capacity) 44*
Table 2.2: *Standby Power Usage for HTC Kaiser 53*
Table 2.3: *HTC Kaiser Average Backlight Power Consumption 55*
Table 2.4: *Baseline Backlight Power Consumption (Rounded to the Nearest 10 mW) 56*
Table 2.5: *Talk-Time Power Consumption 57*
Table 2.6: *Power Consumption of Multimedia Applications/Devices 58*

Table 3.1: *Differences between Managed and Unmanaged Code 60*
Table 3.2: *Examples of When to Use P/Invoke for Features Not Implemented in Managed Code 61*
Table 3.3: *Built-in Support for Marshaling Common .NET Data Types 78*
Table 3.4: *.NET Compact Framework P/Invoke Support for Arrays 91*
Table 3.5: *Marshal Class Members Supported by the Compact Framework 101*
Table 3.6: *Design-Time Steps and Runtime Steps for Communicating between Managed and Native Code Using the MessageWindow Class 112*

Table 4.1: *Controls That Support Complex Data Binding 124*
Table 4.2: *Control Properties Used with Complex Data Binding 124*
Table 4.3: *The PMEs of a Binding Object 127*
Table 4.4: *Support for Properties Inherited from Control 132*
Table 4.5: *Support for Methods Inherited from Control 134*
Table 4.6: *Support for Events Inherited from Control 135*
Table 4.7: *PMEs Specific to the DataGrid Control 136*

Download at WoweBook.Com

Table 4.8: *PMEs for the DataGridTableStyle Object 137*
Table 4.9: *PMEs for the DataGridColumnStyle Object 137*
Table 4.10: *Types of Regions of the DataGrid Control 144*
Table 4.11: *PMEs of the HitTest Object 145*
Table 4.12: *Obtaining DataGrid and Bound DataTable Information 147*

Table 5.1: *File Manipulation Classes 184*
Table 5.2: *Byte-Level I/O Classes (Streams) 186*
Table 5.3: *Data-Specific I/O Classes (Higher-Level Readers/Writers) 187*
Table 5.4: *The EncodingClass Properties and Methods 190*
Table 5.5: *Common Encodings 191*
Table 5.6: *Registry Methods for Creating or Opening Registry Keys 214*

Table 6.1: *Persistent Storage Options Available on Windows CE–Powered Devices 224*
Table 6.2: *Data Synchronization Options on Windows CE–Powered Devices 225*
Table 6.3: *Classes for Accessing, Maintaining, and Displaying Data 229*
Table 6.4: *Commonly Used Namespaces for Data Access 232*
Table 6.5: *Unsupported SQL Server Functionality in SQL Server CE 257*
Table 6.6: *Supported SQL Server Functionality in SQL Server CE 258*
Table 6.7: *The Execute Methods of the SqlCeCommand Class 264*

Table 7.1: *Five Northwind Tables Used by the LINQ Sample Application 345*
Table 7.2: *State Table for Our Business Objects 364*
Table 7.3: *XML Support in .NET 2.0 378*

Table 8.1: *Methods and Properties of the SqlCeRemoteDataAccess Class 399*
Table 8.2: *Parameters of the Pull Method 401*
Table 8.3: *Impacts of Publication Design 413*
Table 8.4: *Synchronization Errors 445*

Table 9.1: *Available RAPI Functions 453*
Table 9.2: *Comparison between Win32 and .NET Parameter Types 481*

Table 10.1: *Transports and Sample URIs 518*
Table 10.2: *System Bindings and .NET Compact Framework Support 520*

Table 11.1: *Desktop .NET Framework Drawing Namespaces in the .NET Compact Framework 556*
Table 11.2: *System.Drawing.Graphics Methods for Text Drawing 560*
Table 11.3: *System.Drawing.Graphics Methods for Raster Drawing 560*
Table 11.4: *System.Drawing.Graphics Methods for Vector Drawing 561*
Table 11.5: *.NET Compact Framework Methods for Accessing a Graphics Object 562*
Table 11.6: *System Colors in the .NET Compact Framework 572*
Table 11.7: *Named Colors in the .NET Compact Framework 574*

Download at WoweBook.Com

Table 11.8: *Color Triplets for Common Colors 575*
Table 11.9: *Sources for Bitmaps and Associated Bitmap Class Constructors 582*
Table 11.10: *Size Comparison for Three 100 × 100 Images in Various Image File Formats 590*
Table 11.11: *System.Drawing.Graphics Methods for Vector Drawing 603*

Table 12.1: *Text-Drawing Features and Sample Programs 622*

Table A.1: *Microsoft Guidelines for .NET Naming 644*
Table A.2: *Hungarian Prefixes for Standard Value Types 649*
Table A.3: *Hungarian Prefixes for Usage and System Classes 650*
Table A.4: *Hungarian Prefixes for System.Text Classes 651*
Table A.5: *Hungarian Prefixes for System.Collection Classes 651*
Table A.6: *Hungarian Prefix for Microsoft.Win32 Class (Desktop .NET Framework Only) 651*
Table A.7: *Hungarian Prefixes for System.Windows.Forms Classes 651*
Table A.8: *Hungarian Prefixes for Microsoft.WindowsCE.Forms Classes (.NET Compact Framework Only) 653*
Table A.9: *Hungarian Prefixes for System.Drawing Classes 653*
Table A.10: *Hungarian Prefixes for System.Data Classes 653*
Table A.11: *Hungarian Prefixes for System.Data.SqlServerCe Classes 655*
Table A.12: *Hungarian Prefixes for System.IO Classes 655*
Table A.13: *Hungarian Prefixes for System.Threading Classes 656*
Table A.14: *Hungarian Prefixes for System.Xml Classes 656*

Table B.1: *Windows API Objects and Object Support Functions 660*

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

Foreword

WELCOME TO 2009, the year that mobile development has finally gone mainstream.

If you'd asked me back in 2003 when I thought Windows Mobile development was going to take off, I would've told you 2003. Heck, I was connecting Pocket PCs to SAP for large companies back then. Wasn't everyone else seeing what I was seeing? No. I was wrong then and I've been wrong every year this decade.

We had a powerful mobile platform, a powerful development runtime in the form of the .NET Compact Framework, powerful tools in the form of Visual Studio, and the backing of the world's largest software company. We also had an ecosystem of forward-thinking Mobile and Embedded MVPs to help evangelize the technology while keeping the pressure on Microsoft. As a long-time serial entrepreneur, I should've known better that you can't turn conservative companies into early adopters overnight. Sometimes having the first-mover advantage like the Pocket PC and early Smartphones isn't necessarily an advantage at all. We were all just early.

Apple was really late to the party. The iPhone was so late that it leapfrogged everyone else and was an overnight success. With its stunning UI, desktop browser, iPod music, and cool games, it became an indispensable "must-have." Platforms like the Palm, Windows Mobile, and the Blackberry pioneered this space, but the iPhone changed the game. As Apple released its SDK, its Objective C programming language, and an on-device App Store, I watched in amazement as developers came out of the woodwork to sell their wares, not unlike what I had done years before on Handango.

Download at WoweBook.Com

Luckily, this rising tide has lifted all boats. In fact, Windows Mobile outsold the iPhone last year, and we're turning up the heat this year with v6.5. A touchable, responsive, translucent UI supports panning and flicking to navigate with or without a stylus. We're back in the browser game big time with the inclusion of Internet Explorer from the desktop with the super-fast IE8 JavaScript engine for blazing AJAX performance for our mobile Web and widget developers. Microsoft My Phone lets you sync your photos, text messages, music, contacts, and much more with the cloud so that a lost phone doesn't mean the end of the world for your data. Last and certainly not least, Marketplace allows Windows Mobile developers to start selling their managed and native apps directly from the phone while keeping 70 percent of the revenue.

Now is the time to be a Windows Mobile developer.

In this book, two of the world's top developers are going to walk you through the most powerful mobile development runtime, the .NET Compact Framework 3.5. Combined with the productivity of Visual Studio 2008, there's no faster way for you to build the next killer mobile app and get it into the new marketplace where it will be seen by millions. Whether you want to build a line of business application for large enterprises or a fast-paced game for consumers, Paul Yao and David Durant will show you how to get it done in a way that's easy to understand.

—Rob Tiffany
Mobility Architect, Microsoft
http://blogs.msdn.com/robtiffany

Download at WoweBook.Com

Preface

WE FEEL PRETTY FORTUNATE. During the past three decades, we have traveled the world teaching programmers what we know. Between the two of us, we have led hundreds of classes and taught thousands of professional programmers. We enjoy working with the people who are inventing the future, and we have been fortunate enough to have been at the cutting edge of several waves of innovation in software development technology. We have learned much from our students; this book is one way to thank them.

We worked together on the first book published on the subject of Windows programming, *Programmer's Guide to Windows* (Sybex, 1987). Long out of print, in its day the book helped many programmers tackle the challenges presented by Windows Version 1.01. That version of Windows came out in November 1985. In those days, developers used computers running MS-DOS with no hard drive and no network.

Things have changed a lot since then. Today's pocket-size computers have more RAM and CPU power than a 1980s desktop system. Today's desktop systems have more computing power than the 1980s mainframes. Along the way, development tools have improved: online programming references, context-sensitive help, and graphical editors. As an industry, we are much more productive today than we were back then.

While tools have improved, so has the complexity of programming interfaces. The .NET Framework and its derivatives provide Microsoft's best-organized interface library; but it is still very large and very intricate. With enough time, most programmers can master these intricacies. But time is often a programmer's least available resource. Who has enough

Download at WoweBook.Com

time to learn about a new technology while building new software? Yet that is what is needed.

Our primary mission is to support you, the software engineer, in saving time. This book distills many years of research and sample code to give you the information you need in a way you can use. We do the same thing in our training classes, in our magazine articles, and in the talks we give at conferences. We concentrate our efforts on helping software engineers become more productive. Time saved in learning about software development issues can be focused on meeting the specific needs of the end-users whom you are supporting (or, perhaps, on taking some time off between projects to recharge your mental and physical batteries).

What You Need to Use This Book

To make the most of what this book has to offer, you are going to need a few things, as described in the following subsections.

Hardware

Software developers have historically had the fastest and most capable computer systems they could get their hands on. Developing for smart devices such as the Pocket PC and the Smartphone is no different. To get started, we recommend you have the following:

- Desktop system compatible with Microsoft Windows XP/Vista/Windows 7
- Minimum 1.6 GHz CPU (2.2 GHz recommended)
- Minimum 384 MB of RAM (1,024 MB recommended)
- Minimum 50 GB hard drive (200 GB recommended)
- Windows Mobile–powered device, such as a Windows Mobile Standard Edition device (Smartphone) or a Windows Mobile Professional Edition device (Pocket PC)

Strictly speaking, you do not need a Windows Mobile device because you can run your software on an emulator that runs on your development system. You will, however, eventually want to test your software on real devices, the same ones you expect your users to use. The emulator technology is very good—more than good, in fact. Today's emulators provide an excellent simulation of the software environment found on a device. But there are still important differences—mostly in the hardware

Download at WoweBook.Com

and associated device drivers—and all of these differences mean that an emulator should not be the only test device that you use for your software. We recommend testing your software on real Windows Mobile hardware—a superset of the various devices that you expect your users to have.

Software

The development tools require Microsoft Windows, including all 32-bit versions after Microsoft Windows XP. Support for 64-bit development systems is available for Windows Vista and also for Windows Mobile 6 and later. This means you want one of the following:

- Microsoft Windows XP Professional with Service Pack 2 or later
- Microsoft Windows Server 2003 with Service Pack 1 or later
- Microsoft Windows Vista with Service Pack 1 (32-bit or 64-bit)
- Microsoft Windows Server 2008 (64-bit); you need to install the Desktop Experience feature as well as the Microsoft Windows Mobile Device Center
- Microsoft Windows 7

With the right operating system in place, you can then use the software development tools. The first item in the following list is required; the other items are "nice-to-have" tools:

- Microsoft Visual Studio 2008 (required)
- Windows Mobile Software Development Kit (SDK); download new SDKs when Microsoft introduces new Windows Mobile devices
- Power Toys for .NET Compact Framework 3.5, needed to build WCF clients (see Chapter 10)
- P/Invoke Wizard, available from The Paul Yao Company, for help in creating declarations needed to call native code from the .NET Compact Framework (download a demo from www.paulyao.com/pinvoke)

.NET Compact Framework Versions

Windows Mobile devices today ship with Version 2 of the .NET Compact Framework. But the latest version—and the one we used for this book—is Version 3.5. Which one should you target?

Download at WoweBook.Com

Two factors will influence your choice: your target audience (or target market) and what features you need in your software. In the first edition of this book, we enthusiastically recommended that developers target the latest (highest-numbered) version. With new technology, it makes sense to ride the wave of the latest and greatest. In those days, everyone needed to download and install the .NET Compact Framework to make use of it.

But things have changed—and so has our recommendation. In general, you are going to be better off targeting Version 2.0 of the .NET Compact Framework, since that is the one that is shipping on more devices today. All other things being equal, then, be conservative in your choice of what version to target.

There are, however, lots of great new things in Version 3.5 of the .NET Compact Framework. If you need to use *L*anguage *IN*tegrated *Q*uery (LINQ), you need Version 3.5. The same is true if you want to use the Windows Communication Foundation (WCF) in your application development. Those are the two biggest changes; this book has a chapter for each of these important topics. Obviously, if you need something in the newer version, that is what you should target.

The Sample Code

Download the book's sample code here: www.paulyao.com/cfbook/code.

When you install the sample code directory tree from the Web site, you will see three top-level directories.

- `.\CS` contains all the C# samples.
- `.\CPP` contains C/C++ samples.
- `.\Tools` contains binaries of useful tools.

Each .NET Compact Framework sample is available in two languages: C# and Visual Basic .NET. Some samples are written in C++, using the Windows API (also known as Win32).

Within the two samples directories (`.\CS` and `.\CPP`) you will find a directory for each chapter. Within each chapter directory you will find another set of directories for all the samples in that chapter.

Download at WoweBook.Com

The Target Audience for This Book

For the first edition of this book, we assumed no prior .NET programming experience. Four years later, we assume some experience with .NET programming on desktop or server systems. Perhaps you have built thick client applications for desktop systems, or—more likely—have dug into building .NET-based Web sites. Whichever is the case, you have invested in .NET programming and are ready for more advanced topics.

If you have experienced writing programs that use the .NET Framework, you are going to find much that is familiar. The C# language, for one thing, has the same syntax in Windows Mobile as on desktop .NET development. The fundamental data types that support interoperability among different languages on the desktop also play a core part of interoperability with smart-device programs.

One thing that may be surprising to desktop .NET Framework developers is the extent to which they might need to rely on P/Invoke support to call the underlying Win32 API functions that Windows CE supports. While the desktop .NET Framework provides an extensive set of classes that minimizes the need to call outside the framework, the .NET Compact Framework provides a reduced set of classes to meet the size constraints of mobile and embedded devices.

To help programmers move from the .NET Framework to the .NET Compact Framework, throughout the book we provide some detailed discussions of differences between the two frameworks. In the many workshops we have taught, we have observed the knowledge and skills of programmers who are experienced with the .NET Framework transfer quite readily to the .NET Compact Framework.

The primary challenge comes from an experience we refer to as "stubbing your toe"—tripping over a familiar desktop feature (whether a class, an enumeration, an attribute, or an operating system feature) that is not present on smart mobile devices. When this happens, you have found a limit on the support available in the .NET Compact Framework (or, perhaps, a limit on the support of the underlying Windows CE operating system). The attitude you take in dealing with such experiences will play a big role in determining how successful you are in .NET Compact Framework programming—and how enjoyable you will find it. We have observed that the programmers who excel with device development are the ones who are able to see in these limitations an enticing challenge and an opportunity to explore new ways to solve old problems.

Download at WoweBook.Com

We extend our very best wishes to you on your .NET Compact Framework development, whether for the Pocket PC, the Smartphone, or some other Windows CE–powered smart device. We look forward to seeing you in one of our workshops or at an industry conference or trading comments with you online (contact us via e-mail at training_info@paulyao.com).

—Paul Yao, Seattle, Washington
David Durant, Peachland, North Carolina
July 2009

Download at WoweBook.Com

Acknowledgments

ANY LARGE PROJECT—and a book is nothing if not a large project—takes a lot of time to do right. What makes it worthwhile is the support of the people you really care about.

Paul Yao wants to thank his son, Jasper, who provided the time and space necessary to work on the book and also the fun when Dad needed time off from writing and coding.

David Durant would like to thank his wife, Betty, who endured it all while providing tranquility throughout.

At Addison-Wesley, thanks are due to Martin Heller, who first contacted us about this project and believed in us enough to help push for the insanity of attempting to write two books as part of one writing project. Thank you, Martin; we owe you another lunch or two on Mulberry Street!

We also thank Joan Murray, the acquisitions editor for this project, for her work on getting approval for this project and for coordinating the many players that support the creation of a new manuscript. Thanks also to Olivia Basegio, editorial assistant. And thanks to Susan Zahn for help in chapter formatting.

Thanks also to the current (and past) board members of the Windows Embedded Developers Interest Group (WE-DIG). Current and past board members include Joe Broxson, Jon Fincher, Mike Hall, Martin Hernandez, Constanze Kratel, Mike Salamone, Gabriel Spil, and Ronald Vanderputten. This group has helped us stay in touch with the latest in developments in all things related to Windows Embedded.

Download at WoweBook.Com

At Microsoft, we got a lot of help and technical support. On the .NET Compact Framework team, thanks are due to Seth Demsey for prompt and friendly support for what probably seemed like an endless stream of questions. We also wish to thank Craig Neable, Jamie De Guerre, and Ben Albahari, who are also on the .NET Compact Framework team. In the SQL Server CE group, we thank Kevin Collins for help on the *MSDN Magazine* article, which serves as the foundation for Chapters 6 and 8. On the Visual Studio team, thanks are due to Amit Chopra, Anand Balachandran, Mark Cliggett, Benjamin Wulfe, and Ritchie Hughes.

A big word of thanks to the extended Windows CE core team, which has always been quick to provide help and support when we had questions, comments, or just needed someone to buy us a cup of coffee. Our thanks go to Todd Brix, Karen Carter, Jason Demeny, Mike Hall, Scott Horn, Jeana Jorgensen, Jay Loney, Mark Miller, Mark Mullen, Chip Schnarel, and Mike Thomson. Thanks also are due to the MVP program for the help that MVPs have provided in keeping us in touch with the inner workings of the Microsoft mother ship, and to the uber-coordindator of all things MVP, Michael Fosmire.

Outside Microsoft, thanks are due to the industry experts on Windows CE who have been so generous with their tips, insights, long-term prognostication, senses of humor, and errant puns. Thanks are due to Doug Boling, Ginny Caughey, Neil Cowburn, Nat Frampton, Bill Mar, Steve Maillet, Chris Muench, Michael Salamone, and Chris Tacke.

We wish to thank Ronald Vanderputten at Intermec Corporation for his friendship and support as well as helpful information on power management issues addressed in Chapter 2. Also at Intermec Corporation, thanks to Andy Harding for contributions to Chapter 5 content.

Perhaps not many readers are aware of this, but one reason Addison-Wesley books have such top-notch technical content is due to the extensive technical review each book receives prior to publication. This involves hiring experts on each subject to read chapters and to submit critiques to the authors. We found this a very valuable process, and we wish to thank the reviewers who participated in reviewing the first edition of our book: Darrin Bishop, Bill Draper, Alex Feinman, Fabio Ferracchiata, Gerard Frantz, Greg Hack, Doug Holland, Andrew Krowczyk, Ivo Salmre, Jonathan Wells, and Michael Yuan. On the second edition, we wish to thank Rich Crane, Mike Erickson, Jason Estes, Troy Magennis, Brian Noyes, and Rebecca Riordan.

Download at WoweBook.Com

In addition to the technical review of the text that Addison-Wesley conducted, we made all the chapters of the book available at The Paul Yao Company Web site while we were writing the book. We were inspired by Bruce Eckel, who pioneered the idea of open reviews for programmer books. We did this to solicit comments and suggestions and were very pleased with the response we got. Among the reviewers who participated in this effort, we would like to especially thank Richard Rosenheim for his many detailed comments. Thanks are also due to Andrew Byrne, Chris Craft, and Maarten Struys (eMVP) for the many suggestions they submitted to our book review Web site. The following reviewers also submitted comments and suggestions that provided us with useful insights into how we could make the text and code better: Henri Birecki, Lars Black, Jon Box, Steve Broadhead, Tim Chaffee, Christos Chorattides, Gayla R. Cocullo, Ray Collins, Mark Dale, Michael Drake-Brockman, Mehdy Draoui, Michelle Forren, David Andrew Giglio, Chris Howlett, Sasha Imamovich, Kris Luyten, Enio Ohmaye, Ian Payne, John Perkins, Bill Ritch, Gary Scheitlin, Ronald van der Putten, Tom Vande Stouwe, Asheesh Vashishtha, John Verbrugge, Paul Webster, David Welden, and Troy Wolbrink.

Thank are also due to the staff members at ISInc Training Centers of Sacramento, California, for their cooperation and support.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

About the Authors

WITH MORE THAN forty years of experience between them teaching complex programming technologies, Paul Yao and David Durant are uniquely qualified to explain the ins and outs of the .NET Compact Framework. This team wrote the first book published on the subject of Windows programming, *Programmer's Guide to Windows,* in 1987. This was the must-have book for programmers working on Windows 1.x.

Since then, Paul and David have dedicated themselves to teaching programmers how to take advantage of just about every Microsoft technology there is, including Win32, Visual Basic, ASP, ASP.NET, ADO, ADO.NET, SQL Server, SQL Server CE, the .NET Framework, Windows CE, Pocket PC, Platform Builder, and even device drivers. This team has taught thousands of programmers, written dozens of magazine articles, and spoken at countless industry forums. In short, this team has pioneered every major technology from Microsoft.

Paul Yao is a principal with Enovative Energy Solutions (www.enovativenergy.com), an energy services business that provides performance tuning services for businesses and non-profit organizations looking to optimize their energy usage while minimizing their environmental impact. He is also president of The Paul Yao Company (www.paulyao.com), which provides educational and consulting services on Microsoft Windows–based software technologies. He has written a total of six Windows programming books and lives in Seattle.

Paul is a board member of the Windows Embedded Developers Interest Group (WE-DIG). This group holds monthly meetings on

Download at WoweBook.Com

Microsoft's main campus in Redmond, Washington, on topics of interest to developers targeting Windows Embedded technologies, including Windows Mobile, Windows Embedded CE, and Windows Embedded Standard. For details, visit www.we-dig.org.

David Durant is CEO of Durant Associates and lead author of *Programmer's Guide to Windows.* David began his computer science career in 1968 as a COBOL programmer. He was a presenter at both the first ORACLE User Conference and the first Microsoft SQL Server User Conference, and he taught Windows API programming at Microsoft University, where he received the Microsoft Award for Excellence. David was an early proponent of .NET and is a frequent contributor to industry journals and forums. He is often on the road leading seminars on .NET-based technologies for corporate clients worldwide.

David lives in Peachland, North Carolina, with his wife, Betty. His primary hobby is visiting his grandchildren.

Download at WoweBook.Com

1
Mobile Phone Programming

This chapter reviews the two main Application Programming Interfaces (APIs) for Windows Mobile: Win32 and the .NET Compact Framework. Managed code memory management is explored in great detail.

THIS DEVICE FAMILY boasts many different names: "cell phone" in the United States, "mobile" in Europe, "Handy" in Germany, and "converged device" among some industry pundits. Each falls short in describing the extent of what devices in this category can do. To call them "phones" is like calling a PC a "data entry terminal." These terms capture one aspect of their respective devices; yet no single term can begin to express what is possible. Just as people are continually pushing the frontiers of PC uses, so too are new uses for cell phones continually being found. We have no doubt that some readers of this book are contemplating even more new ways to make use of such devices.

Among the many interesting qualities of mobile phones is that they are personal devices. Some might consider a laptop to be personal, since many people carry one wherever they go. And yet, far more people carry a mobile phone wherever they go. Walk down a city street, sit in a coffee shop, cruise a shopping mall; in each of these places, almost everyone has a mobile phone in hand or close by. These devices are used to communicate (by voice or text), to plan (in a calendar or task list), and to enjoy media (music, pictures, or video).

If you are new to Windows Mobile application development and are coming from the world of developing for Windows on desktop and server systems, you will encounter many familiar things. For example, the same

Download at WoweBook.Com

development tool set—Microsoft Visual Studio—is used for both Windows Mobile and Windows desktop and server application development. All of these environments have a common set of programming interfaces: the Win32 (or "native") API and the .NET (or "managed") API.

The presence of familiar elements provides a nice starting point, but obviously a mobile phone is very different from a desktop system. One has a small display screen while the other has a large screen (or multiple screens, in some cases). One is battery-powered, the other uses wall—or "mains" as the British like to call it—power. Table 1.1 summarizes some differences between a Windows Mobile phone and a typical Windows desktop PC.

A close look at Table 1.1 reveals marginal similarities but also significant qualitative and quantitative differences. For example, every Windows Mobile system has two CPUs. On the surface, this makes the CPUs on a Windows Mobile phone appear similar to the dual core CPUs that are common on desktop systems. The two CPUs on a Windows Mobile phone are not the symmetric type found on desktop systems, but each plays a rather different role in an asymmetric multiprocessing architecture. One CPU runs end-user

TABLE 1.1: Comparison between a Windows Mobile Phone and a Windows Desktop PC

Item	Windows Mobile Phone	Desktop PC
CPU	Dual core—400 MHz ARM and 200 MHz digital signal processor (DSP)	Dual core—2 x 3.1 GHz processor
CPU clock speed	400 MHz	3.1 GHz
RAM	128 MB	2 GB
Storage	64 MB + microSD memory card slot	250 GB hard drive
Power supply	5-watt-hour rechargeable battery	200 watt
Display screen (minimum pixel resolution)	Windows Mobile Standard (172 x 220) Windows Mobile Professional (240 x 320)	VGA (800 x 600) minimum
Input device	Phone keypad QWERTY keyboard (optional) Touchscreen (optional)	QWERTY keyboard Mouse

Download at WoweBook.Com

applications built for the Windows CE operating system; the second CPU controls the phone's communication functions and is typically a specialized I/O processor rather than a general-purpose CPU. Your Windows Mobile applications run on only one of the CPUs: the one running Windows CE. You must move past the apparent similarities to understand the real differences.

A Windows Mobile phone—like any mobile phone—has less of everything compared to a typical desktop (or even laptop) system. The processor that is present is slower than the processor on a desktop system. There is less program memory and less storage space. Screen size is smaller, and input devices are significantly more limited.

One area where a Windows Mobile phone is more similar to a desktop Windows system is in software. In addition to the obvious end-user similarities, such as a Start button and similarly styled Control Panel applets, the underlying programming interfaces are quite alike. And so a programmer who has learned to write .NET programs on desktop (or Web server) systems will find a lot that is familiar in the .NET Compact Framework. This fact enables programmers to make the move to Windows Mobile development more easily than if they had to learn a whole new programming model with a whole new programming interface.

That similarity contains the seeds of another potential problem. While the programming interface may be similar, the resources for a mobile phone are far less than the resources for a desktop system. It takes time for programmers to understand, for example, that an approach to solving a problem on the desktop might not scale down to a mobile phone.

You cannot, for example, put an enterprise accounting system onto a mobile phone. What you can do, however, is to use a mobile phone to collect data that will eventually end up in that same accounting system. A travel expense application, for instance, might be one part of an accounting system that would fit onto a mobile phone. Such an application might allow data entry on the road, which could be very convenient for a traveling worker. Another example is a time card application, which consultants could run on mobile phones. Such an application could capture data that can later be sent into a centralized accounting system.

This smaller-scale approach needs to be applied to individual applications. The previously mentioned travel expense application might be a good candidate for a mobile phone application. But then again, it might not be. Success depends on whether such an application will work well on devices without a mouse and QWERTY keyboard. Success also depends on

Download at WoweBook.Com

whether the data entry fields can be laid out on the very small screen real estate that is found on a typical mobile phone. There is nothing inherently "small" about a travel expense application; instead, the application design and development teams must work together to do things in a mobile phone way instead of in a desktop way.

Before you can even begin to approach these issues, you must make some decisions about the programming interface (or interfaces) that you are going to use. While the focus of this book is on the .NET Compact Framework, it would be wrong to assume that this is always the best tool for every job. In fact, some situations require an alternative, which is often the Win32 API. Selecting the right API, then, is the next topic we explore.

1.1 Selecting an Application Programming Interface

Many development teams struggle with picking an appropriate API. Given the long-term impact of this decision, it is an important issue to address. While we understand the benefits of managed code, it is not the only choice and is certainly not always the best choice for every project. There are no silver bullets,[1] either in software development or in other areas of life.

For you to pick the right tool, you must have some understanding of the available tools. For the sake of completeness, Table 1.2 summarizes the universe of mobile and embedded APIs to give you some idea of your deployment options. We have arbitrarily divided the available APIs into four categories, with one column per category. The interesting APIs for most Windows Mobile developers are in the first two columns: "Thick Client APIs" and "Internet APIs."

The items in Table 1.2 represent a broad set of Microsoft technologies that developers are likely to hear about. We provide this list, in part, to provide background for developers new to Windows Mobile. Not every technology is an exact substitute for every other one.

For building a stand-alone "thick client" application, you have two real choices: Win32 and NETCF. In the discussion that follows, we dig into the more salient features of these APIs, the strengths and weaknesses of each, and when to choose one over the other. You can get the benefits of both APIs through a managed-code feature known as P/Invoke, which allows

1. For a discussion of why there are no silver bullets for software developers, see Fred Brooks' classic work, *The Mythical Man-Month.*

Download at WoweBook.Com

TABLE 1.2: The Universe of Mobile and Embedded APIs

Thick Client APIs	Internet APIs	Managed Spinoffs	Older / Obsolete
The Win32 API (Win32)	Browser-based application in Internet Explorer Mobile	.NET Micro Framework	Microsoft Foundation Class (MFC)
.NET Compact Framework (NETCF)	Rich Internet applications (RIAs)	XNA Framework	eMbedded Visual Basic
	Live Mesh (Mesh Operating Environment, or MOE)		
	Cloud computing (Azure operating system)		

calls from .NET Compact Framework code into native DLLs (see Chapter 3, Platform Invoke).

For client/server applications running in connected scenarios, four APIs are worth considering: browser-based applications, rich Internet applications (RIAs), Live Mesh, and the Azure cloud services operating system. As of this writing, only the first item—Web browser–based applications—is fully available and deployable on today's Windows Mobile devices. For building RIAs, you have few choices today: Adobe's Flash Lite is the only mainstream product available; in addition, Microsoft has announced Windows Mobile versions of its Silverlight offering, which will bring a second RIA choice to developers. Live Mesh allows for data synchronization between desktop and Windows Mobile devices. And finally, there is the cloud computing platform known as Microsoft Azure, which was in beta as this book goes to press. While Azure is primarily a server-side technology, Windows Mobile systems can use such "cloud computing" services in the form of a Web services client or through the Windows Communication Foundation (WCF) client.

We include the managed spinoffs in this list because they show Microsoft's commitment to managed code. The .NET Micro Framework is a single-threaded embedded operating system that runs entirely on managed code. The best-known products associated with this runtime are the SPOT (Smart Personal Objects Technology) watches, which were first available in early 2004. While SPOT watches have been discontinued, the

Download at WoweBook.Com

underlying tiny operating system is available for developers who want to build embedded systems with an even smaller footprint than that targeted by Windows Embedded CE.

The XNA Framework is a managed-code API for game development. XNA was derived from an earlier version of the .NET Compact Framework. As of this writing, Microsoft has shipped Version 3.0 of XNA along with an updated version of the XNA Game Studio (Visual Studio with XNA plug-ins). While XNA is not yet able to target Windows Mobile, the list of devices that can be targeted from a common code base includes Windows XP, Windows Vista, the Xbox 360, and the Microsoft Zune media player.

We include two old and obsolete technologies: MFC applications written in C++ and eMbedded Visual Basic applications. Microsoft is not planning any new investments for either tool, which makes them poor candidates for new projects. They are worth mentioning because they were supported in earlier versions of Windows Mobile and you may encounter them in discussion groups or read about them in older blog postings.

For some readers, you may have inherited Windows Mobile projects that use one of these older APIs. While we discourage their use for new projects, we also do not suggest that you automatically get rid of code written with these APIs. Since there is always a cost to developing—or even porting—code, you want to consider all of your options when considering how to support any such code that you may be responsible for. For our thoughts on this subject, see the sidebar, To Rewrite or Not to Rewrite.

To Rewrite or Not to Rewrite

When should you rewrite older code? This issue is especially pertinent when older code was written with older, "obsolete" tools. Should you rewrite old code or should you leave it alone? Our default answer is that you should avoid rewriting existing code because of the cost to write, debug, test, and deploy code. If you have code that works, why replace it?

The work of everyone in a business—from the president to everyone in the engineering department—should be focused on keeping the end-user happy. We are fond of quoting Jeff Bezos, founder of Amazon, when he says the end-user experience matters the most. You have to ask yourself whether rewriting code is more important than doing something else that will materially impact your end-users. Maybe the two are one in the same, which increases your ability to justify a rewrite.

Download at WoweBook.Com

Remember that your job is to spend your time and effort on making a positive impact on your business, and not to undertake technology projects for the sake of technology, or even for the sake of your own curiosity. Thriving businesses are built on keeping the customer happy. To do anything else threatens the health of a business, which directly threatens your prospects for long-term employment.

This question—to rewrite or not to rewrite—is more a business question than a technology question. Prioritize your engineering requirements and do what is important to your customers. As Tom Peters suggests in his book *In Search of Excellence,* be sure to talk to your customers.
Here are some guidelines to help you make a good decision.

- **Don't replace working code.** Is this obvious? We think so. And yet, we hear stories about developers who have working code that they wish to replace. Among the bad reasons we have heard is replacing native code with managed code because managed code is somehow better. We recommend against this approach. Existing native code that works is always better than new managed code that has yet to be written.
- **Don't replace code just because you didn't write it.** Developers love to write new code, and enjoy less having to maintain code written by someone else. Are you suffering from the Not-Invented-Here (NIH) syndrome? Are you new to a project and find that the original author is no longer on the project? Maybe you should talk to the original author to figure out what he was trying to do rather than just rewrite his code.
- **Have a good reason to rewrite code.** Good reasons include adding new capabilities, providing better performance, and offloading work previously done to the operating system or into hardware. Bad reasons include "because managed code is better than native code" and "because we want to learn about this new API."

The developers of Microsoft's Office Suite (Word, Excel, etc.) have a very conservative rewrite policy. Word—which was first released in 1988—contains C code that is 20 years old. And this is for a business unit that sells more than $10 billion in product every year.

We did hear of one effort to push managed code that backfired. At Microsoft, there was a push to put as much managed code into Windows Vista as possible. This, more than anything else, contributed to the delays in shipping Vista. And here is the kicker: Owing to performance hits in managed code, most of the Vista managed code had to be replaced with native code.

Continues

Download at WoweBook.Com

There are always hidden costs when you port existing code. There are no magic bullets and there is no free lunch. When you decide to rewrite existing code, make sure you have good reasons for doing so.

1.1.1 The Win32 API

The full name for the Win32 API is "Windows 32-Bit API." Code written to this API is also called *native code* and *unmanaged code.* The latter term is somewhat pejorative, implying perhaps *unmanageable.* This term is often used to contrast Win32 code with .NET code, which is typically known as *managed code.* The implication is that .NET code provides greater manageability. One way this statement is true is because .NET code provides automatic memory management, which we cover in more detail later in this chapter.

The Win32 API forms the core and the foundation for Windows Mobile (and for Windows Embedded CE). Most of the core features of the .NET Compact Framework come from features provided by the Win32 API. This is certainly true at the lowest (kernel) level of the application stack. Higher up the application stack, the .NET Compact Framework does add quite a bit of its own "secret sauce," which we describe later in this chapter in the discussion of NETCF.

Knowledge of the Win32 API is important, even for developers targeting the Compact Framework. There will almost always be a situation when a Compact Framework program needs to access a feature that is present in the underlying operating system but is not supported by the Compact Framework. In such situations, developers will need to use a feature of the Compact Framework known an Platform Invoke (P/Invoke) to call the Win32 functions in the underlying DLLs (see Chapter 3, Platform Invoke).

Microsoft did not invent Win32 for its mobile and embedded platforms, but rather adopted it from the core API used on its desktop and server operating systems. One of the strengths of Microsoft's mobile and embedded strategy is that much is borrowed from desktop technologies, which means programmers—and some code—can transition from the desktop and server systems onto projects that use Microsoft's mobile and embedded offerings.

One difference from the desktop is that on Windows CE, the Win32 API uses only the Unicode character set. Another difference is that only a subset of the APIs on the desktop is available—primarily, those APIs that support client-side features. For example, Windows Mobile has a Web

Download at WoweBook.Com

browser, but no Web server is supported. Among the common APIs, the mobile and embedded platforms generally have fewer functions than are found on the desktop. It is very common for a programmer to get tripped up by the situation caused when a favorite function is not supported; we refer to this as "stubbing your toe." It does not kill you, but it is unpleasant. When that occurs, we encourage you to persist in searching for a suitable substitute. Even when one function is not available, there are often similar suitable functions that can serve as replacements. (This is not *always* the case, however, but it is the case often enough.) The reason that functions are missing is a good one: With less storage space and fewer RAM resources, Microsoft eliminates redundancy to help shrink the footprint of the operating system image. This is an important advantage when building mobile and embedded operating systems, and is one reason that mobile phones are not built using desktop operating systems.

1.1.1.1 *Win32 Strengths*

The major strength of the Win32 API is that it can be used to create the smallest and fastest executables. Some might dispute the issue of "smallest," since Compact Framework applications can be pretty small. But when you add in the Compact Framework runtime (5 MB as bundled on an HTC Kaiser), Win32 easily wins the "smallest" category. Since Version 5.0, Windows Mobile has come preloaded with a version of the Compact Framework. Whether preloaded or not, the runtime does occupy memory, and the smaller memory use is still an important Win32 benefit worth keeping in mind.

Win32 creates the fastest running executables. We make this claim for several reasons. One is that Win32 executables are built for the instruction set of a given platform's CPU. All Windows Mobile platforms have an ARM processor, and Visual Studio builds executables using the ARM4 instruction set. NETCF executables, on the other hand, contain instructions from the Microsoft Intermediate Language (MSIL) instruction set. No CPU currently runs the MSIL instruction set. Instead, MSIL is converted in a just-in-time (JITted) fashion when an executable is run.[2] This is a fairly fast operation, but the cost is more than zero—it does take time. This cost becomes less important for code that is run repeatedly, since once the MSIL-to-native conversion is done, that code can be reused as long as it resides in RAM.

2. On the desktop, the performance of managed code can be improved by using the Native Image Generator (ngen.exe) tool. This tool is not supported in the Compact Framework.

Download at WoweBook.Com

Related to the subject of execution speed is that of real-time performance. In this area, Win32 is a better choice than managed code. In comparison to managed code, Win32 code gets more consistent response time in terms of how threads are scheduled. While Compact Framework code gets the benefit of the memory manager, that support comes at a price. During garbage collection, all managed threads are frozen to prevent threads from changing system memory state while the memory manager is shuffling memory. Because the timing of garbage collection cannot be predetermined, Microsoft recommends that any real-time operations be performed on native threads rather than on managed-code threads.

Win32 code is easier to debug than managed code. On the surface, this seems like a surprising claim to make, because Visual Studio provides the same set of features for debugging both native and managed code. Where you see the difference is when a problem is caused in some underlying Win32 code, which—when debugging managed code—requires a transition through the P/Invoke layer into the underlying Win32 code. When debugging Win32 API code, you can trace virtually every line of code and see all that is happening.[3] This is not so with managed code, since the Visual Studio debugger does not support debugging through a managed-to-native transition.

And finally, there are some situations when you must use the Win32 API because that is the only way to do certain things. For our list of when Win32 is required, see the sidebar, When Win32 Is Required.

When Win32 Is Required

Some developers are such fans of managed code that they suggest managed code for every situation when Win32 is not explicitly required. While we refrain from making that strong a recommendation, here is a list to help you identify when you must use the Win32 API. Our hope is that it helps you identify those areas early so that you can include them in your planning and resource allocation. We have three categories of situations when Win32 is required: (1) low-level code, (2) operating system extensions, and (3) server-side (as opposed to client-side) code.

3. In some cases, special debuggers are needed, such as those found in the Windows Embedded CE Platform Builder tool kit, to trace into operating system code.

Download at WoweBook.Com

Low-level code: Device driver and system service code must be written using the Win32 API. One distinguishing characteristic of Windows Mobile and Windows CE is that Win32 is the only API supported for device driver development. This is in contrast to what you find on desktop operating systems such as Windows Vista, where the device driver programming interface looks nothing like the Win32 API (and even less like .NET code).

These components cannot run managed code because they are native-mode processes. Device drivers live in a native process, DEVICE.EXE, and operating system services run in another native process, SERVICES.EXE. Developers who are familiar with the full .NET Framework might notice that this feature—known as hosting of managed DLLs in a native process—is, in fact, supported on desktop and server systems. That feature is not supported in the .NET Compact Framework.

It is probably worth mentioning that some individuals are so enthusiastic about managed code that they have attempted to write managed-code device drivers for Windows CE.[a] The idea is not entirely bad, as Microsoft has created one embedded operating system—the .NET Micro Framework—in which device drivers are, in fact, written in managed code. We hope such wonders will never cease. In spite of these examples, the Win32 API is required for device drivers and operating system services in both Windows Mobile and Windows Embedded CE.

You will also want to use the Win32 API when writing any code that executes during or shortly after system boot time. You should avoid the use of managed code because of load ordering issues, meaning that you cannot be sure the components needed by the managed-code libraries will be fully initialized. If the underlying components are not up and running, there is a possibility that the managed-code libraries will fail in unexpected ways.

There are several ways to run programs at system startup. One involves a system registry entry at `HKEY_LOCAL_MACHINE\Init`. To be on the safe side, we always use native code for startup programs. This is not to say that managed code could never be used, but you would have to make sure you set up the dependencies correctly. The main issue involves making sure the operating system is able to support the runtime for the .NET Compact Framework. To avoid such problems, we suggest using only native code for startup code.

Continues

a. One person who comes to mind is Chris Tacke, a Microsoft Embedded MVP who, among his other accomplishments, worked on the Smart Device Framework (www.opennetcf.com).

Download at WoweBook.Com

The second way to run a program at system startup involves creating an entry for the system's Start menu. Just as on the desktop, you can tie a program to the system shell by creating an entry in the appropriate directory. The entry can be either an executable file or a shortcut (*.lnk) file in the `\Windows\StartUp` directory. In this case, a .NET Compact Framework program can safely be used since the operating system itself is in a stable enough state to support managed code.

Operating system extensions: Extensibility is a key tenet for all Microsoft platforms, and this includes Windows Mobile. The most obvious way to extend a platform is to create a stand-alone application program in an .exe file. In some cases, operating system executables allow third parties to provide a dynamic link library (DLL) that is loaded and run by the OS component. Here is a list of some of those DLLs. All of these must be built as native executables.

- Application setup: A setup DLL can be deployed inside a deployment (CAB) file for any application setup steps that require custom code to be executed.
- Control Panel applets: DLLs with an extension of *.CPL and that support a specific set of entry points can be used to plug into the system Control Panel.
- Custom Software Input Panel (SIP): Windows Mobile devices with touchscreens allow for the creation of custom on-screen keyboards that are enabled with a special DLL.
- User interface skin: Windows Embedded CE (but not Windows Mobile) supports customizations to the user interface look and feel through the creation of user-interface skins. Part of this implementation takes the form of native DLLs.
- "Today screen" customization: The look and behavior of the main screen of a Windows Mobile device can be customized using a home screen DLL.
- Shell extensions: In addition to the today screen, other aspects of the Windows Mobile user interface, such as the File Explorer, can be customized using a shell extension DLL.

Server-side code: In a world of client/server computing, Windows Mobile—and its underlying Windows Embedded CE operating system—were built for client-side computing. In keeping with the design of its host platforms, the Compact Framework enables the client-side part of the .NET Framework libraries. Anything that looks like a server-side feature in the full

Download at WoweBook.Com

.NET Framework was excluded from the Compact Framework. For this reason, the Compact Framework does not support any server-side capabilities. Creating the following, then, will require that you rely on the Win32 API.

- Web server: Who would have thought that Windows Mobile (or Windows Embedded CE) might be useful as a Web server? In fact, there are third-party solutions that enable this support. In addition, you could write your own Web server, but it would require quite a bit of code, including all of the network plumbing code.
- Web services server: The Compact Framework helps you create Web services clients, but has no support for building a Web services server. You must instead write your own support using the Win32 API and the native socket library.

In summary, some situations require the Win32 API including for device drivers, when extending the operating system, and to build server-side code. Otherwise, managed code suffices.

1.1.1.2 *Win32 Weaknesses*

Charles Petzold, the Windows programming author, calls Win32 the "assembly language of Windows." Like CPU-level machine language, Win32 provides a set of low-level primitives. These primitives are not built to be programmer-friendly but rather to make operating system features available to application programs. The Win32 API is notoriously quirky and inconsistent.

One of its quirks involves the lack of inherent support for object-oriented programming (OOP). Most programmers would agree that OOP is a good thing. OOP concepts of encapsulation, inheritance, and polymorphism provide structure and enhance discipline in code. While OOP is pervasive today, it was not when the first version of Windows was being built. Win32—and its Win16 predecessor—were created before OOP tools were widely available to PC programmers. That is the main reason the Win32 API is a procedural programming interface and not an object-oriented programming interface. While it might be argued that Win32 is object-based—with object creation APIs and their associated handles—the API has none of the features normally associated with OOP.

The most significant weakness of the Win32 API is that it requires programmers to be very careful when creating and destroying operating system objects. A failure in this regard creates a memory leak. A memory leak on

Download at WoweBook.Com

any system is not good. On a battery-powered device with limited memory (compared to a desktop system), a memory leak can be a very bad thing. Left alone, memory leaks may lead to system crashes and the need to reboot frequently. Tools are available to help detect and eliminate memory leaks, such as those from Entrek Software.[4] Also see Appendix B, Windows API Allocation and Cleanup Functions, for a complete list of all Win32 allocation and the associated cleanup functions.

1.1.2 The .NET Compact Framework

Microsoft first created the .NET Framework for desktop and server systems, and then built the .NET Compact Framework as a smaller, client-oriented framework. The two frameworks have a lot in common, including the supported data types as well as a common set of names for namespaces, classes, properties, methods, and events. These common elements were meant to ease the transition for programmers from desktop and server programming to device programming. The common architecture also enables code sharing between desktop and device programming.

In fact, it is quite easy to write Compact Framework code that can run on a desktop or server system. Several factors make this possible, including source-level compatibility and binary compatibility. At the level of the source code, the Compact Framework has an identical naming scheme as the full .NET Framework. Going the other way—taking a full .NET Framework and running on the .NET Compact Framework—is more challenging because the device framework supports a subset of the full framework.

To develop for both desktop and device, create separate Visual Studio projects. The Visual Studio tool set offers minimal support for the simple reason that few developers have expressed the need for this kind of development support. We expect the need to grow, especially as developers build a common code base to run on devices, on desktops, on server systems, and, eventually, on the Microsoft Azure cloud operating system.

At the level of binary compatibility, both frameworks share a common (PE) file format and a common machine-level instruction set (MSIL). This means, for example, that a single managed-code binary file can be deployed to both desktop systems (e.g., Windows Vista) as well as Windows Mobile devices. The operative word here is that it *can* be deployed, but there are obstacles to this support for binary compatibility.

4. For details on the Entrek Toolbox, visit www.entrek.com.

Download at WoweBook.Com

One obstacle involves calls from managed code to Win32, which are enabled through P/Invoke. While both desktop and device systems have support for Win32, the underlying DLLs have different names. For example, the `MessageBox` function is deployed in USER32.DLL on the desktop and in COREDLL.DLL in Windows Mobile. This is a small difference, but because the P/Invoke feature requires a hard-coded DLL name, any calls to Win32 functions break binary compatibility.

Another obstacle involves the supported set of .NET namespaces and classes. Any reference to an unsupported property, method, or event causes an exception. Therefore, it is helpful to incorporate exception handling when attempting to build binaries that are deployed on different platforms. This will allow you to identify and hopefully fix those situations where an unsupported element is referenced. One way to address this issue involves targeting the .NET Compact Framework first, since its set of supported classes, methods, and properties are smaller than those found on the desktop .NET Framework. Once your code is written, you can then target the full framework to validate that you have, in fact, built an executable file that can support binary portability. Along the way, you want to make sure to avoid calling any of the managed libraries that are specific to the .NET Compact Framework. Portable DLLs and namespaces start with "System" (e.g., `System.Windows.Forms`), while the nonportable DLLs and namespaces typically start with "Microsoft" (e.g., `Microsoft.WindowsMobile.dll`).

1.1.2.1 *Compact Framework Strengths*

When should you use the .NET Compact Framework to develop software? You should do so when you are building anything that takes advantage of the strengths of the .NET Compact Framework. Highlights of these strengths include the following.

- Data forms: The Compact Framework has a rich set of user interface controls such as labels, text boxes, menus, and so forth. To support the widest range of target devices, target the Standard devices instead of Professional devices, since Standard devices support a smaller number of controls.
- Portability: As discussed earlier, managed code enables both source-level and binary compatibility. Win32 provides source-level but not binary compatibility of executable files. In other words, binary portability allows a single managed DLL to be deployed and run on both

Download at WoweBook.Com

desktop systems and devices. Native DLLs do not allow binary portability, but source-level portability is supported. In other words, with the Win32 API you must build an x86 DLL for desktop systems and a separate, ARM DLL for devices.

- String handling: Managed code does especially well in terms of string handling. Two string classes provide a good start (`String` and `StringBuilder`). Regular expressions provide a rich set of string parsing features (see the `System.Text.RegularExpressions` namespace).
- Data handling: The Compact Framework has rich support for reading and writing relational data through the ADO.NET classes. Support includes access to data stored in a local database (SQL Mobile) as well as remote databases (SQL Server 2005/SQL Server 2008).
- XML: The Compact Framework provides a rich set of services for working with XML, such as parsing and serializing XML-formatted data.
- Web services client: As a client-based runtime library, the Compact Framework provides two different mechanisms to connect to a Web services server: the traditional .NET Web services interface as well as the newer, Windows Communication Foundation (WCF), approach.
- Web browser client: Through its support for the `WebBrowser` control, the Compact Framework supports the creation of a customized Internet browser experience.

1.1.2.2 *Compact Framework Weaknesses*

For experienced .NET Framework programmers, the biggest issue with the Compact Framework will be getting used to the subset that is supported. While this sounds simple, we use the phrase "stubbing your toe" to refer to the frustration that is typical when any familiar part of any desktop API is not present on a device. At such times, it is best to take a deep breath and remember that its compact size is part of the charm of the Compact Framework. And then, there are always other ways to accomplish most tasks. In part, the frustration comes when you think you have only one line of code left to write and then you suddenly discover that the one line you had in mind involves a method or property that is not present. And one line turns into one week of reinventing the wheel.

Less frustrating, but equally important, is the set of desktop .NET Framework features that are absent from the Compact Framework. One area that is missing is support for server-side code. For example, you cannot use the Compact Framework for an ASP.NET-style Web server. Nor,

Download at WoweBook.Com

for that matter, can you use the Compact Framework to support the server side of a Web service.

Also among the major features that are missing from the Compact Framework is the ability to host managed code within a native process. If that feature should ever be added, it will enable some interesting scenarios, such as the ability to write device drivers and services using managed code. Alas, we do not have that ability today.

Another weakness of the Compact Framework is poor support for real-time programming. We touched on this already among the strengths of the Win32 API, and refer you back to that section.

1.1.3 The Web Browser

Aside from building thick clients, another application style is represented by browser-based applications. Windows Mobile has a built-in browser. The earliest browser was known as the Pocket Internet Explorer (PIE). The latest version of this browser is currently called Internet Explorer Mobile. It provides a decent browser experience, but is not at the same level as that found on desktop systems.

From the IEMobile blog, here is a list of the standards supported in Internet Explorer Mobile:[5]

- HTML: Supported to Version 4.01
- XHTML: Basic support, to Version 1.01
- WAP: Versions 1.2.1 and 2.0
- CSS: Level 1, Level 2 (partial), Mobile Profile 1.0, Wireless (WAP) CSS 1.1
- Scripting/DOM: JScript 5.6, DOM Level 0, DOM Level 1 (partial), DOM Level 2 (partial)
- XML: MSXML3
- ActiveX: These are supported, but only those built for ARM and not for x86

What is lacking, however, is support for the following.

- VBScript: Only JavaScript is supported.
- Automatic downloading of ActiveX controls.

5. This list comes from the blog of the IEMobile team. For an update, visit http://blogs.msdn.com/iemobile/.

Download at WoweBook.Com

While no public information is available as of this writing, there is speculation that a future version of Windows Mobile will provide support for a more capable Web browser. Until that future version of Windows Mobile ships, however, you need to make your plans around what you know is available.

1.1.4 Rich Internet Applications (RIAs)

RIAs represent a new class of applications that provide some of the advantages of thick clients and Web applications with perhaps fewer of the disadvantages. This class of application is expected to grow in importance in the coming years, so regardless of whether you need to use this technology, it helps to be aware of what this is and what it can do for you.

The so-called "thick client applications" refer to Win32 or Compact Framework applications. Such applications are installed onto a given platform—whether Windows Vista or Windows Mobile—and tend to stay there until they are uninstalled. Such applications run very well by themselves, but great effort is required to update them or to provide access to outside (server-based) data sources.

Of course, quite a few technologies have been created to enable thick client applications to interact with the outside world. What we mean are technologies such as distributed COM, remote procedure calls (RPCs), and, more recently, Web services. All of these are good things, but none of these enhancements change the fact that the thick client does its best work when dealing with local hardware and the local operating system.

Enter Web applications, which reside on Web servers and run inside browsers. Web applications are quick and easy to update, which gives them an advantage over thick client applications. But Web applications are not as responsive as thick client applications, which is why some pundits suggest that "www" really stands for "world-wide wait."

There are two RIA environments worth discussing here: Adobe has Flash for the desktop and Flash Lite for mobile phones. Microsoft has Silverlight for the desktop and has announced plans for a Windows Mobile–compatible runtime. Both of these environments operate through a runtime within an ActiveX control within the Web browser. RIAs are therefore centrally deployed and easily updated, like Web applications.

Unlike Web applications, however, RIAs run within a client-side runtime engine. This architecture gives RIAs the ability to have the interactivity and graphical richness of a thick client application. RIAs are typically more portable than thick client applications, so they can be written once and deployed to support multiple client platforms. This also means RIAs

Download at WoweBook.Com

are less likely to support esoteric or unique hardware, such as a fingerprint reader or other specialized device-specific enhancements. Still, their main attraction comes from supporting a common, multiplatform API.

1.2 Memory Management

A thorough understanding of memory management is necessary to appreciate the improvements that are provided by managed-code development. When most people hear the term *managed code,* they think about automatic data cleanup. While that is an important part of the picture, memory is used for other things besides holding allocated data. In addition to data, memory is also used to hold code and to hold metadata. The memory used by managed-code programs can be divided into three parts:

1. Metadata maps allocated by the class loader for each class
2. The JITted code pool allocated by the JIT compiler for each JITted method
3. Garbage collector pools (commonly referred to as "the heap") that provide instance data for allocated objects

The Scope of Managed Memory

When we teach .NET programming, we encounter a lot of confusion about the location of the heap where instance data gets allocated and the relationship between allocated memory and a program.

All of the memory allocated for a program resides in the context of an application domain.[a] When a program terminates, all of the allocated memory is automatically freed. This includes the memory for metadata maps, JITted code, and instance data.

a. An operating system process can hold multiple application domains. By default, there is one application domain per operating system process.

1.2.1 Metadata Maps

Every .NET program and library file contains **metadata,** a dictionary of the types defined in a module.[6] At build time, the compiler creates metadata,

6. By "module" we mean an application (.exe) file or a dynamic link library (.dll) file. Some programming environments use the term to refer to a source file.

Download at WoweBook.Com

which the execution engine uses at runtime. Metadata provides a rich snapshot of all the type information in the managed-code module. Every .NET type is defined in an executable module file as metadata.

Various utilities allow you to view metadata, including ildasm.exe and the Object Browser in Visual Studio. Metadata can also be accessed programmatically using a feature of the .NET Compact Framework known as Reflection.

At runtime, objects are instantiated by using type information, and code is verified for compatibility with the supported types. When a class is encountered in managed code, the loader creates an in-memory description of the class. Among its other elements are a method table for all of the methods defined in the class, including instance and static methods.[7] Once created, a metadata map for a class stays in memory until a program terminates.

The JITted code pool contains memory allocated by the JIT compiler, a subject we discuss next.

1.2.2 The JITted Code Pool

The biggest memory savings that managed code provides over native code is one that few programmers think about, yet its impact on overall memory use is profound. We are referring to the way in which code gets loaded into memory. The "executable" part of a managed executable file consists of MSIL instructions. When loaded into memory, the MSIL instructions are converted into native machine instructions (JITted by the JIT compiler).

This represents a change over how code is loaded for native modules under Microsoft Windows. Win32 programs and DLLs are staged as memory-mapped files. Portions of Win32 modules are brought into memory by memory faults when code, data, and resources are accessed. When a particular method gets called in native code, the page with the method is loaded into memory. Except for the rarest of cases,[8] this approach brings more of the executable into memory than is needed, strictly speaking. With a granularity of 4K pages on a typical system, it is almost certain that more memory is consumed in a Win32 module than by a comparable managed-code module. Managed heaps have a granularity of 4 bytes instead of the 4K used when loading elements from a native module.

---

7. Box and Sells cover this topic in great detail in Chapter 6 of *Essential .NET,* Volume 1 (Addison-Wesley, 2003).
8. To be 100% memory efficient, a native-code method needs to meet two conditions: (1) The size must be exactly one page (1K or 4K, depending on the CPU), and (2) the method must start on a page boundary. If either of these conditions is not met, memory is wasted.

Download at WoweBook.Com

As the execution engine converts IL to native machine instructions, each JITted method gets added to a RAM-resident collection known as the JITted code pool. Like other any allocated memory, JITted code stays around as long as it is needed; in some circumstances, the garbage collector flushes the JITted code pool. Because of the profoundly negative implications for system performance, this is used only as a last resort for memory reclamation.

The key design choice that saves system memory is this: methods are JITted on an as-needed basis rather than blindly by memory page. All method calls are routed through a method table that belongs to the class that contains the methods, including static methods, constructors, destructors, and properties as well as instance methods. When the method table is first created, every entry points to a **thunk**,[9] a tiny code stub that calls the loader for that method. Thunks have been supported by every version of Windows. The first time a method is called, the thunk routes the call to the JIT engine, which responds by loading the method from the executable file and creating the native code. The JIT engine then redirects the entry in the method's table to the newly generated code. The JIT behavior of the Compact Framework runtime is different from the behavior in the full .NET Framework. Application developers do not need to do anything special to benefit from the extra care that is taken to minimize the size of the application's working set.

The **working set** of managed code loaded by the JIT engine is the set of methods that have actually been used. Once JITted, a method stays in memory in case it is needed again. Under severe memory pressure, however, the memory manager can purge JITted code to free memory for other uses. These purges can have a big (negative) impact on program performance, however, which is why the JITted code pool is the last place the memory manager looks when system memory is low.

The **granularity** of loading native-code is 4K (a page boundary); managed-code has a much finer granularity designed to load individual methods as those methods are needed. When a method is first called, the IL code from the module (.exe or .dll) file is converted into native code so that only the actual methods used by a program are JITted and loaded into memory. No extra "slop" gets added for portions of methods which may or may not ever get called.

---

9. While the term seems to have fallen out of favor with the advent of .NET, the implementers of the shared source CLI seem to like *thunk*—it appears more than 1,300 times in Rotor source code.

Download at WoweBook.Com

By contrast, when native-code methods are loaded into memory, the granularity is set by the CPU as a virtual (4K) memory page. This design means that memory is wasted when native code is loaded into memory. The greatest benefits are probably in shared libraries that contain a large number of methods, of which a small number are called from any given program. The benefits are probably least evident with small programs, when all native code might be loaded with a few load operations.

JIT Caching and the Native Image Cache

The .NET Compact Framework's caching of JITted code is significantly simpler than that done on the desktop. With its greater storage capacity, the desktop maintains the native image cache as part of the global assembly cache. Programmers can add assemblies to the disk-bound portion of the native image cache on the desktop using a tool called ngen.exe. No such tool exists for the .NET Compact Framework because there is no native image cache stored in the file system.

1.2.3 Garbage Collector Pools

Garbage collection is the reclamation of memory[10] that has been allocated and used but is no longer needed. In the .NET Compact Framework, the portion of memory from which object instance data is allocated is referred to as the garbage collector pools. Many programming environments require a programmer to explicitly release resources. The various .NET frameworks provide automatic garbage collection. Garbage collection was not invented for .NET; instead, automatic garbage collection has been available in earlier systems, including Smalltalk, Visual Basic, and Java.

Win32 programmers have always needed to pay attention to manually cleaning up system objects they allocated. Two factors make the Win32 API prone to memory leaks: (1) the lack of automatic memory management and (2) the large number of functions that allocate memory. Memory leaks are not inevitable, but they require significant diligence on the part of programmers to prevent. Several third-party tools were created to help

10. Other resources besides memory may need to be managed as well, including file handles and other operating system objects. For the purposes of this discussion, though, *memory* is a satisfactory term.

Download at WoweBook.Com

locate memory leaks. The Catching Windows CE Memory Leaks sidebar discusses a tool created for Windows CE developers.

Catching Windows CE Memory Leaks

Seattle-based Entrek Software provides a tool called CodeSnitch that detects memory leaks in code running on Windows CE systems. The tool can highlight the specific line of source code that allocated a leftover object. For details, visit www.entrek.com.

For programmers building COM/ActiveX components, the issue of memory cleanup has always centered on reference counts. The rule is simple: A component maintains a reference count; when a reference count reaches zero, a component destroys itself. In practice, an enormous amount of energy has been expended—both by component developers and by component users—getting components to shut down properly.

For .NET programmers, automatic garbage collection means better use of two scarce resources: programmer time and smart-device memory. Programmer time is saved because automatic garbage collection means rarely having to worry about object cleanup. System memory is saved because, compared with native Win32 executables, managed code is lazily allocated on an as-needed basis, and managed data is systematically tracked so that unused memory can be effectively reclaimed.

Some of the data allocated by managed code is *not* subject to automatic garbage collection. Managed code, after all, can allocate both managed data and unmanaged data. It is this second category—unmanaged data, sometimes called native data—that a managed-code programmer must manually release because there is no automatic garbage collection of native data.

1.2.4 Garbage Collection and Data

In looking at freeing memory used by data, we must address both managed code and native code. After all, .NET Compact Framework programmers probably need native objects more than when using the full .NET Framework. We find this so because the .NET Compact Framework has fewer features than the full .NET Framework. Managed code can allocate both managed data and unmanaged data. Native code, by contrast, is able

Download at WoweBook.Com

TABLE 1.3: Possible Interactions between Code and Data

	Managed Data	Native Data
Managed code	Allocate: using new operator Free: automatic for pure managed objects; other steps required for mixed managed/native objects	Allocate: use members of `Marshal` class or through native function calls (P/Invoke) Free: `Marshal` class members of native function calls
Native code	Allocate: no support Free: no support	Allocate: use native functions Free: use native functions

to allocate only unmanaged data, not managed data. Table 1.3 summarizes the possible interactions between the two types of code and data.

Managed code allocates managed data from restricted heaps controlled by the managed execution engine. When managed objects are allocated from this heap, pointers provide the reference stored in a program's variables. But these pointers are implicit pointers and cannot be accessed like pointers in C (and other languages) for pointer arithmetic. Managed data can be moved by the memory manager without your code noticing that it has been moved. And when no longer in use, managed data is reclaimed by automatic garbage collection.

When passing pointers from managed code into native code, you generally do not need to worry about the pointers being made invalidated by having the objects moved. One exception occurs when the native code holds onto a pointer for an extended period of time. To address this situation, make sure to pin the data into place so that the memory does not move and the pointer stays valid over an extended period. (For details on pinning managed objects, refer to the `GCHandle`[11] class.)

Native data can be allocated either from managed code or from native code. Such data is unmanaged because the managed execution engine does not manage it. This means that any programmer who allocates native objects must be sure to write code to release those objects. When referring to what such code does, we use the term *manual memory management,* to parallel the automatic garbage collection provided by the runtime for managed data.

11. `System.Runtime.InteropServices.GCHandle`

Download at WoweBook.Com

In this discussion of garbage collection, we start with a look at managed data: its allocation, management, and reclamation by the execution engine. We then turn our attention to unmanaged data and the native Win32 functions that allocate and free system objects.

1.2.5 Automatic Garbage Collection

One of the more convenient features of managed code is automatic garbage collection of managed data. Garbage collection is a relatively time-consuming process that the system postpones as long as possible. Normal allocation, by contrast, is very fast because the allocator does not use any kind of "best-fit" algorithm. Instead, new objects are always allocated next to the previously allocated object. If you have ever visited a large event where parking attendants direct incoming cars to the next free space, you have a good idea of how managed data is allocated. As each object is allocated, an entry is made into an *allocated object map.* This map plays a role in the first step of object cleanup, the mark and sweep phase.

Garbage collection is triggered when an allocation request causes the amount of allocated memory to exceed a certain threshold. The garbage collector starts by waiting for all managed-code threads to freeze at safe points. Threads running in native code—sometimes called *unmanaged threads*—are not frozen when the garbage collector is run, although any threads returning to managed code while the garbage collector is running are frozen.

Garbage Collection in the .NET Framework and the .NET Compact Framework

The garbage collectors in the desktop version of the .NET Framework and in the .NET Compact Framework share a number of features. Most of the behavior described here applies equally to both.

One difference is that the desktop garbage collector makes its decisions about how it operates based on how long an object has been alive. Objects are grouped by generations, and the desktop supports three generations. In the .NET Framework, the static `MaxGeneration` property in the `GC` class has a value of `2` (which means three generations are supported: `0`, `1`, and `2`). By contrast, the .NET Compact Framework does not use this method of object aging, as indicated by the value `0` for the `MaxGeneration` property in the `GC` class.

Download at WoweBook.Com

Memory reclamation is performed in a stepwise fashion, from least to most painful as far as impact on system performance is concerned. When a step causes memory use to drop below the critical threshold, garbage collection ceases. There are three phases of garbage collection.

1. Mark and sweep.
2. Compact the heap.
3. Flush the JITted code pool.

The following subsections briefly discuss each of these phases in turn.

1.2.5.1 *Phase 1: Mark and Sweep*

During this first phase of garbage collection, the garbage collector searches for objects that are unreachable. The garbage collector knows how to find all memory allocated in managed code by walking the stack of all threads, looking for references to managed objects. Some objects refer to other objects, and this continues until every stack frame for every thread in a process (or, more accurately, within an application domain) has been taken into account. Objects referenced from static variables are reachable, as are objects referenced from local variables. In addition, object references nested in other objects, in arrays, and in collections are considered reachable.

Each object that can be reached, through either a direct or an indirect reference on the stack, is reachable data. Using the set of objects in the allocated object map as a starting point, the garbage collector identifies the reachable objects and removes them from this set. Any leftover objects are unreachable. They represent memory that is occupied but not used; the memory occupied by such objects is reclaimed by the garbage collector. If enough memory is reclaimed from unreachable objects, garbage collection stops. Otherwise, it continues to the next phase.

1.2.5.2 *Phase 2: Compact the Heap*

The second phase of garbage collection is to compact the heap. In this phase, reachable objects are moved to consolidate the space that is made available by freeing objects. This step is important because the managed-code allocator allocates only from the very end of the heap. By compacting the heap, the garbage collector can reclaim memory from unreachable objects that were not located at the end of the heap.

Download at WoweBook.Com

1.2.5.3 *Phase 3: Flush the JITted Code Pool*

When all else fails, the final step taken by the garbage collector is to free memory from the JITted code pool. The manner in which memory is chosen depends on the processor because each CPU has its own JIT engine. In the JIT compiler for the StrongARM processor, the CPU for Pocket PCs, a "least recently used" algorithm helps to select which methods to discard.

1.2.6 Special Handling of Managed Data

A few special cases deserve discussion. Each causes a change to the garbage collector or to how the garbage collector deals with specific objects.

1.2.6.1 *Calling the Garbage Collector*

Portions of the garbage collector are exposed through the `GC`[12] class. By calling the `Collect` static method, for example, a program can request the garbage collector to run through its first phase of operation to reclaim memory associated with unreachable objects. While it is not a good programming practice for programs to call the garbage collector, such calls can be used for testing and development purposes. For example, by forcing garbage collection, a program could get a truer sense of the memory required when another `GC` class method, `GetTotalMemory`, is called.

Calling the garbage collector is considered a poor practice because the garbage collector is part of the runtime's automatic memory management, and it has been tuned to operate when needed. In general, you want the garbage collector to run as infrequently as possible because all managed-code threads must be frozen prior to running the garbage collector, and this uses up precious CPU cycles (which on a mobile device also drains the battery). Making frequent manual calls to the garbage collector can cause performance problems for programs.

The `WM_HIBERNATE` Message

Windows CE has its own garbage collection–like mechanism that is part of the native, unmanaged-code support libraries: the `WM_HIBERNATE` message. This message gets sent to all the top-level windows in the system when available system memory falls below a specific threshold (about 200K).

Continues

12. Fully qualified name: `System.GC`.

Download at WoweBook.Com

In spite of the name, this message has nothing to do with sleep (of either the operating system or applications), nor does it relate to power management. Instead, it has a single meaning: System memory is low. The requested response is to free whatever memory a program can afford to release. If available memory subsequently falls below 160K, the Pocket PC shell starts to send `WM_CLOSE` messages to top-level windows, asking them to close. To detect this message, a .NET Compact Framework program needs to have a way to listen to the message traffic of a visible, top-level window.

What do you do when you detect this message? To free managed memory, you can call the `GC.Collect` method. This reasonable choice *might* ease some of the operating system's memory pressure, but there is no guarantee. Instead, the garbage collector will likely free unused objects—that is, perform garbage collection—but to make more memory available to the system, something else is required. That something else involves shrinking the size of the heap itself.

The best thing a .NET Compact Framework program can do is to free any native data that it has allocated. This has a more direct impact on making more system resources available to Windows CE than calls to the managed runtime's garbage collector.

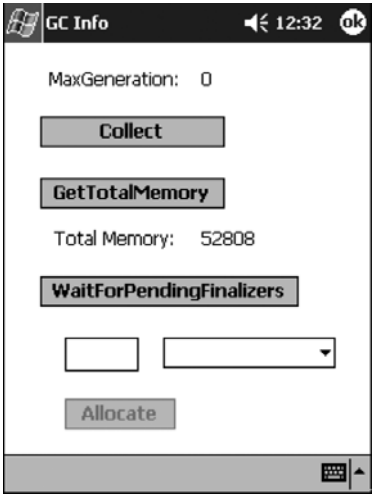


FIGURE 1.1: The `GCInfo` Sample Calls Methods in the `GC` Class

Download at WoweBook.Com

Figure 1.1 shows the output from one of this book's sample programs, `GCInfo`. This program calls various `GC` class members and also allocates and frees arrays of objects to demonstrate the effect of allocation on the memory used by a program. (This figure shows the maximum generations supported by the garbage collector—the public property `MaxGeneration`—as zero, meaning that all objects are treated as the same age.)

1.2.6.2 *Manual Cleanup Using the* `Dispose` *and* `Finalize` *Methods*

As a general rule, automatic garbage collection works well, and programmers do not need to worry about the timing or manner by which objects are cleaned up. In a few cases, however, objects are wrappers for a resource that requires either immediate cleanup or custom cleanup. This might be a file, opened in exclusive mode, that needs to be closed, or an unmanaged library that needs to free some scarce resource or free some unmanaged memory. The garbage collector is neither timely, for it normally runs only when there is a shortage of managed memory, nor aware of native (a.k.a. "unmanaged") objects, so it cannot reclaim the memory occupied by native objects.

Two methods are available for implementation by your class to help with the timely disposal of exclusive and unmanaged resources: `Dispose` and `Finalize`. The former is intended primarily for disposing the exclusive resources, such as a file that has been opened with a nonsharable setting; while the latter is intended for the disposal of native resources, specifically the freeing of native objects and the memory they hold. The two can even be used together. The `Finalize` method, if it exists, is always called by the garbage collector before it removes the object from memory; it cannot be called by the application. `Dispose`, if it exists, can be called from any code within the application that has a reference to the object; it is never called by the garbage collector.

Microsoft's definition of the `IDisposable` interface, and the subsequent implementation of the `Dispose` method by the class, enables an object to say "Application, when you are done with me, check to see whether I have implemented the `IDisposable` interface. If I have, it probably means I hold an unsharable resource that would best be released right now, rather than my continuing to hold it for the unpredictable length of time that will pass before garbage collection occurs. Call my `Dispose` method and I will release this resource, thus making it available for reuse by this application

Download at WoweBook.Com

or for use by another application. And while I'm at, I'll release any unmanaged resources that I'm holding."

Remember, the garbage collector runs when managed memory becomes scarce; it does not run because files and database connections are becoming unavailable. A relatively long time may pass between when an object becomes eligible for garbage collection and when garbage collection is actually performed. Hence the benefit of calling `Dispose`, which performs cleanup right away.

In some cases, calling `Dispose` is not provided only for the benefit of an application; sometimes calls to `Dispose` are necessary for the health of the system. For instance, forgetting to call the `Dispose` method for a `Graphics` object can cause a program to crash because all instances of a critical resource (a native device context) are busy. This happens when you create a set of `Graphics` objects by calling the `CreateGraphics` method multiple times without a corresponding call to the `Dispose` method to allow cleanup. The documentation makes it clear that you must call the `Dispose` method when you are done drawing. The `Dispose` method frees the native device context wrapped within the `Graphics` class.

Some classes implement a `Close` method in addition to, or in lieu of, a `Dispose` method. Such is the case with a variety of data reading and writing classes with files, XML, and some database classes. For these classes, no call to the `Dispose` method is needed after a call to a `Close` method.

As a rule, the `IDisposable` interface enables the class to provide a standard method that the application can call to allow a class to free critical resources. There is just one problem: What if `Dispose` is not called? That is why there is `Finalize`; the routine that is called by the garbage collector.

1.2.6.3 `Finalize`*: The Last Bastion of Defense*

As we mentioned, when implementing a class that owns a resource other than managed memory (i.e., a resource that requires custom cleanup) you can implement two methods: `Dispose` and `Finalize`. The first method, `Dispose`, provides the public method to be called from users of the class, as already discussed. For most cases, that is the only function called.

There is a potential problem, however. The presence of `Dispose` suggests that cleanup is required. But what happens if that method does not get called? What happens to the resource? The architects of the .NET garbage collector asked themselves that question as well. One approach might have been for the system to detect that the `Dispose` method had not been

Download at WoweBook.Com

called and to initiate the call itself. With that approach, however, the system has to second-guess both the class implementer and the class user.

Instead, we have the `Finalize` method, which provides a last-chance handler for cleanup right before the garbage collector frees memory. Since `Finalize` is called by the garbage collector, the concept of "timely" freeing of resources is inapplicable and even dangerous. Consider an object that holds a reference to a file that has been opened for exclusive use; a reference that has not been shared. Since the object that holds the only reference to the file object is now unreachable, the file object is also unreachable. Therefore, the garbage collector will be calling the file object's `Finalize` method, causing the file to close and to free its resources. And so there is no reason for your object to call the file object's `Dispose` method from within your object's `Finalize` method.

`Finalize` methods are called only by the garbage collector and only when the garbage collector is iterating through a list of objects that are ready to be finalized, a list whose sequence is unknown to the application. In other words, the garbage collector can remove unreachable objects in whatever sequence it chooses. Within a `Finalize` method, references to objects that have already been finalized cause an exception—a fatal error. So, a `Finalize` method must leave alone any referenced objects, on the assumption that the garbage collector has—or will—take care of cleaning up other managed objects that need to be finalized.

What the `Finalize` method must do is free any unmanaged resources; for the garbage collector is completely unaware of unmanaged resources, never frees them, and never reclaims unmanaged memory. Failure to explicitly free unmanaged resources will result in memory leaks.

One last word regarding the relationship of `Dispose` and `Finalize`, and why this relationship should cause you to add functionality to one of them. A `Finalize` method is a last-chance cleanup method when a call to a `Dispose` method is required but has not been made. If the call to `Dispose` has been made, the call to `Finalize` is not necessary. When a `Dispose` method is called, it can tell the garbage collector that the `Finalize` method does not need to be called. This is done with a static method in the `GC` class, `SuppressFinalize`. Incidentally, the `ReRegisterForFinalize` method allows a class to add back to the finalize list an object that had previously been removed by a call to the `SuppressFinalize` method.

Additionally, since `Finalize` is called by the garbage collector, you can be sure that it will be called, at most, once. The same cannot be said for `Dispose`, which is called by the application and which could be called

Download at WoweBook.Com

multiple times. When authoring the `Dispose` method, you must ensure that subsequent calls will not cause harm to occur or cause exceptions to be raised. The most common way to do this is to provide a static `Boolean` variable in which to record the fact that the routine has executed. If it has already been executed, return immediately.

Use the `Finalize` Method or the Destructor?

The name of the finalizing method for all CLI-compatible languages is `Finalize`. This standard name is critical to support interoperability between languages.

However, the C# compiler defines a different syntax for the finalizing method. The syntax matches the syntax in C++ for class destructors, which is defined with a tilde (~) followed by the class name. While this appears to have a different name from the finalizing method in other languages, the actual method name in the IL code is `Finalize`.

The C# compiler adds two other elements: a `try-finally` block and a call to the `Finalize` method in the base class. Because these are the recommended elements for a finalizing method, the C# destructor syntax provides the cleanest and easiest way to implement a finalizing method in C#.

This syntax does pose a potential problem. A C++ programmer who is used to C++ destructor syntax in native code might think the C# destructor syntax behaves in a similar fashion. It clearly does not, so C# programmers must be careful to avoid the *faux amis*[a] in the C# destructor syntax.

a. French for "false friends."

The presence of two cleanup methods adds a complication: A class might have redundant code. The suggested design pattern for manual object cleanup centralizes the cleanup work in a third method called by the other two methods. This third method, which by convention is called `Dispose`, differs from the first `Dispose` method in the number of parameters. The original `Dispose` method has no parameters. The common `Dispose` method takes one parameter: a `bool` flag that indicates which of the first two methods is calling. A value of `true` indicates that a `Dispose` method is calling to dispose of an individual object; a value of `false` indicates that a

Download at WoweBook.Com

`Finalize` method is calling, which means the process is terminating[13] and the garbage collector is calling the `Finalize` method for all objects.

This difference is sometimes important. When our common `Dispose` method is called with a value of `true`, it means a polite caller is advising us that we can clean up. At that point, we want to be sure to clean up any objects that can free up system resources so that our program can continue to run efficiently and well. In particular, we need to call `Dispose` on any objects that are contained in our object (assuming those objects require a call to `Dispose`).

When the common `Dispose` method is called with a value of `false`, it means the process is shutting down and the garbage collector is running down the list of objects that it knows about that need to have their `Finalize` methods called. The objects that are left around are the ones that are lost or forgotten, and as such the garbage collector tries to be as efficient as possible to allow the process to terminate quickly. What must be avoided is making a call to `Dispose` for any contained objects when the garbage collector is calling the `Finalize` methods. There are two reasons: (1) because the garbage collector is already going to call all such objects, and (2) if we reference an object that has already been cleaned up, we risk causing an exception in our program.

Figure 1.2 shows Disposer, a sample program we wrote to demonstrate the operation of the `Dispose` and `Finalize` methods. The five buttons are labeled with five actions that affect the memory use and cleanup of an object.

1. Create Object calls the object constructor.
2. Dispose calls the object's `Dispose` method.
3. Suppress Finalize calls the `GC.SuppressFinalize` method on the object.
4. DeReference sets the object reference to null.
5. Garbage Collect calls the `GC.Collect` method.

Listing 1.1 shows the event handlers for the five buttons in the Disposer sample program. The `ResourceWrapper` class has four methods: a constructor, two `Dispose` methods, and a `Finalize` method. Each method displays a message box when called, to show the result of the action behind each

13. Or for a process with multiple application domains, that a given application domain is shutting down.

Download at WoweBook.Com

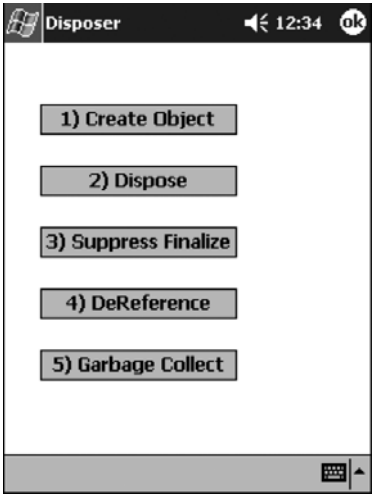


FIGURE 1.2: Disposer Demonstrating the Operation of the `Dispose` and `Finalize` Methods

button. Aside from the message boxes, this class should serve as a starting point for any resource wrapping class that you might wish to write.

LISTING 1.1: Event Handlers That Allocate and Free the `ResourceWrapper` Class

```
ResourceWrapper rw = null;

private void cmdCreate_Click(object sender, EventArgs e)
{
   rw = new ResourceWrapper();
}

private void cmdDispose_Click(object sender, EventArgs e)
{
   if(rw == null)
      MessageBox.Show("rw == null -> Exception! ");
   else
      rw.Dispose();
}

private void cmdSuppress_Click(object sender, EventArgs e)
{
   if(rw == null)
      MessageBox.Show("rw == null -> Exception! ");
   else
      GC.SuppressFinalize(rw);
}

private void cmdDereference_Click(object sender, EventArgs e)
```

Download at WoweBook.Com


```
{
   rw = null;
}

private void cmdCollect_Click(object sender, EventArgs e)
{
   GC.Collect();
}
```

You may need to change the code in Listing 1.1 to call the `Dispose` method for base classes that support a `Dispose` method. We commented out that call for our `ResourceWrapper` class because its base class, `Object`, does not have a `Dispose` method.

The `ResourceWrapper` class code shows, in addition to a constructor, the three standard cleanup methods. The declarations for these methods—two `Dispose` methods and one `Finalize` method—appear here:

```
public void Dispose()
protected void Dispose(bool bDisposing)
~<class_name>
```

Listing 1.2 shows the code for the `ResourceWrapper` class.

LISTING 1.2: The `ResourceWrapper` Class

```
// Simple resource wrapper class to demonstrate managed-code
// support for manual cleanup functions
//
public class ResourceWrapper : System.Object, IDisposable
{
   public ResourceWrapper()
   {
      MessageBox.Show("Constructor called");
   }
   ~ResourceWrapper()
   {
      MessageBox.Show("Finalize called");
      Dispose(false);
   }
   public void Dispose()
   {
      MessageBox.Show("Public Dispose called");
      Dispose(true);

      // Call Dispose when supported by base class.
      // base.Dispose();
   }
```

Download at WoweBook.Com


```
   protected void Dispose(bool bDisposing)
   {
      if (bDisposing)
         MessageBox.Show("Dispose(true) -- are disposing");
      else
         MessageBox.Show("Dispose(false) -- are finalizing");
   }
} // class ResourceWrapper
```

1.2.6.4 *Weak References*

Another special case involving the garbage collector is objects marked with weak references. Memory is a scarce resource and must be managed with care, yet some algorithms run faster when more memory is available. Weak references let you err on the side of using more memory to get better performance but still allow the garbage collector to reclaim that memory if such use causes a shortage of memory for other, more critical needs. For the sake of good performance, it is perhaps better to allow lower-priority data to be discarded than to force the garbage collector to discard code from the JITted code pool.

A weak reference provides a hint to the garbage collector that an object is easily replaceable. The garbage collector interprets as unreachable any object that is accessible only through a weak reference. The same is true for objects nested within an object that has a weak reference. Objects that are reachable *only* through weak references are subject to garbage collection. However, an object that has both a weak reference and a normal reference would not be subject to garbage collection.

Weak references are supported by the `WeakReference`[14] class. When you create a `WeakReference` object, you pass an object reference to a constructor. The object reference is available from the `WeakReference` object through the `Target` property. The Boolean `IsAlive` property returns `true` when the target still resides in memory. To access the contents of the object, copy the target to a regular variable, which then provides a strong reference to keep the object from being garbage-collected while you are using the object.

1.2.7 Manual Memory Management of Native Data

The garbage collector provides automatic reclamation of managed memory. Many managed classes are wrappers around native operating system objects and handle object cleanup. In some cases, the cleanup is automatic, while in other cases it is manual (or, at best, semiautomatic).

14. Fully qualified name: `System.WeakReference`.

Download at WoweBook.Com

But when managed code calls native operating system functions to create native operating system objects, no automatic cleanup is provided for these objects. (That is why such objects are called *unmanaged* objects.) When your program does this, your program must provide manual cleanup—that is, manual garbage collection.

Programmers working in native code must keep track of all memory that is allocated. In some cases, allocation occurs with specific memory allocation functions (such as `malloc`, `LocalAlloc`, `GlobalAlloc`, `HeapAlloc`, `VirtualAlloc`, or `MapViewOfFile`). In other cases, memory allocation is a side effect of making certain operating system calls. The creation of a window, for example, causes memory to be allocated, as does the creation of various graphic drawing objects such as pens, brushes, fonts, and bitmaps. Failure to properly track and clean up memory in native code causes memory leaks. It is possible for a memory leak to cause a program—or the operating system itself—to crash. Appendix B contains a table that summarizes the Windows CE system objects that are available through the Win32 API and the associated allocation and cleanup functions.

In most cases, .NET Compact Framework programmers do not need to allocate and free native operating system objects. Instead, it is better to rely on the .NET Compact Framework classes to deal with the details of managing such objects. This is particularly true for graphic objects, but it is also true for other types of objects. When available, we suggest you do as we do and use the managed-code equivalent for any native operating system object.

1.3 Conclusion

Mobile phones are everywhere, and they are becoming smarter, more personal, and more pervasive each day. Mobile phones are more personal than even the most personal computers, as they are carried everywhere. Your decision to build mobile software means your users can take advantage of your innovations from just about anywhere they might go.

Even though the subject of this book is writing managed code for the .NET Compact Framework, you will probably find that part of your project will involve interacting with the Win32 API as well as with the .NET Compact Framework. In some cases, a pure Win32 solution provides the only way to go. In other cases, you can combine some managed code with some Win32 code. We talk about the use of P/Invoke in Chapter 3 for accessing Win32 functions from your managed code.

Download at WoweBook.Com

One of the most important enhancements provided by the managed-code runtime involves its memory management support. By way of background, this chapter discussed how the managed runtime divides memory into three areas. One memory area holds class metadata; once loaded, class metadata stays in memory. A second memory area is the JITted code pool, which holds methods that have been converted from MSIL to the native machine instructions. The third memory area is made up of the garbage collector pools, which hold instance data for allocated objects.

The garbage collector reclaims memory from unreachable managed objects in a manner that is largely unseen. As we have shown, a program can call methods in the `GC` class to find the amount of memory being used and to force the garbage collector to perform its mark and sweep operation. While you might be tempted to call the garbage collector, in general this practice will cause your application to run more slowly. We suggest that you avoid manually triggering the garbage collector and that you trust that the managed runtime is properly tuned to reclaim memory when it needs to be reclaimed.

Classes that wrap native resources can implement two different kinds of cleanup methods: a `Dispose` method and a `Finalize` method. To help managed-code programmers understand when native operating system resources need to be cleaned up, Appendix B provides a table of native objects and details regarding the functions for allocating and freeing those objects.

Download at WoweBook.Com

2
Extending Battery Life

This chapter promotes the practice of measuring battery usage as a technique for extending battery life.

ALL SOFTWARE DEVELOPERS need to become more power-aware, whether they're targeting battery-powered systems or wall-powered systems. In November 2007, the *New York Times* reported that "energy costs can be 40 percent of the cost of operating a data center."[1] Mobile phone vendors have long known about the need to be power-smart, in part because recharging a phone is inconvenient for their users, and a dead battery means users cannot use their mobile phones and behave like good, paying customers. Mobile phone ads inevitably include talk time and standby time as important, differentiating features of these devices.

Since this book is oriented toward application developers who are building software for Windows Mobile devices, our focus is on Windows Mobile hardware. It is worth noting, however, that this approach can be—in fact, we believe it *must* be—applied to desktop and server systems as well. We do not discuss the measurement tools for such systems; we leave the task of finding appropriate tools for other kinds of systems as an exercise for the reader.

1. Wald, Matthew L. "Taming the Guzzlers That Power the World Wide Web." *New York Times,* November 7, 2007.

Download at WoweBook.Com

2.1 What Is the Problem?

Windows Mobile devices are battery-powered devices. Batteries have improved over the past decade, but those improvements do not even come close to the magnitude of improvements in semiconductor performance over the same period. Chemical batteries, sadly, have no "Moore's Law" equivalent. The irony is that semiconductor advances have led to an increasing demand for power, which must be met using only marginally better batteries. Smarter semiconductors can do more things and so are given more interesting tasks. Such tasks usually require more energy, which for portable devices must come from batteries. Both factors—the greater demands for power and more work given to mobile semiconductor–driven devices—add to the need for taking steps to extend battery life.

Application developers have long ignored battery issues. We see several reasons, the most obvious being simply because they can. Many developers started their careers building software for desktop (or other wall-powered) computers. Energy has long been inexpensive—virtually free—so few worry about it. But things are changing. Wall power is getting more expensive every day. Much (about 72% in 2007) electricity is generated by burning fossil fuels. Oil has long been cheap—in the range of $50 to $60 per barrel for the past decade; earlier this year, however, the price leapt over the (mostly symbolic) figure of $100 per barrel for the first time in history and peaked at more than $140. Fossil fuels are a nonrenewable form of energy, which means less will exist over time, so the price must continue to rise.

Another reason to ignore battery problems involves a turf issue: Whose problem is it, anyway? This line of thinking goes like this: Hardware uses batteries, and batteries themselves are hardware; therefore, battery life is a hardware problem. This line of thinking implies that the primary contributors to this cause are those who work primarily with hardware: silicon vendors, component vendors, and hardware designers. Naturally, this group can contribute, but there is a hole in the basic premise of this reasoning: Software cannot run unless hardware has power. And so, software developers have a vested interest in paying attention to this problem.

A third reason for ignoring battery issues is that it isn't at all clear what an application developer can or should do. Should applications be made smaller? Draw in darker colors? Use fewer threads? None of these specific changes are likely to help, but any application programmer who asks these questions has taken an important first step by realizing that there is something that application programmers can do.

Download at WoweBook.Com

Our basic approach is simple: To manage something, you have to measure it. There are several ways to measure battery usage, and we are going to discuss two: one software-based and the other hardware-based. The software-based approach is very easy and requires only a simple program to begin collecting data (the software is a sample for this book). The hardware-based approach is the more accurate approach, but it is also the more costly because it requires buying a dedicated piece of hardware that very precisely measures battery usage from boot-up to power-down. We expect few readers to adopt this second approach, and we will describe it for the benefit of those with more stringent requirements for measuring battery usage.

2.1.1 Basic Questions to Ask

Finding a solution to any problem starts with asking the right questions. This chapter addresses the following five questions.

1. What factors influence battery life?
2. What tools are available to help measure battery usage?
3. What units of power ought one measure?
4. What do application programs do that wastes power?
5. What can application programmers do to save power?

The remainder of this chapter answers these questions.

2.1.2 Factors Influencing Battery Life

Getting maximum battery life requires cooperation among hardware, software, and end-user. All hardware requires power to operate, but hardware does not need to run at full power all the time. One important element of extending battery life involves reducing the power usage of each hardware component as much as is possible. Even a CPU can be run at a lower power state by reducing its clock speed.

The key to managing peripherals involves having device drivers with support for different power states. Five power states are defined for devices: D0 (full on), D1 (low on), D2 (standby), D3 (sleep), and D4 (off). The only state that a device must support is D0. Other power states need to be supported to be able to save power. An application that really cares about managing the power state of individual peripherals—such as the backlight on the display screen or a wireless network adapter—can do so. Windows Mobile provides a great degree of granularity in moderating

Download at WoweBook.Com

system power usage. However we feel that the need for such granular control is atypical, so we focus instead on more generic power issues.

Application developers probably do not need to worry about battery life every day. But some kind of regular monitoring is needed to detect and respond to unwarranted increases in an application's power usage. For software that is under development, monitoring power usage once per month, for example, allows a development team to notice and respond to an unexpected increase in power consumption before the software is widely distributed to users.

Consider a game under development. The game might consume 5 milliwatts of power when run. By following a schedule of measuring the power usage on a monthly basis, a development team can assure themselves that this number has not changed. What if it does change? Let's say, for example, that after monitoring the power for four months there is a noticeable increase in the power used. Instead of 5 milliwatts (mW), perhaps power usage increases to 100 mW. Is this good or bad? By itself, this data does not tell us what has changed, but the change indicates that it is time to dig deeper.

Perhaps one of the new features during the past month was Bluetooth connectivity, which might be used to enable a game to be played between two devices. Does that justify the energy expenditure? Once again, we do not really have enough information to answer this question. What is important, however, is that the question has been asked, and hopefully, in the process of digging for answers, the development team becomes more aware of power issues. In addition, the team must be sure that power usage drops after the connection terminates. Failing that, the Bluetooth radio might have been left running, and could be draining the battery unnecessarily.

To generalize from this example, an unexpected increase in power usage must lead to determining what has changed and what to do about it. Perhaps the development team started to use a new network connectivity library from a third-party vendor, and the solution might consist of fine-tuning how the application initializes the library. If this does not address the issue, the right choice might involve considering alternatives to using the library (assuming, of course, that the library is the real culprit).

Application development teams need to regularly monitor the power consumption of their software throughout the software development cycle. This is important because, as shown in this example, a single change can increase an application's power consumption. And increases might be

Download at WoweBook.Com

caused by our code or by code that someone else wrote. The source code might be inside an application or inside a dynamic link library (DLL); the causes could just as easily come from native code as from managed code.

2.2 Measuring Battery Usage

To optimize battery usage, you need to measure battery usage. You start by establishing a baseline. How much power does a device require without your software? From that starting point, you then measure the power used while your application software runs. The result might be hard to distinguish from a device's "basal metabolic rate,"[2] or you might find that an application doubles or triples the power usage.

Whatever the result, you need to study this number and make sure it is consistent with what you expect in terms of what it asks of the hardware. For example, an application that activates any radio is going to use more power than one that does not. (Here "radio" means any wireless peripheral, including a GSM or CDMA module, a wireless local area network [WiFi], or Bluetooth.) Most wireless radios consume a lot of power when turned on, and when running unassociated from an access point (or unpaired, in the case of Bluetooth) they consume more power than when associated (or paired). When you look at the power usage curve, you have to make sure that what you see makes sense in terms of what you know your application is doing.

Various system timers prevent accurate measurement of battery usage. To deal with this, you must either disable these timers (the approach we take in our example) or wait for them to expire. The backlight timer, for instance, disables the backlight used for the display screen after a user-settable delay of between 5 and 60 seconds. This timer helps save battery power by cutting power to a peripheral that, depending on how it is configured, may consume several hundred milliwatts. While helpful in terms of overall battery life, this sudden drop in usage can make it difficult to establish a reliable baseline.

Other system timers include two idle timers for foreground and user-interface threads (`ACUserIdle` and `BattUserIdle`), two for background and system threads (`ACSystemIdle` and `BattSystemIdle`), and two that put the system into its lowest power level (`ACSuspend` and `BattSuspend`). The timers with names that start with "AC" are active when a mobile device is

2. The basal metabolic is "the amount of energy expended while at rest," from Wikipedia.

Download at WoweBook.Com

recharging (either via wall power or via USB cable to a PC). The other timers (`Batt...`) are active on battery power. Registry settings determine the value for each of these timers.

2.2.1 Talk Time and Standby Time

Mobile phone vendors advertise talk time and standby time as phone features. Studying the details of some phones can help establish a context for measuring power. Table 2.1 summarizes the advertised claims for four mobile phones.

The choice of handsets in Table 2.1 is somewhat arbitrary. Two of the handsets are very popular and well known (the RAZR and iPhone). The other two are handsets owned and used by one of the authors. For details on the meaning of units used in this table, see the sidebar, Units: Watts or Amperes?

Phone users care about talk time and standby time, which is why phone vendors advertise it. Ironically, as Table 2.1 suggests, the most important contributor to talk time is battery capacity, and not, as one might assume, efficiency of battery usage. A power-hungry phone can provide longer talk time than another handset by means of having a higher-capacity battery. The most battery-efficient phone in Table 2.1, the Motorola RAZR, also has the smallest battery. It needs only 350 mW to enable a connected phone call, and when in standby mode the phone's battery usage drops to 8.7 mW. If you look only at the ratings for talk time and standby time, on the other hand, its performance is similar to that of other, less efficient phones. The RAZR's efficient operation allows the handset to use a smaller battery, which allows the phone to have its slim, compact physical form.

TABLE 2.1: Power Consumption of Four Mobile Phones (Sorted by Battery Capacity)

	Battery			Time (hours)/[Consumption mW]		
Handset	Rating (mAh)	Volts	Capacity (mWh)	Talk (2G)	Talk (3G)	Standby
Apple iPhone	1,400	3.7	5,180	10 [518]	5 [1036]	300 [17.3]
HTC Kaiser	1,350	3.7	4,995	7 [714]	4 [1249]	360 [14]
HTC S710	1,050	3.7	3,885	7 [555]	N/A	175 [22]
Motorola RAZR	680	3.6	2,448	7/[350]	N/A	280 [8.7]

Download at WoweBook.Com

Table 2.1 also shows the difference between mobile phone technologies on talk time. The older, slower technology, known as 2G, for "second generation," is almost twice as efficient as the newer 3G (third-generation) mobile phones. This provides yet another example of how the battle to extend battery life gets increasingly challenging due to advances made in other domains (in this case, in the domain of mobile phone networks). Handset users can extend battery life by disabling a phone's 3G features whenever those features are not being used.

Units: Watts or Amperes?

Lacking a background in electronics, you might wonder about the myriad abbreviations: mAh, mWh, mA, and mW. Here is how to make sense of them.

Mobile phone batteries are rated based on capacity measured in milliampere hours, or mAh. This number indicates how much current, as measured in milliamperes (mA)—or [bf]1/1,000 of an ampere—the battery can provide for a minimum duration of one hour. Linear extrapolation is valid, so using half the power would enable the battery to last twice as long and using twice the power would cause a battery to only last half as long. For example, with its 680 mAh battery, a Motorola RAZR with a fully charged battery can run four hours at a continual drain of 170 mA.

A second number is part of the battery ratings: the device voltage. When you multiply the voltage by the milliampere hours, the result is the capacity of the battery in milliwatt hours (mWh). This number is a better measure of battery capacity because it combines the other two numbers into a single value. Different mobile devices have different voltage ratings. For example, we have encountered mobile devices that had batteries with the following voltage ratings: 3.6, 3.7, 4.2, and 7.2 volts. This is why the use of milliwatt hours—and also milliwatts—allows a fair and valid comparison.

2.2.2 Power Measurement Method 1: Software-Only Approach

The first method for measuring battery usage is a software-only approach. As shown in our `PowerMeter` sample (see Figure 2.1), you can query the current battery usage by calling the Win32 function `GetSystemPowerStatusEx2`. This Win32 function asks the system to populate a `SYSTEM_POWER_STATUS_EX2` data

Download at WoweBook.Com

structure. This data structure has 20 different members, but for our purposes the two most important ones are the current battery voltage (`BatteryVoltage`) and the current battery current (`BatteryCurrent`).

The `BatteryVoltage` element provides the millivolts (mV) for the current battery; for a typical Windows Mobile device this will be between 3,700 (for 3.7 volts) and 4,200 (for 4.2 volts). The `BatteryCurrent` element provides the value in milliamperes of the battery being charged (positive value) or being discharged (negative value). Whatever these two numbers are, the number you are interested in is the product of these two—volts times amperes—which yields the standard measure of power: watts (really milliwatts). To understand why, see the accompanying sidebar, Units: Watts or Amperes?

This tool provides a quick and easy way to take a snapshot of battery usage. And it can certainly help to detect any large changes in power consumption. This approach does have some limitations. One is the low granularity of measurements. Even if you call this function every second, the reported values on the devices we tested change only about every thirty seconds. While such slow change might map to some idealized notion about batteries, it is far from accurate. If you look ahead to Figure 2.4 in the next section, you see that a graph of power consumption looks more like the stock market price of a company over a decade (or perhaps a graph of a CEO's heart rate during a challenging business climate). The point is that the power consumption varies dramatically, a fact that is not at all reflected in the power consumption reported by the software method of measurement.

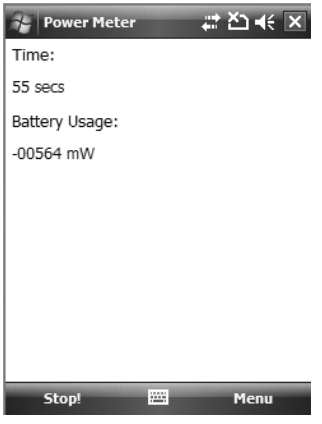


FIGURE 2.1: PowerMeter Sample Displaying Battery Usage

Download at WoweBook.Com

Why is this? It takes power to measure power, and the last thing you want is to drain the battery just from checking the battery state. So, if all you need are snapshots of power usage, the software-only approach may work for you. Otherwise, you'll want to consider the hardware approach. Before examining that approach in detail, the details of our `PowerMeter` program are worth some commentary.

The .NET Compact Framework does not provide any functions for querying the system battery power state. The `PowerMeter` program calls the underlying Win32 function through the Platform Invoke (P/Invoke) feature. (We discuss P/Invoke in more detail in Chapter 3, Platform Invoke). The first thing to remember for all P/Invoke support is the need to add the following line to enable access to the required namespace:

```
using System.Runtime.InteropServices;
```

Listings 2.1 and 2.2 summarize the key parts of the `PowerMeter` sample. Listing 2.1 shows the P/Invoke declarations for one Win32 function and one Win32 data structure. These definitions come from the file winbase.h. We made some minor changes to the data structure. The changes involve making some of the integers signed, to reflect the fact that the reported values can be either positive or negative. We change the type of the following fields from `uint` to `int`: `BatteryCurrent`, `BatteryAverageCurrent`, `BatterymAHourConsumed`, and `BatteryTemperature`.

Listing 2.2 shows the event handler for a timer that fires every five seconds. When it does, a call is made to `GetSystemPowerStatusEx2` to query the current battery usage values. From the values returned in the data structure, the battery drain is calculated by multiplying the voltage by the amperes of current being used.

Listing 2.1: P/Invoke Declarations Needed to Call `GetSystemPowerStatusEx2`

```
[DllImport("coredll.dll")]
public static extern int
GetSystemPowerStatusEx2(
    ref SYSTEM_POWER_STATUS_EX2 pSystemPowerStatusEx2,
    int dwLen, int fUpdate);

public struct SYSTEM_POWER_STATUS_EX2
{
    public byte ACLineStatus;
    public byte BatteryFlag;
    public byte BatteryLifePercent;
    public byte Reserved1;
```

Download at WoweBook.Com


```
    public uint BatteryLifeTime;
    public uint BatteryFullLifeTime;
    public byte Reserved2;
    public byte BackupBatteryFlag;
    public byte BackupBatteryLifePercent;
    public byte Reserved3;
    public uint BackupBatteryLifeTime;
    public uint BackupBatteryFullLifeTime;
    public uint BatteryVoltage;
    public int  BatteryCurrent;
    public int  BatteryAverageCurrent;
    public uint BatteryAverageInterval;
    public int  BatterymAHourConsumed;
    public int  BatteryTemperature;
    public uint BackupBatteryVoltage;
    public byte BatteryChemistry;
};
```

Listing 2.2: Timer Function for Querying and Displaying Battery Status

```
private void timerQueryPower_Tick(object sender, EventArgs e)
{
    var spe = new SYSTEM_POWER_STATUS_EX2();
    int size = Marshal.SizeOf(spe);
    int ret = GetSystemPowerStatusEx2(ref spe, size, 1);

    int mWatts = (int)(spe.BatteryCurrent *
                   spe.BatteryVoltage)/1000;

    nTime += 5;

    labelBattery.Text = mWatts.ToString("d5") + " mW";
    labelTime.Text = nTime.ToString() + " secs";
}
```

2.2.3 Power Measurement Method 2: Dedicated Hardware

The second method for measuring battery usage is a hardware-based approach. It is more precise than the first method because it involves tools that are outside the target hardware. The advantage of this method is much greater accuracy. The software-based approach requires the device itself to do some work, which adds some uncertainty to the numbers that it reports. By contrast, the hardware-based approach does not put any strain on the device itself. Instead, an outside agent provides both power and highly accurate measurements.

A second benefit of the hardware-based approach is greater granularity. As discussed earlier in this chapter, the granularity of the software-based

Download at WoweBook.Com

approach is about thirty seconds. In other words, you get about two measurements per minute. The hardware-based approach, on the other hand, provides many measurements per second.

The hardware-based approach also allows power measurements during the entire device life cycle: from boot-up power through when the system enters idle-time operation on through to shutdown. This is probably not terribly important for most application programmers. For anyone building the hardware or doing the low-level device drivers, on the other hand, this can be very useful information.

A disadvantage to the hardware-based approach to power measurement is cost. Power usage hardware is expensive. It is a highly specialized type of hardware, and does not lend itself to being used on desktop or server systems, nor can you take this hardware and use it to measure the power usage of your home refrigerator.

We use the Power Monitor which was purchased from Monsoon Solutions, Inc. (www.msoon.com), of Bellevue, Washington (see Figure 2.2). (Neither author has any affiliation with this company.) As of this writing, the cost of this device is $741 plus shipping, tax, and export charges. We have heard rumors about there being other devices, but have yet to substantiate those rumors or experiment with using other devices.

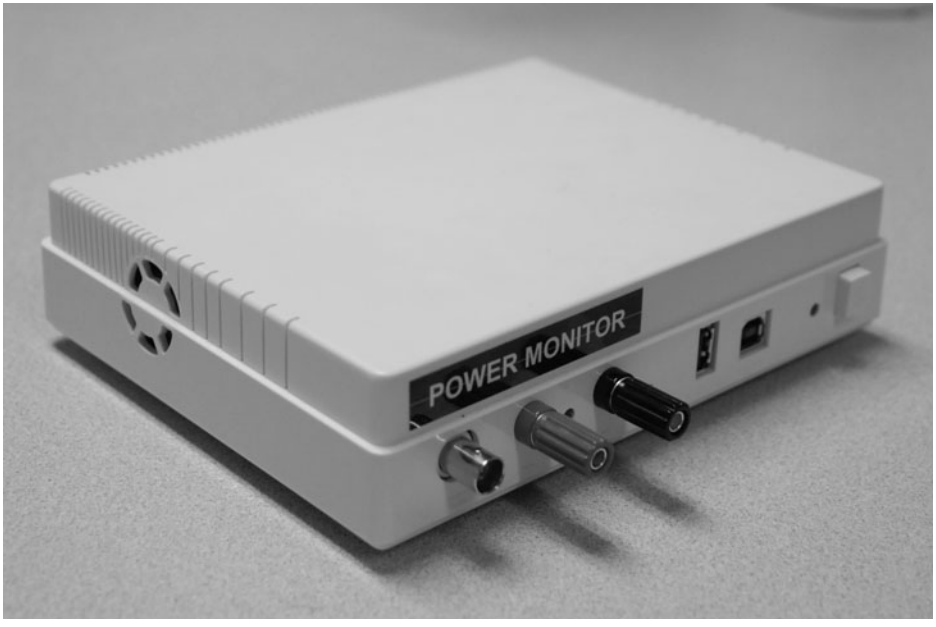


Figure 2.2: Power Monitor from Monsoon Solutions, Inc.
(Photo by Paul Yao)

Download at WoweBook.Com

Figure 2.2 shows a photograph of the Power Monitor. The two posts in the front are the battery connection terminals. When taking measurements, the Power Monitor provides power to your mobile device in place of the device battery. By serving as the power supply, the Power Monitor is able to precisely report on the amount of power that the mobile device is using.

The first time we connected our Windows Mobile device to the Power Monitor, it took a little longer than we expected. We were proceeding with caution because we wanted to avoid making a wrong connection and burning out the battery (or, worse, our Windows Mobile device). As mentioned, the Power Monitor has two battery connections; but when you look at a typical mobile device battery, you notice that there are four (or more!) connectors. Figuring out the correct connectors to use required a bit of care, but it gets much easier when you realize that on many batteries there are clear markings for the positive (hot) terminal and the negative (ground) terminal. The other connectors are for things such as power usage and temperature monitoring.

Monsoon Solutions provides a multipage document to help explain how to prepare a device battery for connection to its Power Monitor. Our summary of these preparation instructions is as follows.

1. Cover the "+" (positive/red) battery terminal with an insulator (tape works well) so that the battery is not providing any power to the mobile device. Test by trying to turn on the mobile device once you have covered the positive terminal. You should not be able to turn it on.
2. Apply a strip of adhesive copper tape over the insulator tape you applied in step 1 so that the copper tape will connect to the mobile device battery terminal.
3. Apply a second strip of adhesive copper tape to the "–" (negative/black) battery terminal.
4. Connect the red lead from the Power Monitor to the battery's "+" terminal.
5. Connect the black lead from the Power Monitor to the battery's "–" terminal.
6. Launch the software and collect your readings.

Figure 2.3 shows an HTC Kaiser hooked up and running on the power provided by a Monsoon Power Monitor. The Power Monitor hardware collects the power usage data, but the hardware itself does not have a display

Download at WoweBook.Com

FIGURE 2.3: Power Monitor Powering an HTC Kaiser
(Photo by Paul Yao)

screen to show its data. Instead, that data gets sent over a USB cable to a desktop Windows system on which you have installed the Monsoon Power Tool software. The software controls the Power Monitor operation and can also collect and display power data. Figure 2.4 shows a screen shot of the Monsoon software showing a graph of some data, in this case the average power consumption of a Windows Mobile device.

2.3 Device Power Study

How much power does it actually take to run a Windows Mobile device? To answer this question, we connected our Power Monitor to an HTC Kaiser, a Windows Mobile 6.1 Professional Edition phone. Manufactured by High Tech Computing (HTC), this phone is available to customers in the United States from AT&T under the product name Tilt. We measured the power usage in response to various user actions.

We referred to this phone earlier in the discussion on talk time and standby time. For your convenience, here is a summary of the phone's pertinent details:

- 1,350 mAh battery
- 3.7 volt battery

Download at WoweBook.Com

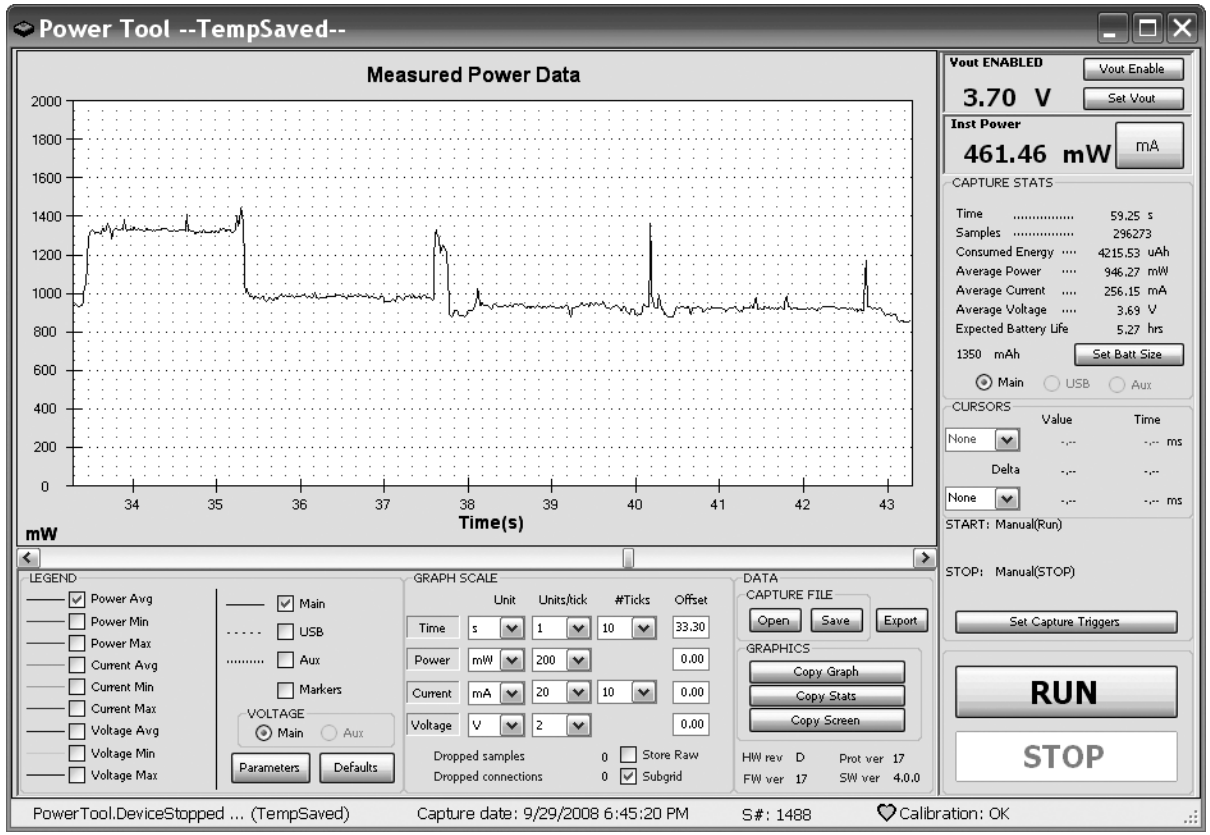


FIGURE 2.4: Monsoon Solutions Power Tool Software Displaying Power Usage Data

From the Library of Lee Bogdanoff

Download at WoweBook.Com

- Capacity of 4,995 mWh
- Talk time (3G): 4 hours (1,249 mW)
- Talk time (2G): 7 hours (555 mW)
- Standby time: 360 hours (14 mW)

We did some measurements using the Monsoon Power Monitor, described earlier. We provide the data here to help give you an idea of how various peripherals consume power. You can make better choices in power usage only when you understand how power is consumed. Our hope is that you will be inspired to take your own measurements and continue to learn more about this important subject.

2.3.1 Standby Power Usage

We started by measuring power usage when the HTC Kaiser is in standby mode. This is the lowest power state, and the state in which our device is likely to spend most of its time. Establishing this baseline provides us a place to begin monitoring what is "normal" so that we can detect when something out of the ordinary is happening. Table 2.2 summarizes our standby measurements.

It is worthwhile to mention something about the methodology we used. We took measurements with all radios off and then with individual radios on. Each value is the average of at least three values taken over a ten-second period, using the capability to average power usage by the software that comes with the Power Monitor hardware. We also rounded to one decimal place, although the software provided many more digits of precision.

TABLE 2.2: Standby Power Usage for HTC Kaiser

State	Milliwatts	Hours of Usage
All Radios Off	4.3	1,162
Infrared-Only On	4.2	1,189
WiFi-Only On	4.3	1,162
Bluetooth On (Passive)	6.7	746
Bluetooth On (Discoverable)	7.3	684
Phone-Only On	10.4	480

Download at WoweBook.Com

The HTC Kaiser is rated at 360 hours of standby time, which translates into power consumption of 14 mW given its battery configuration. All of our measurements show lower power consumption, and therefore a higher number of hours of usage. Don't be too concerned by this fact, however, since some scenarios—such as keeping all radios (including the phone) turned off—are not realistic.

A more realistic scenario, in fact, would be to have the radio on, plus Bluetooth (for a mobile phone headset, say). If you take the phone-only number (10.4 mW) and add in a value for discoverable Bluetooth (7.3 – 4.3 = 3.0 mW) you get a value of 13.4, which is not far from the 14 mW value for the 360 hours of standby time. One unknown factor involves the distance to the cell tower, which would make a difference in the amount of power needed to keep in touch with the tower. All in all, our numbers are at least in the same ballpark as the published numbers for this phone.

What is noteworthy about both the mobile phone radio and the Bluetooth radio is that both stayed active when the phone went into standby. We verified this for both radios. For example, we know the mobile phone radio was active because incoming calls caused the phone to wake up and start to ring. In addition, the blinking blue LED showed us that the phone's Bluetooth support was still active.

We cannot say the same for the other two wireless devices—the infrared device and the WiFi radio. Table 2.2 shows how neither of these caused any additional power to be used, even when they were on. The reason is that both of these radios were shut off when the system went into standby mode. In addition, the device dropped its connection to the WiFi access point. When the system was returned to normal, the WiFi network connection had to be reestablished.

2.3.2 Backlight Power Usage

Our power study addresses the HTC Kaiser shown in Figure 2.3. We measured a variety of hardware peripherals, and we summarize the results here. We used the hardware approach to measuring power consumption for most of our tests, but we did use the software power meter just to see how close the two approaches would be. We first measured how much power was consumed at different screen backlight settings. We had suspected that the display backlight consumed a lot of power, and were eager to test this assertion. Our results appear in Tables 2.3 and 2.4.

The HTC Kaiser has ten backlight levels, which we numbered from 1 to 10 (lowest to highest). The value reported in the "Hardware Measure"

Download at WoweBook.Com

TABLE 2.3: HTC Kaiser Average Backlight Power Consumption

Backlight Level	Hardware Measure	Battery Life (Hours)	Software Measure	Delta
1	167 mW	34	188 mW	+21
2	184 mW	31	201 mW	+17
3	217 mW	26	223 mW	+6
4	259 mW	22	256 mW	-3
5	304 mW	19	317 mW	+13
6	351 mW	16	350 mW	-1
7	397 mW	14	384 mW	-13
8	439 mW	13	438 mW	-1
9	480 mW	12	467 mW	-13
10	522 mW	11	522 mW	0

column in Table 2.4 provides the average power consumption over ten seconds, as reported by the Monsoon software. We calculated the hours of battery life that could be expected at each level of power consumption, to put our numbers into perspective (calculations are based on the standard battery for this device, which has a capacity of 4,995 mWh). Notice that the lowest backlight setting allows you to triple your battery life (to 34 hours) compared to the highest backlight setting (only 11 hours). Clearly, the backlight setting impacts battery life.

We used both hardware and software methods to measure the power usage and were initially pleased to see that the difference (in the column marked "Delta") was relatively small. As we looked more closely at the data—especially the graphical output created by the hardware power monitor software—we realized our numbers seemed a little high. We were, after all, measuring average power consumption, but a close study of the power consumption graph shows an even lower *baseline* power consumption. Figure 2.5 shows a graph of the power consumption during the transition from level 10 to level 1, when the backlight timer expires.

Our initial values (in Table 2.2) are not as accurate because other activities running in the background consume power, creating spikes several times per second. By studying the graph, however, it becomes clear that the baseline power when the backlight is at its highest, brightest setting

Download at WoweBook.Com

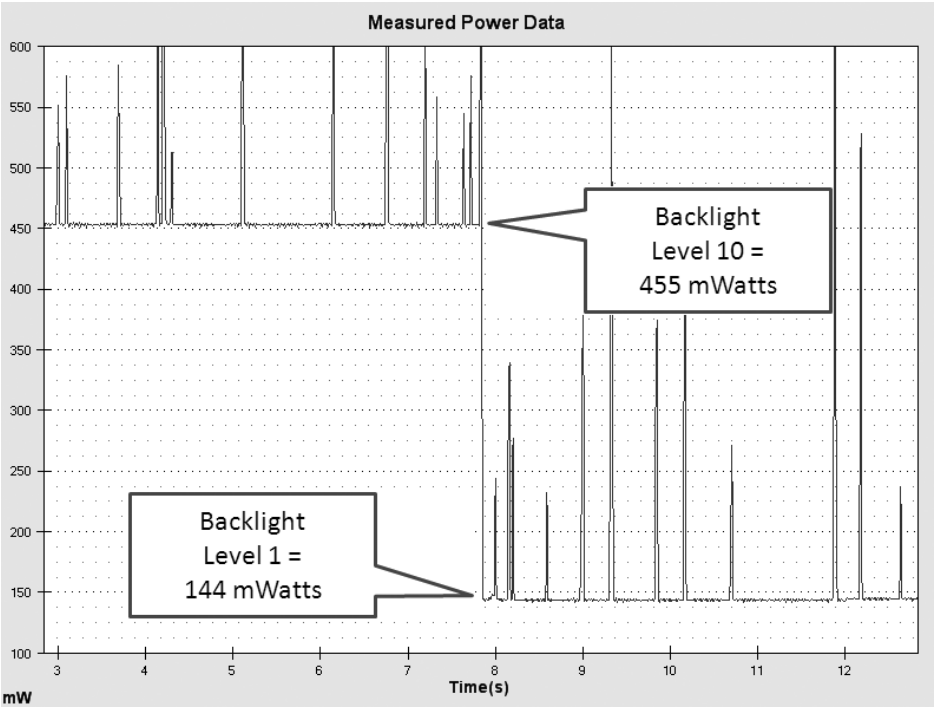


FIGURE 2.5: Power Consumption during Transition from Backlight Level 10 to 1

(10) is closer to 450 mW and not the much higher average (522 mW) that we had found earlier. After the backlight timer expires, the backlight power setting changes to its lowest setting (1), where the amount of power consumed is 140 mW instead of the higher value (188 mW) that we had initially seen. Table 2.4 summarizes the backlight baseline values, as measured from the graph baseline instead of average power consumed.

TABLE 2.4: Baseline Backlight Power Consumption (Rounded to the Nearest 10 mW)

Backlight Level	Hardware Measure	Delta to Next Level
1	140 mW	10 mW
2	150 mW	30 mW
3	180 mW	40 mW
4	220 mW	40 mW
5	260 mW	40 mW
6	300 mW	40 mW

Download at WoweBook.Com

TABLE 2.4: Baseline Backlight Power Consumption (Rounded to the Nearest 10 mW)

Backlight Level	Hardware Measure	Delta to Next Level
7	340 mW	40 mW
8	380 mW	30 mW
9	410 mW	40 mW
10	450 mW	

2.3.3 Power Usage for Communicating

Mobile phones have been around for more than a decade. What started as a talking device that you could carry soon evolved into a talking device with other features, including the ability to take pictures, play music, and play games. The evolution has gone so far that some pundits find it necessary to remind us that mobile phones can still be used for talking. What, then, is the energy cost for talking?

Table 2.5 summarizes our measurements taken for telephone calls. There are two published talk-time numbers: a 2G talk time of seven hours (555 mW consumption) and a 3G talk time of four hours (1,249 mW consumption). Clearly, the numbers in the table are from 3G talking. The table shows the difference between using the default headset and the speakerphone—as you can see, more power is required to use the speakerphone. Also (not surprisingly), it takes more power to talk than to keep silent, since more work needs to be done to compress or decompress the voice stream.

TABLE 2.5: Talk-Time Power Consumption

Talk-Time Power Consumption	Power (mW)	Talk Time
Default Headset—Silence	1,200	4.2 hours
Default Headset—Talking	1,470	3.4 hours
Speakerphone—Silence	1,366	3.7 hours
Speakerphone—Talking	1,674	3 hours

2.3.4 Multimedia Power Usage

In addition to being a phone, the HTC Kaiser can also be used for various multimedia functions, including taking pictures and recording voice

Download at WoweBook.Com

notes. What are the power costs of these operations? It turns out that these are quite expensive, relative to other things you do with this device. The reason is that specialized hardware needs to be turned on and used. Table 2.6 shows the power consumption of the voice recorder and camera applications. Obviously, it costs a lot of power to take pictures. While this is an interesting point, it does not cause us a great deal of concern because the placing of a phone in picture-taking mode tends to be something that is done for only short periods of time.

TABLE 2.6: Power Consumption of Multimedia Applications/Devices

Application	Power Consumption	Usage Time
Voice recorder	560 mW	8.9 hours
Camera application	900 mW	5.6 hours

2.4 Conclusion

To manage the power usage of your application, you must be able to measure the power usage. This chapter discussed two different ways to measure power consumption on a Windows Mobile device. The software-driven approach is inexpensive and easy to do. It has the disadvantage of being somewhat less accurate. The more accurate approach requires dedicated hardware, which gives you the ability to capture a highly granular set of measurements around how power is used. We hope you will take our advice and tune your applications so that everyone gets the benefit of more efficient power usage and extended battery life.

Download at WoweBook.Com

3
Platform Invoke

In this chapter, we discuss Platform Invoke, a .NET feature for calling functions in native Win32 DLLs from managed code.

THIS CHAPTER COVERS Platform Invoke—also known as P/Invoke—a .NET feature for calling functions in native dynamic link libraries (DLLs). Most of the time the .NET Compact Framework provides all the support you need to build programs. Sometimes you need something that is not supported by the .NET Compact Framework but is supported by calling a function in a Win32 DLL. For such cases, there is P/Invoke.

To call a function in a native DLL from managed code, you create a **P/Invoke wrapper.** A P/Invoke wrapper looks just like a managed-code method declaration. One difference is that a P/Invoke wrapper lacks a function body—just the method name, return type, and parameter information are included. In addition, P/Invoke wrappers need a `DllImport` attribute to identify the required DLL and specific function name.

Programmers who have worked with the desktop .NET Framework might wonder how the .NET Compact Framework's support for P/Invoke compares with the support found on the desktop. A section later in this chapter, Comparing P/Invoke Support, provides insight into the differences (section starts on page 112).

Download at WoweBook.Com

3.1 Overview of P/Invoke

P/Invoke connects two different types of code: managed code and unmanaged code.[1] Table 3.1 lists some of the differences between managed code and native (a term we prefer over "unmanaged") code.

Managed code is *portable code* because .NET executables can run on any platform that has the appropriate common language runtime, regardless of the CPU that is present. Managed code is also *safe code* because a broad set of features helps avoid problems that plague native code: Implicit pointers and automatic memory management eliminate memory leaks, and array boundary protection prevents memory overwrites.

TABLE 3.1: Differences between Managed and Unmanaged Code

	Managed Code	Unmanaged Code
Alias	Portable code Safe code .NET code C# code/Visual Basic .NET code	Native code Unsafe code Win32 code C code/C++ code
Supported languages	C#, Visual Basic .NET	C, C++
EXE/DLL file format	Common Intermediate Language (CIL): CPU-independent machine-level language that is converted to native machine instructions at runtime	Native machine language of the target processor; for example, ARM and x86
Memory management	A garbage collector provides automatic cleanup of allocated objects	Application (or driver) code is responsible for cleaning up memory and Win32 objects. Automatic cleanup is provided upon process termination

1. Experienced Windows programmers will see that P/Invoke is like two older Windows technologies: the thunks that date back to Windows 1.x and the proxy/stub elements used in COM to marshal data between operating system processes and across a network in DCOM. Each represents a stateless layer to support a transition—a bridge—between two otherwise disconnected bodies of code. P/Invoke adopts the COM term *marshaling* to refer to the process of passing data into and receiving return parameters back from unmanaged libraries.

Download at WoweBook.Com

TABLE 3.1: Differences between Managed and Unmanaged Code *(Continued)*

	Managed Code	Unmanaged Code
Safetychecking	Yes	No
Arrayboundary protection	Yes	No
Support for pointers	Implicit pointer support is built in; explicit pointers require the `unsafe` keyword. The `Marshal`[a] class supports the creation of pointers for passing to unmanaged code	Explicit pointers widely used

a. `System.Runtime.InteropServices.Marshal`

The term *native code* comes from the fact that the executable file format uses the native machine instructions of the target processor. It is sometimes called *unsafe code* because it lacks the protection from memory leaks, bad pointers, and array overruns that the common language runtime provides. We sometimes call it *Win32 code* because such DLLs are often (but not always) built to directly use the Win32 API.

3.1.1 When to Use P/Invoke

Table 3.2 summarizes some of the situations when you are likely to need P/Invoke, as implemented in Version 3.5 of the .NET Compact Framework. We expect access to custom native DLLs to be the most common of these situations. After all, many native DLLs exist, and P/Invoke provides a way to tap into those libraries.

TABLE 3.2: Examples of When to Use P/Invoke for Features Not Implemented in Managed Code

Feature	Comments
ActiveSync	ActiveSync is a set of services for connecting a desktop PC with a Windows CE device (see Chapter 9, The Remote API). Most ActiveSync services are provided on the desktop itself so that getting .NET support for ActiveSync involves creating a set of P/Invoke wrappers for the desktop .NET Framework. We provide just such a set with the code that accompanies this book. You can download that code from The Paul Yao Company Web site: www.paulyao.com/cfbook/code

Continues

Download at WoweBook.Com

TABLE 3.2: Examples of When to Use P/Invoke for Features Not Implemented in Managed Code *(Continued)*

Feature	Comments
COM components	You can call COM and ActiveX components from your .NET Compact Framework code. This feature is also known as COM Interop
Common dialog boxes	Common dialog boxes provide prebuilt dialog boxes for opening and closing files, allowing a user to select a font or a color, and printing. The Compact Framework supports the dialogs for opening and saving files but not the font and color pickers. In such cases, direct calls to the underlying Win32 functions help fill this need
Custom controls	Custom controls implemented in native DLLs can be supported in Compact Framework Windows Forms applications
Custom native DLLs	These can access exported C-callable functions in a DLL. Such DLLs must be specifically built for Windows CE (or Windows Mobile), targeting a specific CPU (DLLs built for the desktop cannot be used in Windows CE or on Windows Mobile devices)
Microsoft Message Queue (MSMQ)	MSMQ allows for asynchronous communication between different computers
Point-to-point message queues	This interprocess communication feature was introduced in Windows CE Version 4.0. It provides anonymous pipe support for interprocess or interthread communication
Real-time threads	Windows CE provides great real-time support, with very consistent response times for the highest-priority thread in the system and for the highest-priority interrupt. When you have hard real-time requirements, the safest approach to take is to delegate that work to Win32 code. Unexpected delays can occur in managed code due to either of the following: IL JIT (the time taken to convert IL to native machine instructions); or garbage collection (all managed threads are blocked during memory cleanup; native P/Invoke threads run freely without blocking, and block only on return to managed code if garbage collection is running)
Running programs	Call the Win32 `CreateProcess` function to run a program
Shell functions	Some very useful Win32 functions have a prefix of SH, which stands for "Shell." Some of the interesting ones include `SHGetSpecialFolderLocation`, `SHGetSpecialFolderPath`, and `SHBrowseForFolder` (CE .NET 4.0 and later)

Download at WoweBook.Com

TABLE 3.2: Examples of When to Use P/Invoke for Features Not Implemented in Managed Code *(Continued)*

Feature	Comments
Stream interface driver control codes	Windows CE uses *stream interface drivers* for system services such as a Web server (HTTPD), MSMQ, Universal Plug and Play (UPNP), and the Audio Compression Manager (ACM). This type of driver is also used for device drivers for various types of hardware, including the serial port, the infrared port, and various kinds of disk drives. Full access to these drivers requires the Win32 file I/O functions, including the `DeviceIoControl` function to send driver-specific control codes

The other items listed in Table 3.2 are Win32 functions supported by Windows CE but not supported in the .NET Compact Framework itself. The CEFUN tool, also included as a sample with the book's source code, summarizes the available Win32 functions that Windows CE supports. When run on a specific system, this utility provides a list of the functions that are present.

3.1.2 Why We Prefer .NET Compact Framework Classes over Win32 Functions

We prefer using the support provided in the .NET Compact Framework over calling Win32 functions. We have several reasons. First is the automatic memory cleanup. By contrast, Win32 functions that create system objects require you to call the associated cleanup function. For example, if you create a bitmap by calling the Win32 `CreateDIBSection` function, you must later call the `DeleteObject` function to clean up the bitmap's memory.

Failure to clean up causes memory leaks. On a desktop or server system with gigabytes of RAM, memory leaks cause the system to slow down and eventually crash. On a mobile or embedded system, which typically has much less memory than a desktop or server system, a similar memory leak will cause a crash to occur sooner.

A second reason we prefer .NET functions is portability. An application written today—for either the full .NET Framework or the .NET Compact Framework—is more portable than a comparable Win32 program. This portability is largely the result of managed code being built on higher-level abstractions. Microsoft has made and will continue to make significant investments in porting the managed-code libraries to a broader range of

Download at WoweBook.Com

hardware and software platforms. The same cannot be said of the underlying machine instruction sets on which native DLLs are built.

Third, related to portability is the issue of trust and security. Code that uses P/Invoke is less trusted than code that does not—the part of the common language runtime that is responsible for security can look at CIL code and determine whether it is safe to run. When it cannot be verified as being safe, it is called *unsafe code.* The current version of the .NET Compact Framework does nothing to prevent unsafe code from running, but we cannot guarantee that this will not change in the future.

A fourth reason we prefer managed functions is that more work is required to call Win32 functions with P/Invoke. You have to create a function declaration, as well as constants and data structures to support that function. By comparison, managed functions are available by adding references to the required assembly and namespace.

3.1.3 Porting Native Code to Managed Code

We like managed code, but our enthusiasm has a limit. Programmers in our training classes often ask when it makes sense to port native DLLs to run as managed DLLs. Our answer: "It depends."

If you have built and deployed a native DLL, you have invested time and money. We suggest you consider using that DLL as it is. Microsoft will probably continue to support that family of CPUs and the Win32 instruction set for the sake of compatibility with its own existing software. We make this recommendation because we sometimes hear that developers think they need to eliminate their own native libraries for the sake of future compatibility. Before you port native code to managed code, we recommend that you have a solid business case—some real benefit to your customers—to justify the time and energy it will take.

There are, of course, reasons to port. First, libraries with large amounts of user interface code can benefit from being ported because running as a managed-code library can simplify interoperating with .NET Compact Framework–based user interface features. (You may, however, also lose some features because the .NET Compact Framework controls generally support fewer features than their Win32 counterparts.)

A second reason to port native code to managed code is that debugging P/Invoke code can be difficult. Visual Studio does not support debugging between managed and native code when dealing with .NET Compact Framework code (a feature that is supported for desktop code with the desktop .NET Framework). When a P/Invoke function call fails

Download at WoweBook.Com

under the .NET Compact Framework, you get an error message in the form of a .NET exception with few details to help you identify the cause of the failure. To debug a native DLL, you must use Visual Studio's native debugger.

P/Invoke Debugging Tip

If you are creating P/Invoke wrappers for a large number of DLL functions, it might be worth porting your library (or a portion of your library) to the desktop. When you have your P/Invoke code working on the desktop, you can then port the debugged code back to Windows CE and the .NET Compact Framework.

We suggest this approach because the Visual Studio debugger does not support stepping from managed code into native code for the .NET Compact Framework. This feature is supported on the desktop, however. If you need help with a particularly difficult P/Invoke declaration, build a tiny Win32 DLL for the desktop with that one function. Once you figure out the program in your P/Invoke declaration, you can apply the changes to your device-side code.

A third reason to port a native DLL to a managed DLL is to simplify the packaging and setup of your program, particularly when the software you deploy must run on systems with different CPUs with different instruction sets. As of this writing, all Windows Mobile platforms run on the ARM instruction set. This recommendation applies, then, only if you are building code for a non-Windows Mobile platform and you need to support different target CPUs.

3.1.4 Component Object Model (COM) Support

COM support has been an important part of Microsoft's operating system support for many years. COM provides an object-oriented programming (OOP) model for native DLLs. With the rise of .NET, there has been much less emphasis on COM. And yet the need is there because many native COM libraries exist. For example, the Windows Mobile include files have more than 2,400 COM interface definitions.

Managed code can interoperate with COM, through a feature known as COM Interop. The support goes two ways and includes calls from

Download at WoweBook.Com

managed code into COM objects. This feature is supported by creating objects known as runtime callable wrappers (RCWs).

COM objects—which are implemented in native libraries—can also call into managed code. This is done through the creation of objects known as COM callable wrappers (CCWs).

3.2 Creating P/Invoke Declarations

A P/Invoke declaration—often referred to as a P/Invoke wrapper—follows the same rules for regular function declarations: It must be made within a class, have a return type, and have zero or more parameters. In addition, a P/Invoke wrapper needs these elements:

- The `DllImport` attribute
- The `static` keyword
- The `extern` keyword

To understand how these elements are used together to provide access to native DLL functions, it helps to look at some examples. Let's start with a simple P/Invoke declaration.

3.2.1 A Simple Function: `MessageBox`

A common example used for P/Invoke declarations is the `MessageBox` function. That is where we are going to start. The `MessageBox` function is declared in a Win32 include file, winuser.h, as follows:

```
// C++ Win32 declaration
int WINAPI MessageBoxW(HWND hWnd, LPCWSTR lpText,
    LPCWSTR lpCaption, UINT uType);
```

The `W` at the end of the function name in this code marks `MessageBox` as a Unicode function. With few exceptions,[2] all Win32 functions in Windows CE are implemented as Unicode functions, meaning that the string parameters must be 16-bit (UTF-16) Unicode strings. You can either include or omit the `W` in a P/Invoke declaration because if needed the

2. The exceptions are for some network functions, for which both Unicode and multibyte functions are supported—for example, there is both an `InternetOpenW` and an `InternetOpenA` function in wininet.dll.

Download at WoweBook.Com

runtime adds a `W` for you. Both of these are valid declarations for the `MessageBox` function:

```
 [DllImport("coredll.dll")]
public static extern int
MessageBoxW(IntPtr hWnd, String lpText, String lpCaption,
   UInt32 uType);

// A MessageBox declaration without 'W'
 [DllImport("coredll.dll")]
public static extern int
MessageBox(IntPtr hWnd, String lpText, String lpCaption,
   UInt32 uType);
```

This declaration contains a reference to `DllImport`, which is worth discussing on several counts. From the perspective of managed code, it is an example of an **attribute.** In C#, attributes appear between square brackets (`[` and `]`) and fine-tune various managed-code declarations. For example, the `WebMethod` attribute defines Web services functions.

While different from other types of objects, attributes are defined by a class that resides in a namespace. By convention, attribute names have a suffix of `Attribute`. To see the options for `DllImport`, then, you search the help database for the class defined as `DllImportAttribute`.

To use the `DllImport` attribute, we need a namespace reference such as the following:

```
using System.Runtime.InteropServices;
```

With the namespace declaration, we call the Win32 `MessageBox` function like a regular managed-code method:

```
MessageBoxW(IntPtr.Zero, "Call to MessageBoxW", "Message", 0);

MessageBox(IntPtr.Zero, "Call to MessageBox", "Message", 0);
```

The `DllImport` attribute identifies functions in native libraries. An attribute declaration is a constructor for an object, and like other kinds of object constructors, an attribute constructor can accept parameters. The `DllImport` attribute constructor takes a single parameter, a string that is the name of a native DLL.

As you begin to create P/Invoke declarations, you need to proceed slowly because the debugging support and error reporting are limited. Visual Studio does not support tracing a call from managed code into

Download at WoweBook.Com

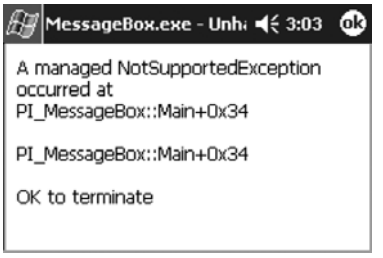


FIGURE 3.1: Example of an Exception Caused by a P/Invoke Error

native code. In some cases, the only error you get displays a message such as the one shown in Figure 3.1. This is often the only information you get that a problem has been encountered in P/Invoke. If you look closely at the error message displayed, you get little information about why there was a problem—only that something is not supported.

3.2.2 Native Function Details

In C#, a function with the `DllImport` attribute must also have the `static` and `extern` keywords. The `static` keyword means you can call the function anytime without having to create an instance of the class—this is the closest thing C# has to a global function that C programmers are so used to. C++ programmers may recognize that the `static` keyword in a C# class function is similar to the same keyword in a C++ class. The `extern` keyword indicates that the actual implementation of a function is in another language besides C#.

The function we are calling, `MessageBox`, takes four parameters: (1) a window handle, (2) message text, (3) caption text, and (4) some flags. A close look at each of these will tell us more about how to pass parameters to this Win32 function.

```
// C++ Win32 declaration
int WINAPI MessageBoxW(HWND hWnd, LPCWSTR lpText,
    LPCWSTR lpCaption, UINT uType);
```

The first parameter, `HWND hWnd`, is a window handle for the message box's parent window. When the message box appears, its parent window is disabled; when the message box is dismissed, the parent is enabled and activated.

Native code uses many different handle types, such as the window handle. Managed code, on the other hand, does not use handles. To bridge

Download at WoweBook.Com

this gap, managed code provides the `IntPtr` type.[3] When there is no parent window, we pass a special value, `IntPtr.Zero`, which is the same as a 32-bit all-zero (`NULL`) native handle. In general, a null native handle is an invalid handle value. In a few cases, a null native handle has special meaning when it is passed. In a `MessageBox` call, a null native handle means no parent window exists for the message box.

Some P/Invoke examples use regular integer types for handles, a technique that works but makes code less portable. Today all implementations of .NET are 32-bit implementations; there may someday be a 64-bit (or 128-bit) implementation. `IntPtr` is always the right size to hold handles (and pointers) for the platform on which a program is running, so its use helps make your code more portable. A second use of the `IntPtr` type is for native memory pointers, a feature that helps unravel some trickier problems encountered in calls to native code.

The second and third parameters, `LPCWSTR lpText` and `LPCWSTR lpCaption`, are strings. To a native function, these appear as string pointers. P/Invoke correctly passes simple string pointers such as the one shown in the call to the `MessageBox` function.

The final parameter, `UINT uType`, is a bit-field flag parameter. This unsigned integer holds flags that indicate the buttons and icons to display in the message box. To fill in the proper bit values, you create .NET definitions for the Win32 flag values. (In a pinch, you can also hard-code constant values; however, this approach makes code harder to maintain, so we suggest you avoid that practice.) Here are the C++ definitions for the legal values to pass in the `uType` parameter, as defined in winuser.h:

```
#define MB_OK                       0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL              0x00000003L
#define MB_YESNO                    0x00000004L
#define MB_RETRYCANCEL              0x00000005L
#define MB_ICONHAND                 0x00000010L
#define MB_ICONQUESTION             0x00000020L
#define MB_ICONEXCLAMATION          0x00000030L
#define MB_ICONASTERISK             0x00000040L
```

When you need to define flags or constant values to call native functions, there are at least two approaches you can take. One approach is to

3. An `IntPtr` is also used for pointers, as demonstrated in the `FindMemoryCard` example later in this chapter.

Download at WoweBook.Com

define a set of constant values. A second approach is to define an enumeration. We show you both in Listing 3.1 so that you can decide which you prefer, along with the associated declaration and call to the `MessageBox` function.

Which approach should you use? One benefit of the enumeration is that once it is defined in your code, Visual Studio can use it to show the available options in a drop-down window as you type your code. This is very handy, and for this reason, we prefer enumerations.

Listing 3.1: Two Approaches for a P/Invoke Wrapper for the `MessageBox` Function

```
// MessageBox.cs - Two ways to call Win32 MessageBox function
//
// Platform Invoke Sample.
// Code from _.NET Compact Framework Programming with C#_
// Authored by Paul Yao and David Durant.
//

using System;
using System.Runtime.InteropServices;

namespace PlatformInvoke
{
   class PI_MessageBox
   {
      public const string strApp = "MessageBox";

      // First approach – flags as integers
      [DllImport("coredll.dll")]
      public static extern Int32
      MessageBox(IntPtr hWnd, string lpText, string lpCaption,
         int uType);

      public const int MB_OK = 0x00000000;
      public const int MB_OKCANCEL = 0x00000001;
      public const int MB_ABORTRETRYIGNORE = 0x00000002;
      public const int MB_YESNOCANCEL = 0x00000003;
      public const int MB_YESNO = 0x00000004;
      public const int MB_RETRYCANCEL = 0x00000005;
      public const int MB_ICONHAND = 0x00000010;
      public const int MB_ICONQUESTION = 0x00000020;
      public const int MB_ICONEXCLAMATION = 0x00000030;
      public const int MB_ICONASTERISK = 0x00000040;
      public const int MB_ICONWARNING = MB_ICONEXCLAMATION;
      public const int MB_ICONERROR = MB_ICONHAND;
      public const int MB_ICONINFORMATION = MB_ICONASTERISK;
      public const int MB_ICONSTOP = MB_ICONHAND;
      public const int MB_DEFBUTTON1 = 0x00000000;
```

Download at WoweBook.Com


```
        public const int MB_DEFBUTTON2 = 0x00000100;
        public const int MB_DEFBUTTON3 = 0x00000200;
        public const int MB_DEFBUTTON4 = 0x00000300;
        public const int MB_APPLMODAL = 0x00000000;
        public const int MB_SETFOREGROUND = 0x00010000;
        public const int MB_TOPMOST = 0x00040000;

        // Second approach – flag values in an enumeration
        [DllImport("coredll.dll")]
        public static extern Int32
        MessageBoxW(IntPtr hWnd, string lpText, string lpCaption,
           MB uType);

        public enum MB
        {
           MB_OK = 0x00000000,
           MB_OKCANCEL = 0x00000001,
           MB_ABORTRETRYIGNORE = 0x00000002,
           MB_YESNOCANCEL = 0x00000003,
           MB_YESNO = 0x00000004,
           MB_RETRYCANCEL = 0x00000005,
           MB_ICONHAND = 0x00000010,
           MB_ICONQUESTION = 0x00000020,
           MB_ICONEXCLAMATION = 0x00000030,
           MB_ICONASTERISK = 0x00000040,
           MB_ICONWARNING = MB_ICONEXCLAMATION,
           MB_ICONERROR = MB_ICONHAND,
           MB_ICONINFORMATION = MB_ICONASTERISK,
           MB_ICONSTOP = MB_ICONHAND,
           MB_DEFBUTTON1 = 0x00000000,
           MB_DEFBUTTON2 = 0x00000100,
           MB_DEFBUTTON3 = 0x00000200,
           MB_DEFBUTTON4 = 0x00000300,
           MB_APPLMODAL = 0x00000000,
           MB_SETFOREGROUND = 0x00010000,
           MB_TOPMOST = 0x00040000
        }

        static void Main()
        {
           // Flags defined as set of const values.
           MessageBox(IntPtr.Zero, "One way to define flags",
              strApp, MB_OK | MB_TOPMOST);

           // Flags defined as members of enum
           MessageBox(IntPtr.Zero, "Another way to define flags",
              strApp, MB.MB_OK | MB.MB_TOPMOST);
        }
     }
}
```

Download at WoweBook.Com

3.2.3 Function Return Values

As in managed code, native functions are not required to return a value. C# programmers specify a `void` return type, just as in C and C++.

A native function can return 32-bit and narrower integers. This means you can return 1-byte integers, 2-byte integers, and 4-byte integers; 64-bit integers, however, cannot be used as return types. As far as using floating-point values goes, neither single nor double floating-point values can be used as the function's return value (although you can accomplish the same thing by passing a by-reference parameter). The C# types that can be used as return values are listed here:[4]

- `bool`
- `byte`
- `sbyte`
- `char`
- `short`
- `ushort`
- `int`
- `uint`
- `IntPtr` (a .NET type, with no C# alias)

When calling native code that returns a data pointer, you declare the function to return an `IntPtr` (which is a 32-bit integer). Once you get the `IntPtr` back from a called function, you next call various members of the `Marshal` class to copy the values to managed objects. You can find examples of using `IntPtr` and `Marshal` later in this chapter.

3.2.4 Getting Started: C-Style Function Declarations

To create a .NET-compatible P/Invoke declaration for a native function, you need a set of Win32 declarations: a function prototype, flag or constant values, and data structure definitions. Where do you get these declarations? There are two choices: the include files and the online documentation.

4. This information also appears in Table 3.3 later in this chapter, which summarizes the use of managed types as parameters when calling unmanaged functions.

Download at WoweBook.Com

Microsoft defines the Win32 API in a set of include files. In the very beginning of Windows programming, all declarations were in a single file: windows.h. That file grew and grew until it had to be split into numerous smaller files—but windows.h is still a good starting point for finding which files are included (with very deep nesting of include files, it is easy to get confused). These include files are available in several different products, most notably with the Microsoft language compilers. Be aware that there are Windows CE–specific versions of the include files (and even platform-specific versions, so that the Pocket PC 2003 has a different set of include files from the Smartphone 2003). A simple solution is to get your hands on the SDK for your platform, which should have the include files you need.[5]

In a few cases, the include files do not have the best declarations, so we rely on the online documentation. Why? Because in some include files the parameters for function declarations do not include a sample parameter name, only the type information. If that were the case with the `MessageBox` function, the declaration might be as follows:

```
// C++ Win32 declaration
int WINAPI MessageBoxW(HWND, LPCWSTR, LPCWSTR, UINT);
```

Without the parameter names, it becomes harder to know the purpose of the middle two parameters. But when you know the parameter names are `lpText` and `lpCaption`, it makes it easier to guess the difference between these two parameters. That is why we sometimes use the online documentation instead of the include files for some of the declarations we use for creating P/Invoke wrappers.

Setting up a P/Invoke call requires a lot of work—you need to create a P/Invoke wrapper for the function, and often you need to include definitions for flag values or other supporting constants. And for some function calls, you might also need to define one or more data structures. If you have only one or two declarations to deal with, it can be fun. If you have more than that, however, it can get boring very quickly. To find out how to access a tool that does the conversion automatically, read the sidebar about the P/Invoke Wizard.

---

5. Developers using Windows Mobile devices, including the Pocket PC 2003 and Smartphone 2003, can find the SDK for these devices on the Web by browsing to www.pocketpc.com. Then follow the links for developers and then downloads.

Download at WoweBook.Com

P/Invoke Wizard

One of us created the P/Invoke Wizard out of necessity. Faced with the tedious task of converting 500 C-style declarations into both C# and Visual Basic .NET, he decided he needed a tool to do the work for him. From that effort, the P/Invoke Wizard was born. The Paul Yao Company then started to sell the tool on its Web site, where it garnered a modest amount of interest. When we sent a copy of the tool to Mike Hall, a friend who works at Microsoft, Mike suggested we find a way to make it available to a wider audience. As a result of Mike's suggestion, a shareware version was created. You can download the tool from the company's Web site[a] and use it to explore the automatic creation of P/Invoke wrappers from C-language declarations.

The P/Invoke Wizard accepts C declarations for functions, enumerations, `#define` constant values, and data structures. You can type these declarations into the program's Edit window, open an include file when you have many wrappers to create, or paste them from the Clipboard. The tool returns a set of .NET-compatible declarations, in your choice of either C# or Visual Basic .NET. Figure 3.2 shows the P/Invoke Wizard with Win32 declarations in the upper input window and the resulting .NET-compatible P/Invoke declarations in the lower output window. For developers creating declarations for the desktop .NET Framework, the tool allows you to enable code generation for P/Invoke features that are not currently supported in the .NET Compact Framework.

The P/Invoke Wizard can open and convert a complete include file. You could, for example, open include files such as winbase.h, wingdi.h, and winuser.h. This tool provides a quick and easy way to convert a large number of functions. We tested the P/Invoke Wizard on all the Win32 include (*.h) files. In addition, we have gotten feedback from users who have used the P/Invoke Wizard with include files from third-party DLLs.

The goal of the P/Invoke Wizard is to save time. We wanted to make it easy for programmers to handle the mind-numbing simple cases, which make up 99% of the work involved with creating P/Invoke declarations. This gives you more time to study and fix the other 1% that require refinement of the output generated by our wizard. You may need to handle some of the cases by hand because the P/Invoke Wizard is not a full-blown compiler, meaning it does not parse the full, rich syntax of the C

Download at WoweBook.Com

and C++ programming languages. It can, however, handle declarations found within include files. We know it works well with the Win32 include files because we used those files in our testing. It can read and create usable declarations for both desktop and Windows CE versions of winbase.h, wingdi.h, winuser.h, and the other include files that make up the Win32 API.

a. To download the P/Invoke Wizard, visit www.paulyao.com/pinvoke.

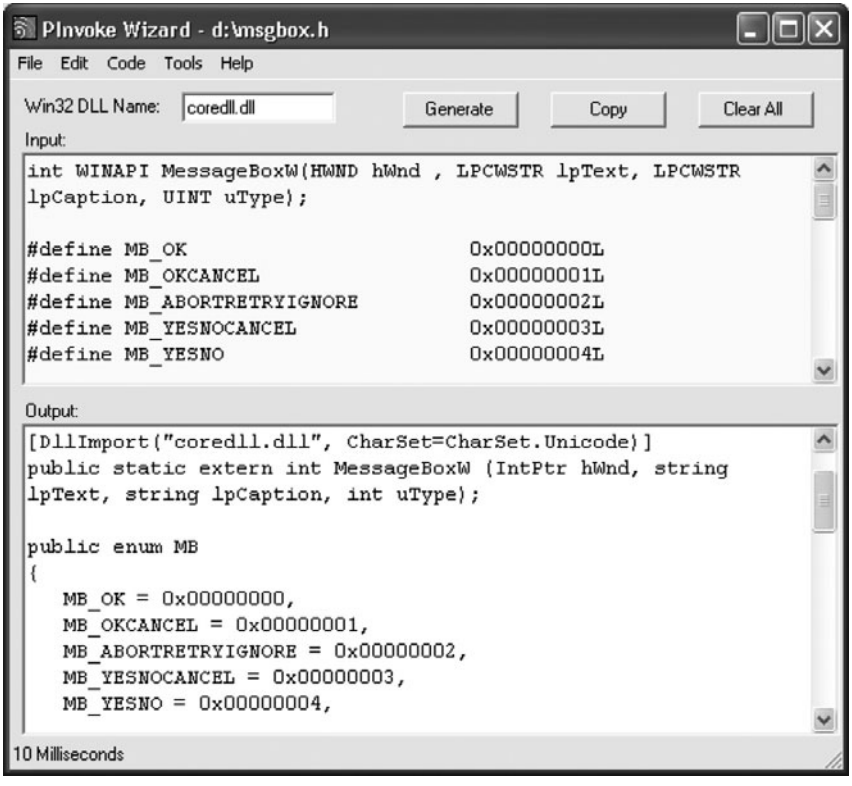


FIGURE 3.2: The P/Invoke Wizard Converting the `MessageBox` Function to C#

3.3 Supported P/Invoke Function Parameters

After you understand the basic P/Invoke syntax, the next step to using P/Invoke is to understand parameter passing. Simple data types—meaning

Download at WoweBook.Com

value types—require no special work on your part. Also, strings work nicely as parameters.

Things get a little more involved with data structures. Structures with value types are the simplest to pass. Structures with reference types—meaning arrays or objects—take a bit more effort on your part but can also be passed.

3.3.1 The Limits of Parameter Passing

There are limits to the built-in support that the .NET Compact Framework provides for parameters to a native function.

- Passing a variable number of parameters is not supported.
- Only the `Cdecl` calling convention is supported and not the other types (`StdCall`, `ThisCall`, `FastCall`).
- Arrays of reference types (objects, strings) are not supported.
- Strings are always marshaled in Unicode.

Remember that our focus is on the .NET Compact Framework. In general, the desktop .NET Framework has richer support for P/Invoke than the .NET Compact Framework does. Given a bit of diligence, however, we have found that we can pass just about all data types to native code—it just takes more work in the .NET Compact Framework. Most other data pointers can be passed into (and, as appropriate, back out of) managed code using a combination of the `IntPtr` data types and members of the `Marshal` class. The `FindMemoryCard` example later in this chapter shows how this is done.

3.3.2 Simple Data Types

Simple data types (value types that are 32 bits and smaller) can be passed as parameters. You can pass such values either by value or by reference. A by-value parameter makes a one-way trip into the called function, while by-reference parameters provide a pointer to the called method so that it can access the original input data (or the output buffer). Such parameters are either output parameters or input-output parameters that allow the data to take a round trip into the called function, with any changes to the data returned to the caller.

The movement of function parameters and return values over the boundary between managed code and native code is known as **marshaling.** Marshaling is easiest when managed and native representations of

Download at WoweBook.Com

Three Uses of Function Parameters

Function parameters are used in three ways: for input only, for output only, and for both input and output. To show the use of parameters, Microsoft has standard notation in its documentation. Input-only parameters are marked `[in]`; output-only parameters are marked `[out]`; and parameters used for both input and output—a rare usage—are marked `[in][out]`.

data are the same, a set that includes most integer types: signed and unsigned 8-bit, 16-bit, and 32-bit integers. Simple types can be passed either by value or by reference. In C#, a by-reference parameter is marked with the `ref` keyword. The keyword is used in two different places: in the function definition and in function calls. When `ref` does not appear, a parameter is a by-value parameter.

Some types can be passed only by reference, including 64-bit integers, single-precision floating-point numbers, and double-precision floating-point numbers. The Win32 code you call must be written to receive a pointer to these types.

Then there are the value types for which there is no built-in marshaling support. `DateTime` and `Decimal` values fall into this category. These types—like all data types—can be marshaled, but extra work is required. Table 3.3 summarizes the .NET Compact Framework's built-in support for marshaling common .NET value types. Table 3.3 also shows whether a given type can be used for a valid return value. All 32-bit (4-byte) and narrower integers can be used as return values. If a data type cannot be used as a return value and if you need to receive data of that type from the called function, you might have to redefine your native-code function to retrieve the return value as a by-reference parameter.

Most items in Table 3.3 are self-explanatory and correspond to data types that exist across a wide range of programming environments. A few worth some extra mention are discussed further in the following subsections.

3.3.2.1 *Boolean*

Many Win32 functions return a Boolean value, as defined by the Win32 `BOOL` type. You might be tempted to use the .NET `boolean` type when building P/Invoke declarations because the names are the same. But you

Download at WoweBook.Com

TABLE 3.3: Built-in Support for Marshaling Common .NET Data Types

.NET Compact Framework Type	Size (bytes)	C# Alias	VB Alias	Win32 Type (by-value)	Win32 Type (by-reference)	Valid As a Return Value
Boolean	1	bool	Boolean	BYTE	BYTE *	Yes
Byte	1	byte	Byte	BYTE	BYTE *	Yes
SByte	1	sbyte		CHAR	CHAR *	Yes
Char	2	char	Char	WCHAR	WCHAR *	Yes
Int16	2	short	Short	SHORT	SHORT *	Yes
Uint16	2	ushort		UINT16	UINT16 *	Yes
Int32	4	int	Integer	INT32	INT32 *	Yes
UInt32	4	uint		UINT32	UINT32 *	Yes
IntPtr	4			HANDLE	HANDLE *	Yes
Single	4	float	Single	(No support)	FLOAT *	
Int64	8	long	Long	(No support)	INT64 *	
UInt64	8	ulong		(No support)	UINT64 *	
Double	8	double	Double	(No support)	DOUBLE *	
String	Varies	string	String	WCHAR *	(No support)	
StringBuilder	Varies			WCHAR *	(No support)	
DateTime				(No support)	(No support)	
Decimal	12	decimal	Decimal	(No support)	(No support)	

From the Library of Lee Bogdanoff

Download at WoweBook.Com

would not get the right results because the managed-code type is one byte. When marshaling, you must use a signed (32-bit) integer for the Win32 `BOOL` type. For example, the Win32 `IsWindow` function, which tells you whether a window handle is valid, is defined as follows:

```
BOOL IsWindow(HWND hWnd);
```

The equivalent P/Invoke declaration is:

```
[DllImport("coredll.dll")]
public static extern int IsWindow(IntPtr hWnd);
```

3.3.2.2 *Signed versus Unsigned Integers*

Table 3.3 lists both signed and unsigned integers. We recommend that you avoid using unsigned types when building P/Invoke declarations. The reason is that the CLI supports only signed integers (except for the 1-byte integer, in which case only the unsigned 1-byte integer is supported).

Visual Basic, for example, does not support unsigned integers. You may have noticed this fact from Table 3.3, which omits Visual Basic aliases for such aliases as `UInt32` and `UInt16`, among others, because such aliases do not exist. Avoiding unsigned integers helps promote interoperability between C# and Visual Basic programs.

This is a simple recommendation, but if you examine the Win32 API, you see how widespread unsigned integers are. Many Win32 functions and data structures contain 16-bit unsigned integers (the Win32 `WORD` type) and 32-bit unsigned integers (the Win32 `DWORD` type). In other words, in making the move from a Win32 type to a CLI-compatible type, you may be forced to convert unsigned integers to signed integers. You may need to use the C# `unchecked` keyword to avoid overflow errors, as detailed in the sidebar, The C# `unchecked` Keyword.

3.3.2.3 *The C# `long` Type*

Programmers who are coming to C# programming with a background in C or C++ should note that a `long` in C# is different from a `long` in C/C++. In C#, `long` is a 64-bit integer, while `long` in C/C++ is a 32-bit integer. A 32-bit integer in C# is an `int`.

3.3.3 Passing Parameters by Value versus by Reference

An important part of every P/Invoke declaration involves deciding whether to pass a parameter by value or by reference. These terms are specific to

Download at WoweBook.Com

.NET programming, but the underlying concepts are not new. The C programming language, which has been around since the late 1960s, supports both types of parameter passing. The following C function passes one by-value integer (`i`) and one by-reference integer (`pi`):

```
// C Code – passing parameters by value and by reference
int Add(int i, int * pi)
{
   return(i + *pi);
}
```

The preceding code is easy to understand for C programmers. What might be new to you—if you are new to C# programming—is how this translates into a valid P/Invoke interface. Here is what the P/Invoke Wizard creates for us:

```
// Add function w/ by-value & by-reference parameters
[DllImport("mydll.dll")]
public static extern Int32 Add(int i, ref int pi);
```

The new element is the keyword `ref`, which says the function call should pass a pointer to the value. You also use this keyword in the function call:

```
int iFirst = 10;
int iSecond = 20;
int iThird = Add(iFirst, ref iSecond);
```

If you forget to use the `ref` keyword, the compiler complains at program build time. As in other cases, here the compiler's syntax checker is the best teacher a programmer could have.

An alternative to the `ref` keyword is an explicit pointer. We do not recommend this, but we mention it for the sake of completeness. The pointer syntax is identical to the pointer syntax that C and C++ programmers know:

```
// Add function with a pointer – requires unsafe keyword
[DllImport("PlaceHolder.dll")]
public static extern unsafe Int32 Add(Int32 i, Int32 * pi);
```

When you compare this declaration with the earlier one, you see two changes: (1) the addition of the `unsafe` keyword and (2) the * in place of `ref`. A third difference, not visible in this code fragment, is best described by the error message generated by the compiler:

Download at WoweBook.Com


```
Unsafe code may only appear if compiling with /unsafe
```

In other words, the `unsafe` keyword requires the `/unsafe` compiler switch. The compiler will create an explicit pointer for you, but only if you are willing to mark your executable as unsafe. It's better, we think, to use the implicit pointer created by the `ref` keyword than to use explicit pointers in your C# code.

The C# `unchecked` Keyword

In most cases, a signed integer in a P/Invoke declaration works just as well as an unsigned integer declaration. C# programmers run into trouble, however, when trying to assign values greater than 0x80000000 to a signed 32-bit integer. The reason? With signed integers, the high-order bit is used as the sign bit, which means signed integers cannot hold the magnitude of values that unsigned integers can. Or to put it another way, the range of signed integers includes both positive and negative numbers, but the range for unsigned integers is in positive numbers only. To account for this difference in the same number of bits, signed numbers cannot be any larger than 0x80000000—which accounts for the fact that numbers larger than this cause an overflow exception. For example, the compiler does not like the following line of code:

```
public const int E_FAIL = 0x80004005; // Compiler dislikes this!
```

The compiler can be placated by casting this value, but at runtime you get an exception complaining about an arithmetic overflow. The solution? Use the unchecked operator to tell the C# compiler not to check for overflows:

```
// Compiler is much happier with this:
public const int E_FAIL = unchecked((int)0x80004005);
```

Our recommendation to avoid unsigned types adds a small complication for C# programmers of having to add the unchecked keyword. What is the impact on Visual Basic programmers of our recommendation? Visual Basic is not as fussy as C#, as demonstrated by the following Visual Basic statement, which causes neither compile-time errors nor runtime errors:

```
Public Const E_FAIL As Integer = &H80004005
```

Continues

Download at WoweBook.Com

Whichever language you use, we still stand by our recommendation: use only signed types for all P/Invoke declarations since signed types enable the greatest amount of interoperability with the different .NET languages and with the .NET runtime itself.

3.3.4 Passing String Parameters by Value

An interesting aspect of strings is that they are passed by value and not by reference. For C and C++ programmers, this is a somewhat counterintuitive aspect of P/Invoke. A string object contains a pointer to a buffer that contains a character array. When you want to pass a pointer to other types, such as an integer, you make them by-reference parameters. From the .NET perspective, however, the value of the string is the pointer—and so we declare string parameters as by-value parameters. As we discuss later in this chapter, arrays passed as parameters are like strings in that they are also declared as by-value parameters.

On the subject of strings, you may have noticed two kinds of strings in Table 3.3: `String` and `StringBuilder`. The `String` class is more efficient with a string that does not change (which Microsoft refers to as *immutable*), and the `StringBuilder` class is more efficient when performing a series of operations on a single string. For example, if you were going to concatenate many strings together, the `StringBuilder` class is likely to be more efficient. You could use the `String` class, but each operation would create a new string—potentially adding memory manager overhead.

As far as passing regular strings and `StringBuilder` strings as P/Invoke parameters, the .NET documentation suggests using the `String` class for one-way trips into managed code and using the `StringBuilder` class for two-way trips into native code and back again. In fact, either can be used for one-way or round-trip calls, but for two-way strings—that is, strings you send into managed code that may have been changed—the `StringBuilder` class is more efficient from the perspective of memory management.

3.3.5 Structures

Support for structures passed as parameters using P/Invoke wrappers plays an important role in letting you access operating system functions because the Win32 API contains definitions for more than 2,000 structures. Many Win32 API functions require the use of pointers to structures as parameters.

Download at WoweBook.Com

Win32 Tip: Caller Always Allocates the Structure for Function Calls

If you are new to Win32 programming, you should be aware that when passing a pointer to a structure to a Win32 function, the caller always[a] allocates the structure. In addition, many Win32 structures contain a field that you must initialize to the structure size (in bytes) before passing that structure to the Win32 function.

a. We say "always," but there are two or three exceptions. However, they are so rare that we are confident in recommending that you always allocate whatever structures are needed by Win32 functions that you may call.

One useful Win32 function that accepts a structure is `CreateProcess`, which you call to start running a program. This function takes a pointer to the `PROCESS_INFORMATION` structure, which receives a pair of handles when a new process is created: a process handle and a thread handle. When creating a new process, you must disconnect both of these handles by calling the `CloseHandle` function to allow the system to properly clean up when the process terminates. Listing 3.2 shows a code fragment from the Start program, with the required P/Invoke declarations to create a new process. This program starts a new process when the user clicks on a button.

Listing 3.2: Passing a Structure to a Win32 Native Function

```
static string strAppName = "Start";

[DllImport("coredll.dll")]
public static extern int
CreateProcess(string pszImageName, string pszCmdLine,
int Res1, int Res2, int Res3, CREATE_FLAG fdwCreate,
int Res4, int Res5, int Res6,
ref PROCESS_INFORMATION pProcInfo);

[DllImport("coredll.dll")]
public static extern int
CreateProcess(string pszImageName, int pszEmptyPath,
int Res1, int Res2, int Res3, CREATE_FLAG fdwCreate,
int Res4, int Res5, int Res6,
ref PROCESS_INFORMATION pProcInfo);

[DllImport("coredll.dll")]
public static extern int CloseHandle(IntPtr hObject);
```

Download at WoweBook.Com


```
[DllImport("coredll.dll")]
public static extern int GetLastError();

public struct PROCESS_INFORMATION
{
   public IntPtr hProcess;
   public IntPtr hThread;
   public int dwProcessId;
   public int dwThreadId;
};

public enum CREATE_FLAG
{
   CREATE_SUSPENDED = 4,
   CREATE_NEW_CONSOLE = 0x10,
}

private void
cmdStart_Click(object sender, System.EventArgs e)
{
   string strPath = textPath.Text;
   PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
   int bOk = CreateProcess(strPath, 0, 0, 0, 0, 0, 0, 0, 0,
   ref pi);
   if (bOk > 0)
   {
      CloseHandle(pi.hProcess);
      CloseHandle(pi.hThread);
   }
   else
   {
      MessageBox.Show("CreateProcess Failed", strAppName);
   }
}
```

3.3.5.1 *Example:* `MemoryStatus`

To pass a managed-code structure to native code, you declare the structure parameter as a by-reference parameter. An example of a function that takes a pointer to a structure as a parameter is the `GlobalMemoryStatus` function, a Win32 function that returns information about system memory. The declaration for this Win32 function takes the following form:

```
void GlobalMemoryStatus(LPMEMORYSTATUS lpBuffer);
```

Here is the equivalent P/Invoke declaration:

```
[DllImport("coredll.dll")]
public static extern
void GlobalMemoryStatus(ref MEMORYSTATUS lpBuffer);
```

Download at WoweBook.Com

The sole parameter has the type LPMEMORYSTATUS, a pointer to a MEMORYSTATUS structure. Both of these types—the structure and the pointer to the structure—are defined as follows:

```
typedef struct _MEMORYSTATUS {
DWORD dwLength;
DWORD dwMemoryLoad;
DWORD dwTotalPhys;
DWORD dwAvailPhys;
DWORD dwTotalPageFile;
DWORD dwAvailPageFile;
DWORD dwTotalVirtual;
DWORD dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

This declaration defines a structure type named MEMORYSTATUS and a pointer type to that structure named LPMEMORYSTATUS.

Win32 Pointers and Hungarian Naming

Hungarian naming is a convention used through the definition of Win32 types, as well as by Win32 programmers when writing code. (We also use Hungarian naming in our .NET Compact Framework programming; for details, see Appendix A, Hungarian Notation for .NET Programs.) Among other things, Hungarian naming uses prefixes as reminders of type information. The LP prefix, for example, stands for "pointer." (It actually stands for "long pointer," a throwback to the days when Windows programmers used both long and short pointers on the segmented memory of early Intel x86 processors.)

The P/Invoke Wizard creates the following C# definition for the MEMORYSTATUS structure:

```
public struct MEMORYSTATUS
{
   public int dwLength;
   public int dwMemoryLoad;
   public int dwTotalPhys;
   public int dwAvailPhys;
   public int dwTotalPageFile;
   public int dwAvailPageFile;
   public int dwTotalVirtual;
   public int dwAvailVirtual;
};
```

Download at WoweBook.Com

The first element in this structure, dwLength, must be set to the size of the structure before the call to the GlobalMemoryStatus function.

After declaring this data structure, you then allocate and initialize the MEMORYSTATUS structure:

```
MEMORYSTATUS ms = new MEMORYSTATUS();
ms.dwLength = Marshal.SizeOf(ms);
```

As occurs so often for Win32 functions, we set a member of the structure—dwLength—to the size of the structure we are passing. To get the structure size, we rely on a static method in the Marshal[6] class: SizeOf. Once the dwLength field has been initialized, we can then call the GlobalMemoryStatus function. Listing 3.3 shows the complete source code for our .NET program that allocates the needed data structure, calls the Win32 function, and reports the results in a message box. An example of the output appears in Figure 3.3.

Listing 3.3: Using P/Invoke in the MemoryStatus Function to Pass Structures That Contain Simple Types

```
// Program Name: MemoryStatus.exe
//
// File Name: MemoryStatus.cpp - Displays system memory status
// by calling a Win32 function and passing a pointer to a data
// structure.
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//

using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MemoryStatus
{
   public class MemoryStatus
   {
      [DllImport("coredll.dll")]
      public static extern
      void GlobalMemoryStatus(ref MEMORYSTATUS lpBuffer);
```

6. Fully qualified name: System.Runtime.InteropServices.Marshal.

Download at WoweBook.Com


```
        public struct MEMORYSTATUS
        {
            public int dwLength;
            public int dwMemoryLoad;
            public int dwTotalPhys;
            public int dwAvailPhys;
            public int dwTotalPageFile;
            public int dwAvailPageFile;
            public int dwTotalVirtual;
            public int dwAvailVirtual;
        };
        const string CRLF = "\r\n";

        public static void Main()
        {
            MEMORYSTATUS ms = new MEMORYSTATUS();
            ms.dwLength = Marshal.SizeOf(ms);
            GlobalMemoryStatus(ref ms);

            string strAppName = "Memory Status";

            StringBuilder sbMessage = new StringBuilder();
            sbMessage.Append("Memory Load = ");
            sbMessage.Append(ms.dwMemoryLoad.ToString() + "%");
            sbMessage.Append(CRLF);
            sbMessage.Append("Total RAM = ");
            sbMessage.Append(ms.dwTotalPhys.ToString("#,##0"));
            sbMessage.Append(CRLF);
            sbMessage.Append("Avail RAM = ");
            sbMessage.Append(ms.dwAvailPhys.ToString("#,##0"));
            sbMessage.Append(CRLF);
            sbMessage.Append("Total Page = ");
            sbMessage.Append(ms.dwTotalPageFile.ToString("#,##0"));
            sbMessage.Append(CRLF);
            sbMessage.Append("Avail Page = ");
            sbMessage.Append(ms.dwAvailPageFile.ToString("#,##0"));
            sbMessage.Append(CRLF);
            sbMessage.Append("Total Virt = ");
            sbMessage.Append(ms.dwTotalVirtual.ToString("#,##0"));
            sbMessage.Append(CRLF);
            sbMessage.Append("Avail Virt = ");
            sbMessage.Append(ms.dwAvailVirtual.ToString("#,##0"));

            MessageBox.Show(sbMessage.ToString(), strAppName);
        }
    }
}
```

Download at WoweBook.Com

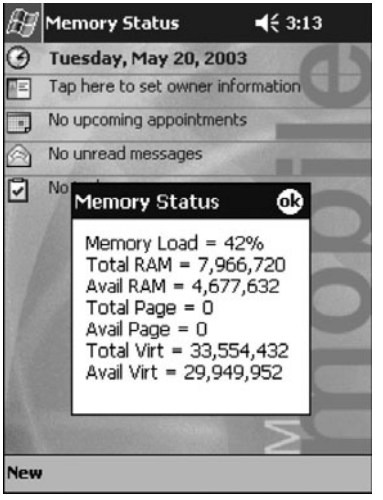


FIGURE 3.3: Results of a Call to the `GlobalMemoryStatus` Function

3.3.6 Type of Types versus Type of Parameters

As far as data types and parameter passing are concerned, there are two related but different concepts to keep straight. One concept deals with value type versus reference type. Another concept deals with passing by-value parameters versus passing by-reference parameters. The two topics are related (hence the similar names), but they are different.

Value types are often passed by value. This means a complete copy gets put on the stack, and the called method can make any change without affecting the original. But be careful! Large structures passed by value involve a lot of copying. In such a case, it might make sense to pass the value type by reference.

Reference types are often passed by value as well. But what gets passed is a reference (a pointer) to the object, not a copy of the object. Any change made by the called method is reflected in what the caller sees automatically. Reference types are very rarely passed by reference because that would mean passing a pointer to a pointer. Strings and arrays, which are both reference types, are normally by-value parameters.[7]

As we discussed earlier in this chapter, strings are a special case. When passed as parameters, they are more like value types—if the called method

7. We say "normally" because it is possible to pass a string by reference, in which case there are two levels of indirection: a pointer to a pointer to the string. An array can also be passed by reference, which creates a pointer to a pointer to the array.

Download at WoweBook.Com

makes a change to the string, that value is not reflected in the caller's copy because strings are immutable. When passing a string to a method that might change the string, you may want to pass the string by reference instead of by value. This is one of the few uses of passing a reference type by reference.

The difference between value types and reference types becomes important when you wish to pass a parameter to a method. This is important regardless of whether the method resides in managed code. In other words, you must pay attention to this distinction whether you are creating a P/Invoke declaration to call a function in native code or writing a managed-code method. Both value types and reference types can be passed as parameters, but the declarations are different for each.

3.3.6.1 *Value Types as Parameters*

Value types are allocated on the stack (what C programmers call *local variables*). Value types that are 32 bits and smaller are normally passed by value. Larger value types and structures that contain value types are often passed by reference. When a value type is passed by value, a copy of the value is put on the stack. That value makes a one-way trip into the called function, and the local copy of that value is not affected.

When a value type is passed by reference, an implicit pointer is put on the stack. The called function uses that pointer to read the contents of the variable. The called function can also change the contents of a variable, which is why we sometimes say that a by-reference parameter makes a two-way trip—down to the called function, then back again with whatever changes the function has made. The example earlier in this chapter with the `GlobalMemoryStatus` function showed the use of a by-reference parameter to allow a Win32 function to fill in the contents of a `MEMORYSTATUS` structure.

3.3.6.2 *Reference Types as Parameters*

Reference types are allocated from the process heap. A reference-type variable contains a pointer that refers to the object. The presence of a pointer in a reference type creates a funny sort of paradox: In P/Invoke declarations, you pass a pointer to a reference type by declaring a by-value parameter. To illustrate this point, here is a different definition for the `MEMORYSTATUS` structure, using a class instead of a structure:

```
public class MEMORYSTATUS
{
```

Download at WoweBook.Com


```
    public int dwLength;
    public int dwMemoryLoad;
    public int dwTotalPhys;
    public int dwAvailPhys;
    public int dwTotalPageFile;
    public int dwAvailPageFile;
    public int dwTotalVirtual;
    public int dwAvailVirtual;
};
```

Declaring `MEMORYSTATUS` as a class makes it a reference type. To accommodate this object in calls to the `GlobalMemoryStatus` function, we change the P/Invoke declaration for the function:

```
[DllImport("coredll.dll")]
public static extern
void GlobalMemoryStatus(MEMORYSTATUS lpBuffer);
```

The change involves removing the `ref` keyword so that the parameter is now a by-value parameter. We must also remove that keyword when we call the function, as shown here:

```
MEMORYSTATUS ms = new MEMORYSTATUS();
ms.dwLength = Marshal.SizeOf(ms);
GlobalMemoryStatus(ms);
```

3.3.7 Arrays

Arrays get passed by value and not by reference, which—as in the case with strings described earlier—is somewhat counterintuitive for some programmers. From the .NET perspective, however, arrays are inherently reference types, so it is the value—a pointer—that gets passed. The underlying native code sees a pointer to the first element of the array, and then it is up to the managed code to figure out what to do with that pointer.

Table 3.4 summarizes the supported array types that can be passed to Win32 functions. The table shows that P/Invoke in the .NET Compact Framework supports passing arrays of all types of integers, but not passing arrays of strings.

When dealing with character data, notice that there are several ways to pass an array of characters to native code. In the .NET Compact Framework, the `Char[]` array can be used to pass an array of Unicode characters. An `SByte[]` array can be used to pass non-Unicode characters (sometimes called ANSI but more correctly referred to as single- and multibyte characters). A `String` implicitly contains a character array, and when we pass a

Download at WoweBook.Com

TABLE 3.4: .NET Compact Framework P/Invoke Support for Arrays

.NET Type	Size (in bytes)	C# Alias	VB Alias	Passing by Value	Passing by Reference
Byte []	Varies	byte []	Byte ()	BYTE *	(No support)
Char []	Varies	char []	Char ()	WCHAR *	(No support)
SByte []	Varies	sbyte []		CHAR *	(No support)
Int16 []	Varies	short []	Short ()	INT16 *	(No support)
UInt16 []	Varies	ushort []		UINT16 *	(No support)
Int32 []	Varies	int []	Integer ()	INT32 *	(No support)
UInt32 []	Varies	uint []		UINT32 *	(No support)
String	Varies	string	String	WCHAR *	(No support)
String	Varies	string []	string ()	(No support)	(No support)

From the Library of Lee Bogdanoff

Download at WoweBook.Com

string (which we pass by value), we are passing a pointer to an array of Unicode characters. While the .NET Compact Framework supports `string` parameters, there is no support for passing an array of strings as a parameter to a native function.

3.3.7.1 *Example: Passing an Array of Integers*

Consider the following C/C++ function, which accepts a pointer to an integer array, adds up the value of each integer, and returns the sum to the caller:

```
//-------------------------------------------------------------
// AddEmUp -- Adds up numbers in an integer array
//-------------------------------------------------------------
extern "C"
int __declspec(dllexport) __stdcall
AddEmUp(int * pi, int count)
{
    int i;
    int sum = 0;
    for (i = 0; i < count; i++)
    {
        sum = sum + pi[i];
    }

    return sum;
}
```

The .NET Compact Framework code to declare and call this function appears next:

```
[DllImport("intarray.dll")]
public static extern int AddEmUp(int [] pi, int count);

private void cmdAdd_Click(object sender, System.EventArgs e)
{
      int [] i = new int [] { 1, 2, 3};
      int c = AddEmUp(i, 3);

      MessageBox.Show("AddEmUp = " + c.ToString());
}
```

Just as we pass a string by value, we also pass an array by value. That value is, of course, a pointer to the array being passed. Because we pass a pointer, the parameter makes a two-way trip. Any changes made by the native code to the contents of the array are reflected in the contents of the array that appears to our managed code.

Download at WoweBook.Com

3.4 A Sample Program: CallWin32

Support for various data types as parameters might change in a future version of the .NET Compact Framework. For this reason, we wrote a program that calls a custom Win32 DLL with each parameter in Table 3.3 both by value and by reference. If a type is not supported, a message box appears to notify you of that fact. Figure 3.4 shows our program in action.

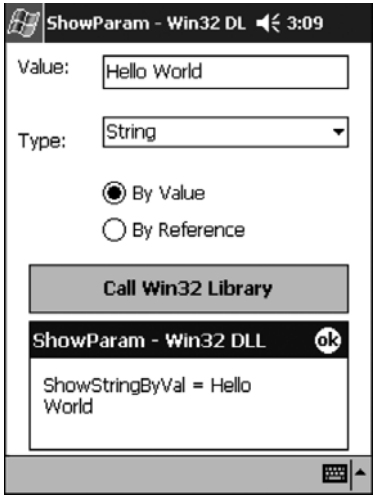


FIGURE 3.4: CallWin32 Showing a String Passed to ShowParam.dll, a Native DLL

A portion of the source to the calling program, CallWin32, appears in Listing 3.4. A fragment of the C++ source code used to create the Win32 DLL appears in the next section, in Listing 3.5. You can get the complete source code for both programs at the Web site for the book's code, www.paulyao.com/cfbook/code.

LISTING 3.4: CallWin32.cs Code That Checks for Support in Calls by Reference and by Value

```
// CallWin32.cs - Declarations and functions to call
// Win32 functions in the Win32 library ShowParam.dll
//
// File Location: \YaoDurant\CS\PlatformInvoke\CallWin32
//
// Code from _.NET Compact Framework Programming with C#_
// Authored by Paul Yao and David Durant.
//
//
```

Download at WoweBook.Com


```
using System;
using System.Text;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace CallWin32
{
   public class CallWin32
   {
      public const string DllName = "ShowParam.dll";

      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowBooleanByVal(Boolean b);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowBooleanByRef(ref Boolean b);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowByteByVal(Byte val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowByteByRef(ref Byte val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowSByteByVal(SByte val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowSByteByRef(ref SByte val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowInt16ByVal(Int16 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowInt16ByRef(ref Int16 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowUInt16ByVal(UInt16 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowUInt16ByRef(ref UInt16 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowInt32ByVal(Int32 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowInt32ByRef(ref Int32 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowUInt32ByVal(UInt32 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowUInt32ByRef(ref UInt32 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowInt64ByVal(Int64 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowInt64ByRef(ref Int64 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowUInt64ByVal(UInt64 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowUInt64ByRef(ref UInt64 val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowSingleByVal(float val);
      [DllImport(DllName, CharSet=CharSet.Unicode)]
      public static extern void ShowSingleByRef(ref float val);
```

Download at WoweBook.Com


```
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern void ShowDoubleByVal(Double val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern void ShowDoubleByRef(ref Double val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern void ShowCharByVal(Char val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern void ShowCharByRef(ref Char val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern void ShowStringByVal(String val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern void ShowStringByRef(ref String val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern
void ShowStringByVal(StringBuilder val);
[DllImport(DllName, CharSet=CharSet.Unicode)]
public static extern
void ShowStringByRef(ref StringBuilder val);

public static void
CallWin32Lib(String strVal, String strType, Boolean bByRef)
{
   //
   // User selection -- Boolean
   //
   if (strType == "Boolean")
   {
      Boolean b;
      if (strVal == "true") b = true;
      else if (strVal == "false") b = false;
      else
      {
         MessageBox.Show("Boolean needs true or false",
            frmMain.strApp);
         return;
      }
      if (bByRef)
      {
         ShowBooleanByRef(ref b);
      }
      else
      {
         ShowBooleanByVal(b);
      }
   }
   //
   // User selection -- Byte
   //
   else if (strType == "Byte")
   {
      Byte bytVal = Byte.Parse(strVal);
```

Download at WoweBook.Com


```
            if (bByRef)
            {
               ShowByteByRef(ref bytVal);
            }
            else
            {
               ShowByteByVal(bytVal);
            }
         }

         //...
         // Portion omitted from book text
         // ...

      }
   }
}
```

3.5 Writing Win32 Dynamic Link Libraries

Up to now, we have discussed how to write .NET code that calls Win32 code. But you might need to write a Win32 DLL that you can call using P/Invoke wrappers. The development tools are not the problem as much as the fact that the syntax of the C/C++ functions is a little tricky.

Installing Native DLLs in Windows CE

Windows CE contains a large number of native DLLs, which you can access from the .NET Compact Framework by using P/Invoke. At times, you will want to call a DLL that is not already part of Windows CE. Whether you want to test a new DLL on a new smart device or test the DLL in the emulator, Visual Studio can help. You can add a native DLL to a .NET Compact Framework project, and Visual Studio then copies the DLL to your smart device (or to the emulator). Here are the steps you follow to make this happen.

Open the Visual Studio Solution Explorer. In the Solution Explorer, summon a pop-up menu with a right-click on the project name. Select the Add | Add Existing Item menu item and then browse the file system for your DLL filename. Visual Studio copies the target DLL to the project directory. When working with a DLL that is changing, we add a post-build entry in the build project to copy the target DLL to the managed code's project directory. You can see an example of this post-build copying by opening the ShowParam.dll project.

Download at WoweBook.Com

3.5.1 Declaring C++ Functions in a DLL

One challenge to writing your own native DLLs is that you can use two languages: C and C++. Each language has its issues, but to simplify things we are going to assume you are using C++. The compiler uses the file extension to determine what language you are using; an extension of .c means C language, and an extension of .cpp means C++. When porting older, C-language source code, change the filename extension and rebuild to make your code conform to the declarations we describe here.

We should also mention that our focus here is on declaring public, exported DLL functions. By default, functions in a DLL are private. UNIX programmers may notice that this differs from how functions are handled in UNIX shared libraries, where all functions are public. In all versions of Windows, a function in a DLL is private unless it is exported. Only exported functions are visible to the outside world, which in the present context means the managed code of .NET Compact Framework applications and .NET Compact Framework DLLs.

The best way to understand the details of function declarations is to study an example. The following function, `ShowInt32ByVal`, was taken from the `ShowParam` example. It shows all of the elements needed to declare an exported C++ function in a DLL.

```
extern "C"  __declspec(dllexport)
void WINAPI ShowInt32ByVal(LONG val)
{
    WCHAR wch[128];
    wsprintf(wch, L"ShowInt32ByVal = %d", val);
    MessageBoxW(NULL, wch, pLibName, 0);
}
```

The `extern "C"` portion of the first line tells the C++ compiler to turn off **name mangling** (sometimes known as *name decorating*), which enables the overloading of function names in your C++ programs. While name mangling is helpful when writing C++ code called by other C++ code, it gets in the way of writing C++ code that can be called from managed code. The `ShowInt32ByVal` function, for example, is renamed to `?ShowInt32ByVal@@YAXJ@Z` without the `extern` declaration (underscoring why this is called *name mangling*).

The `__declspec(dllexport)` portion of the first line tells the C++ compiler to export a function. This is needed because by default all functions are private and hidden. Exporting a function makes it visible—and therefore callable—from outside the DLL.

Download at WoweBook.Com

The WINAPI declaration is also needed, but it hides an important element: the calling convention. The calling convention is important because if you use the wrong one, the function call is guaranteed to fail. It is a frustrating experience to spin your wheels trying to figure out why a function has failed, only to discover that you are using the wrong calling convention.

The calling convention describes how the compiler passes parameters between functions and also who cleans up the stack after a function call. In older Windows code, you sometimes see the PASCAL keyword as a reminder of the calling convention used in 16-bit Windows. (Some people thought this meant Microsoft had implemented Windows in Pascal—an odd idea because Borland, an archrival to Microsoft, had the better Pascal compiler.) The two calling conventions that are interesting to Win32 programmers are represented by the declarations CDECL and STDCALL (portable aliases for the compiler's native __cdecl and __stdcall declarations).

Here is where the desktop and the device diverge. The standard for Win32 function calls on the desktop is the STDCALL calling convention, whereas on Windows CE the standard calling convention is CDECL. To allow for the creation of Win32 code that is portable between these two environments, Microsoft provides the WINAPI calling convention declaration. On the desktop, this is defined as STDCALL, and under Windows CE that declaration is defined as CDECL.

Listing 3.5 contains the source code for ShowParam.cpp, the DLL called in the CallWin32 sample presented earlier in this chapter.

Listing 3.5: ShowParam.cpp Code That Displays Values Called by Reference and by Value

```
// ShowParam.cpp - Display various .NET data types both
// by reference and by value.
//
// File Location: \YaoDurant\CPP\PlatformInvoke\ShowParam
//
// Code from _.NET Compact Framework Programming with C#_
// Authored by Paul Yao and David Durant.
//

#include "stdafx.h"
#include "showparam.h"

LPWSTR pLibName = L"ShowParam - Win32 DLL";

BOOL APIENTRY
DllMain(HANDLE hModule,
```

Download at WoweBook.Com


```
        DWORD  ul_reason_for_call,
        LPVOID lpReserved)
{
    return TRUE;
}

//
// ShowBooleanByVal - Display Boolean value passed by value.
//
extern "C" __declspec(dllexport)
void WINAPI ShowBooleanByVal(BYTE val)
{
    WCHAR wch[128];
    wsprintf(wch, L"BoolByVal = %d", val);
    MessageBoxW(NULL, wch, pLibName, 0);
}

//
// ShowBooleanByRef - Display Boolean value passed by reference.
//
extern "C" __declspec(dllexport)
void WINAPI ShowBooleanByRef(BYTE * val)
{
    WCHAR wch[128];
    wsprintf(wch, L"BoolByRef = %d", *val);
    MessageBoxW(NULL, wch, pLibName, 0);
}

//
// ShowByteByVal - Display Byte value passed by value.
//
extern "C" __declspec(dllexport)
void WINAPI ShowByteByVal(BYTE val)
{
    WCHAR wch[128];
    wsprintf(wch, L"ShowByteByVal = %d", val);
    MessageBoxW(NULL, wch, pLibName, 0);
}

//
// ShowByteByRef - Display Byte value passed by reference.
//
extern "C" __declspec(dllexport)
void WINAPI ShowByteByRef(BYTE * val)
{
    WCHAR wch[128];
    wsprintf(wch, L"ShowByteByRef = %d", *val);
    MessageBoxW(NULL, wch, pLibName, 0);
}
```

Download at WoweBook.Com


```
// ...
// Portion omitted from book text
// ...
```

3.6 Manual P/Invoke Parameter Passing

As we have shown, the built-in support for calling into native code in the .NET Compact Framework is reasonable, but it has limits. You hit those limits when you deal with pointers to structures—especially when a structure contains reference types or arrays. You also hit those limits when you call functions that return linked lists, arrays of arbitrary size, or any of a dozen other cases.

To address these more complex parameter types, you need a more sophisticated way to pass parameters. These techniques require more work because you control specific details of how data gets sent to and retrieved from function calls that cross the managed/native boundary. There are two elements to these techniques: the `IntPtr` data type and the `Marshal` class.

Previously we used `IntPtr` for handles, an artifact of the Win32 API used by the operating system as "magic" numbers to identify system objects. `IntPtr` provides another form of magic: holding native pointers. An `IntPtr` variable can be used to hold pointer, the runtime does not allow pointer arithmetic with an `IntPtr` variable; for that you must convert to an integer type. This data type is important and serves to bridge the gap between the managed and native worlds.

As mentioned, the other element needed is the `Marshal` class. This class contains a variety of static methods. Some of these methods copy data from managed data into native memory, while other methods create new managed objects from a native memory pointer. The `Marshal` class resides in the `System.Runtime.InteropServices` namespace, which is also the home of the `DllImport` attribute used to declare P/Invoke functions.

When digging into the low-level details of marshaling, a final item you must address is the direction of each parameter. As mentioned earlier in this chapter, the online documentation for the Win32 API describes three directional attributes: `[in]`, `[out]`, and both combined as `[in][out]`. Simple data types that are passed as parameters are almost always `[in]` parameters: They get sent to and used by the called function. Parameters that pass a pointer can support any of these three directions, and you must pay careful attention when manually setting up marshaling.

Download at WoweBook.Com

We refer to built-in parameter marshaling as *automatic parameter marshaling* because almost all of the work is done for you. We refer to the more involved kind of parameter passing as *manual parameter marshaling*. (Automobile enthusiasts might notice a parallel between automatic transmission and manual transmission, although this distinction is probably lost on younger drivers who have never driven a manual transmission car or those lucky souls who never learned to drive, such as some residents of New York City whom we know!)

3.6.1 The `Marshal` Class

The `Marshal` class supports copying data from managed memory into native memory and from native memory to managed memory. If you study the online MSDN Library, you will see that the desktop .NET Framework supported methods in this class that allocate native memory and other methods that work with COM objects. None of the memory management or COM support methods are present in the .NET Compact Framework's implementation of `Marshal`. Table 3.5 summarizes some of the members of the `Marshal` class that are supported by the .NET Compact Framework.

TABLE 3.5: `Marshal` Class Members Supported by the Compact Framework

`Marshal` Member	Description
Copy between Managed and Native Memory	
`Copy()`	Copies value type arrays between managed and native memory. Supports CLI integer types, including 64-bit integers. Supports single- and double-precision floating point. Fourteen overloaded methods (seven copy to managed memory; seven copy to native memory)
Copy to Native Memory	
`StructureToPtr()`	Copies managed objects to native memory
`WriteByte()`	Writes a single byte to native memory
`WriteInt16()`	Writes two bytes to native memory
`WriteInt32()`	Writes four bytes to native memory

Continues

Download at WoweBook.Com

TABLE 3.5: `Marshal` Class Members Supported by the Compact Framework *(Continued)*

`Marshal` Member	Description
Copy to Managed Memory	
`PtrToStringUni()`	Creates a managed string from the contents of native memory
`PtrToStructure()`	Creates an object from the contents of native memory
`ReadByte()`	Reads a single byte from native memory
`ReadInt16()`	Reads two bytes from native memory
`ReadInt32()`	Reads four bytes from native memory
Informational	
`IsComObject()`	Hard-coded to return false
`SizeOf()`	Queries the native size of an object instance. Use to set structure size fields needed by some Win32 function calls
`GetLastWin32Error()`	Calls the `GetLastError` function to fetch Win32 error code
`SystemDefaultCharSize`	Read-only field for size of character in default character set (returns 2 in Compact Framework). For portability

Some of the methods in the `Marshal` class let you write to native buffers, which you can then pass as parameters to native functions. Other methods from this class let you read values out of a native buffer into managed data objects. Both writing to and reading from buffers is important because the Win32 API (along with many other C-oriented APIs) makes extensive use of buffers to communicate from a caller to a called function.

3.6.2 Copying to Native Memory

Many parameters to native functions are simple value types passed by value. Some value types are passed by reference. For both cases, the .NET Compact Framework's built-in P/Invoke support is sufficient. A by-reference value-type parameter causes a pointer to get sent to native code, which the .NET Compact Framework knows how to deal with. The .NET Compact Framework can even handle many structures, as long as the structure contains only simple value types. Such structures are sent as by-value parameters, which causes a pointer to get sent to

Download at WoweBook.Com

native code.[8] These are all the built-in cases where the .NET Compact Framework does all the work without the need to use `IntPtr` and `Marshal`.

When dealing with manual parameter passing, when the parameter direction is `[in]` or `[in][out]`, you must initialize the native memory by copying values to the memory before calling the target function. In this subsection, we talk about copying to native memory.

To get some data into native memory, you first allocate a native buffer and store the resulting pointer in an `IntPtr`. You next call a member of the `Marshal` class to copy the data to the buffer. Then you pass the pointer to the native function. The final step is to free the allocated memory.

For example, we redefine the `MessageBox` function to accept an `IntPtr` for the caption instead of `String`, as seen in an earlier example:

```
// MessageBox with an IntPtr caption
[DllImport("coredll.dll")]
public static extern int
MessageBox(int hWnd, String lpText, IntPtr lpCaption,
   UInt32 uType);
```

Notice that the `IntPtr` parameter is a by-value parameter. The code to call this function in response to a button click appears in Listing 3.6.

Listing 3.6: Calling the Native `MessageBox` Function

```
private void button1_Click(object sender, System.EventArgs e)
{
   // Create strings.
   string strCaption = "Caption";
   string strText = "Text";

   // Get count of characters in caption.
   int cch = strCaption.Length;

   // Create a character array from the string.
   char[] achCaption = new char[cch];
   strCaption.CopyTo(0, achCaption, 0, cch);

   // Allocate native buffer.
   IntPtr ipCaption = Marshal.AllocHGlobal((cch+1) *
                        Marshal.SystemDefaultCharSize);

   if (! ipCaption.Equals(IntPtr.Zero))
```

8. Passing a structure by reference generates a pointer to a pointer; this is not used often, but it can be used, for example, when an unmanaged function allocates the memory and returns an array of structures.

Download at WoweBook.Com


```
    {
        // Copy the characters to the native buffer.
        Marshal.Copy(achCaption, 0, ipCaption, cch);

        MessageBox(IntPtr.Zero, strText, ipCaption, 0);

        Marshal.FreeHGlobal(ipCaption);
    }
}
```

The code shows the four steps you always follow when using `IntPtr` and `Marshal` to send parameter data to a native function.

1. *Allocate the memory.* The memory allocation is performed by calling the `Marshal.AllocHGlobal` function, a member of the `Marshal` class.
2. *Copy data to that memory.* We copy the data to that memory by calling the `Marshal.Copy` method.
3. *Pass the pointer as a function parameter,* which in our example is the third parameter to the `MessageBox` function. Notice that we need a different P/Invoke declaration for the `MessageBox` function than we encountered before: one with an `IntPtr` as the third parameter.
4. *Free the memory,* by calling the `Marshal.FreeHGlobal` function.

We might need another step if the called function wrote any values to the buffer. The additional step—after step 3 and before step 4—would be to copy those values from the buffer to managed memory.

We do not need to do this to pass a string to a native function, of course. But this same approach works on any kind of array to handle manual parameter marshaling. A string, after all, is just an array of characters. In Windows CE, of course, that means an array of 2-byte Unicode characters.

Marshaling of the `String` Class

In the desktop .NET Framework, strings are marshaled automatically using a technique similar to the one shown here—memory is allocated and the string is copied to that memory. By contrast, automatic string marshaling in the .NET Compact Framework involves no extra memory and no copying, so it is faster and uses less memory. In the .NET Compact Framework, native functions get a pointer to the managed string's internal memory.

Download at WoweBook.Com

3.6.3 Fine-Tuning Structures with the `MarshalAs` Attribute

When calling a Win32 function with a pointer parameter, the caller allocates the buffer.[9] This is a standard throughout the Win32 API. For structures that contain simple value types, the default marshalling works find. The `FindMemoryCard` example shows how to handle one of the more common situations, namely when a character array is embedded within a structure.

3.6.3.1 *Example:* `FindMemoryCard`

Many Windows Mobile devices support removable storage, such as Compact Flash memory cards or a Secure Digital (SD) card. Even the Emulator has a form of support for removable storage, which it does by making a desktop folder appear as a storage card.

You access the storage cards using a folder name. Although one commonly used name for storage cards is \Storage Card, other variations exist on different devices and on systems with a non-English character set. To use external storage, the best practice is to query the system for the names of the external storage directories.

To determine whether a storage card slot is present, you call the following Win32 functions:

- `FindFirstFlashCard`
- `FindNextFlashCard`

These functions fill a structure named `WIN32_FIND_DATA`, which, incidentally, is the same structure that is filled by the general-purpose Win32 file system enumeration functions, `FindFirstFile` and `FindNextFile`:

```
typedef struct _WIN32_FIND_DATA {
DWORD dwFileAttributes;
FILETIME ftCreationTime;
FILETIME ftLastAccessTime;
FILETIME ftLastWriteTime;
DWORD nFileSizeHigh;
DWORD nFileSizeLow;
DWORD dwOID;
TCHAR cFileName[MAX_PATH];
} WIN32_FIND_DATA;
```

9. Win32 function parameters with double indirection—a pointer to a pointer—involve the called function allocating the buffer. This is rare because normally the caller—not the called function—allocates the buffer. Two examples of a called function allocating a buffer are the RAPI functions `CeFindAllFiles` and `CeFindAllDatabases`, discussed in Chapter 9.

Download at WoweBook.Com

The WIN32_FIND_DATA structure contains one element that prevents automatic parameter marshaling from working: the character array. To make the native data work in a managed-code way, it makes sense to define a structure such as the following:

```
public struct WIN32_FIND_DATA
{
   public int dwFileAttributes;
   public FILETIME ftCreationTime;
   public FILETIME ftLastAccessTime;
   public FILETIME ftLastWriteTime;
   public int nFileSizeHigh;
   public int nFileSizeLow;
   public int dwOID;
   public String cFileName;
};
```

The problem occurs for the very last element, which—from a native-code perspective—would contain a 4-byte pointer to a string object. But that's not what we want; we want to have an in-line character array. We can achieve that by adding a MarshalAs attribute into the data structure definition, as follows:

```
[MarshalAs(UnmanagedType.ByValTStr,SizeConst=MAX_PATH)]
public String cFileName;
```

This declaration assumes that the constant MAX_PATH has been defined somewhere, which we have done as follows:

```
const int MAX_PATH = 260;
```

Listing 3.7 shows the EnumFlash class from the FindMemoryCard sample.

Listing 3.7: The MarshalAs Attribute Making a String Appear As an In-line Array

```
public class EnumFlash
{
  public EnumFlash()
  {
     //
     // TODO: Add constructor logic here
     //
  }

  const int MAX_PATH = 260;
  const int WIN32_TRUE = 1;
```

Download at WoweBook.Com


```
public struct FILETIME
{
   public int dwLowDateTime;
   public int dwHighDateTime;
};

public struct WIN32_FIND_DATA
{
   public int dwFileAttributes;
   public FILETIME ftCreationTime;
   public FILETIME ftLastAccessTime;
   public FILETIME ftLastWriteTime;
   public int nFileSizeHigh;
   public int nFileSizeLow;
   public int dwOID;
   [MarshalAs(UnmanagedType.ByValTStr,SizeConst=MAX_PATH)]
   public String cFileName;
   // public fixed char cFileName[MAX_PATH];
};

public static int INVALID_HANDLE_VALUE = -1;

// Flash-card search functions.
[DllImport("note_prj.dll", EntryPoint = "FindFirstFlashCard")]
public static extern
IntPtr YD_FindFirstFlashCard(ref WIN32_FIND_DATA pfd);
[DllImport("note_prj.dll", EntryPoint = "FindNextFlashCard")]
public static extern
int YD_FindNextFlashCard(IntPtr hFF, ref WIN32_FIND_DATA pfd);
[DllImport("coredll.dll", EntryPoint="FindClose")]
public static extern
int YD_FindClose (IntPtr hFF);

//-------------------------------------------------
//-------------------------------------------------
public static IntPtr
FindFirstFlashCard (ref WIN32_FIND_DATA pffd)
{
   return YD_FindFirstFlashCard(ref pffd);
} // FindFirstFlashCard

//-------------------------------------------------
//-------------------------------------------------
public static bool  FindNextFlashCard (IntPtr hFF,
   ref WIN32_FIND_DATA pffd)
{
   return (YD_FindNextFlashCard(hFF, ref pffd) == WIN32_TRUE);
} // FindNextFlashCard

//-------------------------------------------------
```

Download at WoweBook.Com


```
  //-----------------------------------------------
  public static bool
  FindClose (IntPtr hFF)
  {
     return (YD_FindClose(hFF) == WIN32_TRUE);
  } // FindClose

} // class
```

3.7 Communicating between Native and Managed Code

P/Invoke supports one-way function calls, from managed code to native code. This is different from the support found in the .NET Framework, where callback functions are supported (see the upcoming section, Comparing P/Invoke Support, for more details). In this section, we cover the available mechanisms you can use for communicating the other way—from native code to managed code.

3.7.1 The `MessageWindow` Class

The .NET Compact Framework team recognized the need for native code to call managed code. To address this need, the team added the `MessageWindow` class to the .NET Compact Framework. Incidentally, this is one of only five classes included in the .NET Compact Framework but not in the desktop .NET Framework.

If you are relatively new to the Win32 development environment, read the sidebar on `MessageWindow` for some background information related to the use of `MessageWindow` when communicating between native and managed code.

`MessageWindow` as a Win32-Style Class

Win32 programmers do not write code that handles events. Rather, they write window procedures, one for each window, that process messages. Whenever a user does something such as clicking the mouse or pressing a key, a message is generated and, usually, placed in a queue. These are not MSMQ messages, and they are not placed in an MSMQ queue; this is a separate system message queue that is as old as Windows Version 1.01 on the desktop.

Download at WoweBook.Com

When the Win32 messaging system determines that it is time for a message to be processed, the messaging system delivers the message to the appropriate window procedure, that is, the `WinProc` of that window. A message contains four pieces of information, which become parameters of the window procedure: the window handle, the type of message, 16 bits of additional information, and 32 bits of additional information.

Not only does the runtime generate messages, but the application itself can also define and generate messages by calling the `SendMessage` and `PostMessage` Win32 functions. Because messages can be defined by applications, addressed to a window, and processed by the window procedure of that window, Win32 applications often use messages for communicating within an application or between applications. Messages can even be used for your choice of synchronous or asynchronous interaction: A call to the `SendMessage` function causes an in-line call of the window procedure, while a call to the `PostMessage` function places the message in the queue for subsequent processing and returns immediately to the caller.

Because applications written in native code already use Windows messages for communication, Microsoft wanted the native-to-managed-code communication to use this known method. The `MessageWindow` class is a wrapper around an empty, invisible Win32 window. Like other Win32 windows, it has a window handle. Using that window handle, native code can transmit a message into managed code by using either the Win32 `PostMessage` or the Win32 `SendMessage` function. The result of one of these calls by native code is the automatic invoking of a `MessageWindow` method, the `WndProc` method, in the managed code. Native code communicates using its familiar technique, window messages; managed code communicates using its familiar technique, invocation of methods and access properties of an object.

Using the Win32 message delivery system provides a number of benefits. First, Win32 message delivery is flexible—it can be used when the sender and recipient are on the same thread, when they are on different threads in the same process, and when they are in different processes.

A second benefit of the Win32 message delivery mechanism is that it is guaranteed to be thread-safe. Threads are the unit of scheduling in Windows CE. When two threads share anything, care must be taken to share

Download at WoweBook.Com

cooperatively. A failure to cooperate can result in deadlocks or random failures at random times. When the appropriate safeguards are in place to allow two or more threads to share things without conflict, we say the resulting code is **thread-safe.** If two threads communicate using only Win32 messages, those two threads are guaranteed to be thread-safe with respect to each other.

You must use `MessageWindow` as a base class for a custom class; it will do you no good to create an instance of a `MessageWindow` object. This is because the `MessageWindow.WndProc` method is not an event handler—it is an already defined method. Therefore, you must override it to specify the message-handling functionality you want. Like a Win32 window procedure that gets passed four parameters (a window handle, a message value, and two parameter values—`wParam` and `lParam`), `MessageWindow.WndProc` is also passed these four parameters but within a single structure that also contains a by-reference field for specifying a return value.

All calls to the `WndProc` method must be processed; that is, your derived function must call the base class function.

Here is a template for an overridden `WndProc` method:

```
protected override void WndProc(ref Message msg)
{
   switch(msg.Msg)
      {
         // Handle messages sent to this window.
      }

   // Send unhandled messages to the default handler.
   base.WndProc(ref msg);
}
```

A difference between this function and a Win32 window procedure is the parameters—a Win32 window procedure accepts four, while this function has one. That parameter is a reference to a `Message` structure that, as previously mentioned, has the four properties Win32 programmers expect to see, plus one they might not. Here are the five properties:

1. `IntPtr HWnd`
2. `int Msg`
3. `IntPtr WParam`
4. `IntPtr LParam`
5. `IntPtr Result`

Download at WoweBook.Com

The four expected properties are `HWnd`, `Msg`, `WParam`, and `LParam`. The extra property, `Result`, is for holding the result value from handling a particular message—it is, in other words, the return value.

Table 3.6 (in the following subsection) shows the steps that must be taken in managed and native code at design time and at runtime to make this communications technique work.

3.7.2 Other Ways to Communicate between Native and Managed Code

`MessageWindow` is not the only way to communicate between native and managed code. Other ways include those listed here.

- **Call and block:** Perhaps a good name for this approach would be the "bored messenger approach." The managed application calls into native code and waits for the native function to return something useful. For frequent calls or lengthy waits, a second thread is needed to avoid hanging the application's user interface. Users may get impatient and may shut down a nonresponsive application.
- **Point-to-point message queue:** This operating system feature, introduced with Windows CE .NET (Version 4.0), provides a service that in other operating systems is called an *anonymous pipe.* (For details, see the `CreateMsgQueue` function in the Win32 documentation.)
- **Mutex:** A mutex is an operating system object for communicating between threads to enforce single-threaded access to some shared object. The .NET Compact Framework's `System.Threading.Mutex` class provides a wrapper around this object.
- **Event:** An event is an operating system object that serves as an alarm clock to let one thread wake another thread. The .NET Compact Framework wraps two classes around events: `AutoResetEvent` and `ManualResetEvent`, both in the `System.Threading` namespace.

As you can see, native code can communicate with managed code in quite a few ways. Moving forward, Microsoft has announced that future versions are going to provide even richer support for bridging the gap between managed code and native code. Even when those new features are provided, the set listed in this section should still prove useful.

Download at WoweBook.Com

TABLE 3.6: Design-Time Steps and Runtime Steps for Communicating between Managed and Native Code Using the `MessageWindow` Class

	In Managed Code	In Native Code
Overview	Derive a class from `MessageWindow`, overriding `WndProc` to respond to messages received from managed code	Send messages to window created in managed code using regular Win32 message delivery functions
Initialization	(1) Create an instance of the `MessageWindow`-derived class	(2) Define a variable of type `HWND`. Create a startup routine to accept the window handle value when called from managed code
	(3) Use P/Invoke to call the startup routine defined in step 2, passing the `HWND` parameter into native code	(4) Receive and store the provided `HWND` value
Calling from native code to managed code	(6) The calls made from step 5 result in messages delivered into the managed-code window procedure	(5) Use the window handle received in step 4 to call into managed code using either `PostMessage` or `SendMessage` functions

3.8 Comparing P/Invoke Support

There are a few differences between the desktop and device support for P/Invoke. Some arise from how Windows CE itself works; these are not so much limitations as accommodations for this operating system.

3.8.1 Windows CE–Specific Differences

The first difference involves character sets, with Windows CE supporting only Unicode. Windows XP and Windows Vista, on the other hand, support both the ANSI and Unicode character sets. Each Win32 function has two different versions (at least for the functions that take a character string): one for ANSI and a second for Unicode. The ANSI version has a suffix of `A`, and the Unicode version has a suffix of `W`. For example, `MessageBoxA` takes ANSI parameters, and `MessageBoxW` accepts Unicode parameters.

Download at WoweBook.Com

Windows CE is a Unicode-only operating system.[10] In recognition of that fact, the .NET Compact Framework support for P/Invoke includes only Unicode strings.

A second Windows CE–specific difference involves function-calling conventions. On Windows CE a single calling convention is supported: `CDECL`.

3.8.2 .NET Compact Framework Implementation Details

The final set of differences from the Desktop Framework involves those that arise from the .NET Compact Framework itself.

To reduce memory usage and allow the system to run faster, strings are marshaled without copying the strings. This means native code is allowed to look at the memory inside managed objects. This is different from string marshaling on the desktop, where strings are copied before passing them into native code. In the .NET Compact Framework, on the other hand, a change to a string in native memory—whether the wrapping managed object is a `String` or a `StringBuilder`—is reflected in the managed data object.

3.9 Conclusion

Sooner or later, every programmer who develops on the .NET Compact Framework needs to drill through to the underlying native code. Whether you are accessing a native COM library or calling into a proprietary native DLL, you will be pleased to find that the available support is extensive and, with just a small amount of effort, allows you to do what you need to get the job done.

10. To be precise, Windows CE is a predominantly Unicode operating system. Several of the networking functions—those implemented in wininet.dll, for example—have both ANSI and Unicode implementations. The Windows socket library is an ANSI-only library.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

4
Data Binding to Controls

In this chapter, we examine data binding, the ability to associate non-GUI objects that hold and maintain data with the controls that present and receive that data.

4.1 Data Binding

MOST CONTROLS, especially single-item and list-based controls, display and accept data. Most data, when it is not being displayed or modified within a control, is kept in a database. And we're using the term *database* quite loosely here—loosely enough to include anything from a `List` to a SQL Server database. Because the data is kept in a database and because we need to display and modify it through the use of controls, we need a simple, consistent, property-oriented technique for moving data between our database and our controls. Data binding provides the ability to tie a control to a source of data, simplifying the movement of data within a data-centric application. Data binding specifically was developed to facilitate the movement of data between the business tier and the presentation tier of a traditional three-tiered application, as shown in Figure 4.1.

Consider a Time Tracker application; one that maintains information about projects and their associated tasks. We cannot actually go directly from a database to and from a control, but we can go from an object containing the data to and from a control. The data object might be a `List` (such as a `List` of project task `Structures`), or it might be a database-specific object (such as a `System.Data.DataView` object, which looks much like a relational database table). One benefit of data objects is that they provide

Download at WoweBook.Com

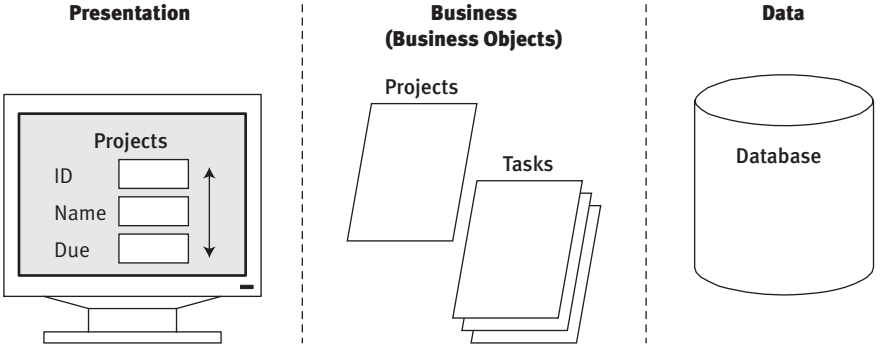


FIGURE 4.1: Application Tiers

the presentation layer independence from the database layer. A `DataTable`, for instance, has the same PMEs—properties, methods, and events, regardless of the database product that persists the data. Thus, your presentation layer code remains the same whether your database is SQL Server CE or a homegrown collection of files.

Data binding is a two-way relationship. Any change in the data object's data will be reflected in the control and vice versa. This eliminates the need to reload the control whenever the data in the data object changes. It also means that any bindable control can be used to update the data object. Even controls that do not accept user input of data (such as the `Label`, `DataGrid`, and `ComboBox` controls) provide for programmatic update of their properties—for example, as in `myLabel.Text = "ABC"`. Data binding ensures that this change occurs in the bound data object as it occurs in the control.

All visible controls are bindable, but some are better suited to data binding than others. Controls that are typically used with data binding include the `Label`, `TextBox`, `ComboBox`, `ListView`, and `DataGrid` controls. The `DataGrid` and `ListView` controls were specifically developed for use with data binding. Figure 4.2 shows project data being displayed in a `DataGrid` control without any styling applied.

Desktop developers are probably aware that in .NET 3.5, the `DataGrid` control has been supplanted by the `DataGridView` control. However, the `DataGridView` control is not available in the .NET Compact Framework, therefore leaving us with the `DataGrid` as our primary tabular display control. We discuss the `DataGrid` control in more detail later in this chapter.

Because the subject of this chapter is data binding, here we are interested only in the movement of data between the data objects and the

Download at WoweBook.Com

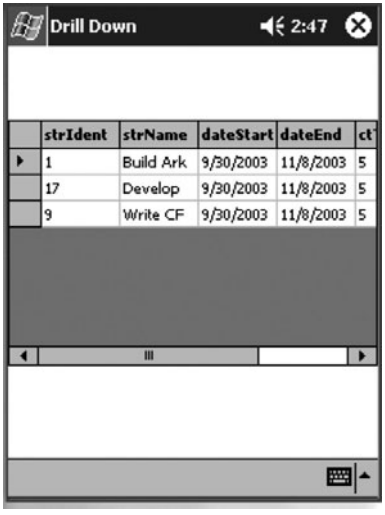


FIGURE 4.2: Project Data Displayed in a `DataGrid` Control

controls. We cover the subject of moving data between the data objects and the underlying databases in later chapters. And because data binding is such an important part of that movement, we will revisit the subject of data binding in Chapter 6, ADO.NET Programming, when we cover binding controls to `System.Data` objects; and in Chapter 7, LINQ (Language Integrated Query), where we bind the output of LINQ queries to controls. Because of the capabilities of `System.Data` objects, we cover row position and hierarchical relationships in greater detail there than here. In many respects, this chapter could be called Data Binding, while Chapters 6 and 7 could be called Son of Data Binding, among other things.

4.1.1 Data-Bindable Controls

Controls differ in their data-binding capabilities and in the method used to do the data binding. Some controls can display more data than others, some accept user input while others are only for display, and some can take advantage of *complex data binding* while others require *simple data binding.*

Consider the `System.Data.DataTable` object, shown in Figure 4.3, which holds the project data mentioned earlier. A `DataGrid` control can display the entire table, while the `ComboBox` control can display only a single column from the table, and single-valued controls such as `TextBox` and `Label` controls can display only a single data element.

Download at WoweBook.Com

Projects

strIdent	strName	dateStart	dateEnd	ctTask
1	Build Ark	3/2/2003	4/10/2003	5
17	Load Supplies	3/2/2003	4/10/2003	5
9	Load Animals	3/2/2003	4/10/2003	5

FIGURE 4.3: A `DataTable` Object That Contains Project Information

Figure 4.4 shows a `ComboBox` control displaying one column of information from a bound `DataTable` object of project data, while the `TextBox` controls display data elements of the currently selected project.

Controls that can display an entire table, or at least a column of that table, use a mechanism called **complex data binding,** which involves the control's `DataSource` property. Controls that can display only a single data element require **simple data binding,** which is done through the `BindingsCollection` property of the control. The terms are slightly confusing because, from a developer's point of view, complex binding is usually the easier of the two to program. All objects that inherit from the `Control` base class have a `BindingsCollection` property, but only objects that can display a list or a table, such as `DataGrid` or `ComboBox` controls, have a `DataSource` property.

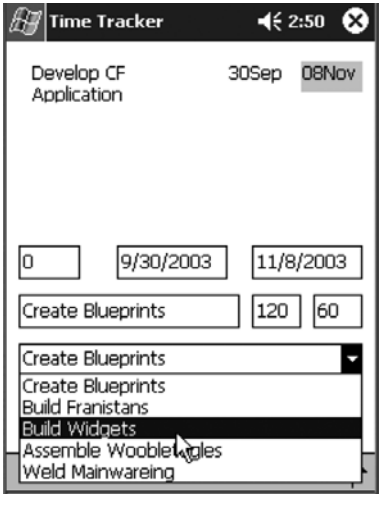


FIGURE 4.4: Different Controls Bound to the Same Data

Download at WoweBook.Com

Not only are there inevitable limitations to the data-binding capability of each control class, but there are also limitations on the data objects that can be bound to those controls in the .NET Compact Framework.

4.1.2 Data-Bindable Objects

As we mentioned earlier, data objects hold data but have no ability to display that data, so the control must display the data. Most .NET Compact Framework controls can bind only to those data objects that have a *tabular* data structure. The major exception to this rule is the `TreeView` control, which is used to display hierarchical data and whose content is normally generated by the piecemeal creation of nodes. In fact, the `TreeView` control, not having a `DataSource` property, does not lend itself to data binding at all.

Having a tabular data structure means that every item in the data object must have the same structure as every other item in that data object. A `List` of `Project` structures qualifies as being tabular because every item in it is an item of the `Project` structure data type. The same would be true for a `List` of `Customers`, for every item would be an item of the `Customer` class. If you had only one `List` object and added both `Project` and `Customer` objects to it, it would not be tabular and it would not be usable for data binding within the .NET Compact Framework environment.

This requirement for tabular structure eliminates certain `System.Data` objects. For instance, a `System.Data.DataTable` and a `System.Data.DataView` are bindable objects because of their tabular structures. However, the `System.Data.DataSet` object is *not* bindable in the .NET Compact Framework environment. This object can contain multiple `DataTable` objects, tables that can be related to each other through the use of keys. Figure 4.5 shows a diagram of a `DataSet` object that contains two `DataTable` objects. Because the `DataSet` object is hierarchical in structure rather than tabular, it cannot be bound to any control in the .NET Compact Framework environment. Either of the `DataTable` objects or any `DataView` object based on those tables could be bound to a control, but the entire `DataSet` object could not be.

To be bindable, a data object must expose its columns as *public properties,* not merely as fields. This is something to keep in mind when you define your own data structures and object classes. The `Task` structure shown in Listing 4.1 could not be used in data binding because its data is exposed as fields, not as properties.

Download at WoweBook.Com

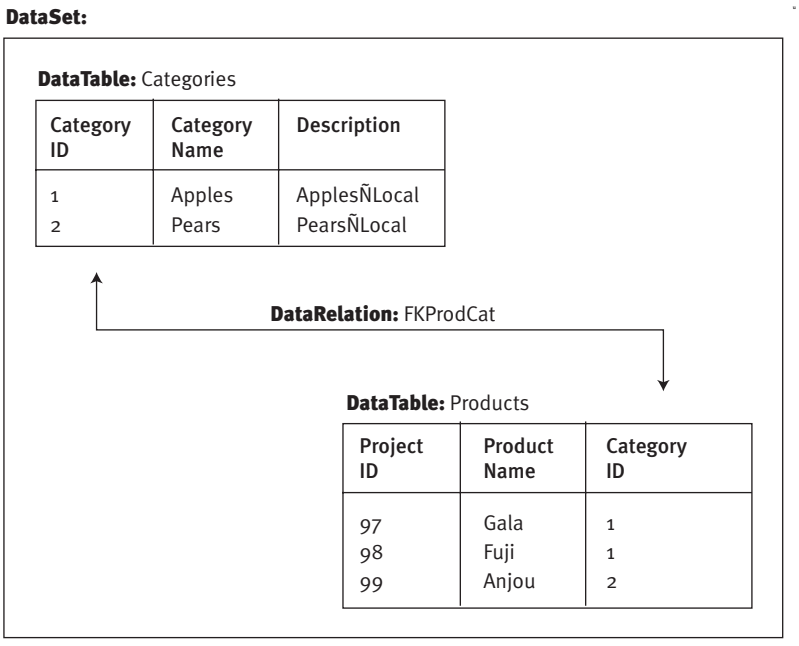


FIGURE 4.5: A `DataSet` Object That Contains Two `DataTable` Objects

LISTING 4.1: An Unbindable Version of the `Task` Structure

```
private struct Task
{
   public string  strTaskIdent;
   public string  strTaskName;
   public Date dateTaskStart;
   public Date dateTaskEnd;
   public int durTaskEstimated;
   public int durTaskActual;
   public string  strTaskComments;
   public void Task( string  strNo,  string  strName,
                     Date dateStart,  Date dateEnd,
                     int durEstimated,  int durActual,
                     string  strComments)
   {
      strTaskIdent = strNo;
      strTaskName = strName;
      dateTaskStart = dateStart;
      dateTaskEnd = dateEnd;
      durTaskEstimated = durEstimated;
      durTaskActual = durActual;
      strTaskComments = strComments;
   }
}
```

Download at WoweBook.Com

To make the `Task` structure usable for data binding, we must modify it to expose its data as public properties, as shown in Listing 4.2.

Listing 4.2: A Bindable Version of the `Task` Structure

```
public struct Task
{
   private string  m_strIdent;
   private string  m_strProjIdent;
   private string  m_strName;
   private DateTime m_dateStart;
   private DateTime m_dateEnd;
   private int m_durEstimated;
   private int m_durActual;
   private string  m_strComments;

   public string  strIdent
   {
      get
      {
         return m_strIdent;
      }
      set
      {
         m_strIdent = value;
      }
   }
   public string  strProjIdent
   {
      get
      {
         return m_strProjIdent;
      }
      set
      {
         m_strProjIdent = value;
      }
   }
   public string  strName
   {
      get
      {
         return m_strName;
      }
      set
      {
         m_strName = value;
      }
   }
   public DateTime dateStart
```

Download at WoweBook.Com


```
{
   get
   {
      return m_dateStart;
   }
   set
   {
      m_dateStart = value;
   }
}
public DateTime dateEnd
{
   get
   {
      return m_dateEnd;
   }
   set
   {
      m_dateEnd = value;
   }
}
public int durEstimated
{
   get
   {
      return m_durEstimated;
   }
    set
    {
       m_durEstimated = value;
    }
 }
 public int durActual
 {
    get
    {
       return m_durActual;
    }
    set
    {
       m_durActual = value;
    }
 }
 public string  strComments
 {
    get
    {
       return m_strComments;
    }
    set
```

Download at WoweBook.Com


```
        {
            m_strComments = value;
        }
    }

    public Task( string  strIdent,
                 string  strProjIdent,
                 string  strName,
                 DateTime dateStart,  DateTime dateEnd,
                 int durEstimated,  int durActual,
                 string  strComments)
    {
        m_strIdent = strIdent;
        m_strProjIdent = strProjIdent;
        m_strName = strName;
        m_dateStart = dateStart;
        m_dateEnd = dateEnd;
        m_durEstimated = durEstimated;
        m_durActual = durActual;
        m_strComments = strComments;
    }
}
```

It is a lot of code in terms of line count but not much in terms of mental challenge or object code size.

Throughout this chapter, we need to illustrate data binding, and to be complete we need to bind to a `System.Collections.Generic` object such as a `List` during some of the examples, and to a `System.Data` object such as a `DataTable` for other examples. Normally, a `DataTable` object is created by executing a `SELECT` statement against a database, but because retrieving data from a database is the subject of Chapter 6, we'll bypass this step and instead create and load a `DataTable` object from scratch.

More specifically, we create a `Projects DataTable` object and a `Tasks DataTable` object and load each from the corresponding `List` objects that we also need to create. Thus, the data will be the same whether we are binding to a data object or to a collection object.

We also write four routines to retrieve project or task data as either a `List` or a `DataTable`. We add a class named `UtilData` to our project and place this code within it. This class will provide data for the sample applications in this chapter. The `UtilData` class code is located in the `BindDataTableToDataGrid` project.[1]

1. You can download all the projects mentioned in this chapter from www.paulyao.com/cfbook/code. Follow the instructions listed on the Web site.

Download at WoweBook.Com

With the `UtilData` class available to provide data, we're ready to start binding that data, beginning with complex data binding.

4.2 Complex Data Binding

We mentioned earlier that both single-item controls, such as `TextBox`, and multi-item controls, such as `ListBox`, can be the recipients of data binding. Because single-item controls are inherently simpler than multi-item controls, you might think that data binding to a single-item control would be easier than data binding to a multi-item control. In fact, just the opposite is true. This is because a bindable data object must contain a tabular array of data, and multi-item controls are designed to display arrays of data. There is a natural match between a data object and a multi-item control.

Table 4.1 lists the controls that support complex data binding; Table 4.2 shows the control properties used with complex data binding. (Note: The `SelectedItem` and `SelectedValue` properties are used to extract data object data after the binding has been completed. They are not used to perform the binding.)

TABLE 4.1: Controls That Support Complex Data Binding

Control	Remarks
`ComboBox`	
`DataGrid`	Displays the entire data object
`ListBox`	

TABLE 4.2: Control Properties Used with Complex Data Binding

Property	Definition	Remarks
`DataSource`	The data object	Required
`DisplayMember`	The data object property (column) to be displayed	Optional for `DataGrid`
`ValueMember`	The property of the data object to be used as the row identifier	Optional; the primary key column of the data object
`SelectedIndex`	The index of the currently selected item	Zero-based

Download at WoweBook.Com

TABLE 4.2: Control Properties Used with Complex Data Binding *(Continued)*

Property	Definition	Remarks
`SelectedItem`	The data object item that maps to the currently selected item	Equals `DataObject[Control.SelectedIndex]`
`SelectedValue`	The `ValueMember` value of the currently selected item	

We'll begin our coverage of complex data binding with the `ComboBox` and then complete the subject when we discuss the `DataGrid` control later in this chapter.

4.2.1 Using Complex Data Binding with `ComboBox` Controls

Something we said earlier bears repeating here: A `ComboBox` is a list of objects, not simply a one-dimensional list of strings, even though we see only a single string displayed within each line of a `ComboBox`. But each `ComboBox` item can be thought of as containing three fields: the display field, the key field, and the indexer field.

To illustrate complex data binding, we'll write an application, named `TimeTracker` (a screen shot of which appears in Figure 4.4 earlier in this chapter), to use complex data binding of the `Tasks` structure with the `ComboBox` control. Later, when we cover simple data binding, we'll further modify the application to data-bind to some `TextBox` controls. The completed application is located in the `BindListToControls` project.

To data-bind to our `ComboBox`, we set three of the properties listed in Table 4.2: one to identify the data source, one to identify which property of the data source is to be displayed, and one to identify which property of the data source is the primary key field.

```
//  Bind the drop-down ComboBox to the Tasks List.
cboxTasks.DataSource = alTasks;
cboxTasks.DisplayMember = "strName";
cboxTasks.ValueMember = "strIdent";
```

This code replaces the code shown next, which iterated through the `List` of tasks and loaded the `ComboBox` programmatically:

```
//  Load Proj data into the ComboBox.
foreach( Task theTask in theTasks )
{
   cboxTasks.Items.Add(theTask.strTaskName);
}
```

Download at WoweBook.Com

When we look at the two code snippets there does not seem to be much benefit to using the former instead of the latter. Both appear to be about the same in terms of size and difficulty. But using data binding gives you added benefits. You can use the control's `SelectedValue` and `SelectedIndex` properties to access the selected item in the data object, or you can pass the `SelectedValue` property to a database routine that requires a primary key as input. And, most importantly, changes made to the data in the control will be reflected in the data object and vice versa. So, complex data binding is really just the opposite of what its name implies—it is simple. But as we mentioned, it is available only with the multi-item controls listed in Table 4.1. For other controls, you must use simple data binding.

4.3 Simple Data Binding

All controls support simple data binding because it is built into the `Control` base class, but it is usually used with controls that display a single value, such as a `TextBox` control. For this reason, simple data binding is a bit more involved than complex data binding. When we bind to a single-item control, we are binding a single data element of the data object to a single valued property of the control, such as the `Text` property. To identify a single data element within a tabular array of data, we need to specify three pieces of information: the identity of the object, which column, and which row.

Because we are binding that data element to a single valued property of the control, we need to specify two more pieces of information: a reference to the control and the name of the property. Thus, simple data binding requires five pieces of information.

Like the information needed for complex data binding, the information required for simple data binding is stored in properties of the control being bound to. In the case of complex data binding, the information is stored in the control's `BindingsCollection` property.

4.3.1 The `BindingsCollection` Property

Each entry in the `BindingsCollection` property is an object of type `Binding` and contains the information necessary to bind one data element of the source data object to one property of the control. Table 4.3 lists the binding-specific PMEs of a `Binding` object.

The following code shows how to bind a column of a data source to a property of a control. In this case, the data object has already been bound

Download at WoweBook.Com

TABLE 4.3: The PMEs of a `Binding` Object

PME	Definition	Remarks
Properties		
`BindingManagerBase`	The object that manages all bindings for this column of this data source	Normally only one binding per data source column
`Control`	The control	
`PropertyName`	The control's bound property	
`DataSource`	The data object	
`BindingMemberInfo`	The data object's bound column	
`IsBinding`	The property that indicates whether this binding is active	
Events		
`Format`	The event raised as the data element is moved from the data object to the control	Allows the application to format incoming data
`Parse`	The event raised as the data element is moved from the control to the data object	Allows the application to unformat outgoing data, converting it to the data type required by the bound data object

to the `ComboBox` control named `cboxTasks`. The data object column whose name is specified by `strColumnName` is being bound to the `Text` property of the `txtTarget` `TextBox` control.

```
txtTarget.DataBindings.Add("Text",
                           cboxTasks.DataSource,
                           strColumnName);
```

When we look at this code, it seems to cover only four of the five pieces of information mentioned earlier. We specified the identity of the control as part of the method call, so that's one piece. Three more pieces of information were specified as parameters of the method call: the control's property, a reference to the data object, and the name of a column in the data

Download at WoweBook.Com

object. So, we're short one piece of information. How do we specify which row contains the data element to be bound to? The answer is, we don't.

We are dependent on the source data object being able to keep track of the "current" row. In database terminology, this means the source object needs a *cursoring* capability. That capability does not exist in a `List` object. Although our `Task` structure's `List` has an indexer method to allow us to access any row, it does not have a property that tracks which row we are currently accessing.

However, we can still bind our `TextBox` controls to our `Task` structure's `List`. We simply do it through the `ComboBox` control, for the `ComboBox` does have a cursor capability, and we have already bound our `List` to the `ComboBox`. The `SelectedItemIndex` property is the `ComboBox` control's cursor. So, once we have bound our `List` to our `ComboBox`, we can then bind that `List`, through the `ComboBox`, to our `TextBox` controls, one binding for each `TextBox`. The following code performs one of these bindings:

```
txtTaskActual.DataBindings.Add(
   "Text", cboxTasks.DataSource, "durTaskActual" );
```

Once we have done this, the `durTaskActual` value of the currently selected row will be bound to the `Text` property of the `txtTaskActual` `TextBox` control. As the user selects a new item from the `ComboBox`, the `TextBox` controls will automatically update, displaying the fields of the newly selected project. Any change in the value of a `TextBox` control's `Text` property, whether by user input or by programmed update, will update the data object; and any change in the data element of the data object will update the `TextBox` control's `Text` property, a change that will immediately be visible to the user.

Note that we bind our `TextBox` controls to the `DataSource` property of the `ComboBox`, not the `ComboBox` itself. That means we bind our `TextBox` controls to our `Task` structure's `List`. Our only reason for using the `ComboBox` in the binding is the fact that the `ComboBox`, not the `List`, is capable of maintaining the knowledge of the current row.

If the data column that we wished to bind to, `txtTaskActual.Text`, were already known to the `ComboBox`, such as the `ComboBox`'s `ValueMember` property, we could write the data-binding statement as shown here:

```
txtTaskActual.DataBindings.Add(
 "Text", cboxTasks.DataSource, cboxTasks.ValueMember );
```

Download at WoweBook.Com

Either variation is acceptable because the third parameter requires that we specify a string that contains the name of the data object column, and the ComboBox's ValueMember and DisplayMember are string properties that identify columns of the bound data object.

In Chapter 6, when we revisit data binding and work with data objects that do not have a concept of *current row,* we will go deeper into the subject of how to programmatically position bound controls to a specific row.

Because data binding results in the movement of data between the not-visible data object and the visible control, you may need to transform the data as it is being moved; that is, you may need to *format* or *parse* the data.

4.3.2 Formatting and Parsing

Simple data binding causes data to flow between a data object and a property of a control. This is not a problem if the data element, like the Text property, is a string. But durTaskActual is not a string; it is an integer, just as dateTaskStart is not a string but a date. What if we need to format this data as it moves from the data object into the control? In fact, we do need to format date fields, for we intend to save screen space by removing the time component before displaying the date. No 12/12/2003 11:45:30:001 dates for us.

We solve the problem of when and how to format bound data by responding to events. What you add to the BindingsCollection of a control is a Binding object, and Binding objects raise events. They raise a Format event as they are moving data into the target control, and they raise a Parse event as they are moving data back into the data object.

We need to handle the Format event in our application so that we can strip off the time component from our two date fields. We do not need to handle the Parse event because our formatted string, such as 02/05/2004, will convert back to a date without a problem. We never had an interest in the time portion anyway, so losing it is not a problem.

To simplify things, we write a short routine to do the data binding and formatting for each field, passing it a reference to the target TextBox and the target property name (which is always Text in our application) as parameters. We name our routine BindToTextbox. We also write an event handler, named Date_Format, for the Format event and use AddHandler to tie it to the Binding object's Format event. Listing 4.3 shows these two routines.

Download at WoweBook.Com

Listing 4.3: The Generic Binding Routine and Format Event Handler

```
private void BindToTextbox( TextBox txtTarget,
                            string  strColumnName)
{
   //  Bind the TextBox to a column of the drop-down
   //     ComboBox's DataSource (the Tasks List).
   txtTarget.DataBindings.Add("Text",
                              cboxTasks.DataSource,
                              strColumnName);
   //  Specify an event handler for formatting those
   //     fields that are of data type DateTime.
   if( txtTarget == txtTaskStart ||
       txtTarget == txtTaskEnd )
   {
      txtTarget.DataBindings[0].Format +=
         new ConvertEventHandler(this.Date_Format);
   }
}

private void Date_Format( object sender,
                          ConvertEventArgs e)
{
   //  Format each date as it is moved by data binding
   //     from the List into a TextBox.
   e.value = e.Value.ToString("d");
}
```

Life, or at least coding, is made a little easier by the fact that each `TextBox` has only one property, `Text`, which uses binding. Thus, each `TextBox` has only one `Binding` object in its `DataBindings` collection, and we can access that `Binding` object as `DataBindings(0)`.

Then we write a routine to do all the binding (the binding to the `ComboBox` and to each `TextBox`), as shown in Listing 4.4.

Listing 4.4: Binding Tasks to a `ComboBox` and Several `TextBox` Controls

```
private void BindTasksToControls()
{
   //  Bind the drop-down ComboBox to the Tasks List.
   cboxTasks.DataSource = alTasks;
   cboxTasks.DisplayMember = "strName";
   cboxTasks.ValueMember = "strIdent";

   //  Bind each TextBox to a column of the drop-down
   //     ComboBox's DataSource (the Tasks List).
   BindToTextbox(txtTaskNo, "strIdent");
   BindToTextbox(txtTaskName, "strName");
```

Download at WoweBook.Com


```
   BindToTextbox(txtTaskStart, "dateStart");
   BindToTextbox(txtTaskEnd, "dateEnd");
   BindToTextbox(txtTaskActual, "durActual");
   BindToTextbox(txtTaskEstimated, "durEstimated");
}
```

And while we're on a roll, we write a routine to unbind everything and to clear out our `TextBox` controls. We need to unbind and then rebind while we are adding new tasks to our data object, and having a routine to do each will help. We make it easier to write this unbinding routine by adding a `Panel` control to our form at design time and then placing the six `TextBox` controls that hold task information into that `Panel` control. With that done, Listing 4.5 shows our unbinding routine.

Listing 4.5: Unbinding the Tasks from the `TextBox` and `ComboBox` Controls

```
private void UnbindTasksFromControls()
{
   //  Iterate through all the controls in the TaskFields
   //     Panel, unbind them, and clear them.
   foreach( Control theControl in panelTaskFields.Controls )
   {
      theControl.DataBindings.Clear();
      theControl.Text = string.Empty;
   }

   //  Unbind the ComboBox.
   cboxTasks.DataSource = null;
}
```

Users can now retrieve, display, and update tasks without requiring any extra code on our part. Updating is built into data binding. In our new version of the application, as the user modifies the contents of a bound `TextBox`, the matching data element in the data object is updated.

So, simple data binding, which is inherited from the `Control` base class, is implemented by specifying a control and its target property, plus a data object that supports cursoring and its source property. It is available for you to use with all controls, and it uses the same PMEs regardless of the control's class.

We have just finished using simple data binding with control classes that were introduced in an earlier chapter. Now it's time to formally introduce a new control, the `DataGrid`, and apply both simple and complex data binding to it.

Download at WoweBook.Com

4.4 The `DataGrid` Control

The purpose of the `DataGrid` control is to display tabular data from a data object, as shown in Figure 4.6.

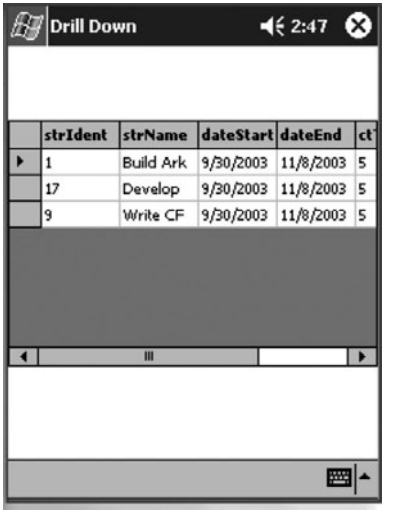


FIGURE 4.6: A `DataGrid` Control Displaying a `DataTable` Object

Tables 4.4 through 4.6 list the PMEs of the base `Control` object class that are supported by the `DataGrid` control, while Table 4.7 lists the `DataGrid`-specific PMEs.

TABLE 4.4: Support for Properties Inherited from `Control`

	DataGrid
Links to Other Controls	
`Controls`	X
`ContextMenu`	X
`Parent`	X
`TopLevelControl`	X
Positional	
`Bottom`	X

Download at WoweBook.Com

TABLE 4.4: Support for Properties Inherited from Control *(Continued)*

	DataGrid
Bounds	X
ClientRectangle	X
ClientSize	X
Height	X
Left	X
Location	X
Right	X
Size	X
Top	X
Width	X
Data Binding	
BindingContext	X
DataBindings	X
DataSource	X
Control State	
Capture	
Enabled	X
Focused	X
Visible	X
Control Appearance	
BackColor	X
Font	X
ForeColor	X
Text	X

Download at WoweBook.Com

TABLE 4.5: Support for Methods Inherited from `Control`

	DataGrid
Thread-Safe	
`Invoke`	X
Object Support	
`Dispose`	X
`Equals`	
`GetHashCode`	
`GetType`	X
`ToString`	
Coordinate Conversion	
`PointToClient`	X
`PointToScreen`	X
`RectangleToClient`	X
`RectangleToScreen`	X
Z-Ordering	
`BringToFront`	X
`SendToBack`	X
Visibility	
`Hide`	X
`Show`	X
Drawing	
`CreateGraphics`	
`Invalidate`	X
`Refresh`	
`Update`	
Control State	
`Focus`	X

Download at WoweBook.Com

TABLE 4.6: Support for Events Inherited from Control

	DataGrid
Garbage Collection and Validation	
Disposed	X
Validated	X
Validating	X
Focus Changes	
GotFocus	X
LostFocus	X
Control Property Changes	
EnabledChanged	X
ParentChanged	X
Resize	X
TextChanged	X
Keyboard Input	
KeyDown	X
KeyPress	X
KeyUp	X
Mouse Input	
Click	X
MouseDown	X
MouseMove	X
MouseUp	X
Drawing	
Paint	X

Download at WoweBook.Com

TABLE 4.7: PMEs Specific to the DataGrid Control

Properties	
ColumnHeadersVisible	RowHeadersVisible
CurrentCell	SelectionBackColor
CurrentRowIndex	SelectionForeColor
FirstVisibleColumn	TableStyles
GridLineColor	VisibleColumnCount
HeaderBackColor	VisibleRowCount
HeaderForeColor	
Methods	**Events**
GetCellBounds	CurrentCellChanged
HitTest	
UnSelect	

4.4.1 Using Complex Data Binding with the DataGrid Control

As we've mentioned, the purpose of the DataGrid control is to display tabular data from a data object, and the easiest way to do it is to bind a data object containing the data to the DataGrid control. For instance, if our application displayed a DataGrid control named dgridDisplay and also used our UtilData class to load a DataTable named dtabProjects, the following code would bind the DataTable to the DataGrid, resulting in the output shown earlier in Figure 4.6:

```
dgridDisplay.DataSource = dtabProjects;
```

In this case, the data binding was very easy. We had to set only one property of the DataGrid control to achieve the binding and display the data.

However, in the code just shown, we deferred all decisions about the appearance of that data to the DataGrid control. In a typical application this would not be the case; you might want to display column titles that were different from the column names of the underlying data object, or not display some of the columns, or alter the left-to-right sequence of the columns, or make some of the columns wider than others. So, with the DataGrid control, the binding was easy, but styling the display is going to require some additional code. It will not be the same code, or even the

Download at WoweBook.Com

same technique, that will be used to format and parse the simple binding example we just presented.

4.4.2 Styling the Display of Data in a DataGrid Control

To control the display of data in a DataGrid, you control the transformation of that data from the data object into the DataGrid. That is, you map columns of the data object to DataGrid columns, specifying additional information about each mapping. Each column mapping is referred to as a DataGridColumnStyle and is itself an object. The complete set of DataGridColumnStyle objects used to map all columns of one data object to the DataGrid is a collection that is a property of a DataGridTableStyle object. A DataGridTableStyle object consists of some individual properties plus a collection of DataGridColumnStyle objects, which taken together define the appearance of one data object to the DataGrid control. Table 4.8 lists the binding-specific PMEs of the DataGridTableStyle object; Table 4.9 lists the DataGridColumnStyle PMEs. As you have found elsewhere in the .NET Compact Framework, both sets of PMEs are much shorter than their desktop .NET Framework counterparts. Here the lists consist solely of properties.

You may need to provide more than one DataGridTableStyle object for a single DataGrid, for you may be planning to bind first one data object to the DataGrid and then a different data object to the DataGrid later in the

TABLE 4.8: PMEs for the DataGridTableStyle Object

PME	Definition
MappingName	The name of the data object
GridColumnStyles	The GridColumnStyles collection

TABLE 4.9: PMEs for the DataGridColumnStyle Object

PME	Definition
MappingName	The data object column name
HeaderText	The text to be displayed in the DataGrid column header
NullText	The text to be displayed in DataGrid cells in lieu of nulls
Width	The width, in pixels, of the DataGrid column

Download at WoweBook.Com

application. Perhaps you first display information about projects within the `DataGrid` and then, as a result of user input, display the selected project's required tasks in the grid. We do this in the next example, which displays project information as shown in Figure 4.7.

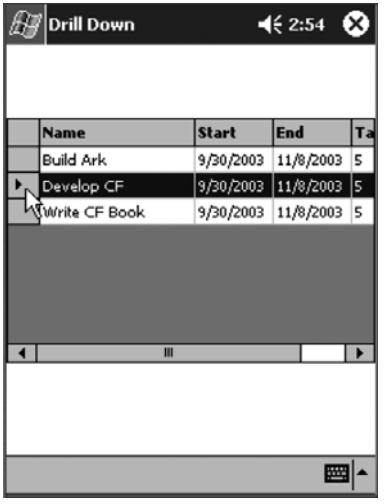


FIGURE 4.7: A `DataGrid` Control with the `Projects` Table Bound to It

Figure 4.8 shows the same `DataGrid` now displaying the tasks required for that project.

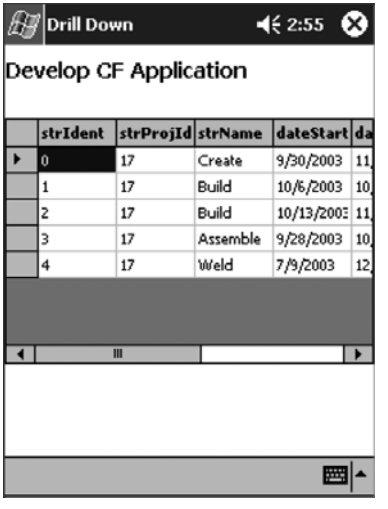


FIGURE 4.8: A `DataGrid` Control with the `DataView` of the `Tasks` Table Bound to It

Download at WoweBook.Com

You are not required to supply a new DataGridTableStyle object each time you change the binding from one data object to another. Instead, you can define the DataGridTableStyle object for each data object in advance and then add all the DataGridTableStyle objects to the DataGrid control's TableStyles collection at the start of the application. When the application assigns a new data object to the DataGrid control's DataSource property, the DataGridTableStyle object for the new data object is invoked. Thus, a DataGrid control has a TableStyles collection, and each TableStyle includes a GridColumnStyles collection.

DataTable objects have a large number of PMEs, most of which are related to the transfer of data between a DataTable and a database. Once again, because this chapter covers transferring data between a data object and a bound control, we will not cover those properties in this chapter, deferring them until Chapter 6. However, two DataTable properties are applicable to creating styles for data binding. The first is the TableName property, which is the property of the DataTable object that serves to identify it. Each instance of a DataTable object has its own name, which can be specified as the table is created, as the following code shows:

```
dtabProjects = new DataTable("Projects");
```

Within a DataTable object, the columns also have names. Both the table name and the column names are needed when styling the display of DataTable data in a DataGrid control. As Tables 4.8 and 4.9 showed, they are themselves properties of the DataGridTableStyle and DataGridColumnStyle objects. The TableName property of the data object becomes the MappingName property of the DataGridTableStyle object, and the column names become the MappingName properties of the DataGridColumnStyle objects.

This requirement that the data objects must have a TableName property for their display to be styled is one of the reasons why a DataTable rather than a List will be used for the remainder of this chapter. A List can be bound to the DataGrid, as we showed in earlier code, but it cannot be styled, because the List class does not have a TableName property.

4.4.3 Creating Table and Column Styles

Table styles and their associated column styles can be defined at either design time or runtime. It's a little easier to understand them if you build them at design time, but you give yourself more flexibility when you

Download at WoweBook.Com

build them at runtime. For this reason, we'll build our first one at design time and then later write the code to create it at runtime.

First, let's look at our data object, decide how we want it to appear within the `DataGrid`, and then create the styles needed to achieve this appearance. We will use the `Projects DataTable`, whose display before styling appears in Figure 4.7, as our data object.

We decide that we do *not* want to display the identity column, duration completed, duration estimated, or comments. We want our `DataGrid` column sequence to be project name, start date, end date, and task count. For a preview of the end result, see Figure 4.13 later in this chapter.

To define our styles, we use Visual Studio .NET. We begin by selecting our `DataGrid` in the Designer window and then choosing the `TableStyles` collection in the Properties window, as shown in Figure 4.9.

This results in the display of the `DataGridTableStyle` collection dialog box. Within this dialog box we click on the Add button and are presented with the information shown in Figure 4.10.

We can name the `DataGridTableStyle` object anything we want; we name ours `dgtsProjects`. We must give the `MappingName` property the value of the `DataTable` object's `TableName` property, which is `Projects`. Then we click on the GridColumnStyles button to add a new column style.

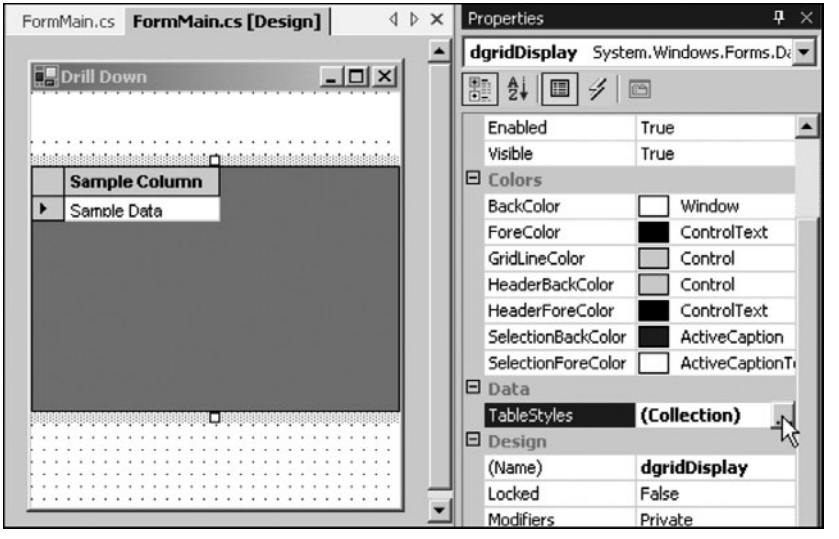


FIGURE 4.9: Choosing the `TableStyles` Collection

Download at WoweBook.Com

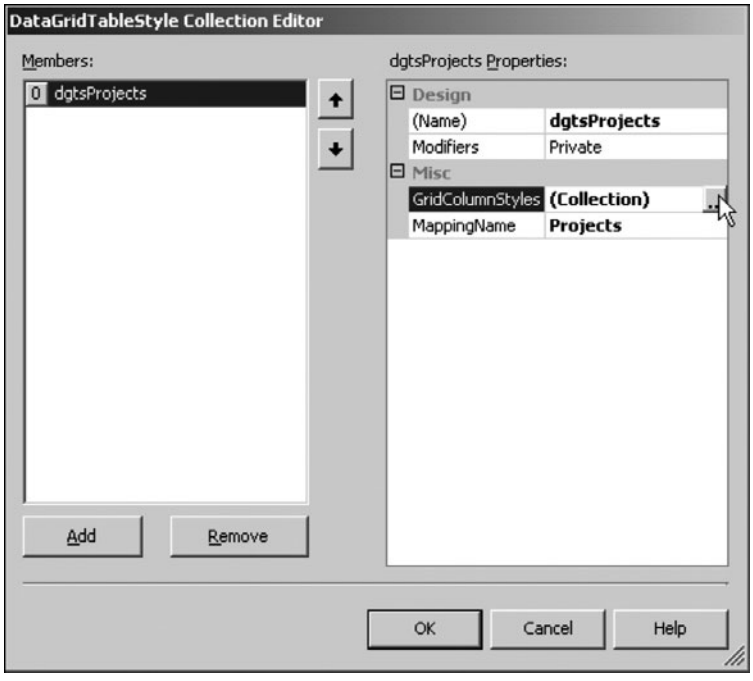


FIGURE 4.10: Adding a Table Style

Once again we are presented with a new dialog box, and once again we click on the Add button and are presented with a `DataGridColumnStyle` collection dialog, which allows us to enter information mapping a column of our data object to the first column of the `DataGrid`. We enter a name for our column style object, decide what we want the column header within the `DataGrid` to say, and specify the data object's column name, in this case, `strName`. We leave the `NullText` specification unchanged and double the `Width` from the default of 50 pixels to 100 pixels. Then we add this column style to our `DataGridTableStyle`, as shown in Figure 4.11.

We repeat this step for each data object column we wish to display in the `DataGrid`. Then we return to the `DataGridTableStyle` collection dialog box and add the `TableStyle` to the `DataGrid`. Now all that remains is to write the following code to bind our `DataTable` to the `DataGrid` and place it within the form's `Load` event:

```
dgridProjects.DataSource = dpProjects.GetProjectsDT();
```

When we run the application, we get the more refined result that appears later in this chapter in Figure 4.13, which is the result we wanted.

Download at WoweBook.Com

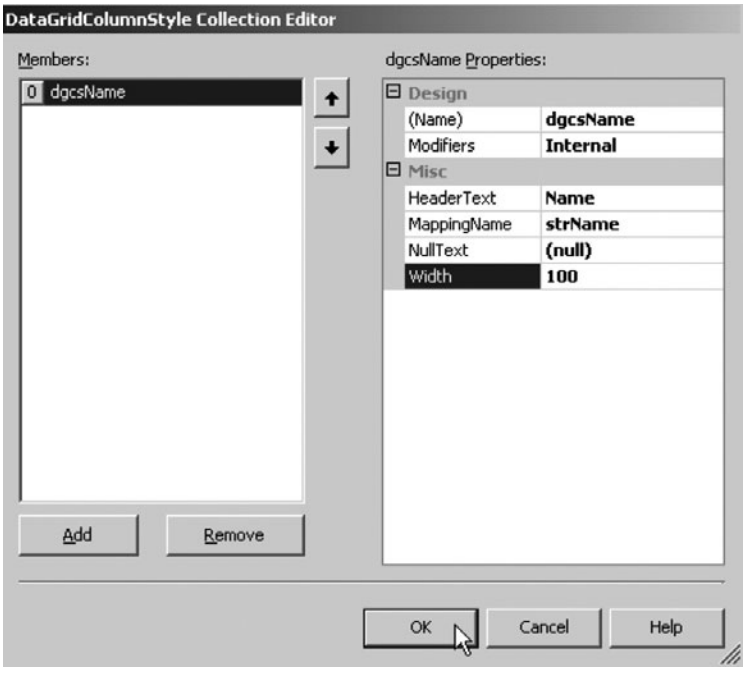


FIGURE 4.11: Entering `DataGridColumnStyle` Properties

4.4.4 Creating Styles at Runtime

We could have built the `DataGridTableStyle` object that we just created in the Designer by writing code instead, and there are some benefits in doing this. It is usually easier to modify code than to modify a design, and it's almost always easier to produce a second `DataGridTableStyle` object by modifying the code of a previous creation rather than by repeating the entire design-time style-creation process. Design time is nice, but it's not very reusable.

Listing 4.6 shows the code to create a `DataGridTableStyle` object that is identical to the one we just created at design time.

LISTING 4.6: Creating `DataGrid` Table and Column Styles

```
internal static void AddCustomDataTableStyle
                              (
                                 DataGrid dgridTarget,
                                 string  strTable
                              )
{
   DataGridTableStyle dgtsStyle;
   DataGridTextBoxColumn dgtsColumn;

   switch (strTable)
```

Download at WoweBook.Com


```
    {
        case "Projects":
            //  Define a Style for the Projects
            //     DataTable or List to be used by
            //     the DataGrid when displaying projects.

            dgtsStyle = new DataGridTableStyle();
            //  Specify that it is to be applied whenever
            //     a data object named Projects is assigned
            //     to the DataGrid's DataSource property.
            dgtsStyle.MappingName = "Projects";
            //  Add columns.
            //  Specify:
            //     Column/field name
            //     Column header
            //     Column width in pixels

            dgtsColumn = new DataGridTextBoxColumn();
            dgtsColumn.MappingName = "strName";
            dgtsColumn.HeaderText = "Name";
            dgtsColumn.Width = 100;
            dgtsStyle.GridColumnStyles.Add(dgtsColumn);

            dgtsColumn = new DataGridTextBoxColumn();
            dgtsColumn.MappingName = "dateStart";
            dgtsColumn.HeaderText = "Start";
            dgtsColumn.Width = 50;
            dgtsStyle.GridColumnStyles.Add(dgtsColumn);

            dgtsColumn = new DataGridTextBoxColumn();
            dgtsColumn.MappingName = "dateEnd";
            dgtsColumn.HeaderText = "End";
            dgtsColumn.Width = 50;
            dgtsStyle.GridColumnStyles.Add(dgtsColumn);

            dgtsColumn = new DataGridTextBoxColumn();
            dgtsColumn.MappingName = "ctTasks";
            dgtsColumn.HeaderText = "Tasks";
            dgtsColumn.Width = 50;
            dgtsStyle.GridColumnStyles.Add(dgtsColumn);

            //  Add the style to the DataGrid.
            dgridTarget.TableStyles.Add(dgtsStyle);

            break;
        case "Tasks":
            break;
        default:
            break;
    }
}
```

Download at WoweBook.Com

To modularize our application, we add a class named `UtilGUI` and place the code from Listing 4.6 within that class. This code is located in the `BindToDataGridStyled` project.

Now that we have the ability to present the data from our data object to our user, let's examine how we handle the user's reaction to this data. That is, let's look at the events raised by the `DataGrid` control and at how to obtain the information we need to respond to those events.

4.4.5 Responding to User Input

When it comes to data binding, the `DataGrid` control is the most capable and the most complex. We saw an indication of this when we did styling, and we see it again as we respond to the events raised by the `DataGrid` in response to user input.

The `DataGrid` control is a read-only control. Although users can designate a current cell by tapping with the stylus, they cannot then enter data into that cell. So, reacting to user input may involve drilling down further into the selected item, or it may require presenting an input control to the user followed by updating the data.

A `DataGrid` control displays cells of data to the user. The surface area of the `DataGrid` control is broken down into regions of different types. These types are listed in the `DataGrid.HitTestType` enumeration defined in the `System.Windows.Forms.DataGrid` namespace and are shown, in brief, in Table 4.10. You can find a more detailed listing in the online help under `DataGrid.HitTestType`.

As shown in Table 4.10, data cells are type 1 cells. The cells located just above the data columns, where the column titles appear, are type 2 cells. The cells just to the left of each row are type 4 cells. The thin dividing line

TABLE 4.10: Types of Regions of the `DataGrid` Control

Type	Location
0	Empty area of the `DataGrid` control
1	Data cell
2	Column header
4	Row header
8	Column resize
16	Row resize

Download at WoweBook.Com

between each column heading is a cell of type 8, while the lines between rows are type 16 cells. Any other area of the DataGrid control is located in a cell of type 0.

As mentioned in the preceding paragraph, the HitTestType of all data cells is 1, which maps to an enumeration name of Cell. In addition, the data type of all data cells is System.String. Regardless of the data type of the underlying bound data object column, the content of a DataGrid cell is always a string.

The data cells, being a two-dimensional, zero-based array, are numbered beginning at the upper left with cell (0,0). At any point in time, at most one data cell is the current cell. The user designates the current cell by tapping with the stylus or by using the Tab key or the arrow keys.

When a user taps on a data cell, that data cell becomes the current cell, which the DataGrid control indicates by highlighting the cell's background. If the user taps on a row header, the DataGrid control highlights the entire row. Even with an entire row highlighted, only one cell is the current cell—the cell whose row number is the same as the highlighted row and whose column number is the same as the previously selected current cell. Thus, if the current cell is cell (2,1)—row 2, column 1—and a user taps the row header of row 0, cell (0,1) becomes the new current cell.

Sometimes it is not the current cell but the cell located under the mouse that is of interest to your application. This is the case not only when the user is tapping on a data cell but also when the user is tapping elsewhere within the DataGrid control, such as within a row header.

The DataGrid control's HitTest method, which takes DataGrid coordinates as input and returns a HitTest object, is used to determine the cell type, row number, and column number of the cell located under the mouse. Table 4.11 lists the PMEs of the HitTest object.

Once the row and column numbers are known, the Item property can retrieve that actual data value from the cell. This is illustrated in Listing 4.7,

TABLE 4.11: PMEs of the HitTest Object

PME	Definition	Remarks
Type	Cell type	DataGrid.HitTestType
Column	Column number	
Row	Row number	
Nowhere	Location of type 0	Static method

Download at WoweBook.Com

which reacts to a MouseMove event by obtaining the row, column, type, and value information for the cell that is under the mouse and displaying that information in the lblDatum set of labels.

Listing 4.7: Displaying HitTest Information

```
private void Grid_MouseMove(object sender,
                            MouseEventArgs e)
{
   //  Store the DataGrid object for early binding.
   DataGrid dgridSender = (DataGrid)sender;

   //  Use the DataGrid's HitTest method to get the
   //     HitTest object for this mouse location.
   DataGrid.HitTestInfo httstInfo;
   httstInfo = dgridSender.HitTest(e.X, e.Y);

   //  Display HitTest properties in the labels.
   lblDatumType.Text = httstInfo.Type.ToString();
   lblDatumRow.Text = httstInfo.Row.ToString();
   lblDatumColumn.Text = httstInfo.Column.ToString();
}
```

So, when responding to mouse events, the application can always determine whether the event is occurring within a data cell or within a particular area of the DataGrid control located outside the data cells.

A more difficult task is to trace a cell of the DataGrid control back to the data element of the underlying data object. You might never need to do this, for data binding ties each cell of the DataGrid control to a data element of the bound data object for you. To change a value in the data object, all you need to do is programmatically change the corresponding value in the DataGrid; you do not need to directly access the data object yourself. Perhaps, however, you need to know the column name or data type of the data element bound to a particular DataGrid cell before you do the update. This will be the case in an in-place editing sample application later in this chapter, for it must check the data object column name to ensure that a certain column is not being updated. So, in case you ever need direct access into the data object bound to a DataGrid control, the technique for doing so is explained here.

To access fields of the data object, you will need to reverse-engineer the mapping you established earlier between the DataGrid control and the data object. Because the .NET Compact Framework DataGrid control does not support sorting, there will be a one-to-one relationship between the

Download at WoweBook.Com

rows of the DataGrid control and the rows of the data object; row 4 in the DataGrid control will be row 4 in the data object, row 0 in the DataGrid control will be row 0 in the data object.

But the same is not true for columns. Columns of the data object can appear in any sequence within the DataGrid control, or they can be absent. Therefore, you must access the current DataGridTableStyle object to determine which data object column maps to which DataGrid column. The current DataGridTableStyle object will be the one whose MappingName property equals the TableName property of the DataGrid control's DataSource property. Then iterate through the GridColumnStyle collection of the TableStyles object. The GridColumnStyle objects will be in the same sequence as the DataGrid columns, and for each column style the MappingName property will contain the name of the data object column.

You can see why we recommend letting data binding do the work for you.

Table 4.12 lists the paths to follow to obtain information about a DataGrid control and its bound object when responding to the CurrentCellChanged event or to mouse events.

TABLE 4.12: **Obtaining DataGrid and Bound DataTable Information**

Reference Number	Datum	Path (Where e is the MouseEventArgs Parameter)
1	Cell under mouse: cell type	`DataGrid.HitTest(e.X,e.Y).Type`
2	Cell under mouse: row number	`DataGrid.HitTest(e.X,e.Y).Row`
3	Cell under mouse: column number	`DataGrid.HitTest(e.X,e.Y).Column`
4	Current cell: cell type	Always 1
5	Current cell: row number	`DataGrid.CurrentCell.RowNumber`
6	Current cell: column number	`DataGrid.CurrentCell.ColumnNumber`
7	Data value of cell located at row X, column Y	`DataGrid[X,Y]`
8	Bounds of cell located at row X, column Y	`DataGrid.GetCellBounds(X,Y)`

Continues

Download at WoweBook.Com

TABLE 4.12: Obtaining `DataGrid` and Bound `DataTable` Information *(Continued)*

Reference Number	Datum	Path (Where `e` is the `MouseEventArgs` Parameter)
9	`TableName` of bound data object	`DataGrid.DataSource.TableName`
10	`TableStyle` used with bound data object	`TableStyle` object whose `MappingName` property equals #9
11	Data object row number of `DataGrid` row number X	X
12	Data object column name of `DataGrid` column number Y	`MappingName` property value of `GridColumnStyles[Y]` of #10
13	Data object column number of `DataGrid` column number Y	Column number of #12
14	Data object data element that maps to `DataGrid` row X, column Y	Value of `DataTable.Rows[X]` of #11
15	Data type of data object data element that maps to `DataGrid` row X, column Y	`GetType` method of #14

To illustrate Table 4.12, we will write a sample application. This application is located in the `DisplayBindingInfo` project.

The application will react to `CurrentCellChanged` events and `MouseMove` events by displaying information about the current cell and the cell located beneath the mouse. This information includes the cell type, row number, and column number of the cell; the contents of the cell; the column name, value, and data type of the bound field in the underlying `DataTable`; and the primary key value of the current row. This information is necessary not only to determine the cell in question but also to access the corresponding data element within the bound `DataTable`.

Normally, we do not respond to `MouseMove` events in our sample applications because most applications must run in a nonmouse environment. In this application, however, we use the `MouseMove` event rather than the `MouseDown` or `MouseUp` event because the information displayed is applicable

Download at WoweBook.Com

to any mouse event, and responding to MouseMove events produces a more animated display.

We use both simple and complex data binding with the DataGrid control in this application. The reason for using complex data binding is obvious—as in past applications, it is the mechanism through which the project information is displayed within the DataGrid. The reason for using simple data binding with the DataGrid control is less obvious but potentially just as beneficial.

4.4.6 Using Simple Data Binding with the DataGrid Control

As with the previous sample application, we have chosen *not* to display the Project Identity field, strIdent, in the DataGrid control. But if the user requests a drill-down operation, desiring additional information about the current row, we need to know the identity value of that row; that is, we need to know the project's primary key value even though it is not displayed in the DataGrid control. It would be ideal if this value were stored in a property of the DataGrid control, but it isn't.

However, we can use simple data binding to achieve this by taking advantage of the fact that a DataGrid control's Text property is normally unused. We can use it to hold the primary key value by binding the DataGrid control's Text property to the primary key column of the data object, in this example, strIdent, as shown in the following code:

```
dgridProjects.DataBindings.Add(
                    "Text",
                    dgridProjects.DataSource,
                    "strIdent");
```

Having the primary key value of the current row available in the Text property is very useful when working with databases because data retrieval routines often require a primary key value as an input parameter.

4.4.7 Accessing DataGrid Information

When we run the application (whose code is presented in this subsection, beginning with Listing 4.8 and ending with Listing 4.12) and request to see information based on the CurrentCellChanged event, we obtain the output shown in Figure 4.12. Note that the Name and Data Type fields refer to the data object, not the DataGrid control, and that the Key field displays the selected cell's primary key value, also from the data object.

Download at WoweBook.Com

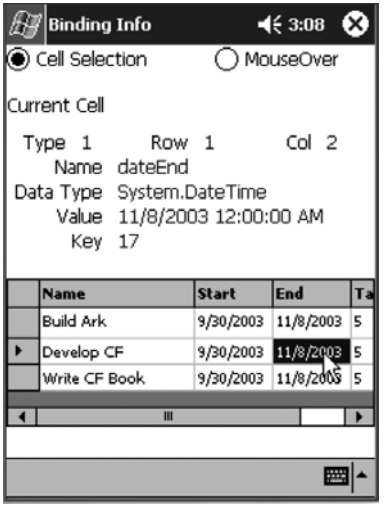


FIGURE 4.12: Displaying Cell Information

We begin our application with some namespace declarations, followed by Designer-generated code, some of which we have removed for the sake of compactness in Listing 4.8.

LISTING 4.8: The Start of the Hit Testing Program

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Data;
using YaoDurant.Data;
using YaoDurant.GUI;

namespace DisplayBindingInfo
{
   public class FormMain : System.Windows.Forms.Form
   {
      internal System.Windows.Forms.Panel panelChoice;
      internal System.Windows.Forms.RadioButton optMouseOver;
               :
               :
               :
      internal System.Windows.Forms.Label lblDatumEvent;
      private System.Windows.Forms.DataGrid dgridProjects;
      private System.Windows.Forms.MainMenu mainMenu1;

      public FormMain()
      {
         InitializeComponent();
      }
```

Download at WoweBook.Com


```
protected override void Dispose( bool disposing )
{
   base.Dispose( disposing );
}
#region Windows Form Designer generated code

static void Main()
{
   Application.Run(new FormMain());
}
```

Next comes the form's `Load` event handler (shown in Listing 4.9), which loads, binds, and styles the data using the utility routines we mentioned earlier.

Listing 4.9: The `Load` Event Handler

```
private void FormMain_Load(object sender, EventArgs e)
{
   YaoDurant.Data.UtilData utilData = new UtilData();
   YaoDurant.GUI.UtilGUI utilGUI = new UtilGUI();

   //  Make the Project table the DataSource.
   //  Make the strIdent field of the currently
   //     selected row the Text property of the DataGrid
   //     control.
   dgridProjects.DataSource = utilData.GetProjectsDT();
   dgridProjects.DataBindings.Clear();
   dgridProjects.DataBindings.Add(
                        "Text",
                        dgridProjects.DataSource,
                        "strIdent");

   //  Use a utility routine to style the
   //     layout of Projects in the DataGrid.
   UtilGUI.AddCustomDataTableStyle(dgridProjects,
                                   "Projects");
}
```

The application handles `MouseMove` events by using the `sender`, `DataSource`, `HitTest`, and `TableStyle` objects to obtain information about the cell under the mouse and its bound data element, as shown in Listing 4.10.

Listing 4.10: Performing Hit Testing as the Mouse Is Moving

```
private void dgridProjects_MouseMove(object sender,
                                     MouseEventArgs e)
{
   //  If the user is not interested in
```

Download at WoweBook.Com


```
//      this event, do not handle it.
if (! optMouseOver.Checked )
{
   return;
}

//  Tell the user what information is being shown.
lblDatumEvent.Text = "Cell Under Mouse";

//  Store the DataGrid object for early binding.
DataGrid dgridSender = (DataGrid)sender;

//  Store the DataGrid's DataSource object.
DataTable dtabDataSource =
   (DataTable)dgridSender.DataSource;

//  Use the DataSource name to retrieve
//      and store the current TableStyle.
DataGridTableStyle dgtsCurrent =
   dgridSender.TableStyles[dtabDataSource.TableName];
//  Use the DataGrid's HitTest method to get the
//     HitTest object for this mouse location.
DataGrid.HitTestInfo httstInfo;
httstInfo = dgridSender.HitTest(e.X, e.Y);

//  Clear the labels that are meaningful only
//     if the mouse is over a data cell.
lblDatumName.Text = string.Empty;
lblDatumValue.Text = string.Empty;
lblDatumDataType.Text = string.Empty;

//  Display HitTest properties in the labels.
lblDatumType.Text = httstInfo.Type.ToString();
lblDatumRow.Text = httstInfo.Row.ToString();
lblDatumColumn.Text = httstInfo.Column.ToString();

//  If the mouse is positioned over a data column...
if (httstInfo.Column != -1 )
{
   //  Obtain the DataSource's column name from
   //     the ColumnStyles collection of the
   //     current TableStyle.
   string  strColumnName =
      dgtsCurrent.GridColumnStyles[httstInfo.Column].
         MappingName;
   lblDatumName.Text = strColumnName;

   //  Obtain the DataSource column's data type.
   lblDatumDataType.Text =
      dtabDataSource.Rows[0][strColumnName].
```

Download at WoweBook.Com


```
            GetType().ToString();

         //  If the mouse is positioned over a data cell...
         if (httstInfo.Row != -1 )
         {
            //           EITHER

            //  Obtain and display the cell value from
            //     the underlying DataSource object.
            lblDatumValue.Text =
               dtabDataSource.Rows[httstInfo.Row]
                                 [strColumnName].ToString();

            //              OR

            //  Obtain and display the cell value from
            //     the DataGrid itself.
            lblDatumValue.Text =
               dgridSender[httstInfo.Row,
                           httstInfo.Column].ToString();
         }
      }
   }
```

The `CurrentCellChanged` event handler is similar, except that it uses the `CurrentCell` object rather than obtaining a `HitTest` object to determine the appropriate cell. Listing 4.11 shows the code.

Listing 4.11: Obtaining Information about the New Current Cell

```
   private void dgridProjects_CurrentCellChanged(
                                        object sender,
                                        EventArgs e)
   {
      //  If the user is not interested in
      //      this event, do not handle it.
      if (! optCellSelection.Checked )
      {
         return;
      }

      //  Tell the user what information is being shown.
      lblDatumEvent.Text = "Current Cell";

      //  Store the DataGrid object for early binding.
      DataGrid dgridSender = (DataGrid)sender;

      //  Store the DataGrid's DataSource object.
      DataTable dtabDataSource =
```

Download at WoweBook.Com


```
            (DataTable)dgridSender.DataSource;

        //  Use the DataSource name to retrieve
        //      and store the current TableStyle.
        DataGridTableStyle dgtsCurrent =
            dgridSender.TableStyles[dtabDataSource.TableName];

        DataGridCell cellCurr = dgridSender.CurrentCell;

        //  Display CurrentCell properties in the labels.
        lblDatumType.Text = "1";
        lblDatumRow.Text = cellCurr.RowNumber.ToString();
        lblDatumColumn.Text = cellCurr.ColumnNumber.ToString();
        //  Obtain the DataSource's column name from
        //     the ColumnStyles collection of the
        //     current TableStyle.
        string  strColumnName =
            dgtsCurrent.GridColumnStyles[cellCurr.ColumnNumber].
               MappingName;
        lblDatumName.Text = strColumnName;

        //  Obtain the DataSource column's data type.
        lblDatumDataType.Text =
            dtabDataSource.Rows[0][strColumnName].
               GetType().ToString();

        //           EITHER

        //  Obtain and display the cell value from
        //     the underlying DataSource object.
        lblDatumValue.Text =
            dtabDataSource.Rows[cellCurr.RowNumber]
                              [strColumnName].ToString();

        //              OR

        //  Obtain and display the cell value from
        //     the DataGrid itself.
        lblDatumValue.Text =
            dgridSender[cellCurr.RowNumber,
                        cellCurr.ColumnNumber].ToString();

        //  Display the primary key of the row, based on
        //     the DataBindings entry that was added
        //     during the Load event.
        lblDatumKey.Text = dgridSender.Text;
    }
```

Download at WoweBook.Com

The final routine in the application, as shown in Listing 4.12, responds to the user's request to switch from `CurrentCell` mode to `MouseMove` mode.

Listing 4.12: Changing Modes

```
      private void optAny_CheckedChanged(object sender,
                                         EventArgs e)
      {
         //  Clear the Labels.
         this.lblDatumEvent.Text = string.Empty;
         this.lblDatumType.Text = string.Empty;
         this.lblDatumRow.Text = string.Empty;
         this.lblDatumColumn.Text = string.Empty;
         this.lblDatumName.Text = string.Empty;
         this.lblDatumValue.Text = string.Empty;
         this.lblDatumKey.Text = string.Empty;
         this.lblDatumDataType.Text = string.Empty;
         this.lblHdrKey.Visible = optCellSelection.Checked;
      }
   }
}
```

Now that we know how to obtain the necessary information to react to user input, we'll write two applications that use this information. Each application will address a common challenge associated with the `DataGrid` control. The first application will provide *drill-down* capability, while the second application will perform *in-place editing*.

4.4.8 Providing Drill-Down Capability

Our drill-down application (located in the `DrillDownSameForm` project) presents project information within the `DataGrid` control, similar to the previous application. When the user taps within a row header, thus selecting a particular project, the application's menu allows the user to request that the application drill down into the selected project.

The application drills down into the project by replacing the information in the `DataGrid` control with a list of the tasks required for the selected project, as shown in Figure 4.13. The application also displays the name of the selected project in a table located above the `DataGrid` control.

The initial display of project information within the `DataGrid` control uses the same database simulation routine and `DataGrid` styling routine we used in the previous application. Like any windows application, desktop or mobile, ours uses a context menu to allow the user to make context

Download at WoweBook.Com

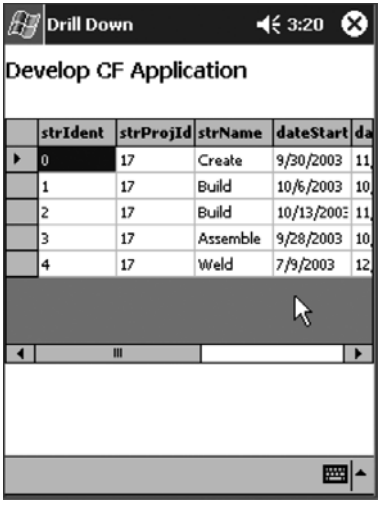


FIGURE 4.13: Response to the Drill-Down Request

sensitive selections. For our current purposes, this context menu is simply the mechanism that allows the user to request a drill-down operation.

Because the routine that drills down into the selected project requires the project's primary key as an input parameter, we once again use both simple and complex data binding.

The complete drill-down application is presented in Listings 4.13 through 4.15. We begin our application with five reference variables to hold information we will need during the course of the application plus two to hold context menu entries, as shown in Listing 4.13.

LISTING 4.13: The Start of the Drill-Down Program

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using YaoDurant.Data;
using YaoDurant.GUI;

namespace DrillDownSameForm
{
   public class FormMain : System.Windows.Forms.Form
   {
      internal System.Windows.Forms.Label lblProjectName;
      private System.Windows.Forms.DataGrid dgridDisplay;
      private System.Windows.Forms.ContextMenu cmenuDrill;
```

Download at WoweBook.Com


```
private System.Windows.Forms.MainMenu mainMenu1;

public FormMain()
{
   InitializeComponent();
}
protected override void Dispose( bool disposing )
{
   base.Dispose( disposing );
}
#region Windows Form Designer generated code

static void Main()
{
   Application.Run(new FormMain());
}

//  A reference to the DataSet
private DataSet dsetTimeTracker;
//  A reference to the Projects DataTable
private DataTable dtabProjects;
//  A reference to the Tasks DataTable
private DataTable dtabTasks;

//  Two context menu items
private MenuItem mitemDrillDown = new MenuItem();
private MenuItem mitemDrillUp = new MenuItem();
```

The reference variables are followed by our `FormLoad` event handler, which performs the data binding and styling, as shown in Listing 4.14.

Listing 4.14: The Form's `Load` Event Handler

```
private void FormMain_Load(object sender, EventArgs e)
{
   // Create utility objects.
   YaoDurant.Data.UtilData utilData = new UtilData();
   YaoDurant.GUI.UtilGUI utilGUI = new UtilGUI();

   //  Set references to data objects.
   dsetTimeTracker = utilData.GetProjectsDataSet();
   dtabProjects = dsetTimeTracker.Tables["Projects"];
   dtabTasks = dsetTimeTracker.Tables["Tasks"];

   //  Make the Projects table the DataSource.
   //  Make the strIdent field of the currently
   //     selected row the Text property of the DataGrid
   //     control.
   dgridDisplay.DataSource = dtabProjects;
```

Download at WoweBook.Com


```
   dgridDisplay.DataBindings.Clear();
   dgridDisplay.DataBindings.Add(
      "Text",
      dgridDisplay.DataSource,
      "strIdent");

   //  Use a utility routine to style the
   //     layout of Projects in the DataGrid.
   UtilGUI.AddCustomDataTableStyle(dgridDisplay,
      "Projects");

   //  Initialize the context menu.
   InitContextMenu();
}
```

When the user taps within the `DataGrid` control, we need to determine whether the tap is within a row header, and if so, we need to present the context menu that allows the user to request a drill-down operation, as shown in the following code:

```
private void dgridDisplay_MouseUp(object sender,
                                  MouseEventArgs e)
{
//  If the user has clicked in a row header,
//     display the context menu.
   if (dgridDisplay.HitTest(e.X, e.Y).Type ==
       DataGrid.HitTestType.RowHeader )
   {
      dgridDisplay.ContextMenu.Show(
                                dgridDisplay,
                                new Point(e.X, e.Y));
   }
}
```

When the user requests a drill-down operation, we call the `DrillDown` routine, passing it the identity value of the selected row:

```
private void mitemDrillDown_Click(object sender,
                                  EventArgs e)
{
   //  The Text property of the DataGrid has been bound
   //     to the strIdent column of the DataSource row.
   //     So, the identity of the selected project will
   //     be in the Text property.
   DrillDown(dgridDisplay.Text);
}
```

Download at WoweBook.Com

Note how much simpler this code is because of the simple data binding that we established during the `Load` event, a binding that causes the row's primary key value to be bound to the `Text` property of the `DataGrid` control.

The `DrillDown` routine, which replaces the information in the `DataGrid` control with the list of required tasks for the chosen project, is shown next. It uses a `DataView` of the `Tasks DataTable`, which it limits to just the tasks of the selected project by setting the `DataView`'s `Filter` property, and then binds the `DataView` to the `DataGrid`. We cover the `DataView` class in greater detail in Chapter 6.

```
private void DrillDown(string  strProjIdent)
{
   //  Note which project is being displayed.
   lblProjectName.Text =
      dtabProjects.Rows[
         dgridDisplay.CurrentCell.RowNumber]["strName"].
            ToString();

   //  Create a view of the Tasks table.
   DataView  dviewProjectTasks = new DataView(dtabTasks);

   //  Set it to display only strProjIdent tasks.
   dviewProjectTasks.RowFilter =
               "strProjIdent = '" + strProjIdent + "'";

   //  Bind it to the DataGrid control.
   dgridDisplay.DataSource = dviewProjectTasks;
}
```

If the user requests a drill-up operation, the code is even simpler, for all that is needed is to clear the `Label` control and to rebind the project table to the `DataGrid`, as shown in the following code:

```
private void DrillUp() {
   //  Bind the Projects DataTable to the DataGrid control.
   dgridDisplay.DataSource = dtabProjects;

   //  Clear the project name display.
   lblProjectName.Text = string.Empty;
}
```

Two other routines, in support of the context menu, are needed to complete our program. The first one initializes the context menu by adding the two possible choices to it:

Download at WoweBook.Com


```
      private void InitContextMenu()
      {
         //  Add "Drill Down" and "Drill Up" entries
         //     to the context menu.
         cmenuDrill.MenuItems.Clear();

         mitemDrillDown.Text = "Drill Down";
         mitemDrillDown.Click +=
            new EventHandler(mitemDrillDown_Click);
         cmenuDrill.MenuItems.Add(mitemDrillDown);

         mitemDrillUp.Text = "Drill Up";
         mitemDrillUp.Click +=
            new EventHandler(mitemDrillUp_Click);
         cmenuDrill.MenuItems.Add(mitemDrillUp);
      }
```

The second routine enables and disables the context menu items so that the user cannot select Drill Down when already down or Drill Up when already up. Listing 4.15 shows the end of the program.

Listing 4.15: The End of the Drill-Down Program

```
      private void cmenuDrill_Popup(object sender, EventArgs e)
      {
      //  If already "Up", the user can only go "Down",
      //     and vice versa.
         if ( dgridDisplay.DataSource == dtabProjects )
         {
            mitemDrillDown.Enabled = true;
            mitemDrillDown.Checked = false;
         } else {
            mitemDrillDown.Enabled = false;
            mitemDrillDown.Checked = true;
         }
         mitemDrillUp.Enabled = ! mitemDrillDown.Enabled;
         mitemDrillUp.Checked = ! mitemDrillDown.Checked;
      }
   }
}
```

So, by understanding how to perform a hit test to determine the mouse location within a `DataGrid` control, by knowing how to obtain information about the current cell, and by making full use of both simple and complex data binding, we can provide drill-down capability to the application with a minimum amount of code.

Download at WoweBook.Com

Let's move on to address the other user input challenge associated with the DataGrid control: in-place editing.

4.4.9 Providing In-Place Editing Capability

A .NET Compact Framework DataGrid control is used for display only. Unlike its desktop counterpart, it does not provide any data modification capability to the user. To overcome the display-only limitation of the DataGrid control, you must provide the data modification capability yourself.

Your application must take two steps to provide in-place editing for a DataGrid control.

1. Present a control capable of accepting input, such as a TextBox, preferably with the text already highlighted, whenever the user indicates that he wishes to modify the contents of the CurrentCell.
2. Provide a mechanism for the user to indicate that he has completed, or is canceling the update. If an update was requested, move the contents of the input control into the CurrentCell. If appropriate, hide the input control.

Here are some things to keep in mind as you design your in-place editing functionality.

- You could respond to an update request in a manner similar to Microsoft Excel by permanently displaying a TextBox control above the grid. Alternatively, you could choose to temporarily display the TextBox control directly over the current cell for the duration of the data entry.
- If you choose to display the input control only while the update is in progress, it will be difficult to automate the edit, update, and cancel steps. That is, it's tricky to make the input control automatically appear with highlighted text as the user taps, tabs, or arrows into a new current cell. It's also hard to update the old current cell as the user moves into a new current cell while the old current cell is being edited.
- You will not need to update the underlying data object because data binding will do that for you as you update the DataGrid control.

Download at WoweBook.Com

Because of the complexity involved in doing "automated" in-place editing, consider providing a manual capability in which the user explicitly requests editing, updating, and canceling. Figure 4.14 shows an application that uses the manual method, with Edit, Update, and Cancel buttons. Once you have a working version of manual in-place editing in your application, you can consider upgrading to automated in-place editing.

In the remainder of this chapter, we will look at providing manual in-place updating first and then address the challenges of automating it. We begin by writing an application to do the manual variation.

In this application (located in the `DisplayBindingInfo` project and shown in Listings 4.16 through 4.19), we use the collocation technique, placing our input control over the current cell while the editing is in progress, as shown in Figure 4.14, and removing it when the update or cancel operation is complete. This emphasizes the in-place aspect of the editing and gives you some experience using the location properties of the `CurrentCell` object.

What we need to do now is create, position, show, and hide a `TextBox` control; and to locate it directly on top of the `DataGrid`'s `CurrentCell`. Table 4.12 earlier in this chapter listed the methods for obtaining the bounds of the `CurrentCell` object and accessing its value. All we need to do now is put this information to work and write our manual in-place editing application.

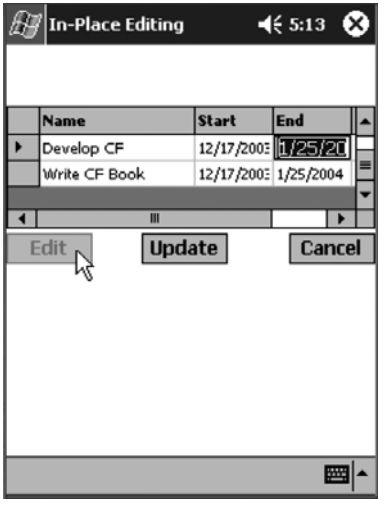


FIGURE 4.14: Manual In-Place Editing

Download at WoweBook.Com

The application displays the DataGrid control and three Command buttons: Edit, Update, and Cancel. The buttons allow users to indicate what they want done. However, because the Task Count field represents the number of required tasks associated with this project, it is a nonupdatable field; we must prevent users from attempting to update this field. The application does this by checking the column name in the bound data object. If users attempt to update a field of the Tasks column, they are presented with a message box informing them it is a nonupdatable field.

As expected, we use the same database simulation routines and DataGrid styling routine for obtaining and displaying information that we have used in the previous applications in this chapter.

When the application starts, the user is presented with a DataGrid control that displays project information. After the user has selected a specific cell and then clicked on the Edit menu selection, the form presents the edit control, located on top of the current cell, as shown in Figure 4.14.

We should mention here that although the TextBox control is collocated with the CurrentCell object, they are not tied together. This will become obvious to the user if he now scrolls the DataGrid control. The cells of the DataGrid will move, but the TextBox will not, resulting in the display shown in Figure 4.15.

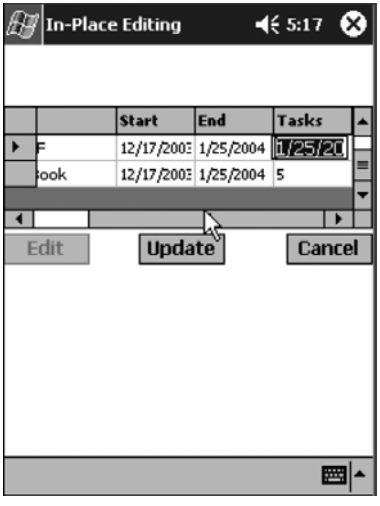


FIGURE 4.15: A DataGrid Control with the Projects Table Bound to It

Download at WoweBook.Com

We begin our application code by declaring a variable that will hold a reference to the input TextBox control, as shown in Listing 4.16.

Listing 4.16: The Start of the Manual In-Place Editing Program

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using YaoDurant.Data;
using YaoDurant.GUI;

namespace InPlaceEditManual
{
   public class FormMain : System.Windows.Forms.Form
   {
      internal System.Windows.Forms.Label Label1;
      internal System.Windows.Forms.Panel Panel1;
      internal System.Windows.Forms.Button cmdCancel;
      internal System.Windows.Forms.Button cmdUpdate;
      internal System.Windows.Forms.Button cmdEdit;
      private System.Windows.Forms.DataGrid dgrdProjects;
      private System.Windows.Forms.MainMenu mainMenu1;

      public FormMain()
      {
         InitializeComponent();
      }
      protected override void Dispose( bool disposing )
      {
         base.Dispose( disposing );
      }
      #region Windows Form Designer generated code

      static void Main()
      {
         Application.Run(new FormMain());
      }

      //  A TextBox control to be used for in-place editing
      private TextBox textEdit;
```

In the form's Load event, we bind the Projects DataTable to the DataGrid control and then apply the styling. Then we create our TextBox control and add it to the DataGrid's controls collection. We want the TextBox to be contained within the DataGrid, not the form, because we want to position the TextBox using DataGrid coordinates and we want to ensure that

Download at WoweBook.Com

when the DataGrid is not visible, the TextBox also is not visible. Finally, we set the application state to "Not Editing." Listing 4.17 shows the code.

Listing 4.17: The Form's Load Event Handler

```
private void FormMain_Load(object sender, EventArgs e)
{
   //  Make the Projects table the DataSource.
   YaoDurant.Data.UtilData  utilData = new UtilData();
   dgrdProjects.DataSource = utilData.GetProjectsDT();

   //  Use a utility routine to style the
   //     layout of Projects in the DataGrid.
   UtilGUI.AddCustomDataTableStyle(dgrdProjects,
                                   "Projects");

   //  Create the TextBox to be used with in-place
   //     editing and add it to the DataGrid control.
   textEdit = new TextBox();
   textEdit.Visible = false;
   this.dgrdProjects.Controls.Add(textEdit);

   //  Set the form's GUI state to "Not Editing."
   this.SetEditingState(false);
```

As shown in Listing 4.18, when the user taps on the Edit button, we check to see whether he is attempting to edit the read-only ctTasks field. If so, we display a message box informing the user that the field cannot be edited; otherwise, we position the TextBox control within the bounds of the CurrentCell object, load the contents of the CurrentCell into the TextBox, highlight the TextBox, show it, set the focus to it, and set the application state to "Editing."

Listing 4.18: Initiating Editing by Displaying the TextBox Control

```
private void cmdEdit_Click(object sender, EventArgs e)
{
   //  Check the DataSource column's name;
   //     if it is ctTasks, do not update.
   DataGridCell cellCurr = dgrdProjects.CurrentCell;
   if ( dgrdProjects.TableStyles["Projects"].
               GridColumnStyles[cellCurr.ColumnNumber].
                  MappingName == "ctTasks" )
   {
      MessageBox.Show("Count of tasks only changes as" +
                      " the result of adding / removing" +
                      " a task.");
      return;
```

Download at WoweBook.Com


```
    }

    //  Position textEdit for in-place editing.
    textEdit.Bounds = dgrdProjects.GetCellBounds(
                                  cellCurr.RowNumber,
                                  cellCurr.ColumnNumber);

    //  Load the CurrentCell's value into textEdit.
    textEdit.Text = dgrdProjects[
                        cellCurr.RowNumber,
                        cellCurr.ColumnNumber].ToString();

    //  Highlight the text.
    textEdit.SelectAll();

    //  Show textEdit and set the focus to it.
    textEdit.Visible = true;
    textEdit.Focus();

    //  Set the form's GUI state to "Editing."
    this.SetEditingState(true);
}
```

When the user clicks the Update button, we set the contents of the `CurrentCell` object to `textEdit.Text` and set the application state to "Not Editing." Data binding automatically updates the bound data object's data element for us.

```
private void cmdUpdate_Click(object sender, EventArgs e)
{
    //  Move the contents of textEdit
    //     into the CurrentCell.
    dgrdProjects[dgrdProjects.CurrentCell.RowNumber,
                 dgrdProjects.CurrentCell.ColumnNumber] =
                 textEdit.Text;

    //  Set the form's GUI state to "Not Editing."
    this.SetEditingState(false);
}
```

When the user taps on the Cancel button, we simply set the editing state to "Not Editing," as shown here:

```
private void cmdCancel_Click(object sender, EventArgs e)
{
    //  Set the form's GUI state to "Not Editing."
    this.SetEditingState(false);
}
```

Download at WoweBook.Com

Finally, Listing 4.19 shows the routine to toggle the application state between "Editing" and "Not Editing."

Listing 4.19: The End of the Manual In-Place Editing Program

```
      private void SetEditingState( bool boolInProcess)
      {
         textEdit.Visible = boolInProcess;
         cmdEdit.Enabled = !boolInProcess;
         cmdCancel.Enabled = boolInProcess;
         cmdUpdate.Enabled = boolInProcess;
      }
   }
}
```

So, by understanding the properties of the `DataGrid` control, the `CurrentCell` object, the styling objects, the data object, and data binding itself, we're able to compensate for the display-only nature of the `DataGrid` control and provide a data-updating mechanism to the user.

Now let's raise the bar one notch and see what problems we encounter when we try to automate our in-place editing capability (i.e., so the input control automatically appears, with highlighted text, as the user taps, tabs, or arrows into a new current cell; and so the cell being edited automatically updates as the user taps a different cell).

4.4.10 Providing Automated In-Place Editing Capability

For reasons we are about to cover, providing in-place editing capability for a `DataGrid` control is more complicated than you might expect. Thus, we keep our functional specification as simple as possible.

- When a cell becomes the `CurrentCell` object, the collocated `TextBox` control with highlighted text must appear.
- The user can cancel the current edit by pressing the Escape key when entering text into the `TextBox` control or by tapping in the row header of the current row and choosing Cancel from the menu. In either case, the cell will *not* be updated, and the `TextBox` control will be removed.
- The user can commit the current update by pressing the Enter key when entering text into the `TextBox` control or by tapping in the row header of the current row and choosing Update from the pop-up menu that appears. In either case, the cell will be updated and the `TextBox` will be removed.

Download at WoweBook.Com

- If the user taps on a new cell while editing the current cell, the old cell being edited will be updated and the edit process will begin at the new cell.

> **NOTE**
> You cannot display a context menu as the result of a tap on the `TextBox` control because `TextBox` controls do not receive mouse messages.

To automate your in-place editing capability, you need to know which events to react to and the sequence in which those events occur. You also need to know that one of those events will occur in the wrong sequence.

When the user taps on a data cell that is not the `CurrentCell` object, the following sequence occurs.

1. The `MouseDown` event is raised.
2. The `DataGrid` control's `CurrentCell` property is set to the new current cell.
3. The `CurrentCellChanged` event is raised.
4. The `Click` event is raised.
5. The `MouseUp` event is raised.

When you automate the presentation of the input control to the user, the correct event to react to is the `CurrentCellChanged` event. This event tells you that the user has selected a new cell, which in turn tells you that you should display the input control for modifying the contents of the cell.

This event sequence doesn't look like a problem, but it is. It is a problem because the `CurrentCellChanged` event is caused by the mouse coming down, not by the mouse coming up. This is different from most other generic mouse-driven events. For example, the `Button.Click`, `ComboBox.SelectedIndexChanged`, and `TextBox.Validating` events are triggered by the mouse coming up, thus assuring you that there will be no subsequent mouse-related activity after you handle the event. This is not the case with the `DataGrid` control's `CurrentCellChanged` event.

If the user taps on a cell and you respond to the resulting `CurrentCellChanged` event by displaying a `TextBox` control containing highlighted text, that text is not going to remain highlighted for long—the user is about to

Download at WoweBook.Com

release the mouse button and the MouseUp event is about to occur. Because your input TextBox control is now located where the mouse is, it will receive the MouseUp-related events, one of which will cause it to unhighlight the text and reposition the text cursor. This is not the behavior you are trying to present to the user.

You might decide to wait until the MouseUp event occurs and then present the TextBox control. That solution will not work because the MouseUp event might never occur; the user might have reached the new CurrentCell object by tabbing or using an arrow key. So, in the CurrentCellChanged event handler you need to know whether the current cell is changing because of mouse activity. In the .NET Compact Framework, you cannot interrogate the status of a mouse except during a mouse event, so you must respond to three mouse-related events: MouseDown, to note whether editing should begin when the CurrentCellChanged event occurs or be delayed until the MouseUp event arrives; CurrentCellChanged; and MouseUp.

If you are editing the CurrentCell object and the user taps on a different cell, the challenge becomes even greater because now you need to close out the editing of the old cell and initiate the editing of the new cell. We will be reacting to the TextBox control's Validating event; the event that notifies you that the entering of data into the old cell is finished. The problem is that, as we have seen, the CurrentCellChanged event occurs as the mouse comes down, while the Validating event is caused by the mouse coming up. Thus, the event that tells you to close out the old cell arrives after the event that tells you to begin editing the new cell. This means that when you react to the Validating and Validated events, the CurrentCell object has already changed from the cell you need to update to the cell you are going to edit. Therefore, you must save the identity of the CurrentCell object as you begin the editing process so that you will update the correct cell.

But updating a cell makes it the CurrentCell object. So, now you not only must store the identity of the old CurrentCell as you begin the edit process, but also must store and restore the identity of the new CurrentCell as you perform the update.

All of this is why we tried to keep our functional specifications simple. The resulting application is located in the InPlaceEditAuto project.

As always, we'll use the same database simulation routines and DataGrid styling routine for obtaining and displaying information that we have used in the previous applications in this chapter.

Download at WoweBook.Com

Because we understood the meaning and sequence of the events that we must handle, our code is as small and simple as we can make it. Not as small and simple as the previous application, but considering the issues related to automating the editing process, this code is quite reasonable. The complete automatic in-place editing application is presented in Listings 4.20 through 4.27.

We begin by declaring the necessary variables, as shown in Listing 4.20.

Listing 4.20: The Start of the Automatic In-Place Editing Program

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.ComponentModel;
using YaoDurant.Data;
using YaoDurant.GUI;


namespace InPlaceEditAuto
{
   public class FormMain : System.Windows.Forms.Form
   {
      private System.Windows.Forms.MainMenu mainMenu1;
      private System.Windows.Forms.ContextMenu cmenuEdit;
      private System.Windows.Forms.DataGrid dgrdProjects;

      public FormMain()
      {
         InitializeComponent();
      }

      protected override void Dispose( bool disposing )
      {
         base.Dispose( disposing );
      }
      #region Windows Form Designer generated code

      static void Main()
      {
         Application.Run(new FormMain());
      }


      //  A TextBox control to be used for in-place editing
      private TextBox textEdit;
```

Download at WoweBook.Com


```
private MenuItem mitemUpdate = new MenuItem();
private MenuItem mitemCancel = new MenuItem();

//  The DataGrid's CurrentCell.  We must save it
//     because a new CurrentCell may be assigned
//     before we are finished with the old one.
private DataGridCell CurrentCell;

//  These fields are used to determine when
//     in-place editing should begin.
private bool boolMouseDriven;
private bool boolDoEdit;

//  This field is to track whether the update
//     should be completed or cancelled.
private bool boolCancelUpdate;
```

As in the previous version of the application, in the form's `Load` event we bind the `Projects DataTable` to the `DataGrid` control and then apply the styling. Then we create our `TextBox` control and add it to the `DataGrid`'s controls collection. We want the `TextBox` to be contained within the `DataGrid`, not within the form, because we want to position the `TextBox` using `DataGrid` coordinates and we want to ensure that when the `DataGrid` is not visible, the `TextBox` also is not visible. We also build the context menu at this time. Listing 4.21 shows the code.

Listing 4.21: The Form's `Load` Event Handler

```
private void FormMain_Load(object sender, EventArgs e)
{
   //  Make the Projects table the DataSource.
   YaoDurant.Data.UtilData  utilData = new UtilData();
   dgrdProjects.DataSource = utilData.GetProjectsDT();

   //  Use a utility routine to style the
   //     layout of Projects in the DataGrid.
   UtilGUI.AddCustomDataTableStyle(dgrdProjects,
      "Projects");

   //  Create the TextBox to be used with in-place
   //     editing and add it to the DataGrid control.
   textEdit = new TextBox();
   textEdit.Visible = false;
   this.dgrdProjects.Controls.Add(textEdit);

   //  Initialize the Update/Cancel context menu.
   this.InitContextMenu();
```

Download at WoweBook.Com


```
      //  The default is:  Do the update.
      this.boolCancelUpdate = false;
   }
```

In the MouseDown event handler, shown in Listing 4.22, we decide whether the context menu should be displayed and also note that mouse activity is in progress.

LISTING 4.22: The MouseDown Event Handler

```
   private void dgrdProjects_MouseDown(object sender,
                                       MouseEventArgs e)
   {
      //  If editing is in progress
      if( textEdit.Visible && e.Button == MouseButtons.Right )
      {
         //  If the user tapped in the row header of the row
         //     being edited, display the Update/Cancel
         //     context menu to the user.
         if( dgrdProjects.HitTest(e.X, e.Y).Type ==
             DataGrid.HitTestType.RowHeader
         &&  dgrdProjects.HitTest(e.X, e.Y).Row ==
             dgrdProjects.CurrentRowIndex )
         {
            cmenuEdit.Show(textEdit, new Point(0, 0));
         }
      }

      //  When the user taps on a data cell or a row
      //     header, the current cell will change.
      //     Our CurrentCellChanged event will need to
      //     know that this was caused by a mouse tap,
      //     rather than by a tab or programmatically.
      boolMouseDriven =
         ( (dgrdProjects.HitTest(e.X, e.Y).Type ==
            DataGrid.HitTestType.Cell)
         ||
           (dgrdProjects.HitTest(e.X, e.Y).Type ==
            DataGrid.HitTestType.RowHeader) );
   }
```

In the CurrentCellChanged event handler, shown in Listing 4.23, we decide whether to begin the editing process or to defer it until the ButtonUp event arrives.

LISTING 4.23: The CurrentCellChanged Event Handler

```
   private void dgrdProjects_CurrentCellChanged(object sender,
                                                EventArgs e)
```

Download at WoweBook.Com


```
{
   // If this is a mouse-caused event, we must wait
   //    for the Click and MouseUp events to complete
   //    before displaying the edit TextBox.  If not
   //    caused by a mouse, we can display the edit TextBox
   //    after this message has completed processing.
   if ( boolMouseDriven )
   {
      boolDoEdit = true;
   }
   else
   {
      this.InitEdit();
   }
   boolMouseDriven = false;
}
```

In the `MouseUp` event handler, we begin the editing process, if needed:

```
private void dgrdProjects_MouseUp(object sender,
                                  MouseEventArgs e)
{
   //  If editing has been requested, we need
   //     to display textEdit to the user.
   if ( boolDoEdit )
   {
      this.InitEdit();
   }
}
```

In the event handlers related to the two methods we gave the user for committing and canceling the edit, we note whether *commit* or *cancel* was requested and then hide the edit `TextBox` control, which will result in the `TextBox`'s `Validating` event being fired. Listing 4.24 shows the code.

Listing 4.24: The Context Menu's `MenuItem_Click` Event Handler

```
private void MenuItem_Click(object sender,
                            EventArgs e)
{
   //  Note which was requested.
   //  Hide the edit TextBox.
   this.boolCancelUpdate = (sender == mitemCancel);
   textEdit.Visible = false;
}
private void textEdit_KeyUp(object sender,
   KeyEventArgs e)
{
```

Download at WoweBook.Com


```
   //  Check to see if the keystroke was enter or escape.
   switch (e.KeyCode)
   {
      case Keys.Enter:
         this.boolCancelUpdate = true;
         textEdit.Visible = false;
         break;
      case Keys.Escape:
         this.boolCancelUpdate = false;
         textEdit.Visible = false;
         break;
      default:
         break;
   }
}
```

When the `Validating` event arrives, we check to see whether we should update or cancel and proceed accordingly:

```
private void textEdit_Validating(object sender,
                                 CancelEventArgs e)
{
   //  To cancel or not to cancel,
   //     that is the question.
   e.Cancel = boolCancelUpdate;
   boolCancelUpdate = false;
}
```

Because of what we did when handling the `Validating` event, the `Validated` event will fire only if the update should be done. Performing the update consists of saving the identity of the new `CurrentCell` object, updating the old `CurrentCell`, and restoring the new `CurrentCell`, as shown in Listing 4.25.

Listing 4.25: The `TextBox` Control's `Validated` Event Handler

```
private void textEdit_Validated(object sender,
   EventArgs e)
{
   //  Do the update.
   //  Two issues must be addressed here.  The cell
   //     we need to update might no longer be
   //     the CurrentCell; and modifying the contents of
   //     a cell makes it the CurrentCell.  Therefore,
   //     we need to save the identity of the CurrentCell
   //     at the start, update the cell whose identity
   //     was saved during InitEdit, then restore the
   //     identity of the CurrentCell.
```

Download at WoweBook.Com


```
   //  Save the identity of the CurrentCell.
   DataGridCell CurrentCell = dgrdProjects.CurrentCell;

   //  Move the contents of textEdit into the
   //     "correct" CurrentCell.
   dgrdProjects[this.CurrentCell.RowNumber,
      this.CurrentCell.ColumnNumber] =
      textEdit.Text;

   //  Restore the identity of the CurrentCell.
   dgrdProjects.CurrentCell = CurrentCell;
}
```

Our routine to initiate the editing process locates the edit `TextBox` control over the `CurrentCell` object, places the `CurrentCell` contents into the `TextBox`, highlights the text, gives it the focus, and notes the identity of the cell being edited. Listing 4.26 shows the code.

Listing 4.26: The Method for Initializing the Editing Process

```
internal void InitEdit()
{
   //  Add the event handlers.
   textEdit.KeyUp += new KeyEventHandler(textEdit_KeyUp);
   textEdit.Validating +=
      new System.ComponentModel.CancelEventHandler
         (textEdit_Validating);
   textEdit.Validated +=
      new EventHandler(textEdit_Validated);

   //  Position textEdit for in-place editing.
   DataGridCell cellCurr = dgrdProjects.CurrentCell;
   textEdit.Bounds = dgrdProjects.GetCellBounds(
      cellCurr.RowNumber,
      cellCurr.ColumnNumber);
   textEdit.Text = dgrdProjects[cellCurr.RowNumber,
                                cellCurr.ColumnNumber].
                                       ToString();
   textEdit.SelectAll();
   textEdit.Visible = true;
   textEdit.Focus();

   //  Save the CurrentCell.  We must save it because,
   //     if the user terminates the edit by tapping on
   //     a different cell, a new CurrentCell will be
   //     assigned before we are finished with the old
   //     one.
   this.CurrentCell = dgrdProjects.CurrentCell;
```

Download at WoweBook.Com


```
         //  Set the form's GUI state to "Editing."
         boolDoEdit = false;
      }
```

Finally, Listing 4.27 shows the routine to build the context menu.

Listing 4.27: The End of the Automatic In-Place Editing Program

```
      private void InitContextMenu()
      {
         //  Add "Update" and "Cancel" entries
         //     to the context menu.
         mitemUpdate.Text = "Update";
         mitemUpdate.Click += new EventHandler(MenuItem_Click);
         mitemCancel.Text = "Cancel";
         mitemCancel.Click += new EventHandler(MenuItem_Click);

         cmenuEdit.MenuItems.Clear();
         cmenuEdit.MenuItems.Add(mitemUpdate);
         cmenuEdit.MenuItems.Add(mitemCancel);
      }

   }
}
```

Figure 4.16 shows our application after the user tapped on the Name column of the third row, entered "A New Project," and then tapped on the Name column of the first row.

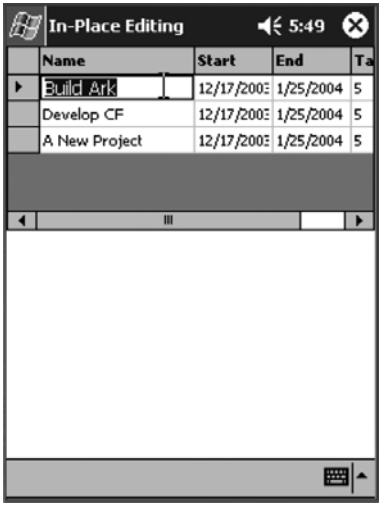


Figure 4.16: Moving to a New Cell

Download at WoweBook.Com

So, having made some good decisions about which events to handle and with an understanding of the cause and sequence of those events, we were able to overcome several challenges and provide an automated in-place editing application using a `DataGrid` control. We can only dream about how many fewer event handlers and how much less code we would have needed if the `CurrentCellChanged` event had been triggered by the mouse button's release, which, in turn, would have caused the `CurrentCellChanged` event to arrive after the `Validated` event.

The challenge of automated in-place editing is definitely the last challenge we are going to attempt in this chapter, for it completes our coverage of data binding, from complex data binding with `ListBox` and `TextBox` controls to simple data binding with any control to both methods of binding with the `DataGrid` control.

4.5 Conclusion

Data binding provides a consistent PME-oriented mechanism for staging information that has been retrieved from a database into standard data objects and then associating those objects with the controls used to present that data to the user. It is a bidirectional process, allowing the changes made within the controls to propagate back to the data objects. In Chapter 6 we will see how to complete the journey, moving the data between the data object and the database of choice.

Controls vary in their data-binding capability. All are capable of simple data binding through the `Bindings` collection derived from the `Control` base class. Additionally, the list-oriented controls, `ListBox` and `ComboBox`, can take advantage of complex binding through the `DataSource` property. The control developed specifically for binding to a data object, the `DataGrid`, is the most capable and the most complex; it provides for tabular displaying, styling, scrolling, and tracking of the current cell. It is the only .NET Compact Framework control written in managed code.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

5 Storage

In this chapter, we examine the .NET Compact Framework's capabilities for accessing local storage, concentrating on file I/O and registry access.

5.1 Smart-Device Data Storage

STORAGE IS AN AREA where Windows CE–powered devices and desktop systems differ significantly. The differences are unseen at the API level, and indeed a common set of APIs—both native and managed—work in both environments. To see the differences, you need to dig down below the level of the programming interfaces.

One difference involves the storage media. Desktop systems rely on rotating disk drives for storage, while Windows Mobile stores data in semiconductor-based storage devices. Among the benefits are performance and durability. Semiconductors allow faster data access times when compared to rotating media. Semiconductor memory is also more shock resistant compared to a rotating hard drive.

The form of semiconductors used for storage has changed over time. Prior to Windows Mobile 5.0, volatile RAM was used for the file system and system registry. Starting with Windows Mobile 5.0, nonvolatile forms of semiconductors, including NAND flash memory, have become the standard for Windows Mobile storage.

Another difference from desktop operating systems is the available file systems. On Windows Vista, NTFS is the standard file system with some support for various forms of the FAT file system for storage cards. With Windows Mobile, a transaction-safe version of the FAT file system, known as Transaction-Safe File System (TFAT), is the type used for the main file system.

Download at WoweBook.Com

While these details are hidden below the programming interface, be aware that the file system uses a 512-byte sector. To help optimize the speed of storage operations, it is prudent to read and write data in larger rather than smaller chunks. Specifically, read and write in chunks of 512 bytes or a multiple of 512 bytes to minimize file system overhead.

In spite of these differences in the physical media, there are many common features at a higher level of abstraction. Both device and desktop storage support a hierarchical file system. And both device and desktop storage support a hierarchical registry.

The Windows Mobile storage model provides for three basic kinds of storage: (1) files in the file system, (2) registry values, and (3) property databases. This chapter covers two of these three types. We cover the use of .NET Compact Framework classes to access the file system and the system registry. We do not cover Windows CE property databases aside from the description in the related sidebar.

Windows CE Property Databases

Most Windows CE features are ported from desktop Windows. However, certain features were created to serve a need not met by existing desktop technologies. Windows CE property databases are one of those technologies. Property databases provide a simple, low-overhead mechanism for storing record-based data. A property database provides, in essence, a very flexible flat-file manager. A fairly simple set of C-callable functions are provided to create records. The names of the functions reflect the Windows CE–centric nature of these functions; each has a prefix of `Ce`. Included among the property database functions are `CeCreateDatabaseEx` to create a new database, `CeOpenDatabaseEx` to open an existing database, `CeSeekDatabaseEx` to search for a record, `CeReadRecordPropsEx` to read a record, and `CeWriteRecordProps` to write a record.

The .NET Compact Framework does not support accessing CE property databases, except through P/Invoke wrappers for these functions. We do not cover that topic in this book. If you need to access CE property databases from a .NET Compact Framework program, you have two choices. One is to first write a set of Win32 functions that do all the main database work and then write a set of custom wrapper functions to pass your database records up to your managed code.

Download at WoweBook.Com

Your second choice involves writing P/Invoke wrappers for direct calls to the CE property database functions. The P/Invoke Wizard described in Chapter 3 can help. We created a set of wrappers for a related set of functions, the desktop Remote API (RAPI) functions. As described in Chapter 9, RAPI functions allow a desktop program to access the object store of a Windows CE device using ActiveSync. We used our wrappers to create the sample programs we wrote for Chapter 9; the complete source code for the wrappers is available online at the book's Web site.

You can use these same wrappers within a .NET Compact Framework program because RAPI functions are identical to device-side functions. In fact, the similarity can sometimes make reading the documentation difficult because you cannot tell whether the RAPI function or the device-side function is being described. There is, for example, a RAPI function named `CeCreateDataBaseEx` that can be used to remotely create a database. There is also a device-based function named `CeCreateDataBaseEx` that does the same thing. Each function takes the same number and type of parameters. So, the P/Invoke wrappers we wrote for the RAPI functions can also be used from within your .NET Compact Framework program to run on a device.

5.1.1 Installable File Systems

In addition to the RAM-based object store, Windows CE supports **installable file systems**, which provide nonvolatile data storage. In addition to the main nonvolatile storage that has become standard on Windows Mobile devices, most devices also provide a mechanism for removable storage media. As of this writing, the most common format is the Secure Digital (SD) card. This is not the only available format, and there is word that a new format, Universal Flash Storage (UFS), may be coming to a Windows Mobile device near you.

Storage cards appear within the file system with a name like `\Storage Card`. There are no standards for this name, and because this is a localizable name, you should avoid hard-coding this name in your code.

Windows Mobile programmers can discover the directory name for storage cards by calling the Win32 functions `FindFirstFlashCard` and `FindNextFlashCard` to enumerate installed file systems. (See the `FindMemoryCard` sample application in Chapter 3.)

Download at WoweBook.Com

5.1.2 The Windows CE File System

The Windows CE file system provides a unified view of all available storage devices, including the following:

- Files and directories from the operating system image (ROM-based)
- Files and directories in the main file system (flash-based)
- Files and directories on installable file systems

The Windows CE Platform Builder documentation describes this as "seamless integration" of each of these different file systems.

The architects of Windows CE simplified the file system structure from the legacy MS-DOS/Windows disk storage architecture. They eliminated drive letters so that Windows CE uses a simpler single-tree file system hierarchy. Instead of 26 possible starting points, there is only one: the root directory (\).

Because Windows CE does not support a "current directory" for a process, you must fully qualify all pathnames in your application. Aside from these differences, the file system directory structure has similarities to desktop file systems: a hierarchical file system with support for long filenames. The maximum file path on Windows CE, as on desktop file systems, is 260 characters.

The Windows CE file system does support memory-mapped files, thus providing a mechanism for interprocess memory sharing. However, the .NET Compact Framework does not support memory-mapped files.

5.1.3 ROM-Based Files

Files located in ROM reside in the `\windows` directory and are read-only by definition. Many have the `system` or `hidden` attribute or both. At system boot time, the object store is initialized with whatever directories may be required. At that time, ROM-based files can be mapped to other directories besides the `\windows` directory.

Platform Builder packages the Windows CE operating system as a ROM image, compressing some of these files and leaving others uncompressed, as determined by the platform developer. Uncompressed ROM files run in place, in what is called an eXecute In Place (XIP) region. An advantage of XIP regions is that ROM images can be mapped directly into virtual memory, allowing system files to be available without using precious program memory. On the downside, some ROM is slower than RAM, so code running in ROM may execute a little more slowly than code located elsewhere.

Download at WoweBook.Com

The \windows folder can also include files from the RAM-based object store. You can copy files and even overwrite some of the ROM-based files that are in the \windows folder. Of course, the file system does not actually copy to or overwrite ROM files, but it does note when you have a newer version of a ROM file in RAM and uses the newer version whenever the file is requested.

Another difference between Windows CE–powered systems and desktop systems is that Windows CE does not support a `PATH` environment variable that determines in what order to search directories when resolving requests to locate programs or shared libraries. Here is the hard-coded search sequence:

- The absolute path (when specified)
- The `.exe` launch directory (when searching for libraries)
- The `\windows` directory
- ROM dynamic link library (DLL) files

While Windows CE has no `PATH` variable, Windows CE from Version 4.1 and later supports a registry key with a search path: `HKEY_LOCAL_MACHINE\Loader`, with the `SystemPath` property. The best resource for details on this subject is the Win32 documentation for Windows CE, specifically the remarks that accompany the Win32 `LoadLibrary` function.

5.2 File I/O

In the .NET Compact Framework—and, for that matter, in the desktop .NET Framework—file I/O and the objects used for handling file I/O can be grouped into three categories.

1. Manipulating directories and files, which includes creating, deleting, discovering, and copying files and maintaining attributes. Table 5.1 lists the `File` and `Directory` classes.
2. Reading and writing bytes to and from a file (i.e., performing low-level file I/O). Byte-level I/O is performed through `Stream` objects, which are shown in Table 5.2 in the Byte-Level I/O subsection.
3. Reading and writing at a higher level, such as the transfer of value data types, Unicode characters, strings, and lines of text. Table 5.3 in the Higher-Level I/O subsection presents these higher-level I/O objects.

Download at WoweBook.Com

The full desktop version of the .NET Framework provides higher-level serialization classes. These classes allow objects to be converted into a flattened, serializable format and then back again into objects. The serialization classes include the following:

- `BinaryFormatter`[1]
- `SoapFormatter`[2]
- `XmlSerializer`[3]

Of these three, the .NET Compact Framework Version 3.5 supports only one: `XmlSerializer`. We will explore using this class to serialize and deserialize objects in this chapter.

5.2.1 The `File` and `Directory` Classes

The managed-code `File` and `Directory` classes, summarized in Table 5.1, provide access to files and directories. They are not used to do the actual I/O, although they can be used to create the objects that are. For instance, `File.AppendText` returns a `StreamWriter` object that can be used to add characters to a file.

TABLE 5.1: File Manipulation Classes

Implementation Class	Description	Available Static Methods

`Directory`	Manipulates directories	`CreateDirectory`
		`Delete`
		`Exists`
		`Move`
		`GetCreationTime`
		`SetCreationTime`
		`GetCurrentDirectory`
		`SetCurrentDirectory`
		`GetDirectories`
		`GetFiles`

1. Fully qualified name: `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`.
2. Fully qualified name: `System.Runtime.Serialization.Formatters.Soap.SoapFormatter`.
3. Fully qualified name: `System.XmlSerialization.XmlSerializer`.

Download at WoweBook.Com

TABLE 5.1: File Manipulation Classes *(Continued)*

Implementation Class	Description	Available Static Methods

`File`	Manipulates files	`Create`
		`CreateText`
		`Copy`
		`Move`
		`Delete`
		`Exists`
		`Open`
		`OpenRead`
		`OpenText`
		`OpenWrites`
		`GetAttributes`
		`SetAttributes`

The `Directory` and `File` classes consist entirely of static methods. Instead of creating instances of objects, you invoke static methods through a class reference, as shown in the following code:

```
using System.IO;
//...
if (! Directory.Exists(strDirName) )
{
   Directory.CreateDirectory(strDirName);
}
Directory.SetCurrentDirectory(strDirName);

if (! File.Exists(strFileName) )
{
   File.Create(strFileName).Close();
}
```

To do the actual I/O you need an I/O object, either a low-level I/O object or one of the higher-level objects that are built on the low-level ones.

5.2.2 Byte-Level I/O

While you use the `File` and `Directory` classes entirely through static methods, the objects used to perform the I/O must be instantiated. The low-level I/O objects, shown in Table 5.2, enable you to read and write arrays of bytes to and from a variety of endpoints in a consistent fashion.

Download at WoweBook.Com

TABLE 5.2: Byte-Level I/O Classes (Streams)

Interface (Abstract Base) Class	Implementation Class	Description	Major PMEs
`Stream`	N/A	Abstract base class; defines reading/writing bytes to a source	`Read Write Seek Close CanSeek`
`Stream`	`FileStream`	Reads/writes bytes to a file	See above
`Stream`	`NetworkStream`	Reads/writes bytes to a network socket	See above
`Stream`	`MemoryStream`	Reads/writes bytes to memory	See above

The `Stream` objects transfer `bytes` to and from a storage medium, such as a file or a network socket. Since a byte is the most basic unit of data transfer, the `Stream` objects provide a basic data transfer capability, independent of the specific storage media. These byte-level I/O classes are also the building blocks for the higher-level file I/O classes.

The abstract base `Stream` class defines the interface—the properties and methods—for transferring bytes to and from a data source. This base class itself has no concept of the source or destination but defines the basic capabilities of stream I/O, which are to read, to write, to seek, and to close. The real work gets done in the classes derived from `Stream`, such as `FileStream`, which incorporate the capabilities of a specific type of storage media. Thus, you use the same properties and methods to write bytes to a file as you do to write bytes to a network socket. The implementation of the underlying class is different for each media type, but the interface is identical.

The `Stream`-derived classes provide access via a common interface for byte-oriented access to a variety of storage media. That's nice if you are working with bytes, but what if you want to read and write Unicode characters (the default Windows CE character set) which has two bytes per character? Perhaps you want to read and write integers, floating-point values, or strings in their native mode. Or maybe you want to work at an even higher level of abstraction, such as reading and writing lines of text.

Download at WoweBook.Com

5.2.3 Higher-Level I/O

The .NET Compact Framework supports some of the higher-level I/O .NET classes. Various reader and writer classes, summarized in Table 5.3, deliver a higher level of abstraction by providing data-specific functionality for any stream regardless of its actual storage media. They do this by wrapping themselves around one of the `Stream` objects that were listed in Table 5.2.

TABLE 5.3: Data-Specific I/O Classes (Higher-Level Readers/Writers)

Interface (Abstract Base) Class	Implementation Class	Description	Key Methods

`TextReader`	N/A	Abstract base class; defines the interface for reading bytes as characters, strings, or lines	`Read`
			`ReadLine`
			`ReadBlock`
			`ReadToEnd`
			`Close`

`TextReader`	`StreamReader` `StringReader`	Reads characters and strings from a stream (`StreamReader`) or from a string (`StringReader`)	See above

`TextWriter`	N/A	Abstract base class; defines the interface for writing bytes as characters, strings, or lines	`Close`
			`Flush`
			`Write`
			`WriteLine`

`TextWriter`	`StreamWriter` `StringWriter`	Writes characters and strings to a stream (`StreamWriter`) or to a string (`StringWriter`)	See above

N/A	`BinaryReader`	Reads value data types from a stream	`Close`
			`Read`
			`ReadBoolean`
			`ReadByte`
			`ReadChar`

Conitunues

Download at WoweBook.Com

TABLE 5.3: Data-Specific I/O Classes (Higher-Level Readers/Writers) *(Continued)*

Interface (Abstract Base) Class	Implementation Class	Description	Key Methods

			`ReadInt16`
			`ReadInt32`
			`ReadString`
			`ReadSingle`
			`ReadDouble`

N/A	`BinaryWriter`	Writes value data types to a stream	`Close`
			`Flush`
			`Seek`
			`Write`

N/A	`XmlReader`	Reads XML from a stream	`Close`
			`MoveToAttribute`
			`MoveToElement`
			`Read`
			`ReadAttributeValue`
			`ReadElementString`
			`ReadInnerXML`
			`Skip`

N/A	`XmlWriter`	Writes XML to a stream	`Close`
			`WriteNode`
			`WriteStartAttribute`
			`WriteAttributeString`
			`WriteEndAttribute`
			`WriteStartElement`
			`WriteElementString`
			`WriteEndElement`

The data-specific I/O classes use a `Stream` object to do the underlying I/O. This `Stream` object can be provided as a constructor parameter, or it may be created by the constructor, as needed.

The `XmlReader` class is the base class for three derived classes: `XmlNodeReader`, `XmlTextReader`, and `XmlValidatingReader`. The `XmlWriter` class is the base class for the `XmlTextWriter` derived class.

The `BinaryWriter` class's `Write` method has 18 overloads (essentially one for each value data type), each capable of outputting the binary image of that

Download at WoweBook.Com

data type. Reading the data by using a `BinaryReader` object, however, requires that you specify a data type–specific routine, such as `ReadSingle`, for each value that you wish to read or that you use its generic `Read` method and then decipher the input bytes yourself. We think it is much easier to use the type-specific methods for reading.

Thus, if you want to write the fields of a structure to a file, you can use a `BinaryWriter` object to do so. You first create a `FileStream` object and then create a `BinaryWriter` object, specifying the `FileStream` object as an input parameter. The `BinaryWriter` is then used to output the contents of the structure to the file, as shown in Listing 5.1. To read the data back in, create a `BinaryReader` object and use the appropriate type-specific reading methods, as we illustrate later in this chapter for the sample application.

Or, if you felt that XML was the appropriate storage format for your data, you could use the XML I/O classes to persist your data in an XML file, as we will also do later in this chapter.

Listing 5.1: Using a `BinaryWriter` Object

```
using System;
using System.Text;
using System.IO;
         :
         :
      private struct ourFontInfo
      {
         public string  strName;
         public float sglSize;
         public int intStyle;
      }

      ourFontInfo fiNotepadCE;
         :
         :
   '  Place data in the fields.
         :
         :
   '  Open a writer for the file.
         BinaryWriter bwrtFile =
            new BinaryWriter(
            new FileStream(strFileName,
                           FileMode.OpenOrCreate,
                           FileAccess.Write,
                           FileShare.None),
            encodeFile);
```

Download at WoweBook.Com


```
'  Write the three fields to the file.
      bwrtFile.Write(textInput.Font.Name);
      bwrtFile.Write(textInput.Font.Size);
      bwrtFile.Write(unchecked((int)textInput.Font.Style));
      bwrtFile.Close();
```

Reader and writer objects wrap themselves around a `Stream` object and can provide their functionality only through their contained `Stream` object. In Listing 5.1, to write the same structure to a socket instead of to a file, you would create a `NetworkStream` object instead of a `FileStream` object. All other code would remain the same.

5.2.4 Encoding and Decoding Data

As data is being transferred, it can also be encoded or decoded. Encoding converts data from Unicode to another character set. Among supported character set conversions, applied as the bytes are being transferred between the storage media and the stream, are 7-bit ASCII, 8-bit Unicode (UTF-8), Simplified Chinese, and Japanese shift-jis. Decoding converts characters from these character sets into Unicode.

Encoding is done by members of the `System.Text.Encoding` class. Pick the encoding you want and specify it as a constructor parameter of the reader or writer object. The encoding properties of reader and writer objects cannot be changed after the object has been created. Table 5.4 shows the common properties and methods of the `Encoding` class, and Table 5.5 shows several of the more common encodings.

TABLE 5.4: The `EncodingClass` Properties and Methods

Properties and Methods	Description
`Convert`	Converts a byte array from one encoding to another
`GetEncoding`	Gets the object that does the encoding for a specified code page or encoding name; e.g., `GetEncoding(1252)` or `GetEncoding("GB18030")` (see Table 5.5)
`Default`	Gets the object that does default (UTF-7) encoding
`ASCII`	Gets the object that does ASCII encoding
`Unicode`	Gets the object that does Unicode encoding
`BigEndianUnicode`	Gets the object that does big-endian encoding

Download at WoweBook.Com

TABLE 5.4: The `EncodingClass` Properties and Methods *(Continued)*

Properties and Methods	Description
`UTF7`	Gets the object that does UTF-7 encoding
`UTF8`	Gets the object that does UTF-8 encoding

TABLE 5.5: Common Encodings

Code Page	Description
932	"Japanese shift-jis"
1200	"UTF-16LE","utf-16","ucs-2","unicode",or"ISO-10646-UCS-2"
1201	"UTF-16BE" or "unicodeFFFE"
1252	"windows-1252"
65000	"utf-7","csUnicode11UTF7","unicode-1-1-utf-7","unicode-2-0-utf-7", "x-unicode-1-1-utf-7", or "x-unicode-2-0-utf-7"
65001	"utf-8","unicode-1-1-utf-8","unicode-2-0-utf-8","x-unicode-1-1-utf-8", or "x-unicode-2-0-utf-8"
20127	"us-ascii","us","ascii","ANSI_X3.4-1968","ANSI_X3.4-1986", "cp367", "csASCII", "IBM367", "iso-ir-6", "ISO646-US", or "ISO_646.irv:1991"
54936	"GB18030"

The reader and writer objects determine how the data is to be streamed; the stream type determines the stream destination. This mix-and-match capability for marrying data-oriented functionality with different types of endpoints highlights the flexibility of .NET file I/O. It allows you to transfer a variety of data types, in a variety of encodings, to a variety of storage media while using (mostly) the same methods for any combination.

5.2.5 Using the I/O Classes

Having introduced the I/O classes, we now proceed to using them. For our example program, we create a .NET Compact Framework version of Notepad, shown in Figure 5.1. The sample allows a user to read, write, and edit the contents of text files, just like Notepad on desktop systems, and to save and restore the application settings to a file. We show two different ways to save the text: one using the `BinaryWriter` object and a custom

Download at WoweBook.Com

`Structure` object, and a second approach later with an `XmlTextWriter` object. The code for the initial version of this application is presented in Listings 5.2 through 5.10 and can be found on the book's Web site in the `Storage` project. Subsequent enhancements to the application are presented in Listings 5.11 through 5.21.

FIGURE 5.1: The Sample Application, a .NET Compact Framework Version of Notepad

Because `OpenFileDialog` and `SaveFileDialog` dialogs display only files and directories located under the My Documents directory, we use `My Documents\YaoDurant\NotepadCE` as the directory for all file I/O performed by this application.

We use the IDE to define the menu shown in Figure 5.1 and then write the code to handle the various selections. As always, we make sure the necessary namespaces have been specified and that our directory path is defined, as shown in Listing 5.2.

LISTING 5.2: The Start of the `Storage` Program

```
using System;
using System.Text;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

Download at WoweBook.Com

The selections on the Encoding Choices menu can all be handled by one routine that sets a `Form` variable named `encodeFile` to the requested encoding. This routine is shown in Listing 5.3. Also shown in Listing 5.3 is the code to initialize the `encodeFile` variable.

Listing 5.3: Handling the Encoding Choices

```
private Encoding encodeFile = Encoding.Default;
                    :
                    :
                    :
private void mitemFFFormat_Click(object sender,
                                 EventArgs e)
{
   //  Set this.encodeFile to the selected encoding.

   this.encodeFile = Encoding.Default;
   if ( ((MenuItem)sender) == mitemFFAscii )
   {
      this.encodeFile = Encoding.ASCII;
   }
   else if ( ((MenuItem)sender) == mitemFFUnicode )
   {
      this.encodeFile = Encoding.Unicode;
   }
   else if ( ((MenuItem)sender) == mitemFFUtf8 )
   {
      this.encodeFile = Encoding.UTF8;
   }
   else if ( ((MenuItem)sender) == mitemFFUtf7 )
   {
      this.encodeFile = Encoding.UTF7;
   }
}
```

Because the text held by the application is really just the contents of the multiline `TextBox` control that overlays most of the form, reading and writing the content means transferring the `TextBox` control's `Text` property to and from the file.

5.2.6 Text File I/O

When the user selects Open from the File menu, the application creates a `System.Windows.Forms` `OpenFileDialog` object (if one has not already been created), sets its initial `Directory` and `Filter` properties, and presents the dialog to the user. After the user has either canceled the dialog or specified

Download at WoweBook.Com

the desired file, the application saves the filename for future reference, creates a `FileStream` object for the requested file, and uses it plus the requested encoding to create a `StreamReader` object. Using the `StreamReader` object, the application reads the entire contents of the file into the multiline `TextBox` control and then closes the file. Listing 5.4 shows the code.

Listing 5.4: Handling the Request for File Open

```
private void mitemFileOpen_Click(object sender,
                                 EventArgs e)
{
   //  Create an OpenFileDialog if necessary.
   if ( fdlgOpen == null )
      { fdlgOpen = new OpenFileDialog(); }
   fdlgOpen.InitialDirectory = "NotepadCE";
   fdlgOpen.Filter =
      "dat files (*.dat)|*.dat|" +
      "txt files (*.txt)|*.txt|" +
      "All files (*.*)|*.*";

   //  Show it.
   switch (fdlgOpen.ShowDialog())
   {
      //  Check the user's response.
      case DialogResult.OK:

         //  Save the file name.
         strCurrentFile = fdlgOpen.FileName;

         //  Open, read, and close the file.
         StreamReader srdrFile =
            new StreamReader(
               new FileStream(strCurrentFile,
                              FileMode.Open),
               this.encodeFile);
            this.textInput.Text = srdrFile.ReadToEnd();
            srdrFile.Close();
         break;

      default:
         break;
   }
}
```

The File menu's `Save` handler checks to see whether there is a current filename. If there is not, it calls the `SaveAs` handler to complete the process.

Download at WoweBook.Com

Once the filename has been specified, the process is almost identical to the Open process; a `FileStream` object is created and used to create a `StreamWriter` object that writes the contents of the `Text` property to the file. Listing 5.5 shows the code.

Listing 5.5: Handling the Request for File Save

```
private void mitemFileSave_Click(object sender,
                                 EventArgs e)
{
   //  If the user has not yet specified
   //     a file name, do SaveAs.
   //  Else,
   //     open, write, and close the file.
   if ( strCurrentFile == string.Empty )
   {
      mitemFileSaveAs_Click(sender, e);
   }
   else
   {
      //  If the file does not exist,
      //     create it.
      if (! File.Exists(strCurrentFile) )
      {
         File.Create(strCurrentFile).Close();
      }

      //  Create Writer.
      StreamWriter swrtFile =
         new StreamWriter(
            new FileStream(strCurrentFile,
                           FileMode.Truncate),
            this.encodeFile);

      //  Write the file.
      swrtFile.Write(this.textInput.Text);

      //  Close the file.
      swrtFile.Close();
   }
}
```

The `SaveAs` handler presents a `SaveFileDialog` object to the user. If the user enters a filename, the handler stores the filename into the `Form` variable and then calls the `Save` handler to complete the process, as shown in Listing 5.6.

Download at WoweBook.Com

Listing 5.6: Handling the Request for File Save As

```
private void mitemFileSaveAs_Click(object sender,
                                   EventArgs e)
{
   //  Get the file name from the user.
   if ( fdlgSave == null )
   {
      fdlgSave = new SaveFileDialog();
   }

   switch (fdlgSave.ShowDialog())
   {
      case DialogResult.OK:
         //  Save the file name.
         strCurrentFile = fdlgSave.FileName;
         //  Save the file.
         mitemFileSave_Click(sender, e);
         break;
      default:
         break;
   }
}
```

So, transferring the text content of our Notepad to and from a file is a matter of using a reader or writer object wrapped around a `Stream` object to specify the file, designate the encoding, and transfer the text.

5.2.7 Binary File I/O

For the second enhancement, to allow saving and restoring the application's settings to a file, we use reader and writer objects that can handle more than just text—some of our settings are the value types `System.Single` and `System.Int32`, as well as `System.String`. For handling this variety, `BinaryReader` and `BinaryWriter` objects are the best choice because they were designed to read and write value data types.

In the mobile world where time and space are limited, reading and writing data in binary format is often advantageous. Binary format saves space because integers, singles, doubles, and so on normally require less space in their value-typed formats than they do as text characters. For example, 1.234567E12 requires 4 bytes of storage as a `System.Single` but 22 bytes (11 x 2) as a `System.String`. Binary format also saves time because the data does not need to be converted to and from `String`, nor does it need to be encoded.

Download at WoweBook.Com

Additionally, desktop programmers may find they are more likely to use binary file I/O in the .NET Compact Framework than on the desktop because .NET's binary serialization classes have not been ported to the .NET Compact Framework. Binary file I/O then becomes a mechanism for "rolling your own" serialization of property values to a file.

To keep things simple, in our example we save and restore only our font settings (`Name`, `Size`, and `Style`). Later in this chapter, we modify the application to use the registry instead of a file for program settings.

Several settings, such as `Font.Style`, are customized integers. That is, although their definition is derived from `Integer`, they are not truly a value data type. Some have the `FlagsAttribute` attribute set. Because the `BinaryReader` and `BinaryWriter` classes are designed to work with value data types, you must cast these derived integers to an integer, and because of the `FlagsAttribute` you will need to use an `unchecked` cast to do it.

5.2.8 Writing Binary Data

Listing 5.7 shows the code for saving the font settings to a file in binary format.

Listing 5.7: Saving Data in Binary Format to a File

```
private void SaveSettingsToFile()
{
   //  Create the directory if it does not
   //     already exist, and make it the
   //     current directory.
   if (! Directory.Exists(strDirName) )
   {
      Directory.CreateDirectory(strDirName);
   }
   Directory.SetCurrentDirectory(strDirName);

   //  Create the file if it does not
   //     already exist.
   if (! File.Exists(strFileName) )
   {
      File.Create(strFileName).Close();
   }

   //  Create a BinaryWriter (wrapped around a
   //     FileStream wrapped around the file)
   //     for output using the user-specified
   //     encoding.
   BinaryWriter bwrtFile =
      new BinaryWriter(
```

Download at WoweBook.Com


```
            new FileStream(strFileName,
                           FileMode.OpenOrCreate,
                           FileAccess.Write,
                           FileShare.None),
            encodeFile);

        //  Write the three fields, each of a
        //     different data type.
        //  Close the file.
        bwrtFile.Write(textInput.Font.Name);
        bwrtFile.Write(textInput.Font.Size);
        bwrtFile.Write(unchecked((int)textInput.Font.Style));
        bwrtFile.Close();
    }
```

The `SaveSettingsToFile` routine is passed the directory name and file name, plus the desired encoding. It begins by using the `Directory` and `File` classes to create the directory and the file, if they do not already exist. This code is technically superfluous, for a `Stream` object will soon be created and one of its constructor parameters specifies whether to "open" or to "create." We include the superfluous code to illustrate the following points.

- The `Directory` and `File` static methods are used without instantiating an object.
- Using the `File` class to create a file also opens the file. A `Stream` object cannot be created for an already open file. Therefore, if you use the `File` class to create a file that you plan to subsequently access with a `Stream` object, you must close the file after you create it.

Once the file has been created and closed, the application creates a `BinaryWriter` object wrapped around a `Stream` object for the file. Opening a `BinaryWriter` object is identical to opening the `StreamWriter` object we used earlier in this chapter, as we showed earlier in Listing 5.5. We specify an encoding because one of the fields that we write is a string field. The encoding parameter, which defaults to UTF-8, is applied to all strings. Once the file is opened and the `BinaryWriter` is created, we simply write each field individually and then close the file. Each of the three `Write` statements calls a different overload of the function, outputting a string, a single-precision floating-point number, and an integer.

Download at WoweBook.Com

5.2.9 Reading Binary Data

Reading binary data is a little trickier than writing it. When writing data, you do not have to pay much attention to data types of individual fields because all the methods for writing data in the `BinaryWriter` class are named `Write`. You leave it up to the compiler to find the correct overloaded method.

When reading data, however, you must pay closer attention to the data type because the `BinaryReader` class has 18 different methods, with the expected type as part of the method name. For example, you call `ReadInt32` to read a 32-bit integer and `ReadDouble` to read a double-precision floating-point number. Although not an overwhelming amount of effort, it is a bit more than required when writing the binary data.

The code for our sample program that reads the application settings from a binary file appears in Listing 5.8. As illustrated in the calls to the `ReadString`, `ReadSingle`, and `ReadInt32` methods, you must pay attention to the expected data types, and the expected sequence, to read binary files.

Listing 5.8: Reading Binary Data from a File

```
private void ReadSettingsFromFile()
{

   //  Make the directory the current directory.
   Directory.SetCurrentDirectory(strDirName);

   //  Create a BinaryReader (wrapped around a
   //     FileStream wrapped around the file)
   //     for input using the user-specified
   //     encoding.
   BinaryReader brdrFile =
      new BinaryReader(
         new FileStream(strFileName,
                        FileMode.OpenOrCreate,
                        FileAccess.Read,
                        FileShare.None),
         encodeFile);

   //  Read the three fields and create the font.
   //  Close the file.
   textInput.Font =
      new Font(brdrFile.ReadString(),
               brdrFile.ReadSingle(),
               (FontStyle)brdrFile.ReadInt32());
   brdrFile.Close();

}
```

Download at WoweBook.Com

5.2.10 Binary I/O and Structures

You can make your I/O code easier to maintain by getting into the habit of organizing related fields into structures and coding structure-specific methods that handle the I/O. When your data is in a structure, it is easy to remember that adding and removing fields from the structure requires a corresponding modification to the I/O routines; it is also easier to specify the correct data types in the read routine if the write routine and the field definitions are located in the same section of code as the read routine.

We create such a structure for our font fields; Listing 5.9 shows that code. The structure contains the fields, a write routine, and a matching read routine.

Listing 5.9: The Font Settings Structure

```
private string  strDirName = @"My Documents\NotepadCE";
private string  strFileName = "Settings.dat";
               :
               :
//  A structure containing font information and
//     the routines to write/restore that information
//     to/from a file.
private struct ourFontInfo
{
   public string strName;
   public float sglSize;
   public FontStyle fsStyle;

   public void WriteToFile(string  strDirName,
                           string  strFileName,
                           Encoding encodeFile)
   {
      //  Create the directory if it does not
      //     already exist, and make it the
      //     current directory.
      if (! Directory.Exists(strDirName) )
      {
         Directory.CreateDirectory(strDirName);
      }
      Directory.SetCurrentDirectory(strDirName);

      //  Create the file if it does not
      //     already exist.
      if (! File.Exists(strFileName) )
      {
         File.Create(strFileName).Close();
      }
```

Download at WoweBook.Com


```
   //  Create a BinaryWriter (wrapped around a
   //     FileStream wrapped around the file)
   //     for output using the user-specified
   //     encoding.
   BinaryWriter bwrtFile =
                  new BinaryWriter(
                     new FileStream(
                              strFileName,
                              FileMode.OpenOrCreate,
                              FileAccess.Write,
                              FileShare.None),
                  encodeFile);

   //  Write the three fields, each of a
   //     different data type.
   //  Close the file.
   bwrtFile.Write(this.strName);
   bwrtFile.Write(this.sglSize);
   bwrtFile.Write((int)this.fsStyle);
   bwrtFile.Close();
}

public void ReadFromFile(string  strDirName,
                         string  strFileName,
                         Encoding encodeFile)
{
   //  Set the current directory.
   Directory.SetCurrentDirectory(strDirName);

   //  Create a BinaryReader (wrapped around a
   //     FileStream wrapped around the file)
   //     for input using the user-specified
   //     encoding.
   BinaryReader brdrFile =
                  new BinaryReader(
                     new FileStream(
                              strFileName,
                              FileMode.OpenOrCreate,
                              FileAccess.Read,
                              FileShare.None),
                  encodeFile);

   //  Read the three fields.
   //  Close the file.
   try
   {
      this.strName = brdrFile.ReadString();
      this.sglSize = brdrFile.ReadSingle();
      this.fsStyle = (FontStyle)brdrFile.ReadInt32();
   }
```

Download at WoweBook.Com


```
            catch
            {
               this.strName = "Tahoma";
               this.sglSize = 9;
               this.fsStyle = FontStyle.Regular;
            }
            finally
            {
               brdrFile.Close();
            }
         }
      }
```

The presence of this structure allows the application to save the font settings using the code shown in Listing 5.10, a much simpler routine than the version shown earlier in Listing 5.7. The structure would also allow for an equal simplification of the restore routine, ensuring that the processes of saving and restoring the value type fields are always synchronized.

Listing 5.10: Using the Structure to Save the Settings

```
private void SaveSettingsToFile()
{
   //  Create an ourFontInfo structure, load it from
   //     TextInput's font, and have the structure save
   //     its contents to the file.

   ourFontInfo fiFont;
   fiFont.strName = textInput.Font.Name;
   fiFont.sglSize = textInput.Font.Size;
   fiFont.intStyle = textInput.Font.Style;
   fiFont.WriteToFile(strDirName,
                      strFileName,
                      Encoding.Default);
}
```

This is a good demonstration of a central principle of object-oriented programming (OOP): encapsulation. We push the messy details of reading and writing the data into the data structure so that once we get it working we can ignore that relatively low-level code. Then users of this structure call the higher-level methods and the work gets done. The simplified code in Listing 5.10 is ample justification for our recommendation that you associate reading and writing methods with structures and classes that must be transferred to streams.

Download at WoweBook.Com

5.2.11 XML File I/O

Another option for saving settings is to save them in an XML file. XML is a text-based format for the exchange of information, standardized by the World Wide Web Consortium. Although XML requires a large number of bytes for the amount of data it carries, its text-based format makes it readable by both people and machines, and its standardization makes it portable across a variety of platforms. For small amounts of data, such as our font settings, it is an excellent format.

XML stores data within elements, which can be nested within other elements, as shown in the XML text listed here:

```
<?xml version="1.0" encoding="us-ascii"?>
<NotepadCe>
   <Font>
      <Name>Microsoft Sans Serif</Name>
      <Size>8.25</Size>
      <Style>0</Style>
   </Font>
   <Textbox>
      <ScrollBars>0</ScrollBars>
      <TextAlign>0</TextAlign>
      <WordWrap>true</WordWrap>
   </Textbox>
</NotepadCe>
```

An element can contain values, such as `Microsoft Sans Serif` and `8.25`, either in the body of the element or in its attributes. In our application, we avoid the use of attributes and keep things simple by writing our settings values as element data rather than as attribute values.

To read and write XML we use an `XmlTextReader` object and an `XmlTextWriter` object. The `XmlTextReader` object uses a `TextReader` object to do the actual I/O (and similarly for the `XmlTextWriter`) while providing us with XML-specific functionality, such as writing the opening tag of an element or setting the number of spaces to be used for indentation.

As we did with binary I/O, we define a structure to hold the necessary settings and give it the ability to read from and write to a file by coding a read method and a write method within the structure itself. The structure will group the settings into two categories: those that are font settings and those that are settings of the multiline `TextBox` control, as shown in the previous XML code.

We begin by defining the element names and a field to hold a reference to the multiline `TextBox` control. Because we are saving the settings of this

Download at WoweBook.Com

TextBox control's font, not the form's font, we do not need a separate variable for the font. We provide an overloaded constructor so that the reference to the multiline TextBox control can be specified as the structure is created. Listing 5.11 shows the code.

Listing 5.11: The Start of the Structure for Saving Settings

```
public struct ourFormInfoXML
{
   //  Field names
   private const string strNotepadCE = "NotepadCe";
   private const string strFont = "Font";
   private const string strName = "Name";
   private const string strSize = "Size";
   private const string strStyle = "Style";
   private const string strTextbox = "Textbox";
   private const string strScrollBars = "ScrollBars";
   private const string strTextAlign = "TextAlign";
   private const string strWordWrap = "WordWrap";

   //  The TextBox whose settings are to be saved
   private TextBox textThis;

   //  Constructor
   public ourFormInfoXML(TextBox textThis)
   {
      //  Store reference to the TextBox.
      this.textThis = textThis;
   }
```

The routine to save the settings creates an XmlTextWriter object and sets it for an indentation size of three spaces, as shown in Listing 5.12.

Listing 5.12: The Start of the Structure's Routine for Saving Settings

```
public void WriteToFile(
   string strDirName,
   string strXMLFileName,
   Encoding encodeFile)
{

   //  Ensure the directory and file exist.
   if( ! Directory.Exists(strDirName) )
   {
      Directory.CreateDirectory(strDirName);
   }
```

Download at WoweBook.Com


```
Directory.SetCurrentDirectory(strDirName);
if( ! File.Exists(strXMLFileName) )
{
   File.Create(strXMLFileName).Close();
}

//  Create the XmlTextWriter.
XmlTextWriter xmlwrtFontInfo =
   new XmlTextWriter(strXMLFileName, encodeFile);

//  Set indentation.
xmlwrtFontInfo.Formatting = Formatting.Indented;
xmlwrtFontInfo.Indentation = 3;
```

Then the routine begins the output of the XML document and writes the opening tag of the `NotepadCE` element (`<NotepadCE>`):

```
//  Begin the document and root element.
xmlwrtFontInfo.WriteStartDocument();
xmlwrtFontInfo.WriteStartElement(strNotepadCE);
```

Next, the routine writes the opening tag of the `Font` element, followed by the three elements that contain data, followed by the closing tag of the `Font` element, as shown in Listing 5.13. Because the three elements that contain data do not have any child elements nested within them, they can be written with a single method call. We have chosen to write the values as strings solely to make the resulting file as readable as possible.

Listing 5.13: Writing the Font's `Name`, `Size`, and `Style` Properties to the XML File

```
//  Begin the Font element,
//     write the three data elements
//     (Name, Size, Style), and
//     end the Font element.
xmlwrtFontInfo.WriteStartElement(strFont);
xmlwrtFontInfo.WriteElementString(
   strName, this.textThis.Font.Name);
xmlwrtFontInfo.WriteElementString(
   strSize,
   XmlConvert.ToString(this.textThis.Font.Size));
xmlwrtFontInfo.WriteElementString(
   strStyle,
   XmlConvert.ToString(
      unchecked((int)this.textThis.Font.Style)));
xmlwrtFontInfo.WriteEndElement();
```

Download at WoweBook.Com

In the same way, the Textbox element and its three nested elements are written, as shown in Listing 5.14.

Listing 5.14: Writing Three of the TextBox Control's Properties to the XML File

```
//  Begin the Textbox element,
//     write the three data elements (ScrollBars,
//     WordWrap, TextAlign), and
//     end the Textbox element.
xmlwrtFontInfo.WriteStartElement(strTextbox);
xmlwrtFontInfo.WriteElementString(
      strScrollBars,
      XmlConvert.ToString(
         unchecked((int)textThis.ScrollBars)));
xmlwrtFontInfo.WriteElementString(
      strTextAlign,
      XmlConvert.ToString(
         unchecked((int)textThis.TextAlign)));
xmlwrtFontInfo.WriteElementString(
      strWordWrap,
      XmlConvert.ToString(textThis.WordWrap));
xmlwrtFontInfo.WriteEndElement();
```

Finally, the NotepadCE element's closing tag is written, the buffer is flushed, and the file is closed:

```
   //  End the root element, end the document,
   //     and flush and close the file.
   xmlwrtFontInfo.WriteEndElement();
   xmlwrtFontInfo.WriteEndDocument();
   xmlwrtFontInfo.Flush();
   xmlwrtFontInfo.Close();
}
```

The read routine used to restore the file is similar in the way that it opens and closes the file, but it is different in the way that it processes the file—it uses a read loop and extracts the nodes it is interested in as they are retrieved.

Listing 5.15 shows the opening code.

Listing 5.15: The Start of the Routine for Reading Values from the XML File

```
public void ReadFromFile(
   string strDirName,
   string strXMLFileName,
   Encoding encodeFile)
{
   //  Temp fields to hold values as they are read
```

Download at WoweBook.Com


```
string tempName = string.Empty;
float tempSize = 0.0F;
int tempStyle = 0;
bool tempWordWrap = false;
int tempScrollBars = 0;
int tempTextAlign = 0;

//  Set the current directory.
Directory.SetCurrentDirectory(strDirName);

//  Create the XmlTextReader.
XmlTextReader xmlrdrFontInfo =
   new XmlTextReader(strXMLFileName);
xmlrdrFontInfo.WhitespaceHandling =
                            WhitespaceHandling.None;
```

The routine then uses the `MoveToContent` method to step over any element(s) at the start of the file that do not contain data. The `MoveToContent` method can be used at any time in the processing of an XML file. It will bypass elements such as comments and processing instructions (elements that often appear at the start of an XML file).

```
//  Skip over nondata stuff at start of file.
xmlrdrFontInfo.MoveToContent();
```

The main processing loop reads each element, checking the element name to see whether it is one we are interested in. More correctly, the loop reads each node. Thus, after the routine has read the value from the `Style` element, the file is positioned at the closing tag of the `Font` element. The next read will position the file at the opening tag of the `Textbox` element, and the read after that will position it at the opening tag of the `ScrollBars` element, where `ReadElementString` can be used to extract the value. This loop is shown in Listing 5.16.

One advantage of using XML over binary file I/O is that we can alter the sequence in which we write the settings values without needing to recode our read routine.

Listing 5.16: The Main Processing Loop for Reading Values from the XML File

```
//  Read each node, looking for the data. If it
//     is a node we are interested in, read
//     the value.
//     Note:  ReadElementString will move to the
//     next node when done.  Do NOT call
//     ReadElementString and Read, or you will
```

Download at WoweBook.Com


```
//     advance the file two nodes instead of
//     one node.
while( ! xmlrdrFontInfo.EOF )
{
   switch( xmlrdrFontInfo.Name )
   {
      case strName:
         tempName =
            xmlrdrFontInfo.ReadElementString();
         break;
      case strSize:
         tempSize =
            float.Parse(
               xmlrdrFontInfo.ReadElementString());
         break;
      case strStyle:
         tempStyle =
            int.Parse(
               xmlrdrFontInfo.ReadElementString());
         break;

      case strScrollBars:
         tempScrollBars =
            int.Parse(
               xmlrdrFontInfo.ReadElementString());
         break;
      case strWordWrap:
         tempWordWrap =
            bool.Parse(
               xmlrdrFontInfo.ReadElementString());
         break;
      case strTextAlign:
         tempTextAlign =
            int.Parse(
               xmlrdrFontInfo.ReadElementString());
         break;
      default:
         xmlrdrFontInfo.Read();
         break;
   }
}
```

Finally, the file is closed and the values are applied to the multiline `TextBox` control:

```
//  Close the XmlTextReader.
xmlrdrFontInfo.Close();

//  Restore the settings that were just read.
```

Download at WoweBook.Com


```
            this.textThis.Font = new
               Font(tempName, tempSize, (FontStyle)tempStyle);
            this.textThis.ScrollBars =
               (ScrollBars)tempScrollBars;
            this.textThis.WordWrap = tempWordWrap;
            this.textThis.TextAlign =
               (HorizontalAlignment)tempTextAlign;
         }
      }
```

To use our structure to save the settings, we create an instance of it and invoke its `WriteToFile` method, as shown here:

```
ourFormInfoXML fiForm = new ourFormInfoXML(textInput);
fiForm.WriteToFile(
   strDirName, strXMLFileName, Encoding.Default);
```

Restoring the settings becomes a matter of invoking our structure's `ReadFromFile` method:

```
ourFormInfoXML fiForm = new ourFormInfoXML(textInput);
fiForm.ReadFromFile(
   strDirName, strXMLFileName, Encoding.Default);
```

When we run and test the application, it produces a file that contains the XML text we listed at the start of this subsection.

This completes a very introductory look at XML. We narrowed our scope to sequential reading and writing of just elements and their values. But even with this narrow scope we saved and restored our information in a file format that was readable and portable.

The Key Points of .NET Compact Framework File I/O

The key points include the following.

1. Use static methods in the `Directory` and `File` classes for creating, copying, deleting, existence testing, and maintaining attributes of files and directories.
2. To read and write to a file, use a `FileStream` object wrapped inside the appropriate reader or writer object.
3. Choose an encoding prior to creating the `Stream` object.
4. Add read and write member functions to structures and classes to encapsulate data transfer between memory and files.

Continues

Download at WoweBook.Com

Once you have determined the type of data you need to read and write (such as binary, text, or XML), choose the appropriate object class for that data type, and you're on your way.

5.2.12 Easier XML Serialization

Serialization is the process of writing the contents, the field and property values, of an object to a location so that those values can be passed on to another application or so that those values can be subsequently used to reconstruct the original object. In .NET, serialization requires the use of three objects: the object being serialized, a `Stream` object, and a `Serializer` object. It also requires a surprisingly small amount of code.

All usable `Stream` objects, such as `FileStream` and `NetworkStream`, derive from the abstract base class of `Stream`. Thus, the same process is used to pass objects between a .NET Web service and a .NET client application as is used to save an application settings object to a file. Under the covers, both activities are making the same `Stream` method calls to pass bytes into a stream of the appropriate class.

The serializer object, like the stream object, is provided by the .NET Framework. This object, which does both serializing and deserializing, needs a reference to the object being serialized and a reference to the stream object. Four lines of code are required to serialize or deserialize an object, as illustrated by the following code, extracted from the LINQ chapter application, that serializes an application defined object to a file. The object being serialized, the `otx` object, is a relatively complex object; consisting of related collections of customers, employees, and orders.

```
XmlSerializer xs =
   new XmlSerializer(typeof(AppToXml));
xs.Serialize(fs, otx);
fs.Close();
```

The nice thing about serialization is that the object being serialized does not need to serialize itself. That is, it does not need to provide any custom code to do the serializing. It is the serializer's responsibility to find the fields and properties of the object being serialized and copy them to or from the stream. Thus, the code that does the serialization does not need to be modified as the design of the object being serialized changes.

Download at WoweBook.Com

5.2.12.1 *Binary Formatting and XML Serialization*

In .NET, there are two kinds of serialization: binary and XML. **Binary serialization,** which is always referred to as *formatting,* is intended for the movement of information within .NET applications, not between .NET applications and the outside world. The object used to do binary formatting, the `BinaryFormatter` object, writes the content of its target object in a layout that is unreadable by other platforms, such as Java, or by most people. The `XmlSerializer` object, on the other hand, converts the target object's field and property values into their W3.org data formats and writes them as standard XML, thus allowing cross-platform applications and humans to read the resulting output. Incidentally, another binary serialization class, `SoapFormatter`, has been deprecated in .NET Framework 3.5 and should be avoided (this is not a problem in Compact Framework programming, since it is not supported).

The advantage of binary formatting over XML serialization is that, in general, it is easier to use as there are fewer restrictions on the object being serialized. The great disadvantage of binary serialization is that it is not available in Compact Framework 3.5. Therefore, we will limit any further discussion of serialization to XML serialization.

5.2.12.2 *XML Serialization*

Objects being serialized via XML serialization have a greater number of restrictions placed upon them than do objects that are being binary-formatted. This is because the `XmlSerializer` class has no inside hooks into .NET, and no backdoor access to the object's allocated memory. It must write the contents of the object into the stream using the same classes and techniques we used when we wrote the content of our application settings object to a file. This leads to the first restriction placed on XML-serializable objects:

Only public, read-write properties and fields will be serialized.

All other information will be lost. The `XmlSerializer` uses reflection to determine the members of the object being serialized/deserialized and then assigns values to, or retrieves values from, those fields and properties; and it does it in the same way that our code retrieved and assigned member values to our application settings object. Thus, member values of the application object being serialized must be accessible by a class defined within the .NET DLL. Hence, they must be public.

Download at WoweBook.Com

Unlike the serialization process, at the start of deserialization the receiving object does not yet exist and must be created by the `XmlSerializer`. This leads to the second restriction placed on XML-serializable objects:

The object's class must have a constructor that requires no parameters.

The `XmlSerializer` must be able to create the object to be deserialized by using the `new` operator. This is how almost all reference type objects are created in .NET. The `XmlSerializer` could not possibly know, or guess, what parameter values would result in successful creation of the target object. It must be able to create the object without specifying any parameters. Hence, the target class must have a public, parameterless constructor.

The third restriction is a bit subtler:

Objects containing circular references will not be XML-serialized.

Thus, if object A has a property containing a reference to object B, and object B has a property containing a reference to object C, and object C has a property containing a reference to object A, neither A nor B nor C is XML-serializable.

This is not a problem for a class that contains only strings and value type fields—numerics, `DateTime` and `Boolean`—such as our applications settings class. But a typical application may contain a significant amount of related data. To be able to save and retrieve this interrelated data to and from a file with just four very simple lines of code, we need to place all of the applications data into a single object, an object that contains references to the other objects of our application, and to serialize that container object.

This is similar to a database design that places a foreign key column in the `Orders` table to link each `Order` to its `Customer`. Both the `Orders` and the `Customers` tables are contained within a database. Backing up the database persists both of the tables and the relationship between them to a file. In the same way, as an upcoming example in the LINQ to Objects section of Chapter 7 will illustrate, we would like our `Customer` objects and their related `Order` objects, placed within a single containing object, to be XML-serializable so that we can save all of the application's data by simply serializing the containing object. If the `Order` object has a `Customer` property containing a reference to its `Customer` object, and that `Customer` object has an `Orders` property containing a collection of references to its `Orders`, we

Download at WoweBook.Com

have a circular reference situation, and neither `Customers`, nor `Orders`, nor the parent container object is XML-serializable.

In the LINQ to Objects section of Chapter 7, where our object design will have reached a typical level of complexity, we'll examine some design solutions for overcoming the aforementioned restriction. In doing so, we will encounter a fourth restriction on XML serialization:

Not all .NET objects are XML-serializable.

There are thousands of .NET classes. Not all of the classes are XML-serializable, and it is not always intuitively obvious which ones are and which ones are not. Consider two classes from the `Systems.Collections.Generic` namespace: the `List` class and the `Dictionary` class. The former is XML-serializable, the latter is not. Learning which .NET objects are XML-serializable, and frequently testing serialization during development, will lead you toward a better design of your application's business objects and their supporting .NET partner objects.

Smart-device applications often need to save the application's data to a file. Decisions that you make regarding the structure of that data within the application's memory will determine whether you can use XML serialization to easily persist application data to device storage. As this book progresses, we will encounter scenarios in which we must design and persist application data between business objects and device storage files.

5.3 Registry Access

The file system and the registry provide a place for persistent data. Both are hierarchically organized. Like the file system, the registry resides in nonvolatile semiconductor memory. These similarities aside, there are clearly important differences between the file system and the registry.

The registry provides a space for storing key name/value pairs. To developers familiar with the desktop registry, the hierarchy in the Windows Mobile registry has a set of familiar-sounding root-level keys, including `HKEY_CLASSES_ROOT`, `HKEY_CURRENT_USER`, and `HKEY_LOCAL_MACHINE`. With their limited resources, devices have smaller, simpler registries than those found on the desktop.

Keys are defined by string names. The registry supports three basic data types: integers, strings, and binary arrays. Each type supports some

Download at WoweBook.Com

variations—you must choose, for example, between 32-bit and 64-bit integers—but the three types are what are supported.

Developers who are completely new to the registry might benefit from drilling into the registry using one of several registry editing tools. Desktop versions of Windows come equipped with a registry editor named regedit.exe. Visual Studio ships with the Remote Registry Editor, which can be used to view both the desktop as well as device registries.

5.3.1 Opening and Creating Registry Keys

To access data in the registry, you start by opening (or creating) a registry key for a specifically named registry node. Once a key is open, you can read or write multiple named values on that node. When done, you must close the key—using a method named `Close`—to allow changes to be saved.

Programming for the registry in managed code, like all managed-code programming, requires access to a specific assembly and namespace. The assembly, mscorlib.dll, is among the default assemblies provided for you by the new project wizard. The required namespace, however, is not among the defaults. You must manually add a reference to the following namespace:

```
using Microsoft.Win32;
```

Two Compact Framework classes provide all that you need to access the registry: `RegistryKey` provides static methods for creating (or opening) nodes in the registry hierarchy, and the `Registry` class provides the methods for reading and modifying the key/value pairs on a specific node. Looked at another way, the `Registry` class gives you access to objects in the left pane of a typical registry editor. Once a key is opened, the `RegistryKey` class provides methods for accessing objects in the registry editor's right pane.

There are several ways to open a registry key. Table 5.6 summarizes the static members of the `Registry` class which create a key for the root of each of the registry's hierarchies. Each method returns a `RegistryKey` object.

TABLE 5.6: Registry Methods for Creating or Opening Registry Keys

Static Method	Root Node Opened
`ClassesRoot()`	`HKEY_CLASSES_ROOT`
`CurrentConfig()`	`HKEY_CURRENT_CONFIG`

Download at WoweBook.Com

TABLE 5.6: Registry Methods for Creating or Opening Registry Keys *(Continued)*

Static Method	Root Node Opened
`LocalMachine()`	`HKEY_LOCAL_MACHINE`
`Users()`	`HKEY_USERS`

It is considered a bad practice to store values in the root of any of the registry hierarchies. For that reason, one of the first things you are likely to do after opening a registry root node is to open a subnode (also known as a *subkey*). For this purpose, the `RegistryKey` class provides two useful methods: `CreateSubKey()` and `OpenSubKey()`.

To check for the existence of a subkey, call `OpenSubKey`. This function returns a null reference if the requested subkey does not exist. Many programmers rely solely on the `CreateSubKey` function, which creates a subkey if it does not already exist. On success, both functions return a `RegistryKey` object.

5.3.2 Reading and Writing Registry Values

Once you have a `RegistryKey` object, you read values by calling `GetValue` and write values by calling `SetValue`. Listing 5.17 illustrates a wrapper class that we created for the sample to handle all the reading and writing of registry values.

LISTING 5.17: `UtilRegistry` Class for the Sample's Registry Access

```
public class UtilRegistry
{
   private string  strYDKey = @"SOFTWARE\YaoDurant";

     private RegistryKey m_RegKey;

   // Default constructor
   public UtilRegistry()
   {
       // Create subkey for our settings.
       m_RegKey = Registry.LocalMachine.CreateSubKey(strYDKey);
   }

   ~UtilRegistry()
   {
       m_RegKey.Close();
   }

   //  Delete a registry key.
```

Download at WoweBook.Com


```
public bool DeleteKey(string strKeyName)
{
   m_RegKey.DeleteSubKey(strKeyName);
   return true;
}


//  Read a System.string  from the registry.
public bool GetValue(string strKeyName,
                     string  strValueName,
                     ref string  strValue)
{
   RegistryKey rkey = m_RegKey.CreateSubKey(strKeyName);
   string strReturn =
       rkey.GetValue(strValueName, strValue).ToString();
   rkey.Close();

   strValue = strReturn;
   return true;
}


//  Read a System.int from the registry.
public bool GetValue(string strKeyName,
                     string  strValueName,
                     ref int intValue )
{
   RegistryKey rkey = m_RegKey.CreateSubKey(strKeyName);
   int iReturn = (int)(rkey.GetValue(strValueName, intValue));
   rkey.Close();

   intValue = iReturn;
   return true;
}


//  Read a System.bool from the registry.
public bool GetValue(string strKeyName,
                     string  strValueName,
                     ref bool boolValue)
{
   RegistryKey rkey = m_RegKey.CreateSubKey(strKeyName);
   int intValue = (boolValue) ?  1 : 0;
   int iReturn = (int)(rkey.GetValue(strValueName, intValue));
   rkey.Close();

   boolValue = (iReturn == 1) ? true : false;
   return true;
}
```

Download at WoweBook.Com


```
   //  Write a System.string  to the registry.
   public bool SetValue(string strKeyName,
                        string  strValueName,
                        string  strValue)
   {
      RegistryKey rkey = m_RegKey.CreateSubKey(strKeyName);
      rkey.SetValue(strValueName, strValue);
      rkey.Close();

      return true;
   }

   //  Write a System.int to the registry.
   public bool SetValue(string strKeyName,
                         string strValueName,
                         int intValue )
   {
      RegistryKey rkey = m_RegKey.CreateSubKey(strKeyName);
      rkey.SetValue(strValueName, intValue);
      rkey.Close();

      return true;
   }

   //  Write a System.bool to the registry.
   public bool SetValue(string strKeyName,
                        string  strValueName,
                        bool boolValue)
   {
      //  Cast the value as a boolean, then use the integer
      //     version of SetValue to set the value into the
      //     registry.
      //  There is no Convert.ToInteger method.  For platform
      //     independence, we cast to the smallest integer,
      //     which will always implicitly and successfully cast
      //     to int.
      return SetValue(strKeyName,
                      strValueName,
                      Convert.ToInt16(boolValue));
   }
}
```

5.3.3 Updating the `Storage` Sample Application to Use the Registry

To illustrate accessing the registry, we modify our sample application to save and restore settings to the registry rather than to a file. We have a head start with our wrapper functions and our utility class already written. In the updated version of our sample application, we create an object

Download at WoweBook.Com

of type `UtilRegistry` and define a set of strings that we use for the registry key name and the various registry values, as shown in Listing 5.18.

Listing 5.18: Declarations for Registry Access

```
//  Variables for registry access
UtilRegistry  urNotepad = new UtilRegistry();

string  strNotepadCE = "NotepadCe";
string  strFont = "Font";
string  strName = "Name";
string  strSize = "Size";
string  strStyle = "Style";
string  strOptions = "Options";
string  strMenu = "Menu";
string  strToolBar = "ToolBar";
string  strScrollBars = "ScrollBars";
string  strTextAlign = "TextAlign";
string  strWordWrap = "WordWrap";
```

The request to save the settings, shown in Listing 5.19, is handled by calling to the various overloaded versions of the `SetValue` method, once for each registry value we wish to save.

Listing 5.19: Saving Settings to the Registry

```
private void mitemSettingsSave_Click(object sender,
                                     EventArgs e)
{
   //    Use the UtilRegistry object to save the
   //       font settings.
   //    UtilRegistry has three overloads for
   //       SetValue: string, integer, and boolean.
   //    Font Size and Style are data types that
   //       derive from integer; they have to
   //       be converted to integer for this call.
   //    Font.Style data type, because it has
   //       the [Flags] attribute, requires an
   //       unchecked conversion.
   urNotepad.SetValue(
      strNotepadCE + @"\" + strFont,
      strName,
      textInput.Font.Name);
   urNotepad.SetValue(
      strNotepadCE + @"\" + strFont,
      strSize,
      Convert.ToInt32(textInput.Font.Size));
   urNotepad.SetValue(
      strNotepadCE + @"\" + strFont,
```

Download at WoweBook.Com


```
            strStyle,
            unchecked((System.Int32)textInput.Font.Style));

        //    Use the UtilRegistry object to save the
        //       Textbox settings, three of which are
        //       boolean, two of which are [Flags]
        //       attributed unsigned integers.
        urNotepad.SetValue(
            strNotepadCE + @"\" + strOptions,
            strMenu,
            this.Menu == this.menuMain);
        urNotepad.SetValue(
            strNotepadCE + @"\" + strOptions,
            strToolBar,
            this.Controls.Contains(tbarCommands));
        urNotepad.SetValue(
            strNotepadCE + @"\" + strOptions,
            strScrollBars,
            unchecked((System.Int32)textInput.ScrollBars));
        urNotepad.SetValue(strNotepadCE + @"\" + strOptions,
            strTextAlign,
            unchecked((System.Int32)textInput.TextAlign));
        urNotepad.SetValue(strNotepadCE + @"\" + strOptions,
            strWordWrap,
            textInput.WordWrap);
    }
```

To restore the settings, we reverse the process by which we saved the settings, calling the `GetValue` method for each value we wish to retrieve, as shown in Listing 5.20.

Listing 5.20: Restoring Settings from the Registry

```
    private void mitemSettingsRestore_Click(object sender,
                                            EventArgs e)
    {
        //  Read Font information, if any, from the registry.
        //  Create a font from that information, or use default
        //     values if no information is in the registry.
        //  Set textInput.Font = that font.
        string  strFontName;
        int intFontSize;
        int intFontStyle;

        strFontName = "Tahoma";
        urNotepad.GetValue(strNotepadCE + @"\" + strFont,
                           strName,
                           ref strFontName);
```

Download at WoweBook.Com


```
intFontSize = 9;
urNotepad.GetValue(strNotepadCE + @"\" + strFont,
                   strSize,
                   ref intFontSize);

intFontStyle = (int)FontStyle.Regular;
urNotepad.GetValue(strNotepadCE + @"\" + strFont,
                   strStyle,
                   ref intFontStyle);
   textInput.Font =
      new Font(strFontName,
               intFontSize,
               (FontStyle)intFontStyle);

//  Read Option information, if any, from the registry.
//  Set the properties from that information, or use
//     default values if no info is in the registry.
bool boolTemp;
int intTemp;

//  .Menu is either menuMain or null.
boolTemp = true;
urNotepad.GetValue(strNotepadCE + @"\" + strOptions,
                   strMenu,
                   ref boolTemp);
this.Menu = boolTemp ? this.menuMain : null;

//  .Controls either contains
//     tbarCommands or it doesn't.
boolTemp = true;
urNotepad.GetValue(strNotepadCE + @"\" + strOptions,
                   strToolBar,
                   ref boolTemp);
if ( boolTemp )
{
   if (! this.Controls.Contains(this.tbarCommands) )
   {
      this.Controls.Add(this.tbarCommands);
   }
}
else
{
   if ( this.Controls.Contains(this.tbarCommands) )
   {
      this.Controls.Remove(this.tbarCommands);
   }
}

//  .ScrollBars
intTemp = (int)ScrollBars.Both;
```

Download at WoweBook.Com


```
        urNotepad.GetValue(strNotepadCE + @"\" + strOptions,
                           strScrollBars,
                           ref intTemp);
           textInput.ScrollBars = (ScrollBars)intTemp;

        //  .TextAlign
        intTemp = (int)HorizontalAlignment.Left;
        urNotepad.GetValue(strNotepadCE + @"\" + strOptions,
                           strTextAlign,
                           ref intTemp);
           textInput.TextAlign = (HorizontalAlignment)intTemp;

        //  .WordWrap
        boolTemp = true;
        urNotepad.GetValue(strNotepadCE + @"\" + strOptions,
                           strWordWrap,
                           ref boolTemp);
           textInput.WordWrap = boolTemp;
     }
```

The last selection that the user can make from the Settings menu is the `Init Settings` selection. The application handles this by deleting the registry key (which removes all its content, subkeys, and data) and then calling the `Restore Settings` handler. That handler, shown in Listing 5.20, restores default values for any missing settings. Because all the settings are now missing from the registry, a complete set of default values will be set. The code appears in Listing 5.21.

Listing 5.21: Initializing the Settings

```
     private void mitemSettingsInit_Click(object sender,
                                          EventArgs e)
     {
        //    Initialize the settings by deleting the
        //       registry key where they are stored and
        //       then restoring from the nonexistent
        //       key to reinstate all default values.
        urNotepad.DeleteKey(strNotepadCE);
        mitemSettingsRestore_Click(this, EventArgs.Empty);
     }
```

5.4 Conclusion

This chapter covered .NET Compact Framework programming to support two types of local storage: the file system and the registry.

Download at WoweBook.Com

Many of the desktop .NET file I/O classes have been ported to the .NET Compact Framework. The most important ones are the `Directory` and `File` classes, which are used for file manipulation. Also available are the various `Stream` classes, which are used for byte-level I/O to a variety of stream types. Finally, the various reader and writer classes provide higher-level I/O to and from streams.

For accessing the registry, two desktop .NET Framework classes are available: `Registry` and `RegistryKey`. The basic design pattern is to open a key for a specific node in the registry hierarchy, read or write named values in that node, and then close the node. You can also call individual functions that read or write values and hide the work of opening and closing the keys from you.

Download at WoweBook.Com

6
ADO.NET Programming

This chapter introduces ADO.NET, a set of .NET Compact Framework classes used for database programming in managed code. Like its counterpart on the desktop, the ADO.NET classes in the .NET Compact Framework provide a convenient, flexible model for operating on memory-resident databases. Three options are presented for the persistent storage of ADO.NET data sets: XML files, Microsoft SQL Server CE databases on a Windows CE device, and Microsoft SQL Server databases (Version 7.0 and later) on a desktop or server version of SQL Server.

A .NET Compact Framework program that manages a significant volume of data is probably going to use the ADO.NET classes to manage that data. ADO.NET provides managed-code support for table-based memory-resident data (in other words, a memory-resident database) and for the transfer of that data to and from a persistent data store, such as a SQL Server database or an XML file. Some of the ADO.NET classes provide the ability to create the memory-resident database and to programmatically manipulate the data in its data tables. Other classes provide the ability to move data between the memory-resident database and various data sources.

Central to the design of the memory-resident database classes is the independence of these classes from any particular database product or storage format. This means that an ADO.NET table might be constructed with data from two or more data sources plus some application-generated data.

Download at WoweBook.Com

Conversely, the classes that transfer data between the memory-resident database and a data source are specific to the data source. For example, there is one set of these classes for use with SQL Server CE databases and a different set (with similar names and nearly identical PMEs) for use with SQL Server databases. These classes, therefore, are referred to as the *data provider classes.*

Table 6.1 summarizes all of the persistent data storage options available to .NET Compact Framework programmers, including ADO.NET. As indicated in this table, ADO.NET supports three different storage types. Two storage types reside in the file system of a Windows CE–powered device: SQL Server CE databases and XML files. These are appropriate for a mobile device, such as a Pocket PC or Smartphone, which needs to run in a self-contained, disconnected mode. Additionally, using the SQL Server data provider, a .NET Compact Framework program can directly access a third type of storage on a connected server: a SQL Server database. This requires a live network connection between the Windows CE client system and a server running SQL Server.

TABLE 6.1: Persistent Storage Options Available on Windows CE–Powered Devices

Type of Storage	Managed API	Native API
SQL Server CE database	ADO.NET with SQL Server CE Data Provider	ADOCE
OLEDBCE	SQL Server 2000 (and later) database	ADO.NET with SQL Server Data Provider
ADOCE	OLEDBCE	Other databases
ADO.NET with database-specific provider	ADOCE	OLEDBCE
XML files	ADO.NET	DOM provided by msxml.dll
Raw data files	`System.IO` classes (see Chapter 5)	Win32 API file access functions
CE property databases	N/A	Windows CE–specific Win32 API functions including `CeCreateDatabaseEx` and others
System registry	N/A	Win32 API functions required (see Chapter 5)

Download at WoweBook.Com

Supported ADO.NET Data Providers

As of this writing, we know of several ADO.NET data providers that are available for the .NET Compact Framework: Microsoft SQL Server CE, Microsoft SQL Server (Version 7.0 and later), IBM's DB2 and DB Everyplace, Sybase SQL (through iAnywhere), and Pocket Access (available from In The Hand, www.inthehand.com).

The following data providers are available for the desktop .NET Framework, but as of this writing they are not available for .NET Compact Framework programs: OLE DB, ODBC, Oracle, and MySQL.

As we implied in the opening part of this discussion, there are two layers of ADO.NET classes. One layer provides memory-resident database programming support, and its classes are not tied to a specific database or file format. Instead, the classes operate on in-memory objects that model relational data tables.

A second layer of classes, the data provider classes, provide the connection between in-memory objects and physical databases that reside in permanent storage. Alternatively, you can choose not to use a formal database and instead allow the ADO.NET classes to store your data in XML files in the Windows CE file system.

We expect that many databases on Windows CE–powered devices will need to be synchronized with databases on desktop and server systems. Table 6.2 summarizes the available options for synchronizing data with desktop and server systems. Two of these techniques are appropriate for synchronizing a device-based SQL Server CE database with a SQL Server database: Merge Replication and Remote Data Access (RDA). We discuss these in detail in Chapter 8.

TABLE 6.2: Data Synchronization Options on Windows CE–Powered Devices

Synchronization Technique	Description
Merge Replication	A publish-and-subscribe model for moving data between a SQL Server CE database and a SQL Server 2000 database. See Chapter 8 for more detail

Continues

Download at WoweBook.Com

TABLE 6.2: Data Synchronization Options on Windows CE–Powered Devices *(Continued)*

Synchronization Technique	Description
Remote Data Access (RDA)	A SQL-driven mechanism for moving data between a SQL Server CE database and a SQL Server 6.5, SQL Server 7, or SQL Server 2000 database. See Chapter 8 for more detail
Web services	A general-purpose RPC mechanism for support and document delivery, supported by the desktop .NET Framework and the .NET Compact Framework. Web services are built on SOAP, an XML-based industry-standard remoting protocol
ActiveSync	A general-purpose mechanism for synchronizing desktop and smart-device data. ActiveSync relies on service providers, which currently must be implemented in native dynamic link libraries (DLLs) that export a specific set of Component Object Model (COM) interfaces. Both desktop and device-side service providers are supported. SQL Server CE does not use ActiveSync for any of its synchronization. However, ActiveSync does support synchronizing a Pocket Access (.cdb) file with a desktop Microsoft Access (.mdb) file. Pocket Access is a discontinued product, but third-party support[a] exists for the Pocket Access file format (in the form of an ADO.NET-compatible data provider), and third-party development tools are also available
Remote API (RAPI)	Part of the ActiveSync support libraries. The RAPI routines provide a general-purpose mechanism for accessing the object store and installable file systems of a smart device from an ActiveSync-connected desktop system. This mechanism can be used to install or back up anything on a Windows CE device. However, RAPI has no specific ADO.NET support or support for SQL databases. RAPI is described in detail in Chapter 9

a. For details on the ADOCE .NET Wrapper available from In The Hand, visit www.inthehand.com.

6.1 Examining ADO.NET

ADO.NET is a set of classes for accessing and managing in-memory data. These classes provide the primary means for bringing data from a database into a managed-code program. The ADO.NET classes provide in-memory data support in a variety of environments for a wide range of application types, including desktop Windows Forms applications, Web Forms applications, Web services, and mobile-device applications. In fact,

Download at WoweBook.Com

a good way to access a database that does not yet have a .NET Compact Framework provider is to wrap access to that database in a Web service. (We cover Web services later in this chapter.)

The ADO.NET classes were designed to work with data that is drawn from a variety of sources or, in the terminology of ADO.NET, from a variety of *data providers.* Whatever the source of the data, the ADO.NET classes allow data to be managed and presented in a standard fashion. As mentioned earlier, to make this happen one layer of the ADO.NET classes is *provider-specific,* meaning those classes are capable of working with only one data provider, such as the `SqlCeConnection` class, which can connect only to a SQL Server CE database, or the `SqlCommand` class, which can submit a SQL statement only to a SQL Server database.

The other layer contains *provider-independent* memory-resident database classes, such as the `DataSet`, `DataTable`, and `DataView` classes. The data contained within these classes has been converted to the ADO.NET data types, and the structure of the data supported by these classes is independent of the database engine that provides the data. Thus, an ADO.NET data set may contain data tables with data drawn from multiple sources. For example, a data set could contain some tables that have been read from a Microsoft SQL Server CE database and other tables read from a Microsoft SQL Server database.

The ADO.NET memory-resident database classes treat data as provider-agnostic, meaning there is no easy way—and, hopefully, no need—to determine which data came from what data source. You can even build and populate data sets and data tables from scratch without reference to a particular database. Additionally, data sets can read and write their data to XML files without the need for any provider-specific software.

6.1.1 A Layered Approach

Because of the capabilities of the ADO.NET classes, you can use either a two-layer approach or a three-layer approach to data management; that is, you can use a memory-resident database to hold your data, or you can bypass the memory-resident database and move the data directly from the data source to its final destination.

In a two-layer approach, you submit SQL queries to the database and retrieve the data into program variables and control properties. You then build SQL statements to make the user-requested changes to the data and submit them to the database (see Figure 6.1). This is often referred to as the *connected approach* because the application must be connected to the

Download at WoweBook.Com

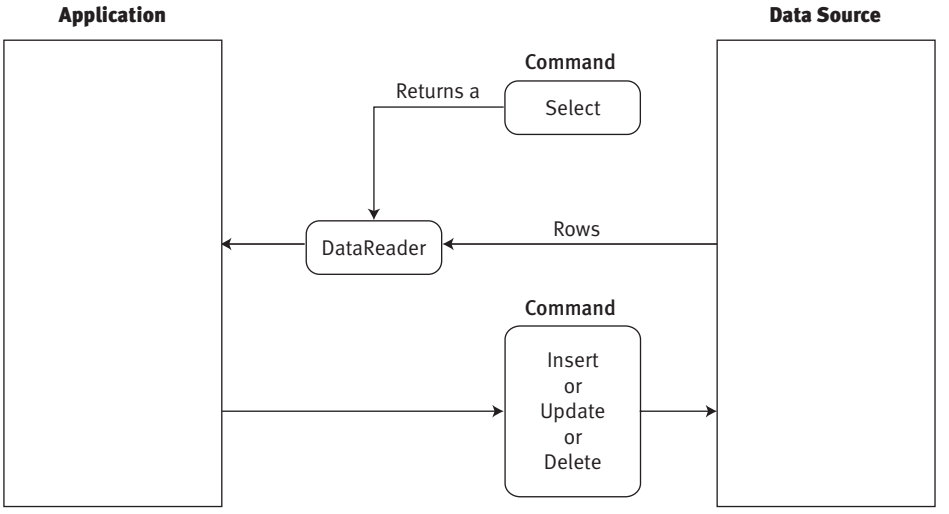


FIGURE 6.1: A Two-Layer (Connected) Approach

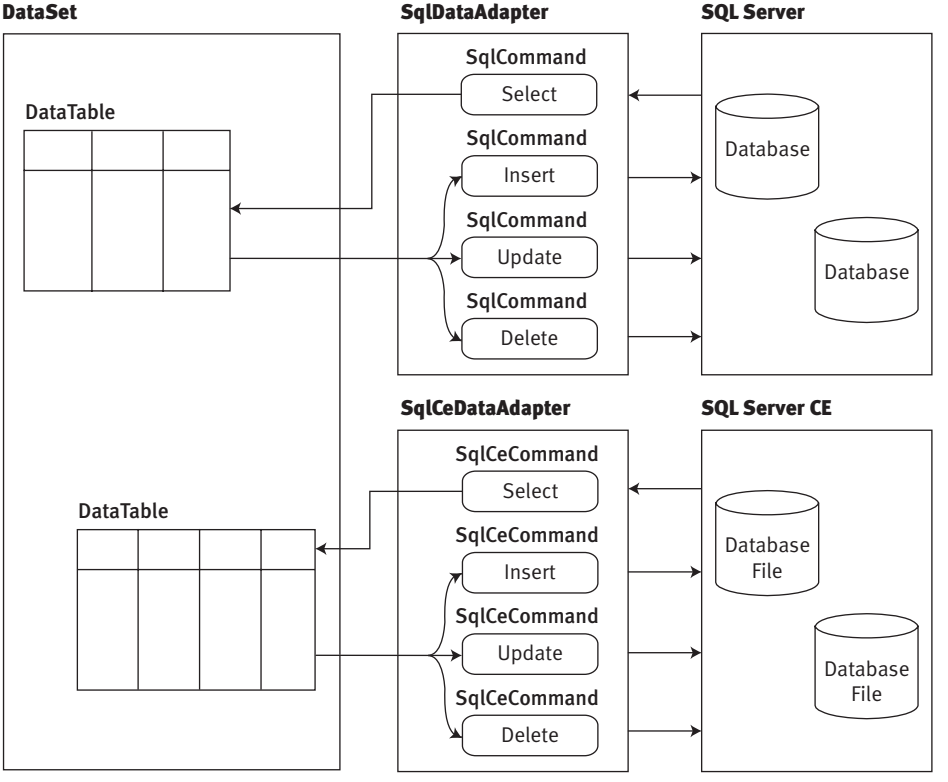


FIGURE 6.2: A Three-Layer (Disconnected) Approach

Download at WoweBook.Com

database while transferring data. Do not confuse this meaning of the term *connected* with being connected to a network. Throughout this chapter, when we use the term *connected,* we mean connected to the data source, such as a SQL Server or SQL Server CE database.

In a three-layer approach, you bring the data into a memory-resident data set, perhaps manipulate the data as it resides in the data set, and then bind some or all of the data to controls on your form. Changes that the user makes to the data in the form are automatically propagated back to the data set; class methods then propagate changes from the data set to the database (see Figure 6.2). This is often referred to as the *disconnected approach,* meaning that the application is not connected to the database while it is accessing the data in the data set.

6.1.2 The ADO.NET Classes

Table 6.3 presents an overview of the classes you can use to access SQL Server CE databases. We discuss many of these classes in detail later in this chapter. They fall into four general categories:

1. File system classes
2. Provider-specific ADO.NET classes
3. Provider-independent ADO.NET classes
4. Data-bindable control classes

TABLE 6.3: Classes for Accessing, Maintaining, and Displaying Data

Category	Class	Purpose and Comments
File system classes[a]	`Directory`	To create, delete, move, and detect directories
	`File`	To create, delete, move, copy, and detect files
Provider-specific classes for SQL Server CE[b] and SQL Server[c]	`SqlCeEngine`	To create and compact a SQL Server CE database
	`SqlCeConnection`, `SqlConnection`	To open a connection to a database. Only one open connection per SQL Server CE database is allowed at a time
	`SqlCeCommand`, `SqlCommand`	To submit a SQL statement to the database. Contains the connection object

Continues

Download at WoweBook.Com

TABLE 6.3: Classes for Accessing, Maintaining, and Displaying Data *(Continued)*

Category	Class	Purpose and Comments
	`SqlCeDataReader`, `SqlDataReader`	To retrieve the rows returned by a command. Can be created only by the command's `ExecuteReader` method. Only one open data reader per SQL Server connection is allowed at a time
	`SqlCeDataAdapter`, `SqlDataAdapter`	To move data between a database and a `DataTable`. Contains up to four (`SELECT`, `INSERT`, `UPDATE`, and `DELETE`) command objects
	`SqlCeCommandBuilder`, `SqlCommandBuilder`	To reverse-engineer a simple `SELECT` statement to create the `INSERT`, `UPDATE`, and `DELETE` statements used by an adapter to move modified data from a `DataTable` back to the database
Provider-independent ADO.NET classes[d]	`DataSet`	To provide a memory-resident database. Contains the provider-independent classes listed below
	`DataTable`	To provide a memory-resident table
	`DataRow`	To provide a row within a `DataTable` object. Contains the data. Has methods to access other rows in other tables based on `DataRelation` objects, such as the `GetChildRows` method
	`DataView`	To sort and filter the rows of a `DataTable` object. Primarily for use with data binding
	`DataRelation`	To specify the parent/child relationship between two `DataTable` objects
	`UniqueConstraint`	To constrain a `DataTable` object
	`ForeignKeyConstraint`	To constrain two `DataTable` objects

Download at WoweBook.Com

TABLE 6.3: Classes for Accessing, Maintaining, and Displaying Data *(Continued)*

Category	Class	Purpose and Comments
Data-binding support classes[e]	Multi-item controls: `DataGrid`, `ListBox`, `ComboBox`	To display one or all columns of a `DataTable` or `DataView` object (when data-bound). Cannot be updated
	Single-item controls: `TextBox`, `Label`, `RadioButton`, `CheckBox`, `Button`, etc.	To display one column of one row of a `DataTable` or `DataView` object (when data-bound). Automatically propagates data changes back to the bound `DataTable`. A `CurrencyManager` object (see below) manages row positioning and data updating
	`DataBindings` collection	To specify data binding between a `DataTable` or `DataView` object and the single-item control; is a property of a single-item control
	`BindingContext`	To contain a collection of `CurrencyManager` objects; is a property of a form. Used by the form to manage all data binding of all contained controls. Created by the .NET runtime
	`CurrencyManager`	To manage the automatic updating of bound `DataTable` objects. Can be used to override the automatic updating of bound `DataTable` objects by canceling the update or forcing the immediate execution of the update. Contains a read-write indexer property that identifies the "current" row of the bound `DataTable` or `DataView`. Created by the .NET runtime. Can be obtained by calling the form's `BindingContext(DataTable/DataView name)` indexer property

a. In the `System.IO` namespace.
b. In the `System.Data.SqlServerCe` namespace.
c. In the `System.Data.SqlClient` namespace.
d. In the `System.Data` namespace.
e. In the `System.Windows.Forms` namespace.

Download at WoweBook.Com

We discussed the file system classes in detail in Chapter 5, and we covered controls used for data binding in Chapter 4. Note that SQL Server permits multiple open connections but only one open data reader per connection, while SQL Server CE is just the opposite, permitting only one open connection but allowing multiple open readers on that connection.

Also note that .NET Compact Framework multi-item controls do not permit users to enter data. As we mentioned in Chapter 4, the `DataGrid` control is for display only, and the `ComboBox` control does not permit direct entry of a value, allowing the user to select only from the list.

6.1.2.1 *Namespace References and Assembly References*

The SQL Server CE–specific ADO.NET classes are located in the `System.Data.SqlServerCe` namespace. There is a similar set of classes in the `System.Data.SqlClient` namespace for use when accessing SQL Server. Other namespaces hold classes for accessing data via Oracle, OLE DB, ODBC, and other providers on the desktop and, if supported, on a device.

Because the classes are located in specific namespaces, you must set some references in your project before writing any code. You need to set a reference to the `System.Data.SqlServerCe` namespace because it holds the classes you must use to access a SQL Server CE database; and you need to set a reference to the `System.Data.SqlClient` namespace when accessing SQL Server. You also need to set a reference to the `System.Data.Common` namespace because it contains private classes used by the data adapter class. And, finally, you need a reference to the `System.Data` namespace because it contains the provider-independent classes listed in Table 6.3. Therefore, among the namespace references and the assembly references you may need are those listed in Table 6.4.

TABLE 6.4: Commonly Used Namespaces for Data Access

Namespace	Located in Assembly
`System.Data`	`System.Data.dll`
`System.Data.Common`	`System.Data.Common.dll`
`System.Data.SqlServerCe`	`System.Data.SqlServerCe.dll`
`System.Data.SqlClient`	`System.Data.SqlClient.dll`

Download at WoweBook.Com

6.1.2.2 *Functionality: Supersets and Subsets*

The minimum set of provider-specific classes, methods, properties, and events that must be supported by a provider has been specified by Microsoft as a set of interfaces. For instance, the PMEs that a `DataReader` must support are defined in the `IDataReader` interface. As you move between SQL Server CE and SQL Server, you use the same properties and methods to access the data regardless of provider. For example, we're about to use the `SqlCeDataReader` class, which has a `NextResult` method. The `SqlDataReader` class and the `OleDbDataReader` class also have this same method, as does the `AdoceDataReader` class in the Pocket Access data provider from In The Hand. They must all have the `NextResult` method because the interface requires it.

The advantage of this standards-driven approach is that the same code can be used to access data from a variety of providers, yet the underlying classes can be optimized to take advantage of the capabilities of the individual data provider. The disadvantage of this approach, especially for SQL Server CE programmers, is that functionality that is inappropriate for SQL Server CE exists in our classes simply because it may be appropriate for SQL Server or Oracle or some other data provider and therefore has been included as part of the standard.

Continuing with our example, the `SqlCeDataReader` class's `NextResult` method provides for the processing of commands that contain multiple `SELECT` statements. But the SQL Server CE database does not support multiple `SELECT` commands. For this reason, the `NextResult` method is not supported and so is completely inappropriate when accessing a SQL Server CE database.

Thus, learning the object model in the `System.Data.SqlServerCe` namespace requires more than just learning the properties and methods—it also involves learning which of those properties and methods are appropriate for SQL Server CE (or whatever ADO.NET data provider you choose to use). One of the goals of this chapter is to focus on the classes, methods, and properties that are important for some typical processing scenarios in SQL Server CE.

Just as the SQL Server CE syntax is a subset of that found in SQL Server, ADO.NET for Windows CE is also a subset of the desktop version. For instance, only three of the eight overloads for the `DataSet.ReadXml` method are implemented. In general, however, ADO.NET for Windows CE has most of the functionality that the desktop version has, and you should not feel constrained.

Download at WoweBook.Com

6.1.3 ADO.NET Error Handling

Before diving into SQL Server CE programming in general, we take a moment to examine error handling. In the .NET Compact Framework, ADO.NET error handling uses the standard `try...catch...finally` technique used in any .NET application. If an error occurs, a `SqlCeException` is thrown. There is only one `SqlCeClient` exception class, `SqlCeException`, not one derived class for each possible error or category of error. Because SQL Server CE is written in unmanaged (native) code, it is written for the world of error numbers and `HResults`, not exceptions. Thus, each error represented by a `SqlCeException` contains its own error information.

The `SqlCeException` class has an `Errors` collection property, consisting of one or more `SqlCeErrors` (each containing the `Source`, `Message`, `NativeError`, and `HResult` properties, which are also properties of the `SqlCeException` class) plus three string parameters and three numeric parameters. In the `SqlCeException` class properties, `Source`, `NativeError`, and `HResult` are equal to the properties of the same name of `SqlCeException.Errors[0]`, while `Message` is equal to `String.Empty`. The `SqlCeException` class also has an `InnerException` property, which is often empty. So, your best plan when handling a SQL Server CE exception is to examine its `Errors` collection. For instance, if a `SELECT` statement contains a misspelled column name, you receive the following information via the `SqlCeException.Errors[0]` object:

`Source`: Microsoft SQL Server Windows CE Edition.
`Message`: The column name is not valid.
`NativeError`: 25503.
`HResult`: -214721900.
`ErrorParameters[0]`: Empty.
`ErrorParameters[1]`: [The name of the misspelled column.]
`ErrorParameters[2]`: Empty.
`NumericErrorParameters[0]`: 0.
`NumericErrorParameters[1]`: 0.
`NumericErrorParameters[2]`: 0.

You should always use error handling. If nothing else, error handling allows your application to present the most meaningful message possible to the user and to exit gracefully, as shown in the following code:

Download at WoweBook.Com


```
private string  strConn = "Data Source=" +
                          @"My Documents\ourProduceCo.sdf";
             :
             :
SqlCeEngine dbEngine = new SqlCeEngine();
dbEngine.LocalConnectionString = strConn;
             :
             :
try
{
   dbEngine.CreateDatabase();
}
catch( SqlCeException exSQL )
{
   MessageBox.Show("Unable to create database at " +
                   dbEngine.LocalConnectionString +
                   ". Reason:  " +
                   exSQL.Errors[0].Message );
}
```

We mention error handling because it is a necessary part of production code. Without it, your code is brittle to the touch of all but the most casual user. With it, your code becomes robust enough to withstand the most brutal attack. That being said, you may notice that our sample code in this book has only a minimal amount of error handling (and sometimes none at all). We do this to keep the focus on the subject at hand and to help make the sample programs easier to understand.

Utility Routines

In any programming environment it is advantageous to have a set of utility routines for performing generic tasks. This is certainly true in the ADO.NET environment, where data is repetitively moved from one tier to another or converted from one format to another. For instance, several ADO.NET methods return an array of data rows, arrays that you might like to convert into a data table so that you can bind it to a control. This "convert from row array to data table" task is a good candidate for being handled by a generic routine. We have written some generic routines for use in this chapter and placed them in the `UtilData` class located in the `UtilData` project at the book's Web site, www.paulyao.com/cfbook/code.

Download at WoweBook.Com

6.2 Working with Data Sets

The ADO.NET `DataSet` class lies at the center of the three-tiered approach and at the heart of this chapter. Understanding the `DataSet` class and its contained objects is a key to successful data management in managed code. Let's recap what we have already said about data sets (including Table 6.3).

- A `DataSet` object is a memory-resident database.
- A `DataSet` object can contain multiple `DataTable` objects.
- `DataTable` objects can have constraints defined for them (e.g., `primary key`, `foreign key`, and `unique`).
- Parent/child relationships can be specified between the `DataTable` objects of a `DataSet` object.
- A `DataView` object sorts and filters the rows of its associated `DataTable` object. Each `DataView` is a view of just one table. A `DataTable` can have multiple `DataViews` defined on it.

Remember that the `DataSet` class and associated classes are located in the `System.Data` namespace and therefore are not provider-specific. That is, a single `DataSet` object can be used to hold the data regardless of the provider from which the data was obtained and regardless of the environment in which it is being used. The `DataSet` class is used today in Web services, Web applications, and Windows applications, as well as mobile applications.

`DataSet` objects provide four major benefits for the mobile application developer.

1. They can contain data from a variety of remote data sources.
2. They function while disconnected from those data sources.
3. They can persist their data locally as XML files through their `ReadXml`/`WriteXml` methods.
4. They display their data through data binding.

Although `DataSet` objects do not have all the functionality of SQL Server CE (see the `DataSet` Class Limitations sidebar), their capability to retrieve, disconnect, data-bind, persist locally, reconnect, and update remotely is ideal for lightweight remote applications that do not need the full functionality of SQL Server CE. Additionally, if the application does

Download at WoweBook.Com

require SQL Server CE, DataSet objects are still the best way to display the data to the user and to persist all of the data (or subsets of it) as XML files.

DataSet Class Limitations

DataSet objects do not have an ad hoc query capability. Instead, the data in a DataSet object is accessed programmatically, not through a SELECT statement. Properties and methods of the data objects are used to access the data. Thus, you access all rows of a data table by iterating through its Rows collection with a For Each statement; you access all rows of a data table that meet a certain criteria through the table's Select method; and you access the rows of a relationship by using the parent row's GetChildRows method.

6.2.1 Creating and Accessing DataSet, DataTable, and DataView Objects

Creating a DataSet object is easy: You simply "new" it, with the option of specifying a name, as follows:

```
dsetDB = new DataSet("AnyName");
```

Within a DataSet object, you can create a DataTable object and populate it from scratch, as shown in Listing 6.1.

Listing 6.1: Building a DataTable Object from Scratch

```
private void CreateTable() {
      //  Create an empty table.
      DataTable  dtabCustomers = new DataTable("Customers");

      //  Create three columns.
      DataColumn  dcolID = new DataColumn("ID");
      dcolID.DataType = typeof(int);
      dcolID.AutoIncrement = true;
      dcolID.AutoIncrementSeed = 1;
      dcolID.AutoIncrementStep = 1;

      DataColumn  dcolName = new DataColumn("Name");
      dcolName.DataType = typeof(string );

      DataColumn  dcolAddress = new DataColumn("Address");
      dcolAddress.DataType = typeof(string );

      //  Add the columns to the table.
```

Download at WoweBook.Com


```
        dtabCustomers.Columns.Add(dcolID);
        dtabCustomers.Columns.Add(dcolName);
        dtabCustomers.Columns.Add(dcolAddress);

        //  Add a primary key constraint.
        dtabCustomers.Constraints.Add("PKCust", dcolID, true);

        //  Add two rows to the table
        DataRow drowCust;
        drowCust = dtabCustomers.NewRow();
        drowCust["Name"] = "Amalgamated United";
        drowCust["Address"] = "PO Box 123, 98765";
        dtabCustomers.Rows.Add(drowCust);
        drowCust = dtabCustomers.NewRow();
        drowCust["Name"] = "United Amalgamated";
        drowCust["Address"] = "PO Box 987, 12345";
        dtabCustomers.Rows.Add(drowCust);
        }
    }
```

However, the most common way to create and populate a data table is to execute a `SELECT` statement against a database and place the results into a `DataTable` object. These `SELECT` statements can be as simple as `SELECT * FROM Products` or as complex as a multitable join with an aggregate function and grouping. Different data tables in the same data set can be drawn from different databases. We cover the details of moving data between a data set and a database in the Microsoft SQL Server CE and Microsoft SQL Server sections later in this chapter. For now, we are only interested in accessing the data as it resides in the data set and in the movement of data between the data set and the user.

6.2.1.1 *Understanding Data Tables*

A data table consists of a collection of columns and a collection of rows. `DataRow` objects are a collection of `Items`, which are the fields of the row. To access a value in a data table you must specify the row number and column identifier (either column name or number), as shown in the following code, which sets the `CategoryName` column of the second row in the `dtabCategories` data table to `Muzzy`:

```
dtabCategories.Rows[1]["CategoryName"] = "Muzzy";
```

You can also use the `DataTable` class's `Select` method to access rows by value. The following code returns the array of rows whose `CategoryName` is `Widgets`:

Download at WoweBook.Com


```
dsetDB.Tables["Categories"].Select("CategoryName = 'Widgets'");
```

To save the reference to the first row in the array, you would code the following line. (Remember that arrays are zero-based; the first row of this array is row 0.)

```
drowX = dsetDB.Tables["Categories"].Select(
        "CategoryName = 'Widgets'")[0];
```

Accessing columns by name makes your code more readable, but the code will execute more quickly if you access columns by index value rather than by name. A good performance strategy is to look up the row index value for the name once and thereafter access the column by index value.

We often think of a data table as a two-dimensional object, but we should think of it as a three-dimensional object. Each data element (i.e., each field) in a data table can have up to four values, referred to as *versions:* `Current`, `Original`, `Proposed`, and `Default`.

Having four possible values for each field does not necessarily make the data table four times as large, because values exist only when necessary. The `Default` version, for instance, applies at the column level, not the individual field level. There is at most one `Default` value per column, not one for each field.

In addition to multiple possible field versions, individual rows have a status value, such as `Unchanged`, `Modified`, `Added`, or `Deleted`. Understanding the meaning of the versions and status codes, and the relationship between them, is essential when programming for data sets.

To illustrate this, we focus on two field versions, `Original` and `Current`, and two row statuses, `Unchanged` and `Modified`.

When new rows are added to a data table, the `Original` and `Current` values are equal to each other and the status is set to `Unchanged`. When field values change in a data table—either because of changes done by your code or from user actions on a data-bound control—the new value becomes the `Current` value, the old value remains the `Original` value, and the state is set to `Modified`.

At some point in the execution of the program, you decide that you have "completed" your changes, that new values are no longer "new." That is, you want the `Original` value thrown away and replaced with the `Current` value, and you want the row status set back to `Unchanged`. Normally, this desire to "shuffle" versions and statuses occurs because you have pushed the changes back to the underlying database, but it could occur for a variety of other reasons as well.

Download at WoweBook.Com

You accomplish this synchronization of versions and changing of states by calling the DataRow class's AcceptChanges method. The DataSet and DataTable classes also have AcceptChanges methods, which operate by cascading the AcceptChanges call down to the individual rows. Never call the AcceptChanges method just prior to updating the database. The class used for transferring data from a data table to the database, the data adapter class, examines the rows' status first. Data in rows that have their status marked as Unchanged do not cause updates to the database. (See The SqlCeDataAdapter Class subsection later in this chapter.)

Having mentioned the Current, Original, and Default versions of a data row, we need to say a word about the fourth version, the Proposed version. The Proposed version is used to provide transaction commit and rollback capabilities and exists only during the life of a transaction.

6.2.1.2 *Working with Data Rows*

As mentioned earlier, data rows contain the actual data, which can be accessed or modified by specifying the row and the column within the row. Columns can be specified by either name or index.

Data rows have methods for accessing other rows based on data relations that have been defined in the data set. Two of the most common methods are GetChildRows and GetParentRow. For example, the code in Listing 6.2 creates a relationship between the Categories and Products tables and then retrieves an array of all rows containing all products in the Widgets category, assuming that the dsetDB data set already contains the Categories and Products tables.

Listing 6.2: Creating and Using a DataRelation Object

```
//  Define the relationship
//     between Products and Categories data tables.

dsetDB.Relations.Add(
   "FKProdCat",
   .Tables["Categories"].Columns["CategoryID"],
   .Tables["Products"].Columns["CategoryID"],
   true);

//  Select the Widgets row in the Categories data table.
//    Use it to retrieve all Widgets rows from the
//    Products data table.

dsetDB.Tables["Categories"].
```

Download at WoweBook.Com


```
   Select("CategoryName = 'Widgets'")[0].
      GetChildRows("FKProdCat");
End With
```

6.2.1.3 *Introducing Data Views*

A data view is always a view of a single data table, and it provides only two capabilities: sorting and filtering rows. A `DataView` object cannot be used to do joins, evaluate aggregate functions, or filter out columns. It is not intended to behave like a SQL Server view. Its purpose, instead, has to do with data binding.

Every data table has a `DefaultView` property. When this property is first accessed, the `DataView` object is created, and its `Filter` and `Sort` properties are set to `String.empty`. So, its contents are, on first access, identical to the table's contents. Thereafter, you can set the `Filter` and `Sort` properties to anything you want. Or you can create a new view of a table, specifying the table, row selection criteria, sort sequence, and row status criteria, as in the following statement:

```
new DataView(dsetTimeTracker.Tables["Projects"],
             "strIdent = '17'",
             "dateStart ASC, dateEnd DESC",
             DataViewRowState.CurrentRows);
```

Having discussed the `DataSet` object and its contained objects, it's time to cover moving data between the data set and the presentation tier—in other words, *data binding*.

6.2.2 Data Binding

Data binding is the ability to associate an object that holds data with a control that displays and updates that data. We covered the subject in Chapter 4, but now we need to apply it to the `DataTable` and `DataView` classes specifically. The following list summarizes what we covered in Chapter 4.

- The `DataGrid` control can display the entire contents of a bound data object.
- You bind to a `DataGrid` control by setting its `DataSource` property.
- Unlike the desktop `DataGrid` control, the .NET Compact Framework version is read-only. We presented a workaround for this limitation in Chapter 4.
- The `ListBox` and `ComboBox` controls can display one column of a `DataTable`/`DataView` and can use one other column for an identifying key.

Download at WoweBook.Com

- You can bind to a `ListBox` or `ComboBox` control by setting its `DataSource`, `DisplayMember`, and `ValueMember` properties.
- Unlike the desktop `ComboBox` control, the .NET Compact Framework version is read-only. That is, the user can select from the list of choices but cannot enter text into the `ComboBox` control.
- Single-item controls, such as the `TextBox`, `Label`, `RadioButton`, and `CheckBox`, can bind to a single data element of a data object.
- You can bind to a single-item control by adding entries to its `DataBindings` collection.
- The `DataBindings` collection specifies the column to be bound to but not the row. A "current" row of a `DataTable`/`DataView` is always used for binding.
- Data binding is a two-way street. Changes to data made in the control are automatically pushed back to the bound data table.

To lead into our upcoming discussion of data binding, we add the following information to our list.

- Every `DataTable` object has an associated `DataView` object that can be used for data binding. Additional views can be created for each table, but one view is always present once the table has been created.
- `DataTable` objects do not really have a "current" row. Instead, the `DataTable`'s `CurrencyManager` object, located within the form's `BindingContext` property, must be used to position the `DataTable` to a row.

One side note before we begin writing our data binding code: In Chapter 7, we cover the subject of LINQ. Although not directly connected to the subject of data binding, LINQ does provide you with additional options for gathering, creating, filtering, and sorting the data you wish to display. For this reason, there will considerable data-binding code in Chapter 7, and your ability to present data to the user will be extended beyond what is covered in this chapter by the information presented in the next chapter.

We begin by writing a very simple application to illustrate the benefits and issues of data binding. We use a very simple SQL Server CE database to populate the data set. The tables in the data set are the `Categories` and `Products` tables; the relationship between them has been defined within the data set by adding a `DataRelation` object named `FKProdCat` to the data

Download at WoweBook.Com

set. Once the data set has been populated, we use data binding to display the information in the tables to the user. The focus here is on the binding of data between the data set and the controls, not on the movement of data between the data set and the database. We cover the latter in detail later in this chapter when we discuss accessing SQL Server CE and SQL Server databases.

The application, located in the `DataBinding` project on the book's Web site,[1] consists of two forms. The first form, shown in Figure 6.3, consists of two `DataGrid` controls, one for displaying the `Categories` rows and one for displaying the `Products` rows of whichever `Category` the user selects. In other words, we want to reflect the parent/child relationship between categories and products on the form.

The second form, shown in Figure 6.4, displays one product at a time in `TextBox` controls and provides the user with a variety of ways to specify the desired product. It also allows the user to update the product information.

The two forms are not related in a business sense; we are simply using one to illustrate multi-item data binding and the other to illustrate single-item data binding.

The first form is the simpler to program, as we shall see, for it binds data tables to multi-item controls.

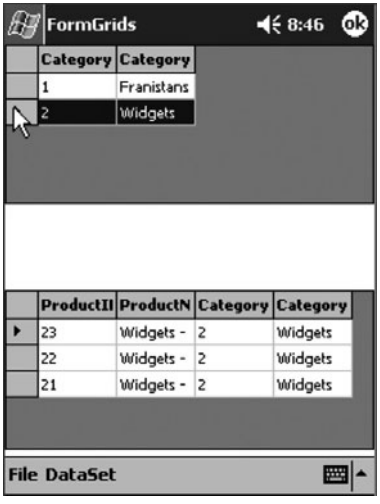


FIGURE 6.3: The Parent/Child Form

1. You can download the projects mentioned in this chapter from www.paulyao.com/cfbook/code. Follow the instructions listed on the Web site.

Download at WoweBook.Com

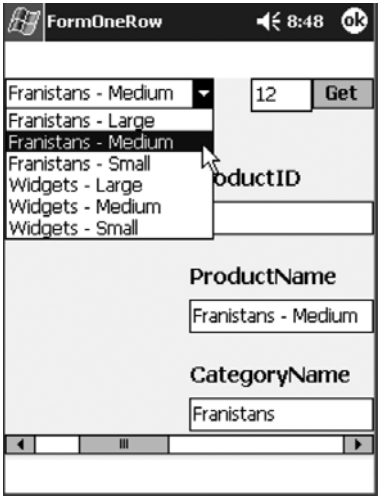


FIGURE 6.4: The Single-Item Form

6.2.2.1 *Binding to Multi-Item Controls*

The form shown in Figure 6.3 declares a private variable to hold the data set named `dsetDB`; creates a new data set, storing the reference in `dsetDB`; and loads `dsetDB` with data from the SQL Server CE database. At this point, we have a data set that contains two data tables and the relationship between them.

To reflect that relationship on the form, we bind the upper `DataGrid` control to the `Categories` data table so that the entire table is displayed within the `DataGrid` control. We bind the lower `DataGrid` control to the default view of the `Products` table because we can easily update the view whenever the user selects a new category. Here is the data-binding code:

```
private void mitemDisplayDS_Click(object sender, EventArgs e)
{
   //  Display the Categories and Products tables
   //     in the parent and child DataGrids.
   dgridParent.DataSource =
      dsetDB.Tables["Categories"];
   dgridChild.DataSource =
      dsetDB.Tables["Products"].DefaultView;
}
```

Whenever the user selects a new category, the application reacts to the `CurrentCellChanged` event by setting the row filter for the `Products` view to select only products of that category, as shown in this code:

Download at WoweBook.Com


```
private void dgridParent_CurrentCellChanged(object sender,
EventArgs e)
{
   DataTable dtabParent = (DataTable)dgridParent.DataSource;
   DataView dviewChild = (DataView)dgridChild.DataSource;
   dviewChild.RowFilter =
      "CategoryID = " +
      dtabParent.Rows[dgridParent.CurrentRowIndex]["CategoryID"];
}
```

Thus, the `DataView` class is the key piece in reflecting the parent/child relationship to the user.

6.2.2.2 *Binding to Single-Item Controls*

When asked to display the second form shown in Figure 6.4 the first form stores a reference to the data set in an `Internal` variable so that the second form can retrieve it and then displays the second form. The second form displays one product row at a time, using `Label` and `TextBox` controls. This makes it inherently more complex than the first form. Binding to single-item controls is more difficult than binding to multi-item controls for three reasons.

1. The user must be given a mechanism to specify which product should be displayed.
2. The `TextBox` controls must display the specified row.
3. The `TextBox` controls can be used to update data as well as display it.

Data binding to single-valued controls, such as `TextBox` controls, requires some additional design decisions. Making good decisions requires an understanding of data-binding internals. So, let's look at the following decisions that must be made and the impacts of each.

1. How does the user designate which row is to be displayed?
   a. *By matching binding.* Bind the data object to a multi-item control, such as a `ComboBox` or `DataGrid`, as well as to the single-item controls, such as the `Label` and `TextBox` controls. The user selects the desired row from the multi-item control.
   b. *By indexing.* Provide a scroll bar or some other position control. The user indicates the relative position of the desired row within the data object. The application positions to that row.

Download at WoweBook.Com

   c. *By search key.* Provide a text box. The user enters a value. The application searches the data object for the row containing the entered value.

2. How should that row be assigned to the control?
   a. *By current row.* The application designates the desired row as the current row of the data object.
   b. *By data view.* The application binds the single-item controls to the data table's `DefaultView` object and then sets the view's `Filter` property to limit the view to just the desired row.
3. When should the data table be updated with the values from the single-item controls?[2]
   a. *When the user moves to a new field?* No. It is not necessary to update the data object if the user is continuing to modify other fields of the same row; wait until the user moves to a new row.
   b. *When the user positions to a new row?* No. Data binding will automatically update the old row values whenever the user moves to a new row.
   c. *When the user indicates that he or she is finished with the edit (e.g., by using a Cancel or Update button or by closing the form)?* Yes. You need to use the data table's `CurrencyManager` object to complete or cancel the update of the current row if the user exits the operation without moving to a new row.

The following subsections discuss these three decisions in more detail.

6.2.2.3 *Designating the Row to Be Displayed*

Figure 6.5 shows the single-item form providing all three methods mentioned previously for designating which row to display. We use this form to cover the issues being addressed here. It's not the most beautiful form we ever designed, but it does illustrate the functionality we want to cover.

Matching Binding Of the three designation methods, the easiest to program is matching binding. Once you bind the single-item controls and the multi-item controls to the same data object, you no longer have to decide how to assign the row to the single-item controls because the user's selection

2. The question of when and how the data should be propagated from the data table down to the database is covered later in the Microsoft SQL Server CE and Microsoft SQL Server sections of this chapter.

Download at WoweBook.Com

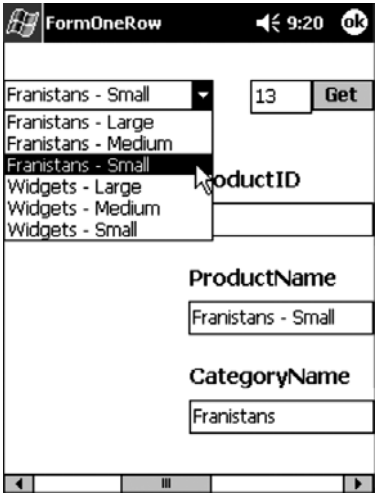


FIGURE 6.5: Three Ways to Designate Which Row to Display

is automatically reflected in the single-item controls. In our example, the binding is done in the form's `Load` event handler, as shown here:

```
//  Bind the ComboBox with the Product names.
comboProductIDs.DisplayMember = strPKDesc;
comboProductIDs.ValueMember = strPKName;
comboProductIDs.DataSource = dtabProducts;
comboProductIDs.SelectedIndex = 0;

//  Bind the DataTable's columns to the text boxes.
textProductID.DataBindings.Add
   ("Text", dtabProducts, strPKName);
textProductName.DataBindings.Add
   ("Text", dtabProducts, strPKDesc);
textCategoryName.DataBindings.Add
   ("Text", dtabProducts, strFKDesc);
```

When the user taps on an entry in the combo box, that row is displayed in the text boxes. If the user enters new values in the text boxes, those values are updated to the data set as soon as the user selects a new row; and the validation events for the text box fire before the row is updated. What more could we want?

When writing the code to do data binding, set the `DisplayMember` property prior to setting the `DataSource` property. Setting the `DataSource` property causes things to happen, some of them visible to the user. Therefore, you want everything in place before setting the `DataSource` property. To

Download at WoweBook.Com

illustrate this, you can take the sample code shown on the previous page and reposition the two calls so that the `DataSource` property is set prior to the `DisplayMember` property. When you run the code you may notice a slight flicker in the combo box.

Indexing Indexing is easy to implement but not as easy as matching binding. And although indexing is easy, it is not obvious. When the user positions the scroll bar to a value of *n,* the application knows that it needs to make row *n* the current row of the data table. Unfortunately, data tables do not have a current row property because the concept of a current row is meaningful only within data binding. To maintain a current row context when not data-bound would be wasted overhead.

Because data binding involves controls and because controls reside within forms, the `Form` class assumes the responsibility for managing the bindings and for maintaining the concept of the current row. When the first binding in the previous code excerpt that involved the `dtabProducts` data table executed, the form created an instance of the `BindingContext` class for managing `dtabProducts` bindings. As the subsequent binding code executed, that object was updated with information about the additional bindings.

When the user tapped on the *n*th entry in the combo box, the form reacted to the combo box's `SelectedIndexChanged` event by noting that the combo box was bound to `dtabProducts` and that the text boxes were also bound to `dtabProducts`. So, the form notified the text boxes to obtain their `Text` property values from row *n* of the data table, and it updated the binding context object for `dtabProducts` to have a `Position` property value of `n`.

To control the current row of a bound data table and thus control which row is displayed in the text boxes, you need to tap into the form's binding management capability. Specifically, you need to obtain the `CurrencyManager` object for the data table (or data view) and set its `Position` property's value to `n`. The `CurrencyManager` object for a data object is contained in the default collection within a form's `BindingContext` object. In our application, this means handling the scroll bar's `ValueChanged` event as shown in the following line of code (assuming the scroll bar is named `hsbRows` and its minimum and maximum values are 0 and the number of rows in the data object minus 1, respectively):

```
this.BindingContext[dtabProducts].Position = hsbRows.Value;
```

Download at WoweBook.Com

This causes the text boxes to update their `Text` property values with the values from the row of the `dtabProducts` table indicated by `hsbRows.Value`.

It seems like we are saying that indexing implies assigning controls according to the current row. Instead, you might wonder what's wrong with binding the text boxes to `dtabProducts.DefaultView`, rather than to the data table itself, and then reacting to the scroll bar's `ValueChanged` event by setting the view's `Filter` property with the primary key of the *n*th row, as shown here:

```
dtabProducts.DefaultView.RowFilter =
        "ProductID = " +
        dtabProducts.Rows[hsbRows.Value]["ProductID"];
```

Now the text boxes are bound to a view that contains only one row, which is the row they display, and the concept of current row becomes meaningless.

You can do it this way, and it works fine until you try to update the fields by entering new values into the text boxes. When you enter data into the text boxes and then reposition the scroll bar thumb, two things go wrong. First, tapping in the scroll bar does not cause the text box to lose focus, and thus the validation events do not fire. Second, you do not move from one row to another row within the bound view; instead, you replace the contents of the bound view with new contents, and thus the values in the underlying data table are not updated.

So, although indexing does not necessarily imply assigning controls by current row, we recommend doing it that way whenever you choose to use indexing. And as you just saw, you can implement that choice in one line of code.

Search Key The third method to use when designating the desired row has the user enter a search value, which fits nicely with the indexing option because both are value-based rather than position-based techniques. The single-item controls can be bound to the data table's default view, as shown in the following code:

```
//  Bind the DataTable's columns to the text boxes.
textProductID.DataBindings.Add
   ("Text", dtabProducts, strPKName);
textProductName.DataBindings.Add
   ("Text", dtabProducts, strPKDesc);
textCategoryName.DataBindings.Add
   ("Text", dtabProducts, strFKDesc);
```

Download at WoweBook.Com

Then the controls can be positioned to the requested row by extracting the user-entered key value and specifying it when setting the default view's `Filter` property:

```
dtabProducts.DefaultView.RowFilter =
   "ProductID = " + textGet.Text;
```

Because `ProductID` is the primary key, this limits the view to, at most, one row.

Using data views to assign the row to the text boxes in this situation does not lead to the data set updating problems that option had when used in conjunction with a scroll bar. When the user begins to enter a new key value after entering new field values in the old row, the focus does shift, the validation events are called, and the underlying data table is updated.

Conversely, assigning rows according to the current row, which is position-based rather than value-based, does not fit well with having users designate rows by search keys. No single property or method of the `DataTable`, `DataView`, or `DataRow` class translates a primary key value into a row number. If you need this type of translation, you need to write your own routine to provide it.

6.2.2.4 *Assigning the Controls to a Row*

Because, as we have just seen, this subject is so tightly tied to the issue of designating the row to be displayed, we have already covered it. We summarize with these points.

- If the user is designating the desired row by selecting from a multi-item control bound to the same data object as the single-item controls, do nothing.
- If the user is designating the desired row by entering an index value, bind your single-item controls to the data table, obtain the binding context for the data table from the form, and set the `Position` property to the index value.
- If the user is designating the desired row by entering a search value, bind your single-item controls to the data table's default view, and use the `Filter` property to accept only rows that contain the search value.

Download at WoweBook.Com

6.2.2.5 *Updating the Bound Data Table*

We said earlier that data binding is a two-way street: Changes made in the control propagate back to the data set. However, bound controls do not update the underlying row until they are repositioned to a new row. Normally, you want this behavior because the row is probably out of sync with itself as its individual fields are being entered and updated, and it is best to wait for the automatic update that occurs when the user moves to a new row.

It is always possible for the user to complete the editing of a row and not move to a new row. If the user positions to a row, makes a change to the contents through a bound control, and then closes the form, the change the user made is not persisted—it is lost. This is because the application never repositioned to a new row after the change was made, and therefore the data table was not modified.

This is why forms tend to have specific Update and Cancel buttons on them and why we handle not only the `Click` events of those buttons but also the form's `Closing` and `Deactivate` events. To programmatically complete or cancel an update (i.e., to force or prevent the transfer of data from the single-item controls to the data table), use the `CurrencyManager` object's `EndCurrentEdit` or `CancelCurrentEdit` method, respectively. This is the same `CurrencyManager` object used earlier in this chapter to specify the current row of a data object. For instance, the following code reacts to the form's `Closing` event by completing the edit of the current row, thus causing the data in the controls to propagate to the row:

```
using System.ComponentModel;
   :
   :
   private void FormUpdate_Closing(object sender,
                                   CancelEventArgs e)
   {
      //  Force the current modification to complete.
      this.BindingContext[dtabCategories].EndCurrentEdit ();
   }
```

The `CurrencyManager` object has both an `EndCurrentEdit` method to force the update to occur and a `CancelCurrentEdit` method to prevent the update from occurring, as well as the `Position` property that is used to specify the current row.

Download at WoweBook.Com

So, the key points to consider when using data binding to update data set data is how to let the user position the controls to a specific row and how to prevent lost or unintended updates to that row.

This concludes our discussion of moving data between the data set and the presentation layer. For the rest of this chapter we examine moving data between the data set and persistent storage.

6.2.3 Reading and Writing a Data Set as XML

A database is not the only source for or destination of data set data. XML files can also be the storage media for the data. Unlike database access, which requires a separate provider-specific class to perform the transfer, XML file I/O is done by the `DataSet` class itself. After all, an XML file is a text file of information in a standard format; therefore, access to it is provided in a standard manner by derived classes of the `Stream` class.

The `DataSet` class has four methods for performing XML file I/O:

1. `WriteXml`: writes the contents of a data set as XML to a file
2. `WriteXmlSchema`: writes the structure of the data set as an XML schema to a file
3. `ReadXml`: reads the contents of an XML file into a data set
4. `ReadXmlSchema`: reads the contents of an XML schema file and builds the data set structure from it

The XML schema is translated to and from `Constraint` and `DataRelation` objects in the data set, as well as the `DataTable` objects. Unlike relational data, which is flat in structure, XML is hierarchical. The `DataRelation` classes are the mechanism for expressing the hierarchy within the data set. For instance, the following code, which we used earlier in Listing 6.2, creates a `DataRelation` object that specifies the parent/child relationship between the `Categories` and `Products` tables:

```
//  Add relation to the DataSet.
dsetDB.Relations.Add(
   "FKProdCat",
   dsetDB.Tables["Categories"].Columns["CategoryID"],
   dsetDB.Tables["Products"].Columns["CategoryID"],
   true);
```

This relationship has been in our data set ever since we first built the data set. If we now execute the `WriteXml` method, as shown in the following code

Download at WoweBook.Com

excerpt, and then view the contents of the file, we can see the generated XML (see Figure 6.6):

```
//  The XML file
private string strXMLFile =
                        @"My Documents\ourProduceCo.xml";
         :
         :
private void mitemWriteXml_Click(object sender,
                                 EventArgs e)
{
   dsetDB.WriteXml(strXMLFile);
}
```

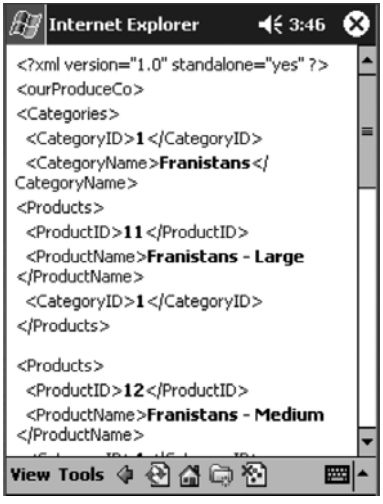


FIGURE 6.6: The XML Generated by Executing the `WriteXml` Method

The XML shown in Figure 6.6 not only contains the data but also reflects the relationship between the tables; the first category element is the first entry in the file, and it contains the product elements for all its products, then the next category element, and within it all its product elements, and so on. The need to display nested relationships in this manner is the reason why the `DataRelation` class has a `Nested` property.

When we added the `FKProdCat` data relationship to our data set in the code shown earlier, we added it with a `Nested` property value of `false`. To ensure that your XML and XML schema reflect the hierarchical nature of your data set, set the `Nested` property of your data relationships to `true`, as shown here:

Download at WoweBook.Com


```
//  Make each relationship a nested relationship
foreach( DataRelation drelForXML in dsetDB.Relations )
{
   drelForXML.Nested = true;
}
```

Or, for a single data relationship, use the following code:

```
//  Make the FKProdCat relationship nested.
dsetDB.Relations["FKProdCat"].Nested = true;
```

It is also possible to obtain nested XML by having matching primary keys and foreign keys defined on the respective data tables, but nested relationships are usually easier to work with because all the information about the relationship is contained within a single `DataRelation` object rather than being spread across two constraint objects.

Thus, .NET Compact Framework data sets give you a convenient way to convert relational data to and from XML format. This, in turn, gives you a way to save data set data that your application has captured to a file without incurring the overhead of using a database to do so.

6.3 Microsoft SQL Server CE

A variety of databases are available for mobile devices, but the one we use throughout these chapters is SQL Server Windows CE Edition, a slimmed-down version of Microsoft's desktop database, SQL Server (in the interest of brevity, we refer to the CE version as "SQL Server CE"). It is the mobile database of choice for .NET Compact Framework programming because it is fully supported in the .NET Compact Framework runtime environment and the .NET Compact Framework development environment.

Most applications written for SQL Server CE have two primary database tasks:

1. Manipulating the data while disconnected
2. Transferring the data to SQL Server when the device is connected to the desktop machine

The first database task involves using the generic ADO.NET classes, which we discussed earlier in this chapter, and the provider-specific classes, which we discuss shortly. For programmers who have worked

Download at WoweBook.Com

with ADO.NET on desktop systems, all of the generic classes on the .NET Compact Framework implementation of ADO.NET are very familiar.

The second major database task, the transfer of data between SQL Server and SQL Server CE, involves not only the participating database engines but also Microsoft Internet Information Services (IIS). Two different mechanisms are available to help with this: Merge Replication and Remote Data Access (RDA). These two topics are the subject of Chapter 8.

Accessing SQL Server CE Databases from Native Code

ADO.NET classes are available only in managed code. But you do not need managed code to access the data in a SQL Server CE database. Two separate native APIs are available for manipulating the data in a SQL Server CE database. One involves a COM object (ADOCE), and the other is an API (OLEDBCE). These components match up with their desktop counterparts, ADO and OLE DB.

6.3.1 SQL Server CE Files

Each SQL Server CE database is stored as a single file on your CE device. The recommended naming convention for the file extension is `.sdf`, and the name of the file is the name of the database. Thus, you open a connection to mydb.sdf, not to mydb. You may associate a password with the database and also encrypt the physical file using 128-bit RSA encryption. Because each database is one file, it is visible in the File Explorer. Figure 6.7 shows a database file selected in the File Explorer.

Because SQL Server CE databases are files in the Windows CE file system, they can be accessed using the `File` and `Directory` classes that we discussed in Chapter 5. For example, you can copy a database file, delete a database file, or test for the existence of a specifically named database file. Your ability to operate on database files is subject to the same restrictions as other files in the file system; for example, you can delete a database file only if the file is not currently in use.

SQL Server CE reclaims space within a database file only when you ask it to; it's not a capability available through the data manipulation language, nor is it provided as a background process. To reclaim space within

Download at WoweBook.Com

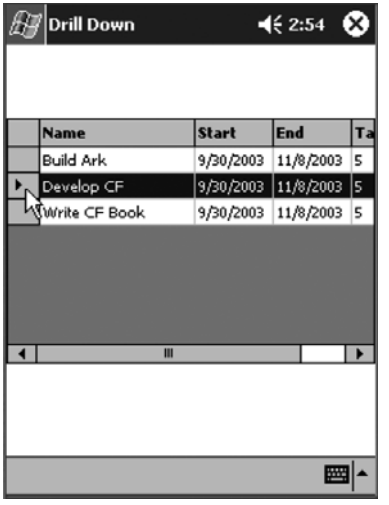


FIGURE 6.7: A Database File Viewed in the File Explorer

a SQL Server CE database, you must use the `Compact` method of the `SqlCeEngine`[3] class.

6.3.2 SQL Server CE Syntax

SQL Server CE programming is simpler than SQL Server programming because the SQL language supported by SQL Server CE is a subset of the SQL available with SQL Server. For example, because a SQL Server CE database is a single-user database whose security is controlled by a file password, there is no need for `GRANT`, `DENY`, and `REVOKE` statements.

Also missing are SQL Server's Transact SQL extensions: no stored procedures, triggers, multistatement batches, or `DECLARE`, `SET`, `IF`, or `WHILE` statements. Even some standard SQL had to be left out; for example, views are not supported. Not all the SQL Server data types could be included, but most of the missing ones can be converted to ones that do exist. For example, Windows CE itself supports only Unicode, and so Unicode is the only character data type (`nchar`, `nvarchar`, `ntext`).

What SQL Server CE does provide is support for tables, indexes, defaults, and referential integrity. It also has the ability to add, modify, and delete rows in those tables using standard SQL Data Manipulation Language (DML). Therefore, your application can manipulate data in SQL Server CE by connecting to the database file and submitting `INSERT`,

3. Fully qualified name: `System.Data.SqlServerCe.SqlCeEngine`.

Download at WoweBook.Com

UPDATE, and DELETE statements to the database. You can write these DML statements yourself, or you can have ADO.NET write them for you as one step in the synchronization of data set updates to the database. Also, the SQL syntax uses single quotes to delineate literals, which is very convenient when you are quoting that SQL within your code—it eliminates the need to quote quotes.

Your SQL Server CE–based application may be more elementary than what you've come to expect from a typical SQL Server application, but it should provide enough functionality to be useful in mobile situations. The lack of the complete Transact SQL syntax is an inevitable limitation given the available memory of a typical Windows CE–powered device. Yet in spite of this size limitation, SQL Server CE provides a more than adequate set of SQL commands for just about any Windows CE–based application you are likely to need.

Tables 6.5 and 6.6 list what functionality in SQL Server is unsupported and supported, respectively, in SQL Server CE.

ANSI Standard SQL

The specification for the SQL syntax is a standard established by an American National Standards Institute (ANSI) Standards Committee. Both SQL Server and SQL Server CE are highly compliant with the ANSI 92 standard. SQL Server has some options for choosing between ANSI compliance and compliance with older versions of SQL Server. These options are not supported in SQL Server CE; ANSI compliance is the only choice in SQL Server CE.

TABLE 6.5: Unsupported SQL Server Functionality in SQL Server CE

Functionality	Comments
DCL[a] GRANT, REVOKE, DENY	These are not needed in a single-user database
DDL[b] Views, triggers, stored procedures, user-defined functions, user-defined data types	Most of these are SQL Server Transact SQL extensions to the ANSI functionality. This is logic that could reside on the SQL Server. It must be coded into the application when using SQL Server CE

Continues

Download at WoweBook.Com

TABLE 6.5: Unsupported SQL Server Functionality in SQL Server CE *(Continued)*

Functionality	Comments
DML[c] `IF-ELSE`, `WHILE`, `DECLARE`, `SET`	Most of these are SQL Server Transact SQL extensions to the ANSI functionality
`INFORMATION_SCHEMA TABLES`	This is replaced by `MSysObjects` and `MSysConstrains` tables

a. Data Control Language.
b. Data Definition Language.
c. Data Manipulation Language.

TABLE 6.6: Supported SQL Server Functionality in SQL Server CE

Functionality	Comments
DDL Databases, tables, data types, indexes, constraints	Only Unicode character types are supported
DML `SELECT`, `INSERT`, `UPDATE`, `DELETE`	
Functions `Aggregate`, `Math`, `DateTime`, `String`, `System`	
Transactions	The transaction isolation level is always `READ COMMITTED`. The limit on nesting is 5. Exclusive lock granularity is table-level and is held for the duration of the transaction. Single-phase commit is the only commit type allowed

SQL Server CE is a little fussier about syntax than SQL Server. Having a smaller footprint means having less code that can "deduce" what you meant. Thus, the following SQL, which is missing the comma between the last column definition and the constraint definition that is required by the ANSI 92 specification, executes as intended in SQL Server but produces a syntax error in SQL Server CE:

```
CREATE TABLE Products
         ( ProductID integer not null primary key
         , ProductName nchar(20) not null
         , CategoryID integer not null
```

Download at WoweBook.Com


```
    CONSTRAINT FKProductCategory foreign key (CategoryID)
                        references Categories(CategoryID)
)
```

Constraints

There are two primary constraints for databases: primary key and foreign key. The primary key constraint specifies the column or columns of a table in which duplicate values are prohibited and a non-null value is required. The product ID in a product table and the order number in an order table are both examples of primary keys. Foreign keys are values that exist as primary keys elsewhere in the database. For instance, the customer number column in an order table is an example of a foreign key because its value must exist in the customer table also.

Column names can be aliased within a `SELECT` statement. These aliases propagate into the data objects and onto the bound controls by default. Thus, the following `SELECT` statement, used to fill a `DataTable` object bound to a `DataGrid` control, would provide the results shown in Figure 6.8:

```
SELECT P.ProductID as ID
     , P.ProductName as Name
     , C.CategoryName as Category
  FROM Products P
  JOIN Categories C on C.CategoryID = P.CategoryID
```

The primary reason to use column aliases is to provide column names for calculated columns and literals or to resolve ambiguous names; it is not the preferred way to specify a column title of a `DataGrid` column. Rather, using the `DataGridColumnStyle` object, covered in Chapter 4, is the preferred way to set `DataGrid` column names because it is independent of the underlying `SELECT` statement.

6.3.3 SQL Server CE Query Analyzer

The primary utility tool for viewing and querying a SQL Server CE database is the SQL Server CE Query Analyzer. Like its counterpart for SQL Server, the SQL Server CE Query Analyzer provides a convenient way to create and submit ad hoc queries.

Download at WoweBook.Com

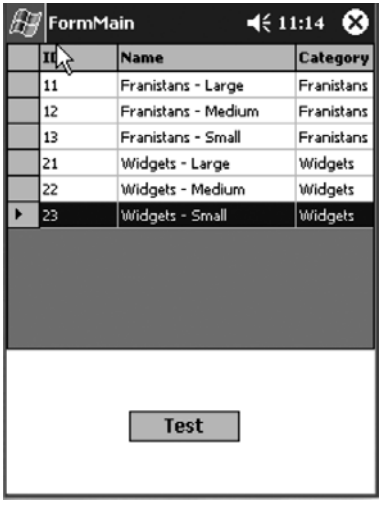


FIGURE 6.8: A `DataGrid` Control with Aliased Column Names

The installation of the SQL Server CE Query Analyzer depends on the development environment onto which SQL Server CE is installed. When you install SQL Server CE, the SQL Server CE Query Analyzer is not installed by default on a device. Once the Query Analyzer (as we'll call the CE version for brevity in the rest of this discussion) is installed on the device, you can run the executable file, Isqlw20.exe, directly from the Start menu or from the directory in which it is installed. Clicking on a database file from within the File Explorer also opens the Query Analyzer.

The Query Analyzer allows you to see structural information about the database (as shown in Figure 6.9) and submit queries to that database (as shown in Figure 6.10).

The Query Analyzer form displays a minimize box, not a close box. If you use the Query Analyzer to examine your database and then click on the minimize box, the Query Analyzer disappears but does not close nor does it close the database it is displaying. Any other application that tries to access the database cannot do so until you either close the database connection in the Query Analyzer (in the Objects tab, click on the tree node with the database name and then click the Stop button on the toolbar) or close the Query Analyzer itself (select Tools | Exit).

To help explain the ADO.NET classes, in this chapter we create a series of sample applications. We begin with database creation and table and index creation; then we explore retrieving, displaying, and updating data

Download at WoweBook.Com

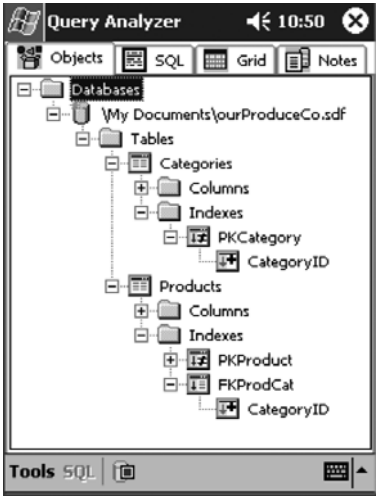


FIGURE 6.9: SQL Server CE Query Analyzer, Objects Tab

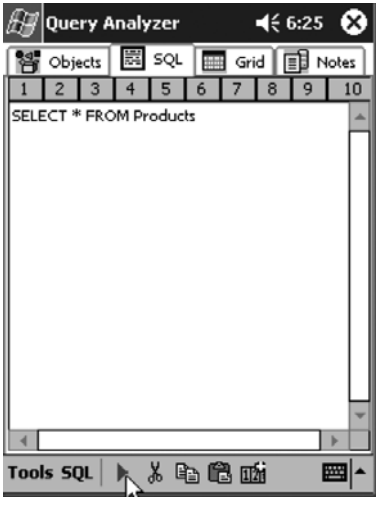


FIGURE 6.10: SQL Server CE Query Analyzer, SQL Tab

using the two-tier approach; and finally we discuss the three-tier approach of creating and using data sets and data binding.

6.3.4 Creating a SQL Server CE Database

Our first sample application for accessing SQL Server CE is located in the `CreateDatabase` project for this chapter on the book's Web site. Shown in Figure 6.11, our application is used to create and populate a database.

Download at WoweBook.Com

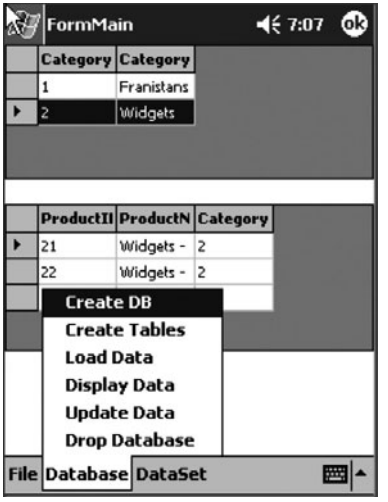


FIGURE 6.11: The `CreateDatabase` Program

To create a SQL Server CE database, we use the `Engine` class located in the `System.Data.SqlServerCe` namespace. Because each SQL Server CE database is a single file, all that we need to tell the `Engine` object is the path and name of that file. We can do this either in the `Engine` class constructor or by setting the object's `LocalConnectionString` property. In either case, we must specify the filename prior to executing the `CreateDatabase` method of the `Engine` class because this method does not take any parameters.

Thus, the code to create a SQL Server CE database looks like Listing 6.3.

LISTING 6.3: Creating a SQL Server CE Database

```
private string  strFile = @"My Documents\ourProduceCo.sdf";
private string  strConn = "Data Source=" +
                          @"My Documents\ourProduceCo.sdf";


private void mitemCreateDB_Click(object sender, EventArgs e)
{
   if ( File.Exists(strFile) ) { File.Delete(strFile); }

   SqlCeEngine dbEngine = new SqlCeEngine();
   dbEngine.LocalConnectionString = strConn;
   try
   {
      dbEngine.CreateDatabase();
```

Download at WoweBook.Com


```
   }
   catch( SqlCeException exSQL )
   {
      MessageBox.Show("Unable to create database at " +
                      strFile +
                      ". Reason:  " +
                      exSQL.Errors[0].Message );
   }
}
```

Although the `Engine` class has a `CreateDatabase` method, it does not have a `DropDatabase` method. You must drop a database by submitting a `DROP DATABASE` statement to the SQL Server CE database or by using the `File` class to delete the file.

6.3.5 Populating a SQL Server CE Database

Now that we have created our SQL Server CE database, we can populate it with tables, indexes, and data. You might think that we use the `Engine` class to do this, but we do not. The `Engine` class operates only on an entire database, not on the individual components within the database, such as tables. In Chapter 8 we look at populating a SQL Server CE database from a connected SQL Server database, but in this chapter we focus on creating and maintaining the database. Therefore, to populate our SQL Server CE database, we use standard SQL DDL statements, such as `CREATE TABLE` and `CREATE INDEX`, and the standard SQL DML language statements `INSERT`, `UPDATE`, and `DELETE`.

Submitting SQL statements to a database requires two classes: one to open the connection and one to submit the statement.

6.3.5.1 *The* `SqlCeConnection` *and* `SqlCeCommand` *Classes*

The `SqlCeConnection` class opens a connection to a database and thus needs the name of the database file. The `SqlCeCommand` class submits one SQL statement at a time to the database using an `Execute` method and needs to know the connection object to use and the SQL statement to be submitted. The `SqlCeConnection` object and the DML statement are properties of the `SqlCeCommand` object. They must be set and the connection must be opened before the command's `Execute` method can be called. There are three possible `Execute` methods, summarized in Table 6.7.

Listing 6.4 shows the code for creating two simple tables.

Download at WoweBook.Com

TABLE 6.7: The `Execute` Methods of the `SqlCeCommand` Class

Method	Function
`ExecuteNonQuery`	Executes a SQL statement that returns no rows, such as `INSERT` or `CREATE`
`ExecuteScalar`	Executes a SQL statement that returns just one value, such as `SELECT SUM(Value) FROM Orders WHERE CustomerID = "ABCD"`
`ExecuteReader`	Executes a SQL statement that returns multiple columns or multiple rows

LISTING 6.4: Creating Two Tables

```
private void mitemCreateTables_Click(object sender, EventArgs e)
{
   SqlCeConnection  connDB = new SqlCeConnection();
   SqlCeCommand  cmndDB = new SqlCeCommand();

   connDB.ConnectionString = strConn;
   connDB.Open();

   cmndDB.Connection = connDB;

   cmndDB.CommandText =
      " CREATE TABLE Categories " +
      "  ( CategoryID integer not null " +
      "          CONSTRAINT PKCategories PRIMARY KEY " +
      "  , CategoryName nchar(20) not null " +
      "  )";
   cmndDB.ExecuteNonQuery();

   cmndDB.CommandText =
      " CREATE TABLE Products " +
      "  ( ProductID integer not null " +
      "          CONSTRAINT PKProducts PRIMARY KEY " +
      "  , ProductName nchar(20) not null " +
      "  , CategoryID integer not null " +
      "  , CONSTRAINT FKProdCat " +
      "       foreign key (CategoryID) " +
      "       references Categories(CategoryID) " +
      "  )";
   cmndDB.ExecuteNonQuery();

   connDB.Close();
}
```

Download at WoweBook.Com

SQL Server CE automatically creates indexes for the three constraints specified in Listing 6.4. You need to submit `CREATE INDEX` statements only for indexes that are not covered by constraints.

Listing 6.5 shows the code for inserting rows into the tables.

Listing 6.5: Adding Rows to Tables

```
private void mitemLoadData_Click(object sender, EventArgs e)
{
   SqlCeConnection  connDB = new SqlCeConnection();
   SqlCeCommand  cmndDB = new SqlCeCommand();

   connDB.ConnectionString = strConn;
   connDB.Open();

   cmndDB.Connection = connDB;
   cmndDB.CommandText =
      " INSERT Categories " +
      "   (CategoryID, CategoryName)" +
      "   VALUES (1, 'Franistans' )";
   cmndDB.ExecuteNonQuery();
   cmndDB.CommandText =
      " INSERT Categories " +
      "   (CategoryID, CategoryName)" +
      "   VALUES (2, 'Widgets' )";
   cmndDB.ExecuteNonQuery();

   cmndDB.CommandText =
      " INSERT Products " +
      "   (ProductID, ProductName, CategoryID)" +
      "   VALUES (11, 'Franistans - Large', 1 )";
   cmndDB.ExecuteNonQuery();
   cmndDB.CommandText =
      " INSERT Products " +
      "   (ProductID, ProductName, CategoryID)" +
      "   VALUES (12, 'Franistans - Medium', 1 )";
   cmndDB.ExecuteNonQuery();
   cmndDB.CommandText =
      " INSERT Products " +
      "   (ProductID, ProductName, CategoryID)" +
      "   VALUES (13, 'Franistans - Small', 1 )";
   cmndDB.ExecuteNonQuery();
   cmndDB.CommandText =
      " INSERT Products " +
      "   (ProductID, ProductName, CategoryID)" +
      "   VALUES (21, 'Widgets - Large', 2 )";
   cmndDB.ExecuteNonQuery();
   cmndDB.CommandText =
      " INSERT Products " +
```

Download at WoweBook.Com


```
      "   (ProductID, ProductName, CategoryID)" +
      "   VALUES (22, 'Widgets - Medium', 2 )";
   cmndDB.ExecuteNonQuery();
   cmndDB.CommandText =
      " INSERT Products " +
      "   (ProductID, ProductName, CategoryID)" +
      "   VALUES (23, 'Widgets - Small', 2 )";
   cmndDB.ExecuteNonQuery();

   connDB.Close();
}
```

Because no data is returned by the SQL statements in Listing 6.5, we use the `ExecuteNonQuery` method.

So, two classes, a few methods and properties, some standard SQL, and we have a populated database. Granted, it is a very simple database, but we would use the same classes, properties, and methods to produce a more complex one.

6.3.6 Retrieving and Displaying Data

Now that we have populated our SQL Server CE database with some data, we'll look at how to display that data to the user. We start with the two-tier approach, which uses the `SqlCeDataReader` class, and then move on to the three-tiered approach based on the `DataSet` class.

6.3.6.1 *The* `SqlCeDataReader` *Class*

The `SqlCeDataReader` class allows your program to access the rows returned by a SQL query. It provides read-only, forward-only access to one row at a time. Once positioned at a row, you use methods of the `DataReader` class to access the information located in the individual fields of that row. Many of these methods tend to be very data-specific, and those whose names begin with the word `Get` treat the fields as a zero-based array of data. Thus, if the underlying `SELECT` statement were `SELECT ProductID, ProductName FROM Products`, the call to retrieve the `ProductID` field (field 0) from the `DataReader` object's current row would be one of the following lines of code:

```
intVar = drdrDataReader.GetInt32(0);

//  or the slightly less efficient

intVar = drdrDataReader.GetValue(0);
```

Download at WoweBook.Com


```
//  or

intVar = drdrDataReader.Item["ProductID"];
```

Similarly, the call to retrieve the `ProductName` field would be one of the following lines of code:

```
strVar = drdrDataReader.GetString(1);

//  or

strVar = drdrDataReader.GetValue(1);

//  or

strVar = drdrDataReader.Item["ProductName"];
```

You cannot create a `DataReader` object directly; you must use the `SqlCeCommand` object's `ExecuteReader` method to create the `DataReader` object. When first created, the `DataReader` object is open but not yet positioned at a row. You have two ways to position it to a row.

The first is to use the `DataReader` object's `Read` method to move sequentially through the rows. The second is to use the `DataReader` object's `Seek` method to position to a specific row. The `Seek` method requires the following conditions.

- The underlying database table is indexed.
- Additional properties of the `SqlCeCommand` object are set before executing the `ExecuteReader` method.
- The index name and key value(s) are specified in the call to the `Seek` method.
- The call to the `Seek` method is followed by a call to the `Read` method.

The advantage of using the `Seek` method is performance. In fact, performance is the very reason why the .NET Compact Framework development team added the `Seek` method to the product beginning with Version 1.1. Using the `Seek` method is almost always faster than using the `WHERE` clause of a `SELECT` statement for locating specific rows. The disadvantage is that the `Seek` method provides access to only one table at a time, and knowledge of the index is required. The former means you cannot combine the `Seek` method with the joining of two or more tables; the latter goes against

Download at WoweBook.Com

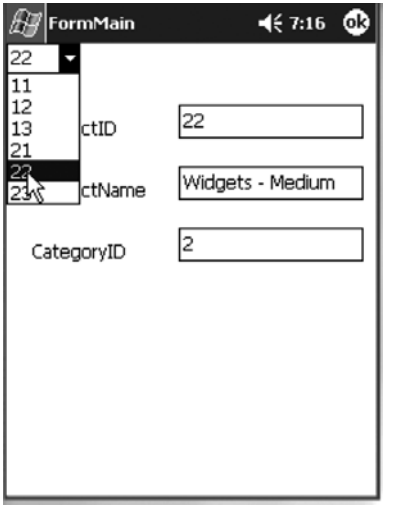


FIGURE 6.12: The `DataReader` Program

the very grain of relational database theory. Of course, many programmers have gone against the grain when it was the faster way to go.

To illustrate both the `Read` and `Seek` methods, the following sample application, shown in Figure 6.12 and located in the `DataReader` project on the Web site, loads a `ComboBox` control with product key values. When the user clicks on the `ComboBox` control, the fields of the selected product are displayed in the `TextBox` controls.

Listing 6.6 shows the code for this application.

LISTING 6.6: Source Code for the `DataReader` Program

```
using System;
using System.IO;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.Data.Common;
using System.Data.SqlServerCe;

namespace DataReader
{
   /// <summary>
   /// Summary description for FormMain
   /// </summary>
   public class FormMain : System.Windows.Forms.Form
   {
     internal System.Windows.Forms.Label lblCategoryName;
```

Download at WoweBook.Com


```
internal System.Windows.Forms.Label lblProductName;
internal System.Windows.Forms.Label lblProductID;
internal System.Windows.Forms.TextBox textCategoryName;
internal System.Windows.Forms.TextBox textProductName;
internal System.Windows.Forms.TextBox textProductID;
internal System.Windows.Forms.ComboBox comboKeys;

public FormMain()
  {
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
 }
 /// <summary>
 /// Clean up any resources being used.
 /// </summary>
 protected override void Dispose( bool disposing )
 {
    base.Dispose( disposing );
 }
 #region Windows Form Designer generated code


 /// <summary>
 /// The main entry point for the application
 /// </summary>

 static void Main()
 {
    Application.Run(new FormMain());
 }


 //  File path and name
 private string  strFile = @"My Documents\ourProduceCo.sdf";

 //  Connection string
 private string  strConn = "Data Source=" +
                           @"My Documents\ourProduceCo.sdf";

 //  Select product keys.
 private string strGetProductIDs =
    " SELECT ProductID " +      "   FROM Products ";

 //  Select one product, joined with its category.
 private string strGetOneProduct =
    " SELECT ProductID, ProductName, CategoryName " +
    "   FROM Products P " +
    "   JOIN Categories C on C.CategoryID = P.CategoryID " +
```

Download at WoweBook.Com


```
      "  WHERE P.ProductID = ";

   //  Used to bypass the SelectIndexChanged event
   //     during the loading of the ComboBox
   private bool boolLoading = true;


   private void FormMain_Load(object sender, EventArgs e)
   {
      //  Display a close box.
      this.MinimizeBox = false;

      //  Let the form present itself.
      Application.DoEvents();

      //  Ensure that the database exists.
      if (! File.Exists(strFile) )
      {
         MessageBox.Show(
            "Database not found.  Run the CreateDatabase " +
            "program for this chapter first.  Then run " +
            "this program.");
      }

      //  Load product keys into the ComboBox
      //     and select the first one.
      LoadProductIDs();
      comboKeys.SelectedIndex = 0;
   }

   private void comboKeys_SelectedIndexChanged(object sender,
                                               EventArgs e)
   {
      //  A product key has been selected; retrieve
      //     and display the corresponding product.
      if (! boolLoading )
      {
         LoadOneProduct((int)comboKeys.SelectedItem);
      }
   }

   private void textProductName_Validated(object sender,
                                          EventArgs e)
   {
      //  Update this product row in the database.
      UpdateSelectedRow(int.Parse(textProductID.Text),
                        textProductName.Text);
   }

   private void LoadProductIDs()
```

Download at WoweBook.Com


```
{
   //  Clear the ComboBox.
   comboKeys.Items.Clear();

   //  A connection, a command, and a reader
   SqlCeConnection connDB =
      new SqlCeConnection(strConn);
   SqlCeCommand cmndDB =
      new SqlCeCommand(strGetProductIDs, connDB);
   SqlCeDataReader drdrDB;

   //  Open the connection.
   connDB.Open();

   //  Submit the SQL statement and receive
   //     the SqlCeReader for the results set.
   drdrDB = cmndDB.ExecuteReader();

   //  Read each row.  Add the contents of its
   //     only column as an entry in the ComboBox.
   //     Close the reader when done.
   while ( drdrDB.Read() )
   {
      comboKeys.Items.Add(drdrDB["ProductID"]);
   }
   drdrDB.Close();

   //  Close the connection.
   connDB.Close();

   //  Start responding to ComboBox's
   //     SelectedIndexChanged events.
   this.boolLoading = false;
}

private void LoadOneProduct( int intProductID)
{
   //  A connection, a command, and a reader
   SqlCeConnection  connDB = new SqlCeConnection(strConn);
   SqlCommand  cmndDB = new SqlCommand(strSQL,connDB);
   SqlCeDataReader drdrDB;

   //  Open the connection.
   connDB.Open();

   //  Set the Command object to retrieve
   //     rows from one table via one index.
   //  Then retrieve the reader.
   cmndDB.Connection = connDB;
   cmndDB.CommandType = CommandType.TableDirect;
```

Download at WoweBook.Com


```
   cmndDB.CommandText = "Products";
   cmndDB.IndexName = "PKProducts";
   drdrDB = cmndDB.ExecuteReader();

   //  Retrieve the first (only) row from the
   //     Products table that has the ProductID
   //     that was selected in the ComboBox.
   //  Load the fields into the form's controls.
   //  Close the reader.
   drdrDB.Seek(DbSeekOptions.FirstEqual, intProductID);
      if( drdrDB.Read() )
      {
         LoadControlsFromRow(drdrDB);
      }
   drdrDB.Close();

   //  Close the connection.
   connDB.Close();
}


private void LoadControlsFromRow(  SqlCeDataReader drdrDB)
{
   //  Transfer the column titles and the field
   //     contents of the current row from the
   //     reader to the form's controls.
   lblProductID.Text = drdrDB.GetName(0);
   textProductID.Text = drdrDB.GetValue(0).ToString();
   lblProductName.Text = drdrDB.GetName(1);
   textProductName.Text = drdrDB.GetValue(1).ToString();
   lblCategoryName.Text = drdrDB.GetName(2);
   textCategoryName.Text = drdrDB.GetValue(2).ToString();
}

private void UpdateSelectedRow(int intProductID,
                               string strProductName)
{
   //  A connection and a command
   SqlCeConnection connDB = new SqlCeConnection(strConn);
   SqlCeCommand cmndDB = new SqlCeCommand();

   //  Open the connection.
   connDB.Open();

   //  Update the product name for the selected product.
   cmndDB.Connection = connDB;
   cmndDB.CommandText =
      " UPDATE Products " +
      " SET ProductName = " + "'" + strProductName + "'" +
      " WHERE ProductID = " + intProductID;
```

Download at WoweBook.Com


```
            cmndDB.ExecuteNonQuery();

            //  Close the connection.
            connDB.Close();
        }
    }
}
```

The `LoadProductIDs` routine submits a `SELECT` statement and uses a `DataReader` object to process all the rows retrieved by that `SELECT` statement. Note that the properties of the `SqlCeCommand` object must be initialized before a `DataReader` object can be created. Because the reader is not positioned to the first row when created, the `Read` method must be called at the start of the iterative loop, not at the end of the loop. In general, this makes for cleaner code and is a definite improvement over past ADO classes.

The `LoadOneProduct` routine uses the `Seek` method followed by the `Read` method to retrieve a single row from the database. Note also that a call to the `Read` method must follow the call to the `Seek` method. Without this call to the `Read` method, no row would be in the reader. Although there would be no row in the reader if the `Read` method call were missing, no exception would be raised. From ADO.NET's point of view, nothing has gone wrong; you simply have an empty reader.

As you can see from the output shown earlier in Figure 6.12, all the columns came from the `Products` table, a built-in limitation of the `Seek` method. Listing 6.7 shows an alternative version of the `LoadOneProduct` method. It uses a `WHERE` clause rather than a `Seek` method index to specify the desired product. As the output in Figure 6.13 shows, this alternative version of the `LoadOneProduct` method can gather fields from more than one table, resulting in the more informative `CategoryName` field associated with each product, rather than the less informative `CategoryID` field. However, as mentioned earlier, this version of the method executes more slowly.

Listing 6.7: An Alternative Version of the `LoadOneProduct` Method

```
private void LoadOneProduct( int intProductID)
{
   //  Append the desired ProductID to the SELECT statement.
    string  strSQL = strGetOneProduct + intProductID;

   //  A connection, a command, and a reader
   SqlCeConnection  connDB = new SqlCeConnection(strConn);
```

Download at WoweBook.Com


```
   SqlCommand  cmndDB = new SqlCommand(strSQL,connDB);
   SqlCeDataReader drdrDB;

   //  Open the connection.
   connDB.Open();

   //  Submit the SQL statement and receive
   //     the SqlCeReader for the one-row
   //     results set.
   drdrDB = cmndDB.ExecuteReader();

   //  Read the first (only) row.
   //     Display it.  Close the reader.
   if ( drdrDB.Read() )
   {
      LoadControlsFromRow(drdrDB);
   }
   drdrDB.Close();

   //  Close the connection.
   connDB.Close();
}
```

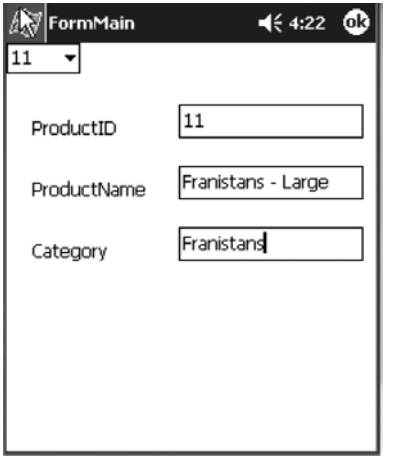


FIGURE 6.13: The Output from a Join Using the Alternative `LoadOneProduct` Method

We mentioned earlier that the `DataReader` object returned from the `ExecuteReader` method is not yet positioned to the first row. But it is already open, and it is your responsibility to close it. Unlike the desktop `DataReader` objects, which live in the multiuser environment, the `SqlCeDataReader` object was written for the mobile-device world. This means you can have

Download at WoweBook.Com

multiple readers open on the same connection at one time. This is a somewhat necessary capability, given that you are allowed only one open `SqlCeConnection` object per database at a time.

The current rows of a `DataReader` object and a `DataRow` object are not the same thing. A `DataRow` is an object unto itself, while the current row of a `DataReader` is not. As we saw, we access the contents of the current row of a `DataReader` object by using methods and properties of the `DataReader` itself.

6.3.7 Updating a SQL Server CE Database

Now that we have presented data to the user, we'll continue by allowing the user to modify that data and submit those modifications to the database. We already presented an example of this when we first populated our database. So, the code you are about to see is very similar to code you have already seen.

The application that allows the user to change the product name does so by reacting to the `textProductName.Validated` event as shown here:

```
private void textProductName_Validated(object sender,
EventArgs e)
{
   //  Update this product row in the database.
   UpdateSelectedRow(int.Parse(textProductID.Text), textProductName.Text);
}
```

The called routine updates the product row as shown in Listing 6.8.

Listing 6.8: The `UpdateSelectedRow` Method

```
private void UpdateSelectedRow(int intProductID,
                               string strProductName)
{
   //  A connection and a command
   SqlCeConnection connDB = new SqlCeConnection(strConn);
   SqlCeCommand cmndDB = new SqlCeCommand();
   //  Open the connection.
   connDB.Open();

   //  Update the product name for the selected product.
   cmndDB.Connection = connDB;
   cmndDB.CommandText =
      " UPDATE Products " +
      " SET ProductName = " + "'" + strProductName + "'" +
      " WHERE ProductID = " + intProductID;
   cmndDB.ExecuteNonQuery();
```

Download at WoweBook.Com


```
    //  Close the connection.
    connDB.Close();
}
```

If the application allowed more than one field to be updated, we would set a "needs updating" flag within the `Validated` event handler of each field and react to the `ComboBox` control's `SelectedIndexChanged` event and the form's `Close` event by doing the update at that point, if needed.

If this application is the only application accessing a database, it does not need to open and close the connection each time a query is submitted to the database. In that situation, we could move the `Open` and `Close` calls into the form's `Load` and `Closed` events, respectively.

You have now seen how to use the `SqlCeConnection`, `SqlCeCommand`, and `SqlCeDataReader` classes to retrieve data from a SQL Server CE database, present that data to the user in controls on the form, accept changes from the user, and submit those changes back to the database. And that, in turn, completes our coverage of SQL Server CE in the two-tiered application.

6.3.8 The `SqlCeDataAdapter` Class

It is time to address the three-tiered approach, the approach that has you move data between your SQL Server CE database and a data set. You cannot do that without using and understanding the `SqlCeDataAdapter` class, the primary means for moving data between a database and a data set.

6.3.8.1 *Using a* `SqlCeDataAdapter` *Object to Retrieve Data*

Just as the `SqlCeCommand` class has a property that holds the `SqlCeConnection` object to be used, the `SqlCeDataAdapter` class has a `SelectCommand` property that holds the `SqlCeCommand` object to be used to retrieve the data. The `SqlCeDataAdapter` class also holds three other command properties, all optional, for inserting database records (`InsertCommand`), updating database records (`UpdateCommand`), and deleting database records (`DeleteCommand`).

It also has the `Fill` method, which is used to connect to the database, execute the `SELECT` command, create the data table if necessary, and place the selected rows into the data table. Thus, the following code creates a data table named `Categories` in a previously created data set named `dsetDB`, connects to the `ourProduceCo` database, submits the `SELECT` statement, and places the returned rows into the `Categories` table:

Download at WoweBook.Com


```
SqlCeDataAdapter daptDB =
      new SqlCeDataAdapter(
         "SELECT * FROM Categories",
         "Data Source=My Documents\ourProduceCo.sdf");
daptDB.Fill(dsetDB, "Categories");
```

When you look at the code, it seems that only one object, the `SqlCeDataAdapter` object, is being created, for there is only one `new` function call. Actually, four objects are created; one is then disposed, and the remaining three are kept. The overloaded constructor shown here created a connection object and a command object from the parameters we provided, as well as creating the data adapter object. A reference to that command object is now stored in the adapter's `SelectCommand` property, and a reference to the connection object is now stored in the command object's `Connection` property. A fourth object, a data reader object, was created, was used to retrieve the rows, and was then closed and disposed.

The second parameter of the `Fill` method is the name to be given to the new data table in the data set. This name is completely unrelated to any table name in the underlying database. It may be the same name as a database table name, but no inherent relationship is assumed based on the choice of name.

The `Fill` method leaves the connection object in the state in which the method found the object. If the connection is open at the time the method is called, the connection is left open. If the connection is closed at the time the method is called, the `Fill` method opens the connection, retrieves the rows, places them into the data table, and closes the connection.

The `Fill` method creates a new data table if a data table of the same name does not already exist. If the data table does exist, primary key values are used to determine whether each incoming row should create a new row or update an existing row.

At this point it's reasonable to ask, "Why is a separate object needed to move the data between the database and the data set? Why can't the data table just 'pull' the rows in, similar to the way an ADO `Recordset` object pulls data into itself by using its `Open` method?"

A data table cannot load itself because it is provider-independent. That is, the `DataSet` and `DataTable` classes have no knowledge of SQL Server, Pocket Access, or SQL Server CE. They do not know how to perform such tasks as translating the underlying data types into .NET data types or obtaining schema definitions from the underlying data source. We need a

Download at WoweBook.Com

provider-specific class to act as the middleware. When the database is SQL Server CE, `SqlCeDataAdapter` is that class.

Although not mandatory, it is a good practice to have a separate instance of `SqlCeDataAdapter` for each data table instance. If your application uses the `SqlCeDataAdapter` class to update the database, having a separate `SqlCeDataAdapter` object for each table is highly recommended.

6.3.8.2 *Using a* `SqlCeDataAdapter` *Object to Update a Database*

Remember that the `SqlCeDataAdapter` object lies between the `DataSet` object and the underlying database. The `SqlCeDataAdapter` object is used to pull the data from the database and place it into the data table, and it can be used to push any changes made to the data in the data table back to the database. Just as a `SqlCeDataAdapter` object has a `SelectCommand` property, it also has `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties, each holding a `SqlCeCommand` object. These commands are used to push changes made to the data tables back to the database. They are called by the `SqlCeDataAdapter` object's `Update` method. You can call them directly, but normally you let the `Update` method do it for you.

Coming up with the parameterized SQL statements for these command objects may be very simple or quite difficult. It all depends on the complexity of the `SELECT` statement in the `SelectCommand` property. The `SqlCeDataAdapter` object can build the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` command objects for you if you give it some help (which is provided by the `SqlCeCommandBuilder` class) and if all of the following conditions are true.

- The `SelectCommand` property's `SELECT` statement accesses just one database table.
- The table being accessed has a primary key or unique constraint.
- The columns that make up that constraint are in the `SELECT` clause of the `SELECT` statement.
- Each data element in the output of the `SELECT` statement can be traced back to a single data element in the database (no literals, aggregates, or functions).

If your `SELECT` statement meets these criteria, and if you have created a `SqlCeCommandBuilder` object and associated it with the `SqlCeDataAdapter` object prior to calling the `SqlCeDataAdapter` object's `Update` method, the

Download at WoweBook.Com

SqlCeCommandBuilder object builds the Insert, Update, and Delete command objects for you.

To illustrate this, we add a simple form to our project for updating category information, as shown in Figure 6.14.

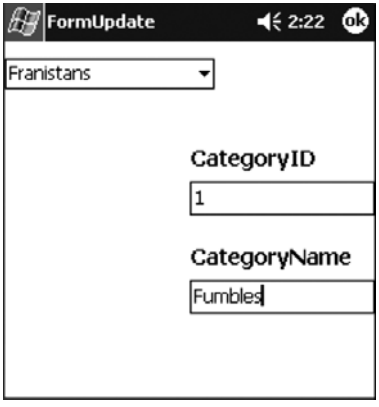


FIGURE 6.14: The Category Modification Form

Listing 6.9 shows the code for this form. Most of it is familiar, consisting of creating and loading the data set and binding to the controls. New concepts in the code are the creation of the SqlCeCommandBuilder object in the Load event handler and the update-specific code in the Closing event handler.

LISTING 6.9: The Category Modification Form Class

```
using System;
using System.Data;
using System.Data.Common;
using System.Data.SqlServerCe;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace CreateDatabase
{
   /// <summary>
   /// Summary description for FormUpdate
   /// </summary>
   public class FormUpdate : System.Windows.Forms.Form
   {
      internal System.Windows.Forms.Panel panelCategory;
```

Download at WoweBook.Com


```
internal System.Windows.Forms.ComboBox comboCategoryIDs;
internal System.Windows.Forms.TextBox textCategoryName;
internal System.Windows.Forms.Label lblCategoryName;
internal System.Windows.Forms.TextBox textCategoryID;
internal System.Windows.Forms.Label lblCategoryID;

public FormUpdate()
{
   InitializeComponent();
}

protected override void Dispose( bool disposing )
{
   base.Dispose( disposing );
}

#region Windows Form Designer generated code

//  Connection string
private string  strConn =
   "Data Source=" + @"My Documents\ourProduceCo.sdf";

//  The data set, adapter, table
private DataSet dsetDB;
private SqlCeDataAdapter daptCategories;
private DataTable dtabCategories;

private bool boolLoading = true;


private void FormUpdate_Load(object sender, EventArgs e)
   //  Present a close box.
{
   this.MinimizeBox = false;

   //  Create the data set.
   dsetDB = new DataSet("Produce");

   //  Create the data adapter.
   daptCategories = new
      SqlCeDataAdapter("SELECT CategoryID, CategoryName " +
                       "  FROM Categories",
      strConn);

   //  Create the command builder for the adapter.
   SqlCeCommandBuilder cbldCategories =
      new SqlCeCommandBuilder(daptCategories);

   //  Create and fill the data table, and
   //     save a reference to it.
```

Download at WoweBook.Com


```
            daptCategories.Fill(dsetDB, "Categories");
            dtabCategories = dsetDB.Tables["Categories"];

            //  Bind the combo box with the category names.
            comboCategoryIDs.DataSource = dtabCategories;
            comboCategoryIDs.DisplayMember = "CategoryName";
            comboCategoryIDs.ValueMember = "CategoryID";
            comboCategoryIDs.SelectedIndex = 0;

            //  Load labels with data table column names.
            lblCategoryID.Text = "CategoryID";
            lblCategoryName.Text = "CategoryName";

            //  Bind the data table's columns to the text boxes.
            textCategoryID.DataBindings.Add
               ("Text", dtabCategories, "CategoryID");
            textCategoryName.DataBindings.Add
               ("Text", dtabCategories, "CategoryName");

            //  Give the panel some tone.
            panelCategory.BackColor = Color.Beige;
            //  Loading is finished.
            boolLoading = false;
            comboCategoryIDs.SelectedIndex = 0;
         }

         private void textCategoryName_Validated(object sender,
                                                 EventArgs e)
         {
            //  Force the current modification to complete.
            this.BindingContext[dtabCategories].EndCurrentEdit();
         }

         private void FormUpdate_Closing(object sender,
                                         CancelEventArgs e)
         {
            //  Force the current modification to complete.
            this.BindingContext[dtabCategories].EndCurrentEdit();

            //  Push data set changes back to the database.
            daptCategories.Update(dsetDB, "Categories");
         }
      }
   }
```

Regardless of whether you are letting the `SqlCeCommandBuilder` object build the adapter's three modification commands or you are building them programmatically yourself, calling the `SqlCeDataAdapter` object's

Download at WoweBook.Com

Update method pushes changes that have been made in the data set back to the database. The Update method does this by examining the row status of each row of the data table and then executing the Insert, Update, or Delete command as necessary.

As we mentioned earlier, never call the DataTable object's AcceptChanges method just prior to calling the SqlCeDataAdapter object's Update method. The AcceptChanges method sets all row statuses to Unchanged, which means the Update method would find no rows to update.

Unlike the adapter class for use with SQL Server, the SqlCeDataAdapter object does not detect conflicts when updating a database. Conflict detection prevents an application from modifying data that has been modified by some other application since the first application read the data. This ensures users that they are updating data whose values are still the values being presented to the user. In a multiuser environment, conflict detection is a very beneficial guardian; in the single-user world, it is a lot of overhead. If the user runs two applications to update the same data at the same time, the SqlCeDataAdapter object quite rightly lets the user trash his or her own updates if the user chooses to do so.

Even though you use an adapter to bring data into a data set, using it to push changes back to the database is always optional. You can always write your own DML statements and place them into the CommandText properties of the adapter's command objects. As we mentioned, if the SELECT statement is sufficiently complex, the SqlCeCommandBuilder object cannot generate the Insert, Update, and Delete statements; you have to write them yourself.

You might want to write those statements yourself for additional control or application documentation reasons. Your application can always create a connection and a command object and use the command object's ExecuteNonQuery method to submit an update to a database. And for reasons we will explain in the upcoming Microsoft SQL Server section, you are more likely to provide your own updating when connected to SQL Server rather than to SQL Server CE. But first we examine one last aspect of SQL Server CE programming: obtaining information about a database's schema.

6.3.9 Querying Schema Information

In an environment where data is being collected from a variety of sources and stored in your data tables, knowing the schema of that data can be

Download at WoweBook.Com

very important. Historically, database engines have always provided a method for querying information about a database schema, such as the names of the tables, the names and data types of the columns, and the names and included columns of the primary keys and foreign keys.

The database engines all did this differently, mainly because their internal storage mechanisms for holding this information varied from engine to engine. To improve cross-engine compatibility, the ANSI Standards Committee introduced a standard syntax for querying schema information, known as the `INFORMATION_SCHEMA` views. For instance, using this syntax you can query the database for a list of table names by submitting the following query:

```
SELECT TABLE_NAME
  FROM Information_Schema.TABLES
 WHERE TABLE_TYPE = 'TABLE'
```

SQL Server CE schema querying capability differs significantly from that of SQL Server and from the ANSI standard. This is necessary to maintain the smaller footprint of SQL Server CE. Many of the ANSI standard tables related to constraints have been removed and replaced by a single table that is more appropriate for a single-user database: the `MSysConstraints` table. For instance, the following ANSI standard query obtains foreign key information when submitted to SQL Server; but it yields a syntax error when submitted to SQL Server CE:

```
--***
--***  SQL Server version
--***
SELECT C.CONSTRAINT_NAME
     , PK.TABLE_NAME
     , PK.COLUMN_NAME
     , FK.TABLE_NAME
     , FK.COLUMN_NAME
  FROM Information_Schema.REFERENTIAL_CONSTRAINTS C
  JOIN Information_Schema.CONSTRAINT_COLUMN_USAGE PK
    ON PK.CONSTRAINT_NAME = C.UNIQUE_CONSTRAINT_NAME
  JOIN Information_Schema.CONSTRAINT_COLUMN_USAGE FK
    ON FK.CONSTRAINT_NAME = C.CONSTRAINT_NAME
```

Instead, the following query must be used with SQL Server CE to obtain foreign key information:

```
--***
--***  SQL Server CE version
```

Download at WoweBook.Com


```
--***
SELECT C.CONSTRAINT_NAME
     , C.REFERENCED_TABLE_NAME
     , C.REFERENCED_COLUMN_NAME
     , C.TABLE_NAME
     , C.CONSTRAINT_COLUMN_NAME
  FROM MSysConstraints C
  WHERE C.CONSTRAINT_TYPE = 1
```

In the world of SQL Server CE, schema queries are very useful tools. As we discuss in Chapter 8, the data in your SQL Server CE database may have been imported from a connected database such as SQL Server. You may need to ascertain schema information after the data has been received. Additionally, as you move data from your SQL Server CE database into a data set, you may wish to reproduce the database schema, such as constraints and relationships, into the data set.

As an example of querying schema information, the `UtilData` class located in the `UtilData` project on the book's Web site has a shared method, `BuildDataSet`, which does the following:

- Creates a data set
- Finds all the tables in a specified SQL Server CE database
- Uploads them to the data set
- Finds all the constraints in the specified SQL Server CE database
- Creates the corresponding relations and constraints in the data set
- Returns the data set

Listing 6.10 presents the code.

Listing 6.10: Building a Data Set from a Database

```
public static DataSet BuildDataSet( string  strDB )
{
   DataSet dsetWork;

   if (! File.Exists(strDB) ) { return(null) ; }

   string  strConn = "Data Source=" + strDB;
   SqlCeConnection connDB;
   connDB = new SqlCeConnection(strConn);
   connDB.Open();

   dsetWork = new DataSet("ourProduceCo");
```

Download at WoweBook.Com


```
   SqlCeDataReader drdrTableNames;
   drdrTableNames = GetTableNames(connDB);
   while ( drdrTableNames.Read() )
   {
      LoadTable(dsetWork,
                connDB,
                drdrTableNames.GetString(0));
   }
   drdrTableNames.Close();

   LoadRelationships(dsetWork, connDB);

   connDB.Close();

   return dsetWork;
}


// ************************************************//
//
//     GENERIC ROUTINES
//

public static SqlCeDataReader GetTableNames
                                ( SqlCeConnection connDB )
{
      SqlCeCommand  cmndDB =
         new SqlCeCommand(strGetTableNames, connDB);
      return cmndDB.ExecuteReader();
}

public static void LoadTable (DataSet dsetDB,
                              SqlCeConnection connDB,
                              string  strTable)
{
   SqlCeCommand cmndDB =
      new SqlCeCommand(strGetTableRows + strTable, connDB);
   SqlCeDataAdapter daptProducts =
      new SqlCeDataAdapter(cmndDB);
   daptProducts.Fill(dsetDB, strTable);
}

public static SqlCeDataReader GetRelationships
                                ( SqlCeConnection connDB )
{
   SqlCeCommand cmndDB =
      new SqlCeCommand(strGetRelationships, connDB);
   return cmndDB.ExecuteReader();
```

Download at WoweBook.Com


```
}

public static void LoadRelationships(
                                DataSet dsetDB,
                                SqlCeConnection connDB)
{
//  Retrieve foreign key information from the
//     database.  For each foreign key, create
//     a relationship in the data set.

//  Create the GetRelationships command object.
SqlCeCommand cmndDB =
   new SqlCeCommand(strGetRelationships, connDB);

//  Execute GetRelationships.
SqlCeDataReader drdrRelationships;
drdrRelationships = cmndDB.ExecuteReader();
string  strRelation;
DataColumn dcolParent, dcolChild;

while ( drdrRelationships.Read() )
{
   //  For each foreign key in the database
   //  Extract and convert name, parent, child info.
   strRelation =
      drdrRelationships.GetString(0);
   dcolParent =
      dsetDB.Tables[drdrRelationships.GetString(1)]
            .Columns[drdrRelationships.GetString(2)];
   dcolChild =
      dsetDB.Tables[drdrRelationships.GetString(3)]
            .Columns[drdrRelationships.GetString(4)];

      //  Add the relation to the data set.
      dsetDB.Relations.Add
                  (strRelation, dcolParent, dcolChild);
   }
   drdrRelationships.Close();

   //  Make each relationship a nested relationship.
   foreach( DataRelation drelForXML in dsetDB.Relations )
   {
      drelForXML.Nested = true;
   }
}
```

Download at WoweBook.Com

NOTE

The `SqlServerCe` namespace contains the classes for accessing a Microsoft SQL Server CE database. You need a connection and a command object, plus either a `SqlCeDataReader` or a `SqlCeDataAdapter` object. The data reader lets you retrieve rows into your application one at a time, forward only. The data set is a memory-resident database that allows for programmatic access to data from a SQL Server CE database and from other sources. It provides data binding and automatic updating.

Interestingly, all of these points are as true for Microsoft SQL Server as they are for Microsoft SQL Server CE. But programming for one is not exactly the same as programming for the other, as we are about to see.

6.4 Microsoft SQL Server

If, as we said earlier, a variety of databases are available for mobile devices, an even greater variety of databases are available for desktop machines and servers. Of these, Microsoft SQL Server was designed with built-in support for .NET Compact Framework applications—it provides synchronization of data between a SQL Server database and a SQL Server CE database (the subject of Chapter 8). The .NET Compact Framework also provides applications with access to SQL Server databases through the provider-specific classes that run on Windows CE devices. Just as there are `SqlCeConnection`, `SqlCeCommand`, and other classes in the .NET Compact Framework for accessing SQL Server CE, there are also `SqlConnection`, `SqlCommand`, and other classes available for accessing SQL Server.

The Microsoft SQL Server provider-specific ADO.NET classes are located in the `System.Data.SqlClient` namespace. Because the provider-specific classes adhere to a common specification, accessing SQL Server from your application is similar to accessing SQL Server CE: You open connections, submit commands, process returned rows with a data reader, and transfer data to and from a data set by using a data adapter. The text within the `ConnectionString` and `CommandText` properties may be more complex when accessing SQL Server, but the properties and methods of the classes are predominately the same.

Earlier in this chapter we introduced SQL Server CE. It is not our intent to present the same level of detail for SQL Server. SQL Server is a far more complex product, one that does not run on a device and one that has already been covered in innumerable books and other sources. Instead,

Download at WoweBook.Com

here we cover some of the differences between SQL Server CE and SQL Server that you might encounter while using the .NET Compact Framework's ADO.NET classes.

We begin by recapping some statements regarding the differences between SQL Server and SQL Server CE as they relate to accessing each from your application.

- Connecting to a SQL Server database is more complicated than connecting to a SQL Server CE database.
  - A network connection to the server must exist.
  - One server can host multiple SQL Servers.
  - One SQL Server can contain many databases.
  - SQL Server provides for detailed user access security.
- The syntax is more complete in SQL Server than in SQL Server CE.
  - Views, triggers, functions, and stored procedures are supported.
  - `GRANT`, `REVOKE`, and `DENY` statements are supported, along with system stored procedures for maintaining user access rights.
  - The ANSI standard `INFORMATION_SCHEMA` views are supported.
- Data integrity is more rigorous in SQL Server than in SQL Server CE.
  - Triggers and stored procedures provide for additional server-side data checking that cannot be overridden by the application.
  - User access permissions can be granted at the table and stored procedure levels.
- Stored procedures are supported in SQL Server but not in SQL Server CE.
  - Parameterized application functionality can be defined on the server and invoked from the client.
  - The `SQLCommand` class contains built-in support for stored procedures.
- Some `System.Data.SqlServerCe` classes, properties, methods, and events are not available in `System.Data.SqlClient`.
  - No `Engine` class exists. Instead, databases must be created by submitting DDL statements.
  - The data reader's `Seek` method is not supported.
- The relationship between a connection and a data reader is different.
  - SQL Server allows multiple open connections but only one open reader per open connection.
  - SQL Server CE allows only one open connection but multiple open readers per open connection.

Download at WoweBook.Com

- Concurrency is handled differently.
  - SQL Server provides concurrency protection.
  - SQL Server CE does not provide concurrency protection.
- More things can go wrong in SQL Server.
  - The need for a remote connection, the granularity of user access security, and the server-side data integrity capabilities all increase the possibility that your application's request can fail and an exception will be raised. (This is a client-side judgment. From SQL Server's point of view, access security and data integrity constraints don't cause things to go wrong, they prevent things from going wrong.)

Despite the major differences between the two products, accessing each from your application is remarkably similar. We hope this emerges from the upcoming discussion of the subject.

We begin with connecting to the database. We use the same sample applications we used when discussing SQL Server CE programming in the previous section of this chapter; we merely change them to access SQL Server's Northwind database, the sample database that is created whenever SQL Server is installed.

Once we have set an assembly reference to the `System.Data.SqlClient.dll` module and optionally added the matching `using` statement shown here, we are ready to create a connection object and open a connection to a SQL Server database:

```
using System.Data.SqlClient;
```

6.4.1 Connecting to SQL Server

We use essentially the same connection class to connect to a SQL Server database as we used to connect to a SQL Server CE database, although the connection classes reside in different namespaces and have slightly different names. (In SQL Server CE, we connect using the `SqlCeConnection`[4] class, whereas with SQL Server we use the `SqlConnection`[5] class.) But both classes adhere to the same Microsoft specification for provider-specific classes, so the PMEs are the same.

The code shown in Listing 6.11 is the code for the `DataReader` application that we originally wrote for SQL Server CE and have now modified

4. Fully qualified name: `System.Data.SqlServerCe.SqlCeConnection`.
5. Fully qualified name: `System.Data.SqlClient.SqlConnection`.

Download at WoweBook.Com

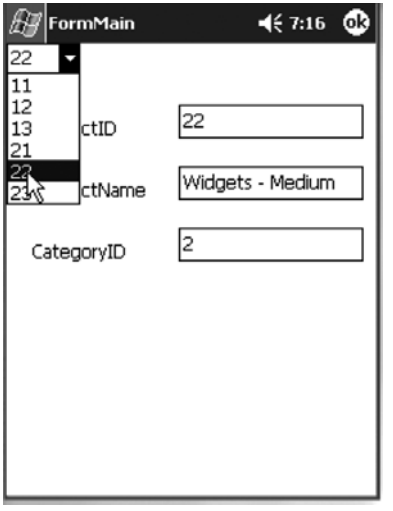


FIGURE 6.15: The `DataReader` Program (Repeated)

for SQL Server. We introduced the `DataReader` application in Figure 6.12 and Listing 6.6 earlier in this chapter. For convenience, in Figure 6.15 we again show the main screen of the `DataReader` program.

As shown in Listing 6.11, we modified the original code to access SQL Server. We did so by changing the statement in which the connection string is defined, by changing `SqlCe` to `Sql`, and by setting a reference to the SQL Server client namespace rather than to the SQL Server CE client namespace. Because the `Seek` method is not supported by the `SqlDataReader` class, we use the version of the `LoadOneProduct` method that we showed earlier in Listing 6.7.

LISTING 6.11: Using a `DataReader` Object with SQL Server

```
using System;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace DataReader
{
   /// <summary>
   /// Summary description for FormMain
   /// </summary>
   public class FormMain : System.Windows.Forms.Form
   {
      #region Assorted Generated Code
```

Download at WoweBook.Com


```
#region Windows Form Designer generated code


//  Connection string
private string  strConn =
   "data source=OurServer;" +
   "initial catalog=Northwind;" +
   "user id=DeliveryDriver;" +
   "pwd=DD;" +
   "workstation id=OurDevice;" +
   "packet size=4096;" +
   "persist security info=False;";

//  Select product keys.
private string strGetProductIDs =
   " SELECT ProductID FROM Products ";

//  Select one product, joined with its category.
private string strGetOneProduct =
   " SELECT ProductID, ProductName, CategoryName " +
   "   FROM Products P " +
   "   JOIN Categories C on C.CategoryID = P.CategoryID " +
   "  WHERE P.ProductID = ";

//  Used to bypass the SelectIndexChanged event
//     during the loading of the ComboBox
private bool boolLoading = true;


private void FormMain_Load(object sender, EventArgs e)
{
   //  Display a close box.
   this.MinimizeBox = false;

   //  Let the form present itself.
   Application.DoEvents();

   //  Load product keys into the ComboBox
   //     and select the first one.
   LoadProductIDs();
   comboKeys.SelectedIndex = 0;
}


private void comboKeys_SelectedIndexChanged(object sender,
                                            EventArgs e)
{
   //  A product key has been selected; retrieve
   //     and display the corresponding product.
```

Download at WoweBook.Com


```
   if (! boolLoading )
   {
      LoadOneProduct((int)comboKeys.SelectedItem);
   }
}


private void textProductName_Validated(object sender,
                                       EventArgs e)
{
   //  Update this product row in the database.
   UpdateSelectedRow(int.Parse(textProductID.Text),
                     textProductName.Text);
}


private void LoadProductIDs()
{
   //  Clear the ComboBox.
   comboKeys.Items.Clear();

   //  A connection, a command, and a reader
   SqlConnection connDB =
      new SqlConnection(strConn);
   SqlCommand cmndDB =
      new SqlCommand(strGetProductIDs, connDB);
   SqlDataReader drdrDB;

   try
   {
      //  Open the connection.
      connDB.Open();

      //  Submit the SQL statement and receive
      //     the SqlReader for the results set.
      drdrDB = cmndDB.ExecuteReader();

      //  Read each row.  Add the contents of its
      //     only column as an entry in the ComboBox.
      while ( drdrDB.Read() )
      {
         comboKeys.Items.Add(drdrDB["ProductID"]);
      }

      //  Close the reader.
      drdrDB.Close();
   }
   catch( SqlException exSQL )
   {
      foreach( SqlError errSQL in exSQL.Errors )
```

Download at WoweBook.Com


```
      {
         MessageBox.Show(errSQL.Message);
      }
   }
   finally
   {
      //  Close the connection.
      connDB.Close();

      //  Start responding to the ComboBox's
      //     SelectedIndexChanged events.
      this.boolLoading = false;
   }
}


private void LoadOneProduct( int intProductID )
{
   //  Append the param ProductID to the SELECT statement.
   string  strSQL = strGetOneProduct + intProductID;

   //  A connection, a command, and a reader
   SqlConnection  connDB = new SqlConnection(strConn);
   SqlCommand  cmndDB = new SqlCommand(strSQL,connDB);
   SqlDataReader drdrDB;

   //  Open the connection.
   connDB.Open();

   //  Submit the SQL statement and receive
   //     the SqlReader for the one-row
   //     results set.
   drdrDB = cmndDB.ExecuteReader();

   //  Read the first (only) row.
   //     Display it.  Close the reader.
   if ( drdrDB.Read() )
   {
      LoadControlsFromRow(drdrDB);
   }
   drdrDB.Close();

   //  Close the connection.
   connDB.Close();
}


private void LoadControlsFromRow( SqlDataReader drdrDB )
{
   //  Transfer the column titles and the field
```

Download at WoweBook.Com


```
         //     contents of the current row from the
         //     reader to the form's controls.
         lblProductID.Text = drdrDB.GetName(0);
         textProductID.Text = drdrDB.GetValue(0).ToString();
         lblProductName.Text = drdrDB.GetName(1);
         textProductName.Text = drdrDB.GetValue(1).ToString();
         lblCategoryName.Text = drdrDB.GetName(2);
         textCategoryName.Text = drdrDB.GetValue(2).ToString();
      }


      private void UpdateSelectedRow(int intProductID,
                                     string strProductName)
      {
         //  A connection and a command
         SqlConnection connDB = new SqlConnection(strConn);
         SqlCommand cmndDB = new SqlCommand();

         //  Open the connection.
         connDB.Open();

         //  Update the product name for the selected product.
         cmndDB.Connection = connDB;
         cmndDB.CommandText =
            " UPDATE Products " +
            " SET ProductName = " + "'" + strProductName + "'" +
            " WHERE ProductID = " + intProductID;
         cmndDB.ExecuteNonQuery();

         //  Close the connection.
         connDB.Close();
      }
   }
}
```

The one statement that we changed to effect the transition from SQL Server CE to SQL Server is shown here in its modified form:

```
//  Connection string
private string  strConn =
   "data source=OurServer;" +
   "initial catalog=Northwind;" +
   "user id=DeliveryDriver;" +
   "pwd=DD;" +
   "workstation id=OurDevice;" +
   "packet size=4096;" +
   "persist security info=False;";
```

Download at WoweBook.Com

As you can see from the connection string, a SQL Server connection is more complex than its SQL Server CE counterpart, which was:

```
private string  strConn = "Data Source=" +
                          @"My Documents\ourProduceCo.sdf";
```

In the new version, we specify the server name, database name, SQL Server username and password, and some other information. We did not code this ourselves—we got Visual Studio .NET to do it for us, and we recommend that you do the same.

To obtain this assistance, your development environment must be able to connect to your development server. Open a new Windows Application project rather than a Smart Device Application project. In a Windows Application project, the entries under the Toolbox's Data tab are enabled and you can drag and drop them onto the form. Figure 6.16 shows a connection object being dropped on the form.

When dropped, a connection object does not appear in the form itself but rather in the blank area beneath the form. Once the connection object is there, you can select it and access its properties. One of your choices in the connection string drop-down list is <New Connection…> (see Figure 6.17). Choosing this item produces the dialog box shown in Figure 6.18.

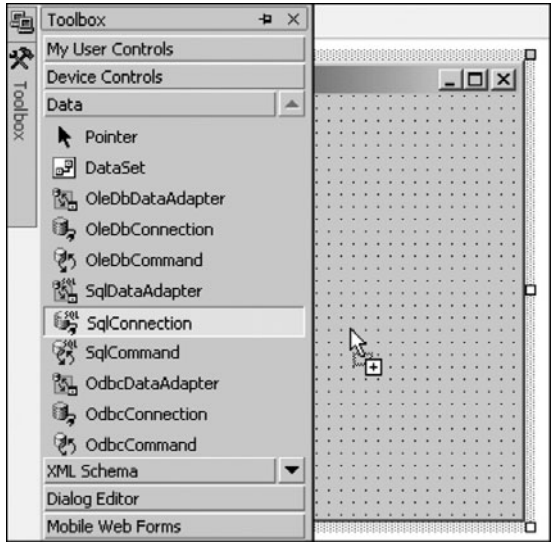


FIGURE 6.16: Dragging and Dropping a Connection Object

Download at WoweBook.Com

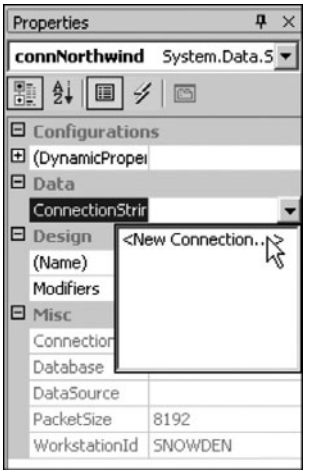


FIGURE 6.17: Requesting a New Connection String

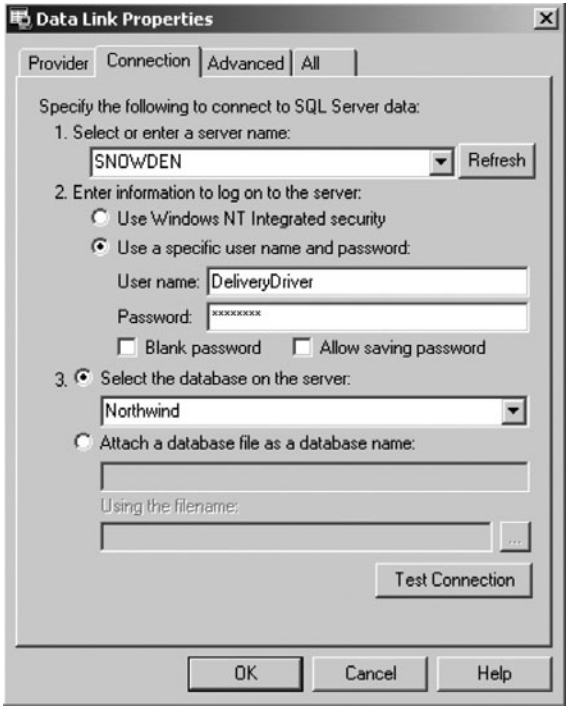


FIGURE 6.18: Specifying Connection Information

Download at WoweBook.Com

Once you enter your information and dismiss the dialog box, the generated connection string appears in the Properties window. From there, you can cut and paste the connection string into your .NET Compact Framework program.

6.4.2 Creating Command Objects

Once you have the connection object, you can move on to creating a command object. As you can see from the code, there is very little difference between the `SqlCommand` class and the `SqlCeCommand` class. Nor is there much difference between the two data reader classes. The only change we had to make in the program is caused by the absence of the `Seek` method in the `SqlDataReader` class.

Although there is very little difference between the two command classes, there can be a great difference in how you use them. This is because SQL Server has stored procedures and SQL Server CE does not, and access to the stored procedures is provided only through the `SqlCommand` class.

6.4.3 Using SQL Server Stored Procedures

A stored procedure is a procedure that is written in SQL, stored in a SQL Server database, and executable by a client program. It can have input and output parameters and returns an integer value. A stored procedure can be as simple as a single `SELECT` statement or as complex as a multitable update with parameter checking and error handling. Stored procedures encapsulate functionality on the server side.

One benefit of stored procedures is that they execute much more quickly than the same logic would in a submitted batch of SQL statements. With a stored procedure, the database engine performs query analysis and optimization and caches the compiled query optimization plan when the stored procedure is first created. On subsequent executions of the stored procedure, the database engine can bypass the analysis, optimization, and compilation. To execute a submitted batch of SQL statements, on the other hand, the entire process must be performed each time a batch is submitted.

Listing 6.12 shows a stored procedure that updates name and price information for a specified product. We use this stored procedure, named `procModifyProductInfo`, during this discussion. It is not the most efficient code ever written, but it is straightforward and readable, and it adheres to the SQL Server convention that successfully executed procedures should return `0`.

Download at WoweBook.Com

Listing 6.12: The procModifyProductInfo Stored Procedure

```
CREATE PROCEDURE procModifyProductInfo
   @ProductID int = null,
   @ProductName nvarchar(40) = null,
   @UnitPrice money = null
AS
BEGIN
   IF @ProductID is null
   BEGIN
      RAISERROR('Product ID not supplied.',10,1)
      RETURN 1
   END

   IF not exists (SELECT *
                    FROM Products
                   WHERE ProductID = @ProductID)
   BEGIN
      RAISERROR('Product ID not on file.',10,1)
      RETURN 1
   END

   BEGIN TRANSACTION

   IF @ProductName is not null
      UPDATE Products
         SET ProductName = @ProductName
         WHERE ProductID = @ProductID
   IF @@ERROR <> 0
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR('Unable to update Products table.',10,1)
      RETURN 1
   END

   IF @UnitPrice is not null
      UPDATE Products
         SET UnitPrice = @UnitPrice
         WHERE ProductID = @ProductID
   IF @@ERROR <> 0
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR('Unable to update Products table.',10,1)
      RETURN 1
   END

   COMMIT TRANSACTION
   RETURN 0
END
```

Download at WoweBook.Com

Client programs execute the stored procedure by submitting the following SQL statement, for example, to the SQL Server. The sample statement shown here changes the name and price of product 18 to "Carnavon Tigers" and $987.65, respectively.

```
EXEC procModifyProductInfo 18, 'Carnavon Tigers', 987.65
```

To submit this SQL statement to the server, your application could set the statement into the `CommandText` property of a command object and then call the `ExecuteNoResults` method. But there is a better way, one that makes it easier to repetitively reexecute the same procedure with new parameter values each time and to access the return value. In your application, perform the following steps.

1. Create the connection and command objects.
2. Set the command object's `CommandType` property to `CommandType.StoredProcedure`.
3. Set the command object's `CommandText` property to the name of the stored procedure.
4. To the command object's `Parameters` collection, add parameter objects for each parameter and for the return value.
5. Set the input value for each parameter.
6. Use the appropriate `Execute` method to cause the execution of the procedure at the server.
7. Repeat steps 5 and 6 for each execution of the store procedure.

Step 4 is the most difficult of these steps. Once again, if your development environment can connect to a development SQL Server, Visual Studio .NET can generate the code for you, with some help from you in the form of a few drag-and-drop operations, as follows.

Once you have created a stored procedure on the development server, open a Windows Application project in Visual Studio .NET and navigate the Server Explorer window to your development SQL Server. Expand the server until you have reached the stored procedure; then drag and drop the stored procedure onto the form, as shown in Figure 6.19.

The resulting command object is added below the form, where you can select it and view its properties. When viewing the properties, you may note that the command object has a `Parameters` collection that displays itself graphically as shown in Figure 6.20.

Download at WoweBook.Com

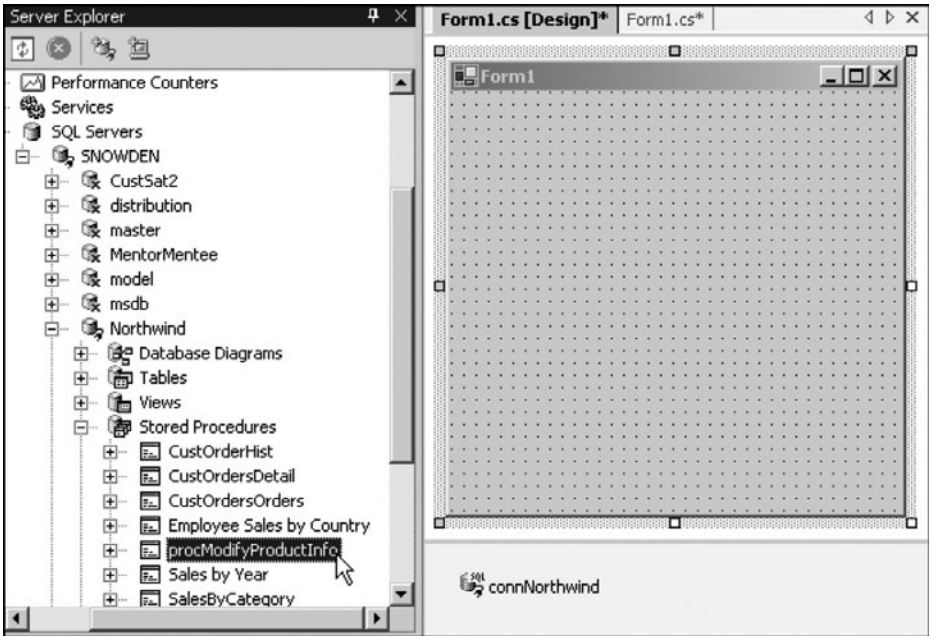


FIGURE 6.19: Adding a Stored Procedure Command to a Project

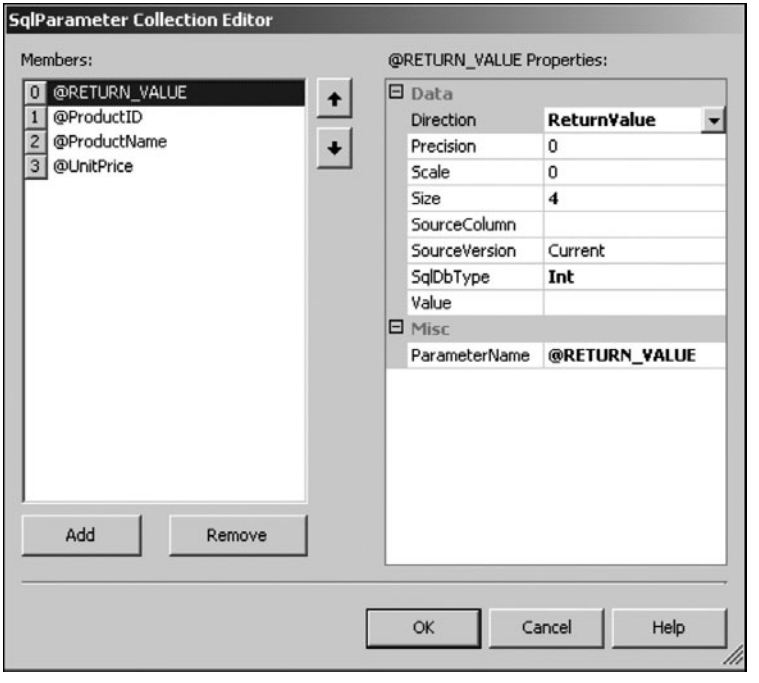


FIGURE 6.20: Graphic Representation of a `Parameters` Collection

Download at WoweBook.Com

Getting back to our example, we do not want the graphical representation of the command; instead, we want the underlying code. So, we set a few final properties for the command object in the Properties window, such as the name (in this case `cmndDB`) and the connection, and then switch to Code View and expand the `Windows Form Designer generated code` region. As we proceed through the code, we come upon the generated code for our command, shown in Listing 6.13.

Listing 6.13: Designer-Generated Code to Create a Command Object

```
SqlCommand cmndDB = new SqlCommand();
        :
        :
//
// cmndDB
//
this.cmndDB.CommandText = "ModifyProductInfo";
this.cmndDB.CommandType = CommandType.StoredProcedure;
this.cmndDB.Connection = this.connNorthwind;
this.cmndDB.Parameters.Add(
   new SqlClient.SqlParameter(
      "@RETURN_VALUE", SqlDbType.Int, 4,
      ParameterDirection.ReturnValue, false,
      ((System.Byte)(0)), ((System.Byte)(0)),
      "", DataRowVersion.Current, null));
this.cmndDB.Parameters.Add(
   new SqlClient.SqlParameter(
      "@ProductID", SqlDbType.Int, 4));
this.cmndDB.Parameters.Add(
   new SqlClient.SqlParameter(
      "@ProductName", SqlDbType.NVarChar, 40));
this.cmndDB.Parameters.Add(
   new SqlClient.SqlParameter(
      "@UnitPrice", SqlDbType.Money, 4));
```

This is exactly what we want and exactly what we did not want to write. So, we cut and paste this code to move it from the Windows Application project into our Smart Device Application project, pasting it into the `UpdateSelectedRow` routine. We remove some of the code that was in the previous version of our application, change all occurrences of `SqlCe` to `Sql`, and add code to move values from the controls into the parameters and to execute the command. Along the way we remove a few superfluous references to `this` and add some minimal error handling. Listing 6.14 shows the resulting code.

Download at WoweBook.Com

Listing 6.14: Stored Procedure Version of the `UpdateSelectedRow` Routine

```
private void UpdateSelectedRow(int intProductID,
                               string strProductName)
{
   //  A connection and a command
   SqlConnection connDB = new SqlConnection(strConn);
   SqlCommand cmndDB = new SqlCommand();

   try
   {
      //  Open the connection.
      connDB.Open();

      //  Initialize the command (including
      //    creating the parameters).
      cmndDB.CommandText = "procModifyProductInfo";
      cmndDB.CommandType = CommandType.StoredProcedure;
      cmndDB.Connection = connDB;
      cmndDB.Parameters.Add(
         new SqlParameter(
         "@RETURN_VALUE", SqlDbType.Int, 4,
         ParameterDirection.ReturnValue, false,
         ((System.Byte)(0)), ((System.Byte)(0)),
         "", DataRowVersion.Current, null));
      cmndDB.Parameters.Add(
         new SqlParameter(
         "@ProductID", SqlDbType.Int, 4));
      cmndDB.Parameters.Add(
         new SqlParameter(
         "@ProductName", SqlDbType.NVarChar, 40));
      cmndDB.Parameters.Add(
         new SqlParameter(
         "@UnitPrice", SqlDbType.Money, 4));

      //  Assign values to the parameters.
      cmndDB.Parameters["@ProductID"].Value =
         int.Parse(comboKeys.SelectedItem.ToString());
      cmndDB.Parameters["@ProductName"].Value =
         textProductName.Text;
      cmndDB.Parameters["@UnitPrice"].Value = 123.45;

      //  Execute the stored procedure.
      cmndDB.ExecuteNonQuery();

      //  Check the SQL Server return value.
      if((int)cmndDB.Parameters["@RETURN_VALUE"].Value != 0)
      {
         MessageBox.Show(
            "You should have already caught a SqlException."
```

Download at WoweBook.Com


```
            );
      }
   }

   catch( SqlException exSQL )
   {
      foreach( SqlError errSQL in exSQL.Errors )
      {
         MessageBox.Show(errSQL.Message);
      }
   }

   finally
   {
      //  Close the connection.
      connDB.Close();
   }
}
```

We placed all the code in one routine to make it more readable. In reality, we would move any code that needed to be executed only once, such as initializing the command and creating the parameters, into a separate routine. What would remain is the code that had to be executed every time, such as assigning values to the parameters, executing the procedure, and examining the return value.

6.4.4 Using Stored Procedures with `DataSet` Objects

Stored procedures can be specified in the `CommandText` property of the command objects that reside within a data adapter; that is, the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands. The command builder object, used earlier in this chapter to generate commands for the data adapter, does not generate commands that use stored procedures. If you want the adapter to use stored procedures, you must create the commands yourself.

The purpose of the data adapter is to move data between the database and the data set, and stored procedures are an excellent way to retrieve and update data (sometimes the only way, as when the updating requirements are very complex or access to the database is channeled through stored procedures for security reasons). Thus, it is beneficial to know how to use command objects based on stored procedures with the data adapter.

Using a stored procedure in a `SELECT` command is quite simple; any command object based on a stored procedure that consists of a single `SELECT` statement can be used as the data adapter's `SelectCommand` property. As long as you have permission to execute the procedure and set the

Download at WoweBook.Com

parameter values before calling the adapter's `Fill` method, the procedure can be used to select the rows that are then loaded into the receiving data table.

Using a command object based on a stored procedure for the adapter's `INSERT`, `UPDATE`, and `DELETE` commands requires an additional piece of information and an extra step. The extra information is required during the creation of each parameter because the value of each parameter must now come from a field in the data table whose data is being updated to the server. Because the data adapter updates one row at a time, the values from each row are loaded into the parameters, the procedure is executed, and then the process continues with the next row. You must specify which column goes with which parameter.

The extra step comes from the fact that the values assigned to the parameters come from the "new" version of the row, which is what you would expect and want. But the parameter(s) that comprise the primary key of the row, which are the values to be used to specify the database row(s) to be updated, must come from the "old" version of the rows. Because primary key values should not be modified, this is, hopefully, a moot point. However, you need to specify it just in case.

For example, earlier in this chapter we added a form to our `DataReader` sample program that allows a user to change data in bound controls. The changes in the bound control data are automatically propagated to the underlying data table. We discussed the added code necessary for the data

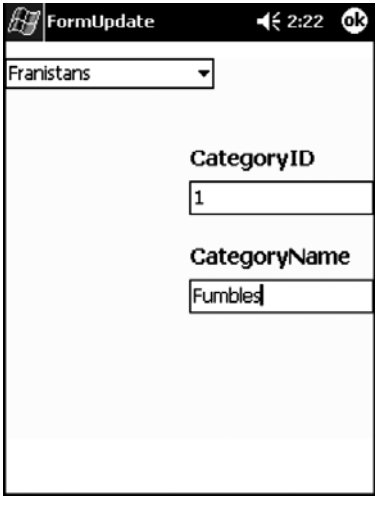


FIGURE 6.21: The Category Modification Form (Repeated)

Download at WoweBook.Com

adapter to propagate the changes from the data table back to the database. This form was shown earlier in Figure 6.14; for convenience, we repeat it here in Figure 6.21.

Now we create a very simple stored procedure, shown in Listing 6.15, which modifies the category name for a given category ID.

Listing 6.15: The Stored Procedure for Category Modification

```
CREATE PROCEDURE procModifyCategoryInfo
   @CategoryID int = null,
   @CategoryName nvarchar(40) = null
AS
BEGIN
   IF @CategoryID is null
   BEGIN
      RAISERROR('Category ID not supplied.',10,1)
      RETURN 1
   END

   IF not exists (SELECT *
                   FROM Categories
                  WHERE CategoryID = @CategoryID)
   BEGIN
      RAISERROR('Category ID not on file.',10,1)
      RETURN 1
   END

   BEGIN TRANSACTION

   IF @CategoryName is not null
      UPDATE Categories
         SET CategoryName = @CategoryName
         WHERE CategoryID = @CategoryID
   IF @@ERROR <> 0
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR('Unable to update Categories table.',10,1)
      RETURN 1
   END

   COMMIT TRANSACTION
   RETURN 0
END
```

This procedure requires two parameters: the category ID of the category to be updated and the new name to be given to that category. Both of these values must come from the rows of the data table.

Download at WoweBook.Com

The code that builds the `UpdateCommand` object for use by the data adapter is shown in Listing 6.16. This code must execute before the `Update` method executes; it can execute either before or after the `Fill` method executes.

Listing 6.16: Building a Command Object That Uses a Stored Procedure

```
//  Initialize the command (including
//    creating the parameters).
SqlCommand cmndDB = new SqlCommand();
cmndDB.CommandText = "procModifyCategoryInfo";
cmndDB.CommandType = CommandType.StoredProcedure;
cmndDB.Connection =
   daptCategories.SelectCommand.Connection;

cmndDB.Parameters.Add(
   new SqlParameter(
   "@RETURN_VALUE", SqlDbType.Int, 4,
   ParameterDirection.ReturnValue, false,
   ((System.Byte)(0)), ((System.Byte)(0)),
   "", DataRowVersion.Current, null));

cmndDB.Parameters.Add(
   new SqlParameter(
   "@CategoryID", SqlDbType.Int, 4, "CategoryID"));

cmndDB.Parameters.Add(
   new SqlParameter(
   "@CategoryName", SqlDbType.NVarChar, 40, "CategoryName"));

cmndDB.Parameters["@CategoryID"].SourceVersion =
   DataRowVersion.Original;
```

In the next-to-last statement in this code, the fourth parameter specifies the column of the data table with the value for the `"@CategoryName"` command object parameter.

The last statement in Listing 6.16 specifies that the primary key value should be the value that was loaded into the data table from the database, not any new value that is now in the data table.

We have discussed the remaining code before; most of it was generated in a Windows Application project with a drag-and-drop operation.

Using stored procedures in a command object provides you a number of benefits. First, it frees you from the details of the SQL code (especially if someone else writes the stored procedure for you). Second, it provides for a more efficient execution of code on the server side. A third benefit, which

Download at WoweBook.Com

we demonstrate in this chapter, is that the presence of a stored procedure makes it easy to set up repetitive execution of the same command while varying the parameters with each execution.

As each stored procedure executes, your program receives the stored procedure's return value, which your program can use to decide whether the procedure executed successfully. Procedure-based command objects are helpful whether they are being used by a data adapter to move data between the database and ADO.NET objects or by your application to operate directly on a database without the involvement of ADO.NET classes.

6.4.5 `DataSet` Objects and Concurrency

Unlike SQL Server CE, SQL Server provides *concurrency protection;* that is, if data within a row of a data table is modified on the server (probably by some other user) after the `Fill` method has executed and before the `Update` method executes, the update of that row fails. After the `Update` method has completed, your application can determine which rows did not successfully update and retrieve their new values from the database. There is no automatic response for this situation. Your application must decide whether to resubmit its values, submit revised values, notify the user, forget it, and so on.

6.4.6 Using Multi `SELECT` Stored Procedures

When doing data modifications via data sets, it is very advantageous to have each adapter's command contain just one `SELECT` statement. As the preceding paragraphs mentioned, one `DataAdapter` should map to one database table. However, if you are retrieving data for display only, this restriction does not apply; and with good reason.

Often you have a stored procedure that executes several `SELECT` statements to gather related information about a certain subject. Such a stored procedure might gather information about an employee, including that employee's orders, the customers for those orders, the line items, and the products for those line items. Listing 6.17 shows a simplified version of such a stored procedure.

Listing 6.17: A Multi `SELECT` Stored Procedure

```
CREATE PROCEDURE dbo.procGetEmployeeII
@EmployeeID int
AS
BEGIN
```

Download at WoweBook.Com


```
SELECT *
  FROM dbo.Employees
  WHERE EmployeeID = @EmployeeID

SELECT *
  FROM dbo.Customers
  WHERE CustomerID IN
      ( SELECT CustomerID
          FROM dbo.Orders
         WHERE EmployeeID = @EmployeeID )


SELECT *
  FROM dbo.Products
  WHERE ProductID IN
      ( SELECT ProductID
          FROM dbo.[Order Details]
         WHERE OrderID IN
             ( SELECT OrderID
                 FROM dbo.Orders
                WHERE EmployeeID = @EmployeeID ))

SELECT *
  FROM dbo.Orders
  WHERE EmployeeID = @EmployeeID

SELECT *
  FROM dbo.[Order Details]
  WHERE OrderID IN
            ( SELECT OrderID
                FROM dbo.Orders
                WHERE EmployeeID = @EmployeeID )
END
```

As before, we'll use a data adapter object to retrieve the data from the stored procedure and place it in a data set. But this time we have five `SELECT` statements in our stored procedure. Therefore, our adapter will need to generate five tables in our data set and load each with the results from a different `SELECT` statement. Before executing the adapter's `Fill` method, we will need to tell the adapter what name to give each table in the data set. We do not need to tell the adapter what data source tables were used, as the adapter knows each incoming result set within the output of a multi `SELECT` command by the names `Table`, `Table1`, `Table2`, and so on, regardless of the underlying database table names.

We match the name we want the adapter to give each data table to the incoming result set's—`Table`, `Table1`, `Table2`, and so on—by adding

Download at WoweBook.Com

entries to the adapter's `TableMappings` collection. In our case, five entries are required; as shown in the following code snippet:

```
dapt.TableMappings.Add("Table", "Employees");
dapt.TableMappings.Add("Table1", "Customers");
dapt.TableMappings.Add("Table2", "Products");
dapt.TableMappings.Add("Table3", "Orders");
dapt.TableMappings.Add("Table4", "Details");
```

The complete code necessary to execute the stored procedure and populate the tables, including the preceding snippet, appears in Listing 6.18. In all, it does the following:

1. Creates an empty data set
2. Creates the connection, command, and adapter objects
3. Sets the `TableMappings` collection property
4. Retrieves the data and loads it into the data set

The `connectionString` and `employeeID` are passed to into the function containing the code.

Listing 6.18: Client-Side Code to Load a Multitable Data Set

```
DataSet dsetEmployee = new DataSet();
SqlDataAdapter dapt;

SqlConnection conn =
   new SqlConnection(connectionString);
SqlCommand cmnd = conn.CreateCommand();
cmnd.Parameters.AddWithValue("@Return", 0);
cmnd.Parameters[0].Direction =
   ParameterDirection.ReturnValue;

dapt = new SqlDataAdapter(cmnd);

dapt.SelectCommand.CommandType =
                     CommandType.StoredProcedure;
dapt.SelectCommand.CommandText =
                     "dbo.procGetEmployee";
dapt.SelectCommand.Parameters
         .AddWithValue("@EmployeeID", employeeID);
dapt.TableMappings.Add("Table", "Employees");
dapt.TableMappings.Add("Table1", "Customers");
dapt.TableMappings.Add("Table2", "Products");
dapt.TableMappings.Add("Table3", "Orders");
```

Download at WoweBook.Com


```
dapt.TableMappings.Add("Table4", "Details");

dapt.Fill (dsetEmployee);
```

When the preceding `Fill` statement has completed its execution, we have our fully loaded, five-table data set, loaded within a single round trip to the server machine.

With the completion of our discussion of multistatement result sets, we come to the last aspect of data set programming that we will cover in this book, and that is the subject of *typed data sets.*

6.4.7 Working with Typed Data Sets

All the data sets we have worked with thus far have been undefined until runtime. That is, not only are their instances not created until runtime, which is true of any class, but their very structure—what tables constitute the data set, what columns constitute the tables—is not specified until runtime. Yet in several cases that structure was known at design time. The multi `SELECT` example that we just saw is such a case. The five tables and their selected columns could be determined by examining the underlying database. In fact, if the stored procedure's `SELECT` statements had been specific in their column lists, the resulting data set structure would be even more apparent.

Typed data sets allow us to provide this structure specification at design time; enabling us to define a class that inherits from `DataSet` and that specifies the structure of our specific data set. Defining the class is quite easy to do. Instead of our writing the entire class, Visual Studio lets us connect to a source database; specify the desired tables, columns, and relationships; and have the class code generated for us. We get the benefits of a strongly typed data set, such as IntelliSense help and structure-specific routines. For instance,

```
OurDataSet.Tables["Products"]
```

becomes

```
OurDataSet.Products
```

And in coding the second line, the moment we enter "`OurDataSet.`", the `Products` table, being a property of the typed data set, shows in the IntelliSense drop-down list.

Download at WoweBook.Com

Typed data sets, like all data sets, call hold data that is retrieved from a data source or data that is created programmatically; but they are most commonly used for holding data that has been retrieved from a database, especially since the underlying database's structure propagates into the application's data structure. This common association between databases and typed data sets is the reason why we waited until we covered the `System.Data` data provider classes before beginning our discussion of typed data sets.

To illustrate the benefits of typed data sets we'll write a simple program, one that retrieves sales data from five different tables, all located in Microsoft's Northwind sample database.

We start by creating a new C# Smart Device application within Visual Studio. We then go to the Product menu and select Add New Item, giving us the Add New Item dialog shown in Figure 6.22. We specify the name for our typed data set class, `SalesData`, and choose Add.

Note that if we had added the data set by using the Data tab of the Toolbox, we would have created an untyped data set, which is not what we want. We must use the dialog shown in Figure 6.22 to have Visual Studio generate a typed data set class for us.

Visual Studio will add an .xsd file and some .cs files to our project. When we view the .xsd file in the Designer, we are given some choices. We choose to right-click and then pick Add | Table Adapter, and are presented with the first of a series of dialogs that will lead us through the definition of one

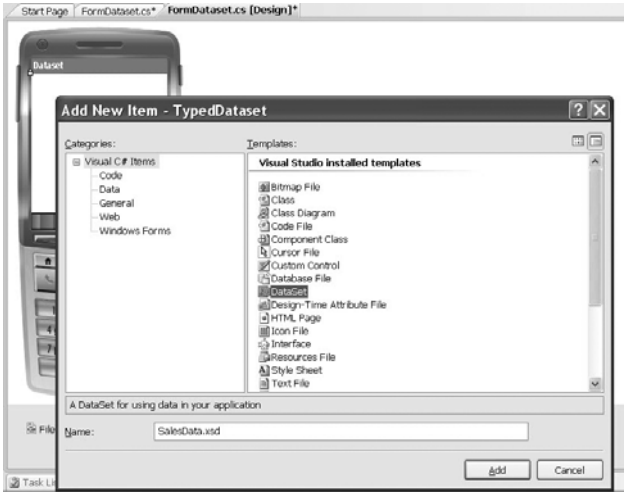


FIGURE 6.22: Adding a New Typed Data Set

Download at WoweBook.Com

data table and its associated data adapter, connection, and command objects. Since our data set will contain five data tables, we will pass through this series of dialogs five times.

The first of these dialogs is shown in Figure 6.23 and lets us specify the underlying database. The next few let us write the command that will retrieve the database data into the data table. In our case, with the Designer's help, we write the command shown in Figure 6.24. We then accept all defaults for the remaining dialogs and arrive at the situation shown in Figure 6.25: a data set that contains one table, a five-column

FIGURE 6.23: Specifying the Data Connection

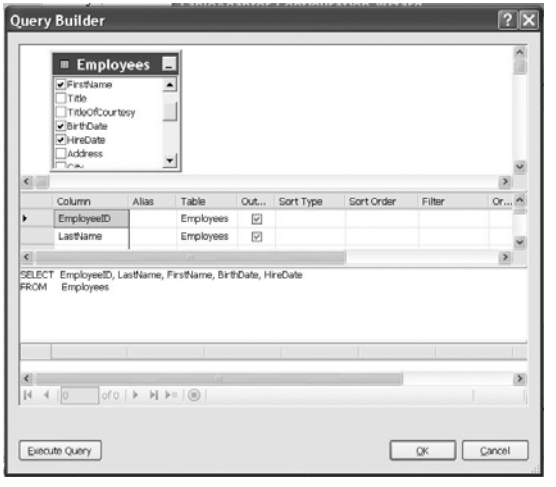


FIGURE 6.24: Coding the Query

Download at WoweBook.Com

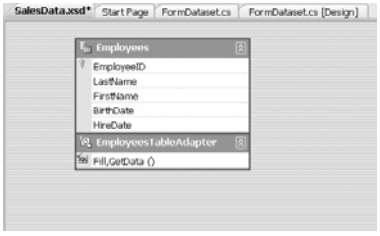


FIGURE 6.25: The Resulting `DataTable` and Its `TableAdapter`

`Employee` table. Note that Visual Studio has detected the presence of a primary key constraint in the database table and has included that constraint in the definition of the data table.

We'll define four more data tables but show the dialogs for only one of them, as the one aspect left to be illustrated is relationships. When we complete the definition of the first of our four remaining data tables, the `Orders` data table, Visual Studio detects a foreign key relationship between `Employees` and `Orders` and adds it to the typed data set, as shown in Figure 6.26.

We add three more table adapters, generating three more data tables and their respective data adapters, and arrive at the typed data set shown in Figure 6.27: a data set that we will use in this sample and again in Chapter 7.

The code for our typed data set class, written by Visual Studio, is in the SalesData.Designer.cs file, which appears in the Solution Explorer when you expand the SalesData.xsd file icon. It contains the code that exposes the data tables as properties of the data set; routines for navigating relationships, such as `ourEmployeeRow.GetOrdersRows();` routines for

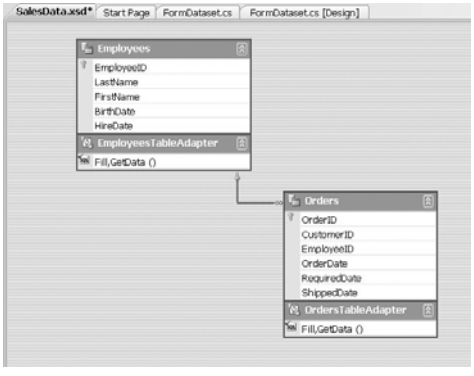


FIGURE 6.26: Two `DataTables` and Their `DataRelation`

Download at WoweBook.Com

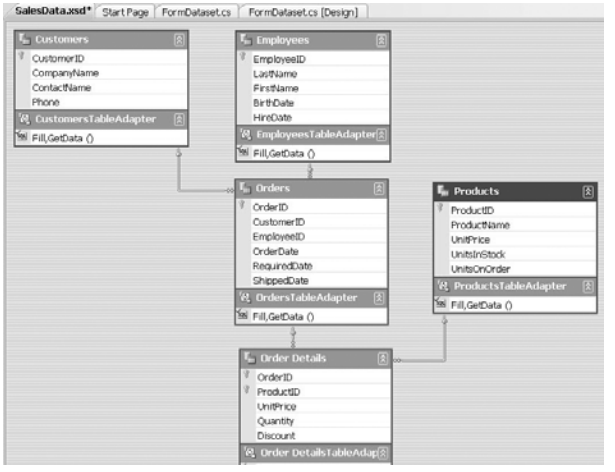


FIGURE 6.27: The Five Tables of Application Data

finding a row given its primary key value, such as `ourEmployeeTable.FindByEmployeeId()`; and others.

What neither the class itself, nor any other class in the project, contains is code to create an instance of our data set and code to fill that instance with data from the database. That code we must write ourselves. Fortunately, there is sufficient code already in the typed data set class to make our share relatively small and easy to write.

For the moment, we will put all the necessary code required to create and load the data set in the `LoadDataset` menu item's `Click` event handler, as shown in Listing 6.19.

LISTING 6.19: Code to Create and Load a Typed Data Set

```
SalesData dsetSales = new SalesData();

EmployeesTableAdapter daptEmployee = new EmployeesTableAdapter();
daptEmployee.Fill(dsetSales.Employees);

CustomersTableAdapter daptCustomer = new CustomersTableAdapter();
daptCustomer.Fill(dsetSales.Customers);

ProductsTableAdapter daptProduct = new ProductsTableAdapter();
daptProduct.Fill(dsetSales.Products);

OrdersTableAdapter daptOrder = new OrdersTableAdapter();
daptOrder.Fill(dsetSales.Orders);

Order_DetailsTableAdapter daptDetail =
```

Download at WoweBook.Com


```
   New Order_DetailsTableAdapter();
daptDetail.Fill(dsetSales.Order_Details);

dgrdOrders.DataSource =
   dsetSales.Employees
      .FindByEmployeeID(1).GetOrdersRows();
```

This code creates a data set containing five empty tables, creates five table adapters and uses them to fill the five tables, and then selects all orders for the employee whose Employee ID is 1 and binds those orders to a data grid, resulting in the display shown in Figure 6.28.

When Visual Studio wrote the code for the data set class, it defined a class for each table adapter, which it placed inside a namespace named `SalesDataTableAdapters`; that is, the name of the data set suffixed by `TableAdapters`. That namespace was placed inside the project's default namespace, in our case, `TypedDataset`. Thus, the code shown in Listing 6.19 requires the following line at the start of the code file:

```
using TypedDataset.SalesDataTableAdapters;
```

Now let us consider a small enhancement that will make our data set class more functional and more flexible. The added functionality will be a `Load` method in the data set class that will pull the data in from the database, allowing us to remove all knowledge of, and dependence upon, the table adapters from the form class's code. The added flexibility will be the option to specify the source database at runtime rather than being bound to the database that was specified at design time. This database designation will

FIGURE 6.28: A Display of One Employee's Orders

Download at WoweBook.Com

be made when the Load method is executed; that is, the connection string value will be a parameter of the Load method.

You might ask, why not specify the connection string when the data data set is created? Why not write an overload of the data set's constructor that takes the connection string as a parameter? We mentioned the answer earlier in this chapter; data sets are not provider-specific and they have no knowledge of where their data came from. Therefore, the data set class does not have a connection or ConnectionString property to set. It is the adapters that have a connection property, and it is there that we must specify the ConnectionString value.

Thus, we move the code for loading the data set from the form's menu event handler into the data set's newly defined Load method. The best location for the Load method is in SalesData.cs, not SalesData.Designer.cs. When the Designer wrote the data set class, it spread the code over the two files mentioned in the preceding sentence. SalesData.cs contains a skeleton partial class and is the place for our code. The other file is where the Designer writes its share of the class; we should not modify that file.

The contents of SalesData.cs are shown in Listing 6.20, and the revised menu handler is shown in Listing 6.21.

Listing 6.20: Partial Class Code for the Typed Data Set

```
using System;
using System.Data;
using System.Data.SqlClient;

using TypedDataset.SalesDataTableAdapters;

namespace TypedDataset
{
   public partial class SalesData
   {
      public void Load(string connectionString)
      {
         EmployeesTableAdapter daptEmployee =
            new EmployeesTableAdapter();
         CustomersTableAdapter daptCustomer =
            new CustomersTableAdapter();
         ProductsTableAdapter daptProduct =
            new ProductsTableAdapter();
         OrdersTableAdapter daptOrder =
            new OrdersTableAdapter();
         Order_DetailsTableAdapter daptDetail =
            new Order_DetailsTableAdapter();
```

Download at WoweBook.Com


```
        daptEmployee.Connection.ConnectionString =
        daptCustomer.Connection.ConnectionString =
        daptProduct.Connection.ConnectionString =
        daptOrder.Connection.ConnectionString =
        daptDetail.Connection.ConnectionString =
                                    connectionString;

        daptEmployee.Fill(this.Employees);
        daptCustomer.Fill(this.Customers);
        daptProduct.Fill(this.Products);
        daptOrder.Fill(this.Orders);
        daptDetail.Fill(this.Order_Details);
      }
   }
}
```

Listing 6.21: Revised Menu Selection Handler

```
private void mitemLoadDataset_Click(object sender,
                                    EventArgs e)
{
   string connectionString =
      string.Format("{0};{1};{2};",
                    "Data Source=RIMROCK",
                    "Initial Catalog=Northwind",
                    "Integrated Security=SSPI");

   SalesData dsetSales = new SalesData();

   dsetSales.Load(connectionString);

   dgrdOrders.DataSource =
      dsetSales.Employees
         .FindByEmployeeID(1).GetOrdersRows();
}
```

So, we now have a data set containing five data tables and five data adapters that support those tables. Thus, we could update the data in the data tables and propagate the changes back to the data source by using the same methods that were available with the untyped data source, as we did earlier in this chapter.

All of this raises the question of combining the last two subjects that we just discussed. Can we have a typed data set that uses a multi `SELECT` stored procedure to retrieve multiple tables of information for display only? The answer is yes, but we must do a few steps slightly differently.

Download at WoweBook.Com

Since we are dealing with multiple result sets of read-only data from a single command, a data adapter is of no benefit. In design mode, when we right-click on the typed data set's design surface, we should choose new *table* rather than new *table adapter.* The bad news is, the Designer will give us less help designing a table than designing a table adapter; the good news is, we will need to write less code.

We can write less code because the Designer always writes a `Load` method for typed data sets; not one that we could use before but one that we can use now. This `Load` method takes two parameters: one is the data reader that will pull the result sets of the multi `SELECT` stored procedure, and another is an array referencing the data tables to be filled. You can use the ADO.NET code shown earlier in this chapter to create the necessary connection and command objects, add the command parameter(s) to the command, and invoke `ExecuteReader` to generate the necessary data reader. This, in turn, puts you in a position to invoke the `Load` method.

> **NOTE**
>
> Programming with ADO.NET for Microsoft SQL Server is similar to programming for Microsoft SQL Server CE. The namespaces are different and the class names are different, but their PMEs are mostly the same. The major differences are caused by the different capabilities of the two engines, not by the classes themselves. Connectivity, security, stored procedures, advanced data integrity, concurrency, and differences in schema definition combine to make programming for SQL Server different from programming for SQL Server CE.

6.5 Web Services

Every day millions of people around the world access Web pages in search of information. Many of these pages contain executable code that gathers the information, formats it into text-based HTML, and returns it to the browser. And for almost as long as people have been doing this, application developers have been asking, "If I can do this, why can't my application do it and do it in an object-oriented way?" That is, "Why can't my application post a request for information to a Web page and receive the response from that page by creating objects and invoking their methods and then, like HTML, send and receive the request/response by using a text-based format so that the process can be platform-independent?"

Download at WoweBook.Com

Out of the desire for platform-independent, standards-based, Web-based exchange of data came Web services. Out of the desire for an object-oriented invocation of these Web services came the Visual Studio .NET development tools we are about to discuss.

6.5.1 XML, XSD, and SOAP

The World Wide Web Consortium publishes a format standard for text-based data exchange. This standard specifies both a format for exchanging data (XML) and a format for specifying the schema of that data (XSD). Web services use these formats to send and receive data. Figure 6.29 shows an XML document containing catalog and product information. Figure 6.30 shows the schema (which is itself XML) for the XML document shown in Figure 6.29.

For the sender and the receiver to be able to transmit the data not just as XML but in an object-oriented fashion (i.e., sending the data as input parameters, output parameters, and return values of class methods), some additional information must be added into each XML transmission. This additional information is formatted in SOAP.

Microsoft defines a Web service as "A URL addressable set of functionality that is exposed over a network to serve as a building block for creat-

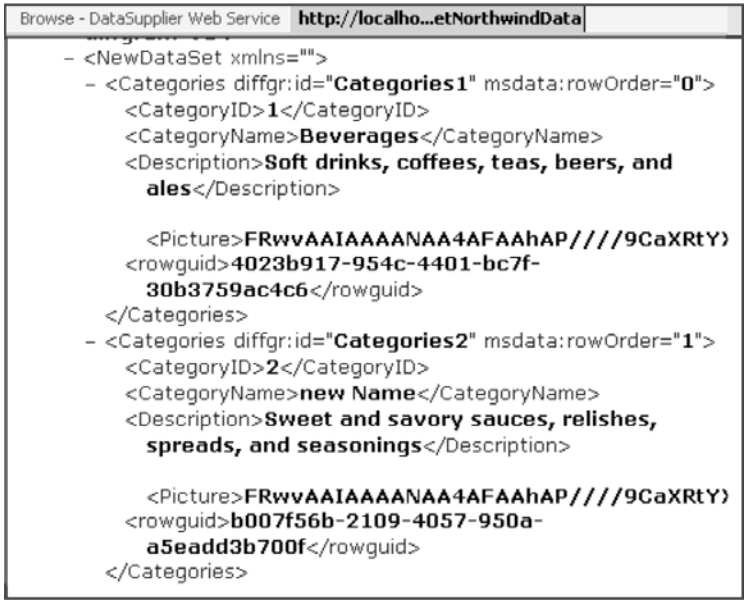


FIGURE 6.29: An XML Document

Download at WoweBook.Com

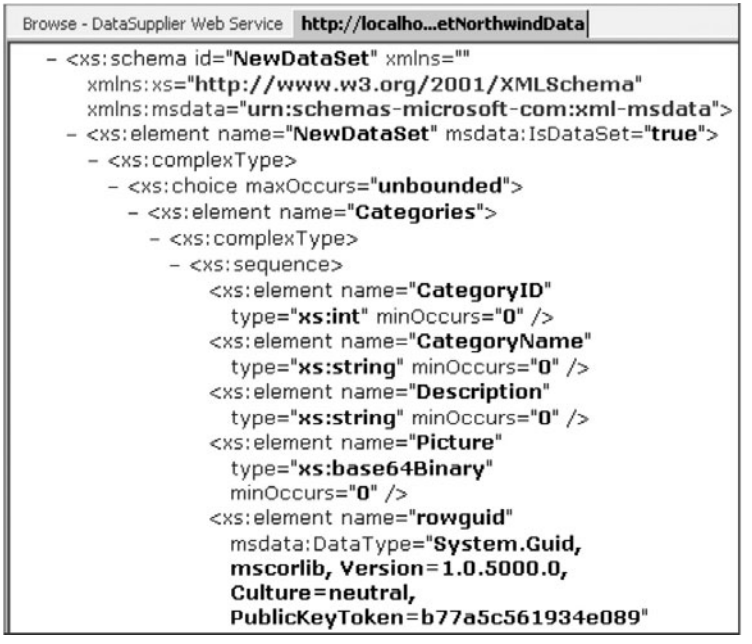


FIGURE 6.30: The XSD Schema for the Document in Figure 6.29

ing distributed applications."[6] That is, a Web service is a Web site, and at that Web site are .asmx pages containing class definitions of objects that can be instantiated and accessed via the Internet. From the perspective of a .NET Compact Framework client, the client instantiates client-side objects that represent those server-side classes and then sets properties and invokes methods. A client-side object that represents a Web service is referred to as the *proxy* object.

Visual Studio .NET will generate proxy object code for you. That code substitutes the transmission of the SOAP messages between the client and the Web site for the method calls you write. That is, the proxy code generates and transmits the XML necessary to invoke the server-side method and parses the returned XML to extract output parameters and the return value.

As the application developer, you code instantiations of objects and invocations of their methods; as an executing assembly, your application posts XML to a URL.

To illustrate the relationship between what you code and what is transmitted, let's write a Web services application. Our intent in this book is not

6. Microsoft Official Curriculum Course #2524, "Developing XML Web Services Using Microsoft ASP.NET."

Download at WoweBook.Com

to make you a Web services developer, but we are going to develop a simple Web service here to illustrate some basic concepts. Understanding the basic development of a Web service will help you write your client applications. Our Web application will consist of two Web services: `Primes` and `DataSupplier`. The first supplies prime numbers, the second supplies data sets. `Primes` is typical of services that do a large amount of processing to generate a small amount of data. `DataSupplier` is just the opposite, generating a large amount of data from a small amount of processing.

6.5.2 A Web Services Application

To create a Web service, we start a new project in the IDE and choose ASP.NET Web Service, as shown in Figure 6.31.

Creating the project creates a Web site. This site is created at the location we specified in the New Project dialog box. It cannot reside on a device, nor can the project be a Smart Device Application project. A Web service cannot run on a mobile device because, currently, the lack of a fixed IP address for a device and the processing power and regular availability required by a Web service preclude this. It is the clients of Web services that can run on mobile devices.

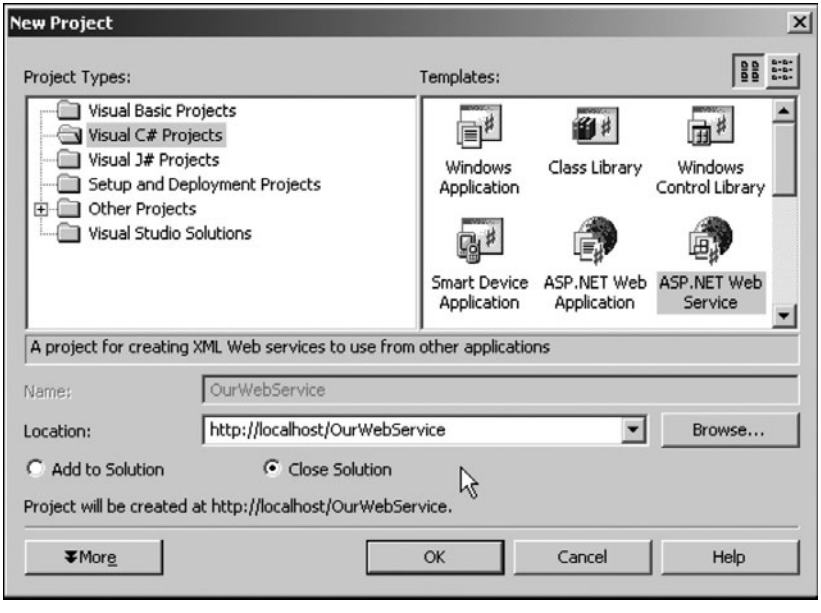


FIGURE 6.31: Creating a New Web Service

Download at WoweBook.Com

The project is created with one Web services page already in place, WebForm1.asmx. Each .asmx page, referred to as a *service,* can have one or more classes defined within it, and one application can have as many pages as it needs. When we open the generated sample code from `WebForm1`, shown in Listing 6.22, we see a single class with just one method defined but commented out.

Listing 6.22: A Web Method

```
      // WEB SERVICE EXAMPLE
      // The HelloWorld() example service
      //    returns the string Hello World
      // Uncomment the following lines then
      //    save and build the project
      // To test this Web Service, press F5

//      [WebMethod]
//     public string HelloWorld()
//     {
//         return "Hello World";
//     }
```

A class method is made visible to clients by adding the `WebMethod` attribute. Simply making a method public is not enough to make it accessible to the client. Any method that does not have the `WebMethod` attribute can be invoked from within the class or assembly but not from a client.

For our first service, `Primes`, we choose something that dovetails nicely with the inherent strengths and avoids one of the common weaknesses of Web services. Specifically, the service involves complex processing and a small amount of data exchange. The service is located in an .asmx page named Primes.asmx that we add to the project. The `Primes` service implements just one class, `Primes`, which has just one method, `GetFloorPrime`, which returns the largest prime number less than or equal to a given number.

The code for the `Primes` service, that is, all the code on the .asmx page, is shown in Listing 6.23.

Listing 6.23: The `Primes` Web Services Class

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
```

Download at WoweBook.Com


```
using System.Web.Services;

namespace WSService
{
   /// <summary>
   /// Summary description for Primes
   /// </summary>
   public class Primes : System.Web.Services.WebService
   {
      public Primes()
      {
         //CODEGEN: This call is required
         //   by the ASP.NET Web Services
      InitializeComponent();
      }

      #region Component Designer generated code

      [WebMethod] public int GetFloorPrime( int Target )
      {
         if( Target <= 1 )
         {
            return 1;
         }
         if( Target == 2 )
         {
            return 2;
         }
         int k = Target - (Target % 2 == 0 ? 1 : 0);
         for( int j=k; j>=1; j-=2 )
         {
            if( IsPrime(j) )
            {
               return j;
            }
         }
         return 1;
      }

      private bool IsPrime( int Candidate)
      {
         for( int j=3; j<=Candidate - 1; j+=2 )
         {
            if( Candidate % j == 0 )
            {
               return false;
            }
            if( j * j >= Candidate )
            {
               break;
```

Download at WoweBook.Com


```
            }
        }
        return true;
    }
  }
}
```

We can test this page the same way we test any Web page that is under development, by right-clicking on the page and selecting View in Browser from the context menu. When we do, the image shown in Figure 6.32 appears in the browser.

It might seem strange that anything shows in the browser. After all, our page has no visible interface; it is just a collection of object classes. But some of the `System.Web.Services` classes, via reflection, generate WSDL, the language for describing the class's interface over the network (see the Web Services Description Language sidebar). Thus, the complete signature of the `Primes` class has just been delivered to the browser for display, in what is called the *help page* for the Web service, as shown in Figure 6.32. The help page was generated by code in the Web service that itself was generated by Visual Studio .NET. When a Web service is being authored using Visual Studio .NET, code is generated that returns a help page whenever the .asmx page is accessed without any query string parameters being provided.

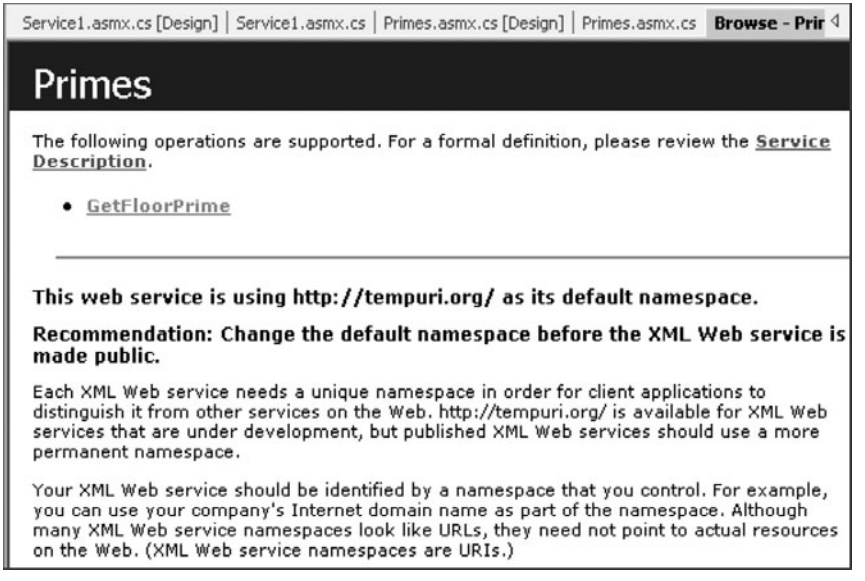


FIGURE 6.32: The `Primes` Web Services Page in the Browser

Download at WoweBook.Com

For instance, accessing the following URL causes the help page to be delivered as the response: www.OurServer.com/WSService/Primes.asmx. The same URL with the attached query string shown here, www. OurServer.com/WSService/Primes.asmx?op=GetFloorPrime, indicates that the `GetFloorPrime` Web method of the service is to be invoked.

Web Services Description Language

The Web Services Description Language (WSDL) is an XML-based language that Web Services use to define the requests they will accept and the responses they will return. From the perspective of a .NET Compact Framework programmer, the most important information in the WSDL is the signatures of the classes. Thus, the WSDL received from a Web service will tell the client the names of the classes and the names and data types of the public properties, methods, parameters, return values, and so on. In other words, the WSDL relays all the information necessary for a client program to create an instance of a class at the Web service and to invoke the class's methods and properties.

Because the browser now has all the necessary information to generate and send the SOAP messages that represent the method calls, we can test the page's class methods, as shown in Figure 6.33.

The test page also shows the SOAP message that will be sent and received by a method call, in this case `GetFloorPrime`, as shown in Figure 6.34.

We test the `Primes` service by asking for the largest prime less than 16. Figure 6.35 shows us the SOAP message returned by the service. Buried within it is "13," the return value from the `GetFloorPrime` method call.

Next we add another Web services page, named `DataSupplier`, to our project and code our second service. This one also has just one class and one method. The method does very little processing and returns a data set that was built when the application first started. This is a common practice for Web services because they often need to bring data into memory, hold it in memory for a period of time, and make it available to clients during that time. The code for this service is shown in Listing 6.24.

Download at WoweBook.Com

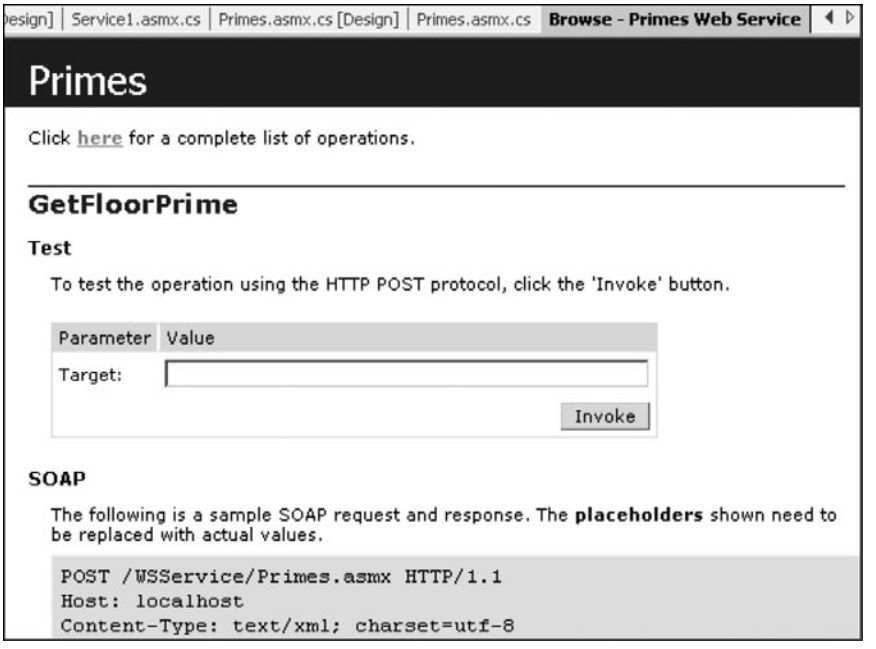


FIGURE 6.33: Testing the Web Services Page's `Primes` Class in the Browser

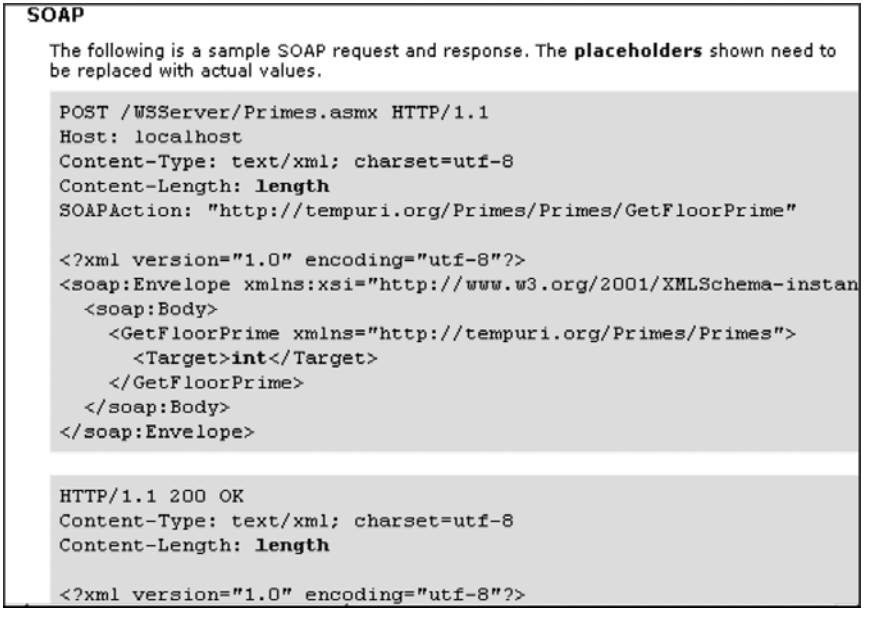


FIGURE 6.34: The Structure of a SOAP Message Request

Download at WoweBook.Com

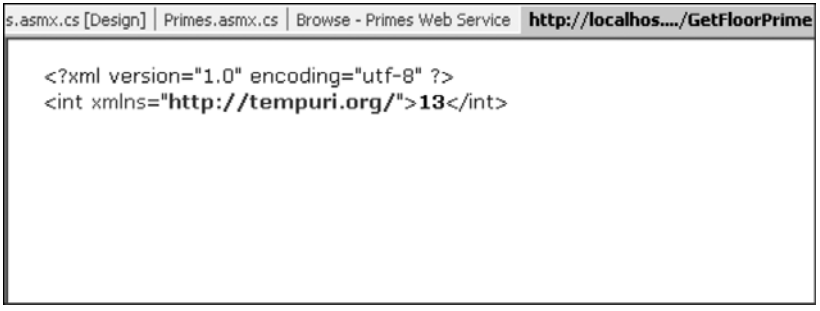


FIGURE 6.35: The SOAP Message Returned by the `GetFloorPrime` Method Call

LISTING 6.24: The `DataSupplier` Web Services Class

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WSService
{
   /// <summary>
   /// Summary description for DataSupplier
   /// </summary>
   public class DataSupplier : System.Web.Services.WebService
   {
      public DataSupplier()
      {
         //CODEGEN: This call is required
         //         by the ASP.NET Web Services
         InitializeComponent();
      }

      #region Component Designer generated code

      [WebMethod] public DataSet GetNorthwindData()
      {
         return Global.dsetDB;
      }
   }
}
```

The code that creates the data set is in the global area of the Web service, Global.asax. It uses the ADO.NET classes we already covered, including

Download at WoweBook.Com

System.Data and System.Data.SqlClient, to create the data set and load the Categories and Products tables from the Northwind database into the data set. Much of the code was generated by dragging the two tables from the Server Explorer window onto the Global.asax Designer. Listing 6.25 shows the code for the .asax page.

Listing 6.25: The Global.asax Web Services Page

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections;
using System.ComponentModel;
using System.Web;
using System.Web.SessionState;

namespace WSService
{
   /// <summary>
   /// Summary description for Global
   /// </summary>
   public class Global : System.Web.HttpApplication
   {
      /// <summary>
      /// Required designer variable
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      public Global()
      {
         InitializeComponent();
      }

      internal
         System.Data.SqlClient.SqlDataAdapter daptCategories;
      internal
         System.Data.SqlClient.SqlCommand SqlSelectCommand1;
      internal
         System.Data.SqlClient.SqlConnection connNorthwind;
      internal
         System.Data.SqlClient.SqlCommand SqlSelectCommand2;
      internal
         System.Data.SqlClient.SqlDataAdapter daptProducts;

      public static DataSet dsetDB;

      protected void Application_Start(Object sender,
                                       EventArgs e)
      {
```

Download at WoweBook.Com


```
    dsetDB = new DataSet();
    LoadDataSet(dsetDB);
}

protected void Session_Start(Object sender, EventArgs e)
{

}

protected void Application_BeginRequest(Object sender,
                                        EventArgs e)
{

}

protected void Application_EndRequest(Object sender,
                                      EventArgs e)
{

}

protected void Application_AuthenticateRequest(
                                         Object sender,
                                         EventArgs e)
{

}
protected void Application_Error(Object sender,
                                 EventArgs e)
{

}

protected void Session_End(Object sender, EventArgs e)
{

}

protected void Application_End(Object sender, EventArgs e)
{

}

private void LoadDataSet( DataSet dsetDB)
{
    daptCategories.Fill(dsetDB, "Categories");
    daptProducts.Fill(dsetDB, "Products");
    dsetDB.Relations.Add(
       "FKProdCAt",
       dsetDB.Tables["Categories"].Columns["CategoryID"],
```

Download at WoweBook.Com


```
            dsetDB.Tables["Products"].Columns["CategoryID"],
            true);
        }

        #region Web Form Designer generated code
    }
}
```

We test this service by viewing it in the browser just as we did the previous service. This is very easy to do because the `DataSupplier` service's only method requires no parameters. That method returns a data set. More correctly, it returns the SOAP message representation of a data set, which is identical to the XML persistence format that `DataSet.WriteXml` produces.

Figure 6.36 shows the output of our test invocation of the `GetNorthwindData` method. As you can tell from the size of the scroll bar thumb, it is a lot of bytes—far more bytes than SQL Server's own format, which is used by the `SqlClient` classes and by the synchronization classes we encounter in Chapter 8.

Figure 6.36 gives you insight into the two downsides of XML: It is verbose and it is not secure. Actually, it is the transport that is not secure. XML is no less secure than HTML, and most of us have allowed our credit card

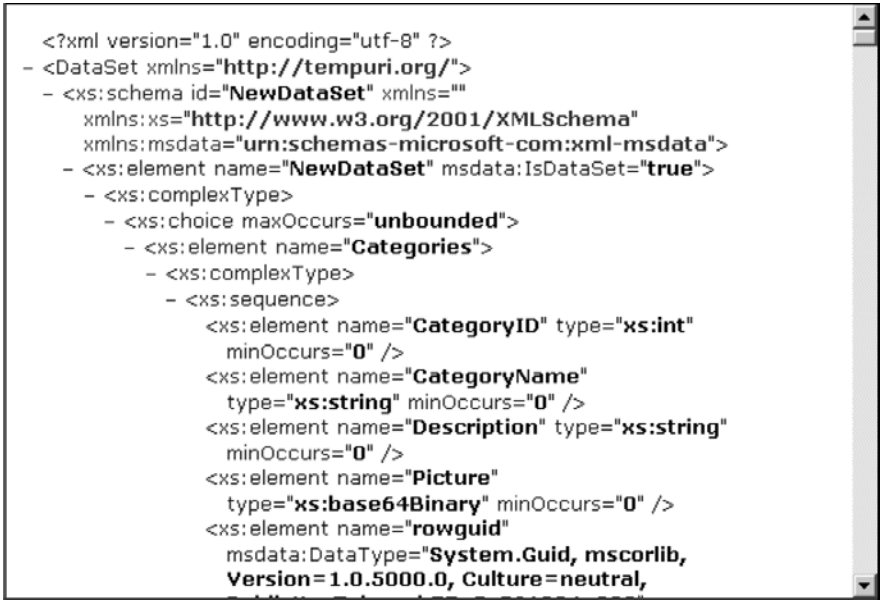


FIGURE 6.36: The SOAP Message Representation of a Data Set

Download at WoweBook.Com

information to be transmitted via that protocol. The key is to use a secure transport protocol, that is, HTTPS (Secure Sockets Layer) instead of HTTP. HTTPS won't solve the verbosity problem, but it will address the security issue.

There are enormous upsides to Web services. The platform independence they provide, the use of the Internet as the transmission mechanism, the adherence to developing standards, the presence of an independent standards committee, and the ease of client program development all make a strong case for using Web services, especially if the clients are requesting small amounts of data from within a secure networking environment.

6.5.3 A Web Services Client Application

To illustrate the ease of developing Web services client programs in Visual Studio .NET, we develop a client application for our two Web services. The application consists of two forms, one for calling the `Primes` service and one for calling the `DataSupplier` service, as shown in Figures 6.37 and 6.38, respectively.

We open a new Smart Device application, add our two forms, and place the controls on them. We are ready to start writing code.

But writing code works best when we have IntelliSense help, and that means having the class definitions available while we code. The compiler also needs those definitions because it must perform type checking during

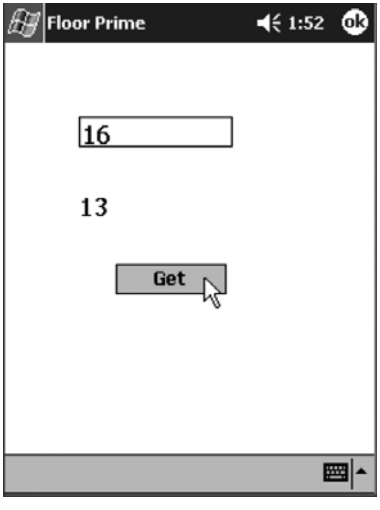


FIGURE 6.37: Using the `Primes` Service

Download at WoweBook.Com

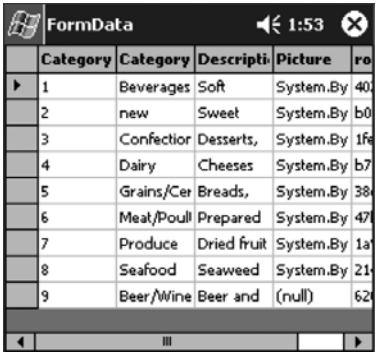


FIGURE 6.38: Using the `DataSupplier` Service

compilation. The place to get those class definitions is from the WSDL that the .asmx pages emit. Just as the browser took full advantage of the WSDL to provide us with a test environment, Visual Studio .NET uses the WSDL to provide us with a development environment. Throughout this book, we have been getting class information by setting references to assemblies. Now we will get our class information by setting our reference to the .asmx page. Either way, we are setting a reference to an object that provides interface definition information for a class.

In our Solution Explorer window, we right-click on References and then we select Add Reference from the context menu, receiving the dialog box shown in Figure 6.39.

We enter the URL of our Web services page, as shown in the figure, and click the Add Reference button, producing a reference entry. We want to rename the reference, which is a good practice in general. Think of the reference name as though it were the namespace name of the Web services class because, when you write code, the reference name will be the namespace name. That is, you create a new `ReferenceName.ClassName` object to access the properties and methods of that Web services class. In our example, we change the reference name to `Primes`, as shown in Figure 6.40.

Then we repeat the process for the other Web services page located at the `WSServer` site, naming it `DataSupplier`.

This makes writing the code quite simple. For our first form, the one that uses the `Primes` class, we create an instance of the `Primes` object and then invoke its `GetFloorPrime` method. The input value is obtained from the text box, and the returned value is displayed in the label. Listing 6.26 presents the code; Figure 6.30 shows the results.

Download at WoweBook.Com

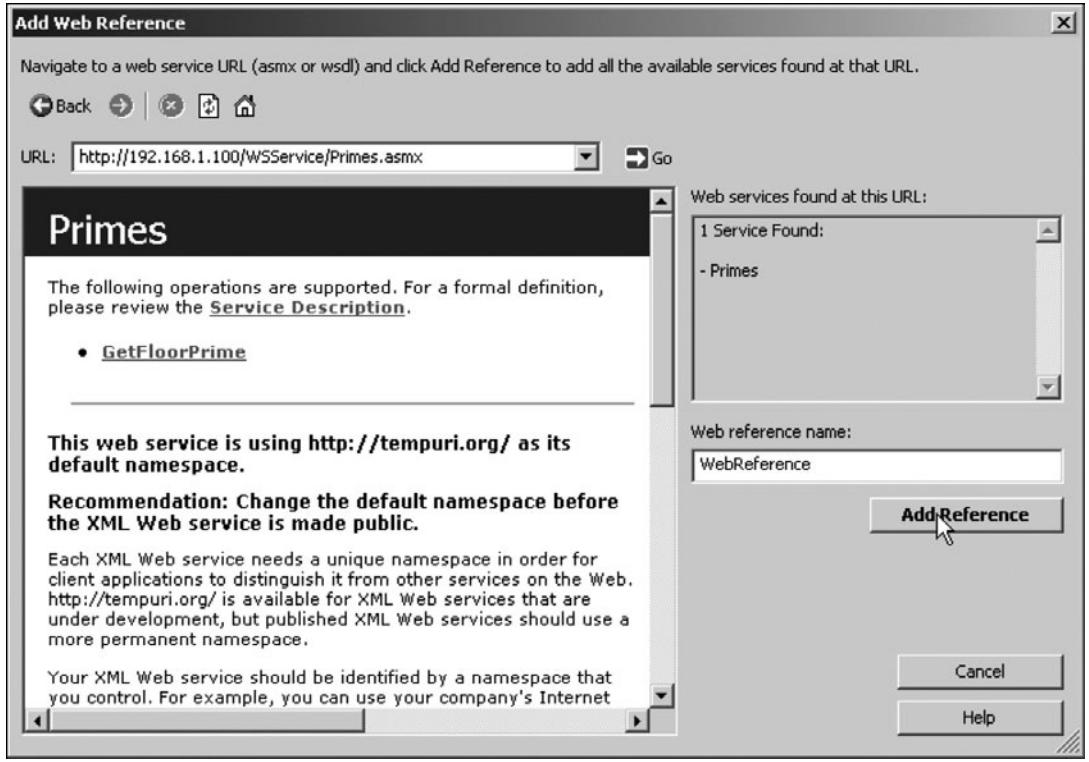


FIGURE 6.39: The Add Web Reference Dialog Box

From the Library of Lee Bogdanoff

Download at WoweBook.Com

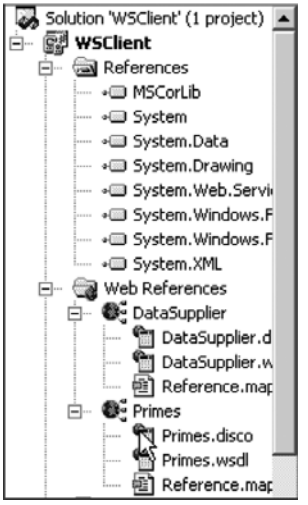


FIGURE 6.40: The Renamed Reference Entry

LISTING 6.26: Calling the `Primes` Client

```
private void cmdGet_Click(object sender, EventArgs e)
{
   Primes.Primes  refPrimes = new Primes.Primes();

   lblFloorPrime.Text =
      refPrimes.GetFloorPrime(
         int.Parse(txtTarget.Text)).ToString();
}
```

We have already seen how much SOAP text had to be transmitted to submit the one integer to the service and receive back the one prime.

Our second form creates an instance of the `DataSupplier` class and retrieves the data set by calling its `GetNorthwindData` method. The `Categories` table from the data set is assigned to the data grid's `DataSource` property. The code, shown in Listing 6.27, is even simpler than that for the first form, but the number of bytes transferred between client and server is much greater, as we already know. Figure 6.31 shows the results.

LISTING 6.27: Calling the `DataSupplier` Client

```
private void FormData_Load(object sender, EventArgs e)
{
   DataSupplier.DataSupplier  refDataSupplier =

      new DataSupplier.DataSupplier();
```

Download at WoweBook.Com


```
   dgridNorthwind.DataSource =
      refDataSupplier.GetNorthwindData().Tables["Categories"];
}
```

> **NOTE**
>
> So, in this section, we have seen the good and the bad of Web services. If you need to make available functionality that requires small exchanges of data, Web services are a good candidate. If you are planning to regularly exchange large data sets with a server machine, Web services are a bad candidate; the SQL Server Merge Replication and Remote Data Access techniques covered in the next chapter might be a better choice.

6.6 Conclusion

We have now completed our coverage of ADO.NET programming. We started with the memory-resident database that is the data set and its contained objects and continued with a discussion of SQL Server CE databases and SQL Server databases. We moved through the various techniques for retrieving, presenting, and updating the information in those databases, emphasizing both the two-tier and the three-tier approaches. Finally, we looked at Web services as a way to retrieve data, including entire data sets, from a remote server.

But it is the rare mobile application that lives in a vacuum. More common are applications that depend on data from the outside world—and the outside world is changing rapidly. In the next chapter, we examine how to transfer data between your mobile application and SQL Server.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

7 LINQ

.NET 3.5 introduces LINQ, a mechanism for manipulating collections of objects. This chapter shows how you can use LINQ capabilities such as selecting, ordering, filtering, and aggregating to simplify the retrieval, display, modification, and persistence of data.

7.1 Overview

When Relational Database Management Systems (DBMSs) first burst on the scene in the 1970s one of their capabilities that helped fuel their universal acceptance was SQL, or Structured Query Language. Prior to SQL, all data was retrieved from DBMSs programmatically. For instance:

> Find the record in this file with this value in this column, then walk this chain of records until you find one with that value in that column, then add that record to a temporary file of records. When finished, sort the temporary file.

The (then-new) concept of asking the DBMS for the data you wanted rather than having to program its retrieval was quickly adopted by users and developers alike. Code such as the following quickly became the preferred method for requesting selective, correlated, ordered data from a set of records:

```
SELECT x.ColA, x.ColB, z.ColC
  FROM TableX x
```

Download at WoweBook.Com


```
  JOIN TableZ z
    ON z.ColA = x.ColA
 WHERE z.ColC = "NS"
   AND x.ColD > 44
 ORDER BY x.ColE DESC, z.ColF
```

Version 3.5 of .NET—on both the desktop and the device—extends this capability to the client side with *L*anguage *IN*tegrated *Q*uery (LINQ). LINQ provides the capability for client programs to query sets of objects, such as a `Table` of `DataRows` or a `Dictionary` of `Customers` or an XML document containing `PurchaseOrder` elements, by invoking object methods that mirror this SQL functionality. Client-side data can be selected, filtered, correlated, aggregated, and sorted by using LINQ's Standard Query Operators; operators whose names reflect their SQL equivalents—`Select`, `Where`, `Join`, `Sum`, `OrderBy`, and many more.

Many desktop programmers see LINQ as a tool for simplifying the access of relational databases. Although this is true for the desktop, it is not, as we will soon explain, true for the .NET Compact Framework. As this chapter and its sample program unfold, the emphasis will be on the great benefit to device applications of LINQ as a manipulator of object sets, be it an array of data objects or a collection of business objects.

The LINQ code to do a `select` from a set of objects might be as simple as the following:

```
myList
   .Where(order => order.CustomerID == "BERGS")
   .Select(order => order);
```

The preceding code would extract the `BERGS` entrants from a collection of objects, each of which had a `CustomerId` property. Note that these three lines are a single C# statement, one that we could have placed on a single line, were it not for the width limitation of this page.

Also, LINQ provides an alternative syntax that allows you to do the same selection thusly:

```
from order in myList
   where order.CustomerID == "BERGS"
   select order;
```

Object-oriented programmers, used to the "." notation, will perhaps be most comfortable with the former style, while database programmers will tend toward the latter. For most of this book, we will use the first style simply

Download at WoweBook.Com

because we want to emphasize LINQ as a manipulator of object sets, but periodically we will show both, for we do not wish to imply that one style is better than the other.

Although the preceding example was quite simple, a LINQ statement might be as complex as the one shown in Listing 7.1.

Listing 7.1: A Long and Complex LINQ Query

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
(
   employee.Orders
      .Where(order =>
         (OrderEntryData.IsActive(order.objectState)))
      .OrderBy(active =>
         active.parentCustomer.CompanyName)
         .ThenByDescending(active => active.NetValue)
         .ThenByDescending(active => active.OrderDate)
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text =
                   sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                           .PadLeft(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate
                           .ToString("dd-MMM-yy")
         })
)
.ToList();
```

The code in Listing 7.1 comes from one of this chapter's sample programs. It does the following:

1. Extracts all the active orders from the `Orders` dictionary object referenced by `employee.Orders`
2. Sorts them into descending order date within descending net value within customer sequence
3. Converts each order into an object of an unnamed class containing
   a. An order ID property named `Id`

Download at WoweBook.Com

   b. A new property named `Text` that is the concatenation of the customer ID / net value / order date propertys
4. Converts that collection into a `List`
5. Assigns that `List` to the data source property of a `ComboBox` such that the `OrderId` of the order subsequently selected by the user can be easily determined

Listing 7.2 shows the alternative syntax for the statement shown in Listing 7.1.

LISTING 7.2: The Same LINQ Query—Alternative Syntax

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
   (from order in employee.Orders
      where AppData.IsActive(order.objectState)
      orderby order.parentCustomer.CompanyName,
              order.NetValue descending,
              order.OrderDate descending
      select new
      {
         Id = order.OrderID,
         Text = String.Format("{0} - {1} - {2}",
                order.parentCustomer.CustomerID,
                order.NetValue.ToString("c")
                        .PadLeft(9).Substring(0, 9),
                order.OrderDate
                        .ToString("dd-MMM-yy"))
      }).ToList();
}
```

One look at the code in Listing 7.1 or Listing 7.2 should make anyone who is new to LINQ realize that some additional information about these LINQ methods and their syntax will be necessary before we can begin programming. But we need to start our LINQ programming discussion even further back than LINQ itself. Since LINQ provides for the manipulation of sets of objects, we must be sure that we understand the set classes that are provided with .NET and the specific capabilities of those classes upon which LINQ is built.

7.1.1 Set Classes in .NET

The .NET set class fall into a variety of differing groups, such as arrays versus collections or generic collection classes versus specialty collection

Download at WoweBook.Com

classes. LINQ doesn't care what category a set class falls into. What it does need, however, is for the class to have implemented the `IEnumerable` interface. If the class has implemented `IEnumerable`, LINQ, like `foreach`, can work with objects of that class. If the class also implements `IList`, LINQ may be able to provide performance optimization for certain operators. But the functionality aspect of LINQ is dependent upon the target class having implemented `IEnumerable`, whether that class is an ADO.NET `DataTable`, a `List<Employee>`, or a class of your own design.

The most commonly used general-purpose set classes, the `System.Array` class and the `System.Collections.Generic` classes, provide the following capabilities that you will often draw upon when writing your LINQ expressions.

- The items within the set can be restricted to objects of a certain class. Thus, we can speak of "an array of strings" or "a list of customers" and be assured that our array will contain only strings and our list will contain only customers.
- Because, as we mentioned earlier, the set class must implement `IEnumerable`, the items within the set can be enumerated. That is, they can be accessed one at a time. We often take advantage of enumeration through the use of the `foreach` statement. Even collection classes that do not provide indexing, such as `Queue` and `Stack`, provide enumeration through the `IEnumerable` interface and thus provide `foreach` capability.
- The set can determine whether an object is in the set. The testing method may be named `Contains` or it may be named `Exists`, but there is always a way to test.

As we write our LINQ code, you will see these common features manifesting themselves within the various LINQ functions that we will be using.

7.1.2 LINQ in the Compact Framework

In .NET 3.5, there are five major variations on LINQ: LINQ to SQL, LINQ to Entities, LINQ to Dataset, LINQ to Objects, and LINQ to XML. The first two, LINQ to SQL and LINQ to Entities, are primarily intended for accessing relational databases and are not available in the Compact Framework. This is why we stated earlier that, in the Compact Framework, LINQ is not primarily a tool for simplifying the access of relational databases, but

Download at WoweBook.Com

rather is a manipulator of collections. Since LINQ to SQL and LINQ to Entities are not available in the Compact Framework, we will confine our efforts to the other LINQ variations, notably LINQ to Datasets and LINQ to Objects.

But before starting on each of the three individually, we should make some comments about LINQ syntax in general. In these introductory comments we will be using the word *collections* in a general sense, a sense that includes both collections and arrays, and definitely not restricting our remarks only to members of the `System.Collections.Generic` namespace.

As we already mentioned, LINQ methods are generic methods of `IEnumerable` classes, meaning they work on collections of objects, just as SQL clauses work on the ultimate RDBMS collection, the rows of a table. Like SQL statements, LINQ methods can return a new collection of objects or they can return a single value. In the code in Listing 7.1 `Where`, `OrderBy`, and `Select` each returned a new collection; one that was subsequently refined by the method that followed it. In all, four LINQ methods were invoked, turning the original `employee.Orders` collection into the `List` that was data-bound to the `ComboBox`.

In defining the collection that will be returned from a LINQ operator, such as `Where`, the `=>` symbol implies that the word preceding the `=>` is a placeholder variable. For each object in the original collection, a reference to that object will be placed into the placeholder variable, the code to the right of the `=>` symbol will be executed, and the result will be used to build the new collection or to compute a value. Thus, both

```
myList.Where(order => order.CustomerID == "BERGS").Select();
```

and our old friend `foreach`, as in

```
foreach (Order order in employee.Orders)
{
   if (order.CustomerID == "BERGS")
   {
      // Add this order to an existing collection.
   }
}
```

access each order in the `employee.Orders` collection via the `order` placeholder variable and use it to generate a new collection.

Download at WoweBook.Com

The => symbol is known by a variety of terms; it is the *lambda expression* operator in formal LINQ terminology, the *anonymous function* in the C# specification, and the *goes to* variable in common usage.

The name of the placeholder variable is up to you. In the complex example shown at the start of this chapter, we used a different name within each of the three methods. We chose `order`, `active`, and `sorted`, respectively, to reflect the progression that was being accomplished. More commonly, a developer would use the same name within each method.

7.1.3 Deferred Execution

Another aspect that makes LINQ unique is that not all of the method calls that comprise the complete transformation need to be declared within a single statement. Thus, the definition of the total LINQ operation can be broken down into separate declarations that will execute as a single operation. For instance, the statement shown earlier,

```
myList.Where(order => order.CustomerID == "BERGS").Select();
```

could have been broken into the following:

```
var bergsCriteria =
   myList.Where(order => order.CustomerID == "BERGS");

         :
         :
         :

List<Order> bergsOrders =
   new List<Order>(bergsCriteria.Select(order => order));
```

The first statement defines the criteria for Berg's orders. The second statement selects all orders meeting the criteria and uses them in the construction of a new list containing just Berg's orders. Note that the first method is not executed at the line where it is declared. Its execution is deferred, becoming part of the execution of the second statement. Therefore, the orders that appear in `bergsOrders` are the `orders` that meet the criteria specified in the first statement at the time the second statement is executed. That is, orders that are in `myList` when the first statement is encountered do not determine the content of `bergsOrders`; it is the orders that are in `myList` when the second statement is encountered that determine the contents of `bergsOrders`.

Download at WoweBook.Com

Also note that `bergsCriteria` is a variable; its contents can be modified at runtime. The program could have placed a `Where` method criterion into `bergsCriteria`, executed the second statement, placed a different `Where` method criterion into `bergsCriteria`, executed the second statement again, and produced a completely different collection of orders. This capability to store what is essentially source code into a variable for subsequent execution is very handy.

A query that can be defined across multiple lines of code is referred to as *composable.* There is a point in time at which a query definition becomes immutable and can no longer be modified. Once a LINQ operator that iterates over the collection, such as `Select`, has executed, no further composition can occur.

With this background out of the way, we are ready to talk about our coding examples and turn our attention to our sample application, which we use to illustrate both LINQ to Datasets and LINQ to Objects. There are three versions of this application: one that uses data sets, one that uses business objects, and one that combines the benefits of both while requiring the least amount of code.

7.2 The Sample Application

Our sample application will be an order entry application. We chose an order entry application as the sample for this chapter for two main reasons. The first is to make it a nontrivial, business-oriented application, one that would demonstrate LINQ's ability to manipulate both ADO.NET data objects and business objects to easily perform commonly required tasks.

The second reason for making it an order entry application is that a familiar source of data is already available: the Northwind database. The Northwind database has been around for a long time now, and its structure is familiar to many, so much so that we have already used it in previous examples. What is important is not whether our application is an order entry application but whether it illustrates ways to use LINQ in your application and the benefits of doing so.

Our application will use five Northwind tables, as shown in Table 7.1. To eliminate much repetition of code and text, we will use a small number of columns from each table, just enough to illustrate the things that a typical smart-device application would need to do, and we will use LINQ to do as many of those things as possible.

Download at WoweBook.Com

TABLE 7.1: Five Northwind Tables Used by the LINQ Sample Application

Northwind Table
Employees
Customers
Products
Orders
Order Details

From a business perspective, the application allows any employee to sell any product to any customer. The employee uses the application to

1. Retrieve all of the employee's orders from the host SQL Server
2. Add, delete, and modify orders
3. Save/restore all application data to a local file
4. Upload all entered information to the host SQL Server

From a developer's perspective, the application will

1. Use ADO.NET classes to retrieve data from a SQL Server database
2. Store that data in a data set (the LINQ to Datasets version) or in business objects of our own design (the LINQ to Objects version) or in a combination of both (the hybrid version)
3. Bind our data to controls
4. Validate data
5. Use XML serialization to move application data to and from the device's object store
6. Upload changes to the SQL Server database

Figure 7.1 shows a schematic diagram of the application.

The entire code of the application is not shown in this book, as doing so would take too much space. If you've seen one regular expression used to validate one string property, you've seen one used to validate them all. If you've seen one business object track changes, you've seen them all track changes. Rather than show all the code, we will focus on one or two

Download at WoweBook.Com

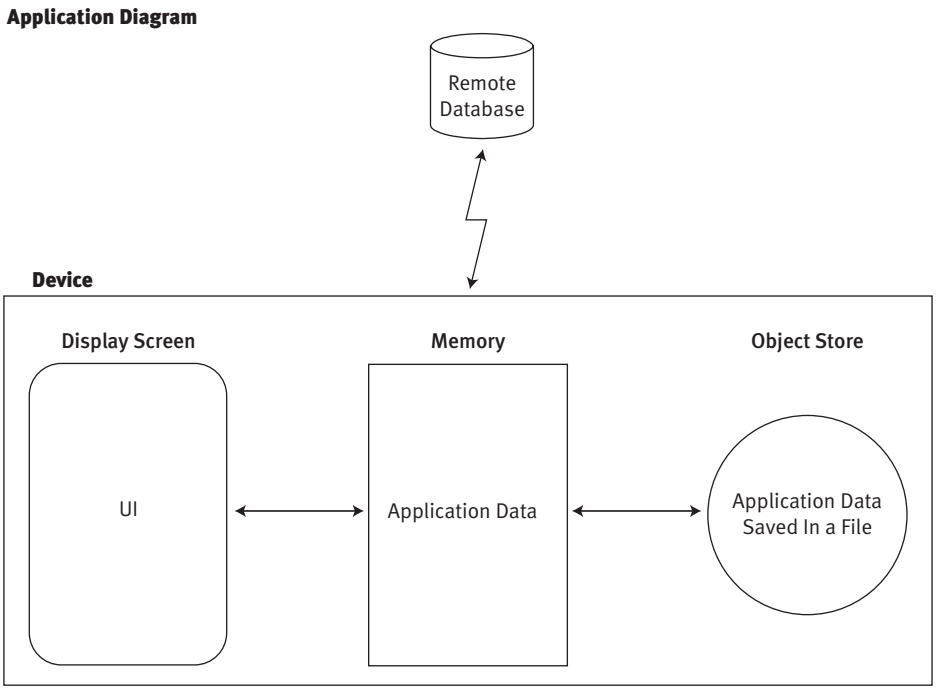


FIGURE 7.1: Sample Application Schematic Diagram

classes, and on the utility routines that are used to provide functionality to all objects, and especially on the use of LINQ within that code. You can find the complete code for all three versions of the application at the book's website. Each version has its own project subdirectory: `LinqToDatasets`, `LinqToObjects`, and `LinqHybrid`.

The main starting point for the application is the retrieval of the employee's orders from the Northwind database. The employee's row must be retrieved from the `Employees` table, the employee's orders retrieved from the `Orders` table, and their details retrieved from the `Order Details` table. Since an employee can sell any product to any customer, all rows must be retrieved from both the `Customers` and the `Products` tables. This information is gathered by a SQL Server stored procedure that takes the employee ID as its only parameter and retrieves the necessary rows from each table. Listing 7.3 shows this procedure.

LISTING 7.3: SQL Server Stored Procedure for Accessing Employee's Order Records

```
ALTER PROCEDURE dbo.procGetEmployee
      @EmployeeID int
```

Download at WoweBook.Com


```
AS
BEGIN
   SET NOCOUNT ON;

   SELECT *
     FROM dbo.Employees
     WHERE EmployeeID = @EmployeeID

   SELECT *
     FROM dbo.Customers

   SELECT *
     FROM dbo.Products

   SELECT *
     FROM dbo.Orders
     WHERE EmployeeID = @EmployeeID

   SELECT *
     FROM dbo.[Order Details]
     WHERE OrderID IN
        ( SELECT OrderID
            FROM dbo.Orders
           WHERE EmployeeID = @EmployeeID )
END
GO
```

The information retrieved by this stored procedure, when brought into the application's memory space, results in the application data structure that is diagrammed in Figure 7.2. Throughout this chapter, we will refer to this data as the *application data,* regardless of whether it is stored in data tables within a data set or in collections of business objects contained within a parent XML-serializable object.

Within the application, the data that was retrieved from the `[Order Details]` table will be known by the one-word name, `Details`. The authors of this book have strong, and not positive, opinions regarding the use of spaces within object names.

Now that we have the server-side stored procedure in place, we turn our attention to the client-side application and the code needed to receive and process the data. We start with the LINQ to Datasets version. Since we'll be using LINQ to Datasets, we can make our schematic diagram a bit more specific, as shown in Figure 7.3.

Download at WoweBook.Com

FIGURE 7.2: The Application's Data

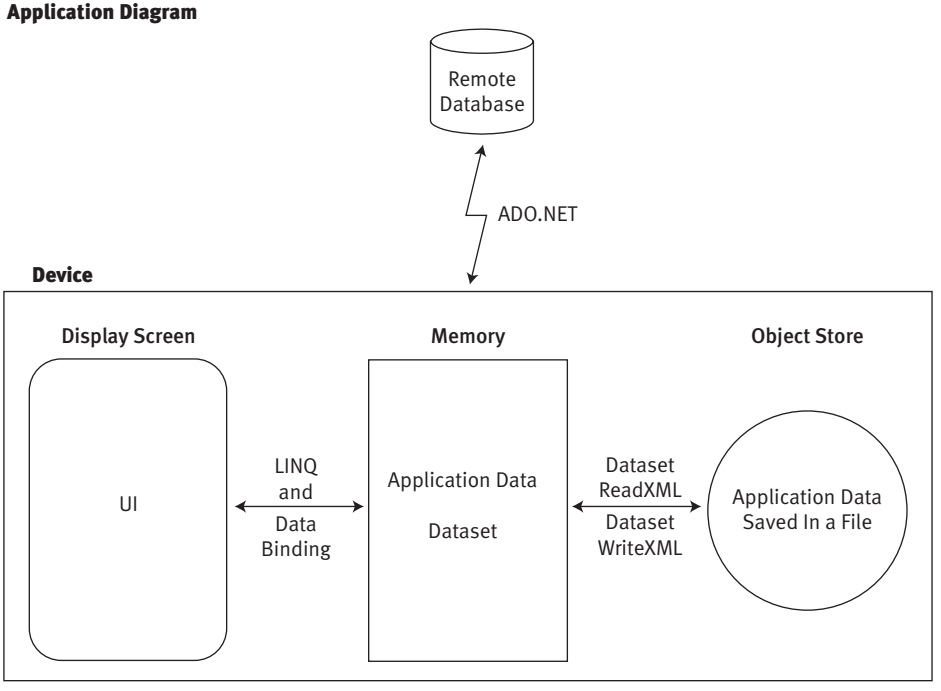


FIGURE 7.3: Sample Application Diagram: LINQ to Datasets

7.2.1 LINQ to Datasets

As we did in Chapter 6, ADO.NET Programming, we'll use a `DataAdapter` object to receive the data from the stored procedure and place it in a data

Download at WoweBook.Com

set. But this time we have five SELECT statements in our stored procedure. Therefore, our adapter will need to generate five tables in our data set and load each with the results from a different SELECT statement. Before executing the adapter's Fill method, we will need to tell the adapter what name to give each table in the data set. We do not need to tell the adapter what data source tables were used, as the adapter knows each incoming result set within the output of a multi SELECT command by the names Table, Table1, Table2, and so on, regardless of the underlying database table names.

We match the name that we want the adapter to give each data table to the incoming result sets—Table, Table1, Table2, and so on—by adding entries to the adapter's TableMappings collection. In our case, five entries are required, as shown in the following code snippet:

```
dapt.TableMappings.Add("Table", "Employees");
dapt.TableMappings.Add("Table1", "Customers");
dapt.TableMappings.Add("Table2", "Products");
dapt.TableMappings.Add("Table3", "Orders");
dapt.TableMappings.Add("Table4", "Details");
```

Listing 7.4 provides the complete code necessary to accomplish the following four tasks.

1. Create an empty Dataset.
2. Create the connection, command, and adapter objects.
3. Set the TableMappings collection.
4. Retrieve the data and load it into the data set.

The connectionString and employeeID are passed into the function containing the code. The resulting DataSet has the structure shown in Figure 7.4 and is the *application data* object for this version of the application. To make all three versions of our application as consistent as possible, we will always place a reference to the application data object into a static field named AppData located in the Program class. In this version, the data set is the application data object.

The code shown in Listing 7.4 does not contain any LINQ statements, for it just loads data from a remote database into a data set on the device. Only after we have placed the data into the data set will we begin to use LINQ to manipulate that data.

Download at WoweBook.Com

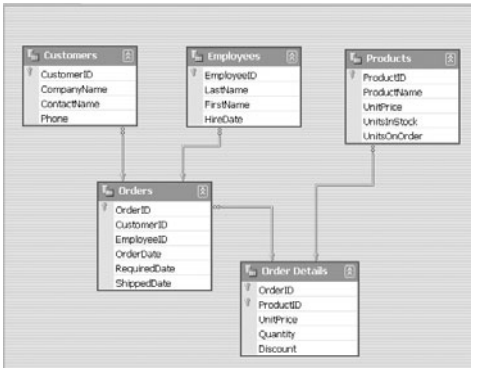


FIGURE 7.4: Structure of Data Set

LISTING 7.4: Client-Side Code to Load a Multitable Data Set

```
DataSet dsetEmployee = new DataSet();
SqlDataAdapter dapt;

SqlConnection conn =
   new SqlConnection(connectionString);
SqlCommand cmnd = conn.CreateCommand();
cmnd.Parameters.AddWithValue("@Return", 0);
cmnd.Parameters[0].Direction =
   ParameterDirection.ReturnValue;

dapt = new SqlDataAdapter(cmnd);

dapt.SelectCommand.CommandType =
                     CommandType.StoredProcedure;
dapt.SelectCommand.CommandText =
                     "dbo.procGetEmployee";
dapt.SelectCommand.Parameters
         .AddWithValue("@EmployeeID", employeeID);
dapt.TableMappings.Add("Table", "Employees");
dapt.TableMappings.Add("Table1", "Customers");
dapt.TableMappings.Add("Table2", "Products");
dapt.TableMappings.Add("Table3", "Orders");
dapt.TableMappings.Add("Table4", "Details");

dapt.Fill(dsetEmployee);
```

Once we have the data, we need to display it to the user. This is where the capabilities of LINQ begin to come in. For example, consider a form, `FormEmployee`, that lets an employee choose which order to work on. This form must display a drop-down list of orders within a `ComboBox`. Combo boxes need two pieces of information per item being listed: the value that

Download at WoweBook.Com

will serve as the identifying key of the item and the text value to be displayed to the user.

Unfortunately, the `Orders` data table has no single column that we wish to display, for no single column contains all of the information we want the user to see. We want to display a composite of fields to the user. This is a common need. How many times has a developer needed to display a drop-down list listing people from a table that had a `FirstName` and `LastName` column but no `FullName` column—or in this case, the need to combine customer ID, order date, and net value into the list?

Thanks to LINQ—with the help of Visual Studio's IntelliSense—we write the code shown in Listing 7.5 to bind rows from the `Orders` data table to the combo box. The result appears in Figure 7.5.

Listing 7.5: Client-Side User Interface Code

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
(
   employee.Orders
      .OrderBy(order => order.parentCustomer.CompanyName)
         .ThenByDescending(order => order.NetValue)
         .ThenByDescending(order => order.OrderDate)
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text = sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                           .PadLeft(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate
                           .ToString("dd-MMM-yy")
         })
)
.ToList();
```

Again, we remind readers that an alternative syntax, the query expression syntax, was shown in Listing 7.2.

In the code, in one statement, we produced a sorted list of objects from a data table, objects of a class that did not exist when the statement began. We defined that class, a class containing `Id` and `Text` properties, within the statement itself. Every row in the data table was used to create an object of

Download at WoweBook.Com

FIGURE 7.5: Results from LINQ Query Displayed on Smart-Device Screen

the new class and that object was placed into a new collection, which in turn was converted into a `System.Collections.Generic.List` object and bound to the combo box.

Like anything related to programming, the placement of parentheses is critical. Had the parenthesis that appeared just before `ToList` appeared after it, the statement would not have compiled. The steps involved in the creation of the new collection of new objects had to be specified in its entirety before that collection could be converted into a `List`.

Another point must be made regarding the `ToList` operator; namely that it exists in this LINQ statement to satisfy a `DataSource` need, not a LINQ need. That is, the `Select` operator generates a collection that implements `IEnumerable`. If an enumerable collection were all that we needed, `ToList` would not have been part of the statement. But a `ComboBox DataSource` requires a collection that implements `IList`, not `IEnumerable`, thus the need for the final step of converting an *enumeratable* collection into a *listable* collection.

We also sorted the rows, not within the data table but for output. Some developers might say, "No big deal. Arrays and many collection classes have always been able to sort the objects contained within them." This is true, but there is a huge difference between that kind of sorting and LINQ sorting. Previous set sorting required that the determination of the fields to be used for the comparison be made by the objects being sorted or by a third, supporting, object and that those objects implement the `IComparer` or `IComparable` interface. In LINQ sorting, we specify the sort criteria in the statement itself; criteria that, as we mentioned earlier in the chapter, can be defined and modified at runtime.

Download at WoweBook.Com

Had we wanted to sort on the concatenated field that we generated within the execution of the statement, we would have done the sort after the select, as shown in the example code in Listing 7.6. In this example, the `Text` property of the on-the-fly defined class, the anonymous class as it is known, is used as the sort key.

Listing 7.6: Sorting Anonymous Objects

```
cbxOrders.DataSource =
(
   employee.Orders
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text = sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                       .PadRight(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate.ToString("dd-MMM-yy")
         })
      .OrderBy(item => item.Text))
   .ToList();
```

One comment about these anonymous classes that LINQ statements generate for you: They are truly anonymous. If you set a break point and drill down into the bound object, you'll see that objects within it are objects of a certain structure but not of any name. This means that no object of another class can be cast as an object of an anonymous type.

It also means that it is sometimes impossible to use LINQ's `Union` method with anonymous classes. Since `Union` can be used only to connect two sets of the same class, and since `Union` cannot base that determination on class name when anonymous classes are involved, it must examine the metadata of the two anonymous classes that are to be `Union`ed and ascertain that they match perfectly.

The authors have occasionally been surprised at what they could and could not `Union`. For instance, the form that is displayed before `FormEmployee`, `FormStart`, also has a combo box. It is used to display a list of employee IDs. Just to experiment, we decided to use a `Union` and to place an additional entry at the start of the list, an entry containing the following instruction to the user: "Select your Employee No". You must use `Union` in this scenario because you cannot add any entries to a combo box after it has been data-bound. You must merge the additional entries into the set

Download at WoweBook.Com

before data binding. The code, which appears in Listing 7.7, works just fine, producing the result shown in Figure 7.6. When we tried to add the same user instruction to the top of the combo box in `FormEmployee`, using the same `Union` clause to do so, it failed. However, if we `Union`ed the user instruction to the end of this list, by swapping the two sides of the `Union`, the code succeeded.

Listing 7.7: Unioning Two Lists

```
private List<string> topOfList =
   new List<string>
      (new string[1] { "<Select your Employee No>" });

private List<int> EmployeeIDs =
   new List<int>(
      new int[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });

cbxEmployeeID.DisplayMember = "Text";
cbxEmployeeID.ValueMember = "Id";
cbxEmployeeID.DataSource =
(
   (
   topOfList
      .Select(instruction =>
         new
         {
            Id = 0,
            Text = instruction
         }))
   .Union
   (
   EmployeeIDs
      .Select(item =>
         new
         {
            Id = item,
            Text = item.ToString()
         }))
 )
 .ToList();
```

The moral of the story is that we should expect that one anonymous class would be different from another anonymous class and that any attempt to cast from one type to another, however implicit it may be, may fail regardless of how similar their definitions may appear.

Download at WoweBook.Com

FIGURE 7.6: Display of `Unioned` Lists

As we just saw, in our LINQ to Datasets application, we use LINQ to extract data from our data set tables, transform it, and bind it to controls. Regarding the non-LINQ functionality of our application, the `Dataset` class was a great help. As we mentioned in Chapter 6, the data set allows us to persist application data to local storage through its `ReadXML` and `WriteXML` methods; and it assists us in delivering changes back to the remote database by providing the data adapter's `Update` method, a method that relies on the `DataRow` class's change tracking and primary key capabilities to build the necessary data modification statements.

What data sets do *not* provide is a central location for business logic. We cannot make the data set validate user input, for there is no way to tell a data set that "WA state driver's license numbers must be one letter, followed by six letters or *s, followed by three digits, followed by two letters" or "No one is allowed to have more than four books checked out of this lending library at a time."

Providing a central location for business logic is a job for business objects. Who better to validate customer ID than the `Customer` class itself? Who better to specify the calculation of an order's net value than the `Order` class? So, let us transition our application to use business objects, for doing so will provide us with a way to illustrate the use of LINQ to Objects.

7.2.2 LINQ to Objects

To cover LINQ to Objects, we will modify the sample application. The new version will

1. Still use ADO.NET classes to retrieve data from a SQL Server database
2. Convert the retrieved data into business objects of our own class design; classes that could be ported without change to a Web or desktop version of the application
3. Use LINQ to Objects, both (1) in the binding of data between our business objects and our UI controls and (2) in the definition of our business objects

Download at WoweBook.Com

4. Place all business objects within a single application data object
5. Use XML serialization to persist the application data object to the device's object store
6. Provide data validation within our business objects
7. Provide change tracking within our business objects
8. Upload changes to the SQL Server database by examining the business objects

In the end, we will be using LINQ to Objects for data binding, for property definitions, and to help with XML serialization.

Figure 7.7 shows the schematic diagram for this version of the application.

If you think this version of the application will require more code than the previous version, you are correct; in fact, it will require significantly more code. This is why there will be a third version later in this chapter, one that attempts to combine the best aspects of the first two in the least amount

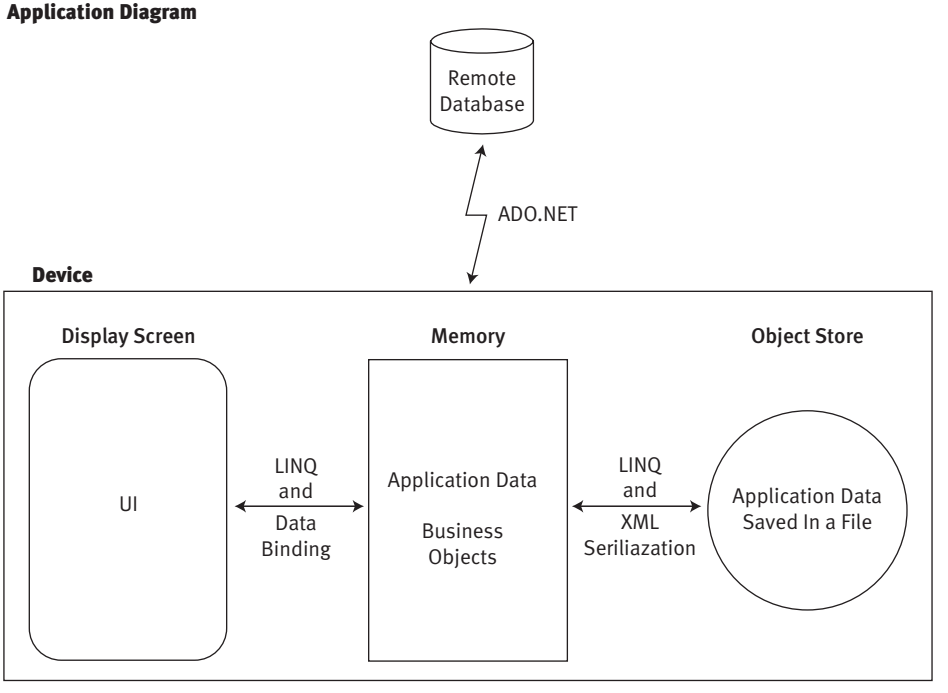


FIGURE 7.7: Sample Application Diagram: LINQ to Objects

Download at WoweBook.Com

of code. In the meantime, we need to focus on LINQ to Objects, for the more complex your application becomes, the more you need classes of your own design that encapsulate the business requirements of your application.

However, we cannot begin our discussion of LINQ to Objects without first discussing objects in general. Specifically, we need to discuss business objects, those objects that we design and write to represent the entities of our application—objects such as appointment, customer, product, and so forth—objects that are often referred to as the *middle tier.*

7.2.2.1 *Business Objects*

Since our business objects are in the middle tier, data is passing through them on its way between the user interface and the host database. It would be nice, then, if our business objects provided some support for this data movement along with their purely business functionality, such as an invoice class's `CalculateOverdueCharge` method.

Support for Disconnected Operations The first area in which a business object can provide support is in data validation. One of the great benefits of properties being defined as two methods, `set` and `get`, rather than as a field is that the object can police the values that are being assigned to it, rejecting those that it finds unacceptable by throwing an `ArgumentException` exception. Again, who better to specify the acceptable amount of money for a withdrawal than the `BankAccount` class itself?

And by placing the criteria for the property values within the property's `set` accessor, one can use that class in any type of application—smart device, desktop, Web, or other—and be assured of consistent and uncircumventable data validation. Also, by placing the validation code in the properties, we need not worry about which event handler is the appropriate place to do validation—the `TextBox`'s `TextChanged` event, the `ListView`'s `Validating` event, or some other handler—for we will not be doing validation in any event handler. The business object, not the UI tier, is the better place for validation logic.

The second area of business object benefit related to data movement is change tracking. We have already seen this capability in ADO.NET's `DataRow` class. Each row tracks whether it has been added, modified, or deleted since the data table was last synchronized with the source database. It would be nice if our business objects provided this same state tracking.

Download at WoweBook.Com

At the start of the workday, when a retail driver cradles his smart device, the application downloads data from the server and creates the business objects. Throughout the day, as information is entered, things are added, modified, and deleted. It would be nice if each business object tracked this fact. Then, at the end of the working day, pushing those changes back to the database becomes a matter of noting what changes occurred to what objects and transmitting them to the matching DBMS rows. This is the technique used by ADO.NET's `DataAdapter` object, which pushed changes from each row in a `DataTable` to a row in the database by matching the primary key value of the `DataRow` with the primary key value of the database table's corresponding row.

A third area of potential benefit, related to change tracking, is concurrency support. When our retail driver of the preceding paragraph downloads data from the host to his device, he expects to have pulled "his" information; his customers, his orders, and so forth. When uploading changes, the driver is not expecting that other individuals have modified that information in the host DBMS during the driver's working day. But it could have happened, and the application may need to deal with this possibility. The application must be able to detect that changes have occurred to the data since the data was retrieved and provide up to four possible choices when conflicts are detected: namely, make the driver's change at the host or do not make the change; and notify the driver or do not notify the driver.

Not changing a record because someone else made a change to that record while you were disconnected from the DBMS is called *optimistic concurrency,* as in you do nothing to prevent others from modifying the data because you are optimistic that they will not do so. Optimistic concurrency means writing an `update` statement that says the following:

```
UPDATE TableX
   SET ColA = this,
       ColB = that
 WHERE ColPrimaryKey = PKValue
   AND Other Columns still contain the values that I downloaded.
```

And therefore, it means remembering the downloaded value, usually referred to as the original value, of a field as the current value is being changed. In the preceding SQL, `Other Columns` might be as few in number as one, as in the case of a SQL Server `TimeStamp` column; or as many as all of the downloaded columns. The inclusion of the primary key in the `WHERE` clause means that one data modification statement must be submitted to

Download at WoweBook.Com

the host DBMS for each object whose state has changed, just as a `DataAdapter` generates one `INSERT`, `UPDATE`, or `DELETE` statement for each altered row.

Don't Miss the Windows Mobile Line of Business Solution Accelerator 2008

Every now and then you come across sample code that is very well written. We recommend that you check out the following sample, which includes lots of smart ways to do things that include, but are not limited to, LINQ. You can download this code at the following URL (or search the Microsoft Web site for "Windows Mobile Line of Business Solution Accelerator 2008"):

www.microsoft.com/downloads/details.aspx?FamilyId=428E4C3D-64AD-4A3D-85D2-E711ABC87F04&displaylang=en

The code on this site is extensive, solid, and well reasoned. In addition, the developers seem to think about business objects the same way we do, which means we think they are smart.

One Row, One Object Each row from each table will be converted into a business object as it reaches our application: one class for employee, one for customer, one for product, one for order, and one for order detail. Figure 7.8 shows each class. If it looks almost identical to the data set diagram shown earlier, it should. What is different between the two figures is the lack of relationships in Figure 7.8. Instead of hard-wired relationships between the classes, the relationships are defined by read-only properties, such as `Order.Details` and `Order.parentCustomer`. We will discuss these properties, and the use of LINQ in defining them, in greater detail later in this section; for now, we need to discuss the reason for avoiding hard-wired relationships in our application data.

Since our data might come from a different source at some future time, we need to be flexible and not tie our business objects to their data source. After all, a customer is a customer regardless of where her information is stored. For this reason, we will write a class that uses ADO.NET to execute the aforementioned stored procedure and process the rows that are returned, turning each into an object of the appropriate class and adding each object to an application data object. By adding every business object,

Download at WoweBook.Com

FIGURE 7.8: The Five Business Objects

such as the employee object, the objects for that employee's orders, and so on, to a single application data object, we can save all application data to device storage by using XML serialization to store the application data object to a file. For further information, see the XML Serialization section of Chapter 5, Storage.

But perhaps that XML serialization won't be as simple as we first think it will be; things never are. Or more correctly, since our application will benefit from having a single XML-serializable object that holds all of our data, perhaps we had best design that object first.

Since the application's data relates to a single employee, you might think that a single employee object that served as the head of a tree of related objects would be the best design. Unfortunately, this design has two snags.

1. First, the `Employee` object would be unserializable because the `Employee` object would have an `Orders` property, or a collection of references to `Order` objects. And each `Order` object would have a `parentEmployee` property containing a reference to the `Employee` object. Thus, the `Employee` object would contain circular references, and that would make it, as we mentioned in Chapter 5, unserializable.
2. Second, as we mentioned earlier in this chapter, all customers and all products must be downloaded to the application, even those that currently have no relationship with the employee. Thus, the `Employee` object would not be connected to all of the other objects needed by the application.

Download at WoweBook.Com

A second design choice would be to have a separate application data object, one that contained dictionaries and lists: a `Dictionary` containing the one and only `Employee` object, a `Dictionary` containing that employee's `Orders`, a `List` containing all the `Details` of those `Orders`, a `Dictionary` containing all the `Customers`, and a `Dictionary` containing all the `Products`. Since `Details` are always retrieved via their parent order and not by their primary key, they can be placed in a `List` rather than in a `Dictionary`, thus saving us the trouble of having to convert the `Detail`'s composite primary key into a single-valued dictionary key.

Each business object, such as `Order`, would contain no references, only its primary key and foreign key values from the database. The primary key values from the database would now be the `Dictionary` keys as well. There would still be an `Employee.Orders` property and an `Order.parentEmployee` property that expose object references, but they would be read-only properties that used LINQ to generate their values on the fly. Since the objects would not actually contain any reference fields, there would be no possibility of circular references. This is the design we will use, as shown in Figure 7.9.

We name our application data class, as we mentioned earlier, `AppData`. The code for the dictionaries and list that comprise this object is shown here:

```
public Dictionary<int, Employee> Employees;
public Dictionary<string, Customer> Customers;
public Dictionary<int, Product> Products;
public Dictionary<int, Order> Orders;
public List<Detail> Details;
```

As you can see, the application data object is very simple and very functional. It is also not XML-serializable. More specifically, its dictionaries are unserializable.

Several Web sites discuss writing your own XML-serializable dictionary class and provide sample code. We chose not to use them because we did not want to introduce any additional nonbusiness-specific classes, and because we wanted to stay with the standard .NET collection classes.

Rather than give up our application data class design, we will take advantage of the fact that, while instances of the `Dictionary` class are not XML-serializable, instances of the `List` class are, in fact, XML-serializable. When the time comes to serialize or deserialize our data, we will simply move each set of `Dictionary` data to or from a `List`. Using LINQ, each `List` can be easily generated from the corresponding `Dictionary`'s `Values`

Download at WoweBook.Com

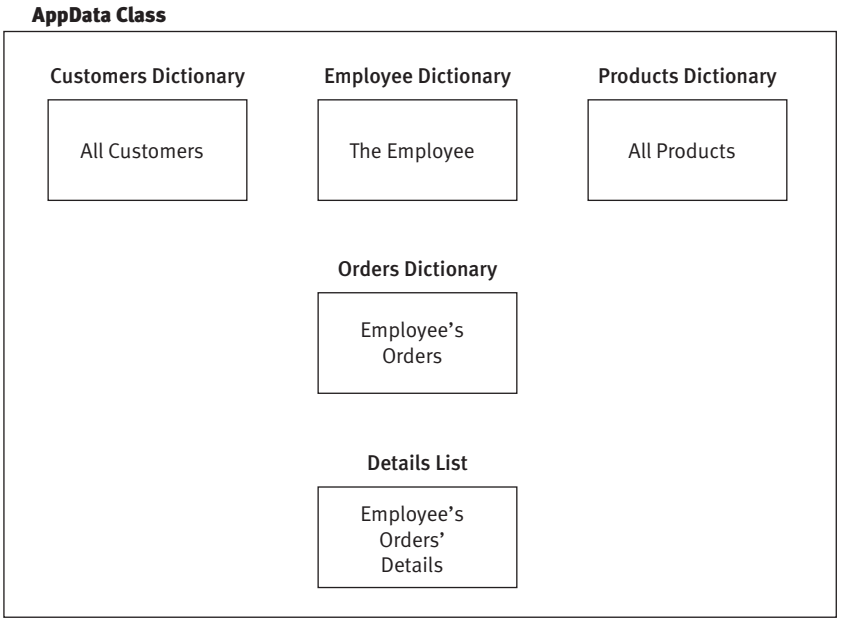


FIGURE 7.9: Structure of the Application Data Object

property by using the `ToList` operator. And each `Dictionary` can be generated from the corresponding `List` via the `ToDictionary` operator if each object's key, such as `OrderId`, is a property of the object. As long as the key is contained within the object, you can always use LINQ to generate a `Dictionary` from a `List` of the objects.

To perform the serialization of application data to and from a file, we write a class named `AppToXml`. This class does the following:

- Contains the application data in five `List` objects
- Serializes and deserializes itself to a file
- Transfers data between its `Lists` and the application's `Dictionarys`

Listing 7.8 shows the serialization code from this class.

LISTING 7.8: Serialization Code from `AppToXml` Class

```
using System;
using System.LINQ;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;
        :
```

Download at WoweBook.Com


```
        :
internal static void WriteAppData(string fullPath)
{
   AppToXml otx = new AppToXml();
   AppData oed = AppData.Current;

   otx.Employees = oed.Employees.Values.ToList<Employee>();
   otx.Customers = oed.Customers.Values.ToList<Customer>();
   otx.Products = oed.Products.Values.ToList<Product>();
   otx.Orders = oed.Orders.Values.ToList<Order>();
   otx.Details = oed.Details;

   FileStream fs = new FileStream(fullPath,
                                  FileMode.Create,
                                  FileAccess.Write);
   XmlSerializer xs =
      new XmlSerializer(typeof(AppToXml));
   xs.Serialize(fs, otx);
   fs.Close();
}
        :
        :
internal static void LoadAppData(string fullPath)
{
   FileStream fs = new FileStream(fullPath,
                                  FileMode.Open,
                                  FileAccess.Read);
   XmlSerializer xs =
      new XmlSerializer(typeof(AppToXml));
   AppToXml otx = (AppToXml)xs.Deserialize(fs);
   fs.Close();
   AppData oed = AppData.Current;
   oed.Clear();
   oed.Employees =
      otx.Employees.ToDictionary(empl => empl.EmployeeID);
   oed.Customers =
      otx.Customers.ToDictionary(cust => cust.CustomerID);
   oed.Products =
      otx.Products.ToDictionary(product => product.ProductID);
   oed.Orders =
      otx.Orders.ToDictionary(order => order.OrderID);
   oed.Details = otx.Details;
}
```

Again, note the use of the => operator. The phrase (`empl => empl.EmployeeID`) says that the `EmployeeID` property is to be treated as the function that returns the object's key value. Thus, for each `Employee` object, a reference to

Download at WoweBook.Com

the object will be placed in `empl`, and the value of `empl.EmployeeID` will be calculated and then used as the `Dictionary` key.

So, even with serialization, a subject not directly associated with LINQ, we find that we get help from LINQ's `ToList` and `ToDictionary` methods, which enable us to convert lists into dictionaries, and vice versa.

Now that we can move our application data to and from a local file, let us examine the objects that comprise this data more closely, especially the parts that use LINQ to Objects.

7.2.2.2 *The Business Objects*

There are five business object classes in our application: `Employee`, `Customer`, `Product`, `Order`, and `Detail` (each `Detail` object represents a line item on an order). We will show more code from the `Order` class than any other simply because it participates in more relationships than any other. And we will be focusing on the use of LINQ within some property definitions and on change tracking. The two subjects are related, as changing a property value can change the object's state. Just as a property's `set` accessor must validate the value being assigned, it may need to update the object's change state. Other actions may also result in an update to the

TABLE 7.2: State Table for Our Business Objects

Action	Old State	New State
Retrieve from host database	None	Unchanged
Create	None	New
Modify	New	No change
	Unchanged or Updated	Updated
	Deleted or Moot	Either no change or `InvalidOperationException`
Delete	New	Moot
	Deleted or Moot	Either no change or `InvalidOperationException`
	Unchanged or Updated	Deleted

Download at WoweBook.Com

object's change state. As Table 7.2 shows, these changes in state are not as straightforward as one might first expect.

`Moot` objects are objects that exist in neither the application nor the host database; they result from a new object being deleted before the application pushes changes to the database, as in "the user changed her mind." Data from `Moot` objects is never passed to the host. `Deleted` objects exist in the database but not in the application. They result in a `DELETE` statement being passed to the database during synchronization.

A `New` object is one that exists in the application but not on the host. It results in an `INSERT` statement being passed to the database. Therefore, modifying a `New` object does not make it an `Updated` object; it remains a `New` object. An `Updated` object is one that is out of sync with the database, and it results in an `UPDATE` statement being passed to the database.

The state names are our own and are similar to those used by the `DataRow` class. They are defined in the application as a set of enumerated values:

```
[Flags()]
public enum ObjectState
{
   Unchanged = 1,
   New = 2,
   Upated = 4,
   Active = 7,    // Unchanged, New, or Updated.
   Deleted = 8,
   Moot = 16,
   NotActive = 24  // Deleted or Moot.
}
```

To help determine which objects are active or inactive, we wrote a trivial routine to examine an `ObjectState` value and determine whether it is `Active` or `NotActive`:

```
internal static bool IsActive(ObjectState state)
{
   return (state & ObjectState.Active) != 0;
}
```

Since change in an object's state is, in part, tied to changing its property values, we will begin our examination of our business object code with properties.

Download at WoweBook.Com

7.2.3 Business Object Properties

The properties of our business objects fall into two general categories: read-write properties that are backed by a private field, such as the `Order.OrderDate` property; and read-only properties, or properties whose values are generated on the fly rather than being backed by a field. In our application, read-only properties are used to reflect the relationships between objects, such as the `Order.Details` property, and to expose calculated values, such as `Order.NetValue`. LINQ code will be used to generate property values in both scenarios.

7.2.3.1 *Read-Write Properties*

It is the read-write group of properties that needs to validate data and update state. To simplify the coding of both requirements, we write two routines for setting property values: one for `ValueType` data and one for `Strings`. The `ValueType` helper routine is a generic routine that allows the user to specify a value plus optional minimum and maximum values for validation, plus pointers to the current and original value fields. It is shown in Listing 7.9.

Listing 7.9: One Routine for Setting Property Values

```
internal static void SetValueProperty<T>(
                                string fieldName,
                                T value,
                                T minAllowable,
                                T maxAllowable,
                             ref T currValue,
                             ref T prevValue,
                             ref ObjectState state)
{
   if (  state == ObjectState.Deleted
      || state == ObjectState.Moot)
   {
      throw new InvalidOperationException
         ("A deleted object cannot be modified.");
   }

   if (value != null)
   {
      IComparable icompValue = (IComparable)value;
      if ((icompValue).CompareTo(minAllowable) < 0
         || (icompValue).CompareTo(maxAllowable) > 0)
      {
         throw new ArgumentOutOfRangeException(
            string.Format(
```

Download at WoweBook.Com


```
                "{0} must be between {1} and {2}",
                fieldName,
                minAllowable, maxAllowable));
         }
      }

      prevValue = currValue;
      currValue = value;

      if (state == ObjectState.Unchanged)
         state = ObjectState.Updated;
   }
```

The `IComparable.CompareTo` routine must be used, as there is no generic implementation of ">", "<", or "==".

The `String` equivalent, obviously, is a nongeneric routine. It uses an optional regular expression parameter for validation rather than minimum and maximum values. Other than that, it bears a strong resemblance to the `ValueType` setting routine. Our `SetStringProperty` method appears in Listing 7.10.

Listing 7.10: A Second Routine for Setting Strings in Read-Write Property Values

```
internal static void SetStringProperty(string fieldName,
                                       string value,
                                       string regex,
                                   ref string currValue,
                                   ref string prevValue,
                                   ref ObjectState state)
{
         :
         :
   if (!string.IsNullOrEmpty(regex)
    && !new Regex(regex).IsMatch(value))
   {
      throw new ArgumentException(
         string.Format(
            "{0} must match this pattern:  {1}",
            fieldName, regex));
   }
         :
         :
}
```

The two regular expressions are used to validate customer IDs and employee names. The latter are restricted to the U.S. English characters:

Download at WoweBook.Com


```
internal static string
   regexpCustomerID = @"^([A-Z]{5})$",
   regexpUSName =
         @"^[a-zA-Z]+(([\ \,\.\-][a-zA-Z])?[a-zA-Z]*)*$";
```

We are indebted to www.RegExpLib.com and one of its contributors, Hayk A, for providing the source of these expressions. Setting a read-write property value is now a matter of calling the appropriate helper routine, as illustrated by the code excerpt from the `Order` class shown in Listing 7.11.

LISTING 7.11: Calling Our Helper Routine to Set `CustomerID` Values

```
private string _CustomerID, _origionalCustomerID;
public string CustomerID
{
   get
   {
      return _CustomerID;
   }
   set
   {
      OrderEntryData.SetStringProperty("CustomerID",
                              value.Trim(),
                              OrderEntryData.regexpCustomerID,
                           ref this._CustomerID,
                           ref this._origionalCustomerID,
                           ref this.objectState);
   }
}


private DateTime _OrderDate, _origionalOrderDate;
public DateTime OrderDate
{
   get
   {
      return _OrderDate;
   }
   set
   {
      OrderEntryData.SetValueProperty("OrderDate",
                              value,
                              DateTime.Today.AddYears(-20),
                              DateTime.Today,
                           ref this._OrderDate,
                           ref this._origionalOrderDate,
                           ref this.objectState);
   }
}
```

Download at WoweBook.Com

We suggest you keep two things in mind regarding property value validation. First, data that is pulled from the database will be validated as the objects containing that data are created, and that data must be able to pass validation. For instance, our allowable `OrderDate` range might seem a little generous, but the Northwind database is well more than ten years old with many of its orders dating back to the mid-1990s. Those orders must successfully download into our application, hence the resulting looseness of our validation criteria. If different validation rules are to apply for new data vis-à-vis downloaded data, the validation routine will need the ability to differentiate between the two.

Second, few things make a developer feel more foolish than providing a default value that fails validation. It's an easy mistake to make, such as requiring a transaction amount to be a positive measurable amount and then supplying a default value of $0.00.

7.2.3.2 *Read-Only Properties*

Our second set of properties, the read-only properties, provides references to related objects and calculated values. For instance, `Order.NetValue` uses LINQ to iterate through the `Details` dictionary, selecting the details of a specific order and returning the sum of their values, whereas `Order.parentCustomer` returns a reference to the order's customer object, an object that is located in the `Customers` dictionary within our application data object. `Customer.Orders` returns not a single reference but rather a `List` of all of the customer's order objects from the `Orders` dictionary, using LINQ to do so.

Remember from an earlier discussion that our business objects do not store references to other objects, for doing so would lead to unserializable objects. Rather, they use the object's primary key and foreign key value properties, such as `CustomerId`, to obtain the related object references from the `Dictionarys`.

The `Order` class has four read-only properties: `parentEmployee`, `parentCustomer`, `Details`, and `NetValue`. Listing 7.12 shows the three that are related to exposing object references.

Listing 7.12: Accessors for Three Read-Only Properties

```
[XmlIgnore()]
public Customer parentCustomer
{
   get
   {
```

Download at WoweBook.Com


```
      return
         OrderEntryData.Current.Customers[this.CustomerID];
   }
}

[XmlIgnore()]
public Employee parentEmployee
{
   get
   {
      return
         OrderEntryData.Current.Employees[this.EmployeeID];
   }
}

[XmlIgnore()]
public List<Detail> Details
{
   get
   {
      return
         (OrderEntryData.Current.Details
            .Where(detail =>
               OrderEntryData.IsActive(detail.objectState))
            .Where(detail =>
               detail.OrderID == this.OrderID))
            .ToList();
   }
}
```

Again, a comment on alternative syntax: If you prefer the query expression syntax, you could write the `Details`' `get` accessor as shown here:

```
Return (from d in OrderEntryData.Current.Details
where OrderEntryData.IsActive(d.objectState) &&
      d.OrderID == this.OrderID
select d)
.ToList();
```

The `XmlIgnore` attribute that is applied to each property is necessary to prevent the property from serializing. We want our object's read-write properties to serialize, for they contain actual data. But our read-only reference-exposing properties are, as mentioned earlier, circular in nature. XML-serializing them is something that we must avoid.

The two `parent`… properties are so simple that they do not need any LINQ code. The LINQ code in the `Details` property accesses the `Details`

Download at WoweBook.Com

dictionary in the application data object, retrieving all active details for this order.

LINQ code is also used in calculating the read-only `NetValue` property of the order, summing the net values of the order's details and returning the total, as shown here:

```
return
   this.Details
      .Where(od => (OrderEntryData.IsActive(od.objectState)))
      .Sum(detail => detail.NetValue);
```

Note that this is an example of a LINQ expression that returns a single value, not a collection of objects.

Properties comprise most of the code contained within our business objects, with a few constructors and default value literals (neither of which require any LINQ code) thrown in.

LINQ also appears in our UI code. The most complex LINQ statement in our application was the one, presented at the start of this section and repeated in Listing 7.13, that displays information in a combo box. It is nearly identical to the one used in the LINQ to Datasets example except that it accesses a dictionary rather than a data table and uses a `where` clause to filter out any inactive orders.

Listing 7.13: LINQ in Support of User Interface Code

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
(
   employee.Orders
      .Where(order =>
         (OrderEntryData.IsActive(order.objectState)))
      .OrderBy(active =>
         active.parentCustomer.CompanyName)
         .ThenByDescending(active => active.NetValue)
         .ThenByDescending(active => active.OrderDate)
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text =
                   sorted.parentCustomer.CustomerID
                + " - "
                + sorted.NetValue.ToString("c")
                          .PadLeft(9).Substring(0, 9)
                + " - "
```

Download at WoweBook.Com


```
                + sorted.OrderDate
                          .ToString("dd-MMM-yy")
        })
)
.ToList();
```

The code, in summary, does the following:

1. Extracts all the active orders from the `Orders` read-only property of the `Employee` object
2. Sorts them into descending order date within descending net value within customer sequence
3. Converts each order into an object of an anonymous class containing
   a. An order ID property named `Id`
   b. A new property, named `Text`, that is the concatenation of the customer ID / net value / order date property
4. Converts that collection into a `List`
5. Assigns that `List` to the data source property of a `ComboBox`, specifying that the `Id` property of the newly created objects will be treated as the key value, and that the `Text` property value will be displayed in the `ComboBox`

The results of this code are shown in Figure 7.10. As we would expect, it looks identical to Figure 7.5, for although we are using LINQ against a generic collection instead of a data table, the LINQ expression is essentially the same.

FIGURE 7.10: Display of LINQ Assembled Data

Download at WoweBook.Com

As before, had we wanted to sort on a property of the newly created anonymous objects, such as the `Id` or `Text` property, our sort would need to occur after the anonymous objects were constructed, and the code would have resembled the code shown in Listing 7.14.

Listing 7.14: Sorting on Anonymous Objects

```
cbxOrders.DataSource =
   (Program.appData.Employees[testEmployeeID].Orders
      .Where(order =>
         (DataOrderEntry.IsActive(order.objectState)))
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text = sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                      .PadRight(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate.ToString("dd-MMM-yy")
         })
      .OrderBy(item => item.Text))
      .ToList();
```

It is interesting to note that the source collection for the LINQ queries shown in Listings 7.13 and 7.14, the `Employee` object's `Orders` property, is itself a LINQ expression. In that regard, it is similar to the `Order` object's `Details` property shown in Listing 7.12. Thus, the code shown in Listing 7.14 is an example of a LINQ expression whose complete definition is divided between two separate C# statements.

Even the code used to submit changes back to the remote database benefits from LINQ, if only slightly. For example, the code that loops through all the orders looking for deleted ones and then sets a parameter value and submits the "execute delete stored procedure" command, goes from

```
foreach (Order order in context.Orders.Values)
{
   if (order.objectState == ObjectState.Deleted)
   {
      cmnd.Parameters.AddWithValue("@OrderID",
                                   order.OrderID);
      cmnd.ExecuteNonQuery();
   }
}
```

Download at WoweBook.Com

to

```
foreach (Order order in context.Orders.Values
            .Where(order =>
               order.objectState == ObjectState.Deleted))
{
   cmnd.Parameters.AddWithValue("@OrderID",
                                order.OrderID);
   cmnd.ExecuteNonQuery();
}
```

This code is the last LINQ code in this version of our application.

Throughout this section, we used LINQ to Objects to access collections of objects; filtering, sorting, aggregating, converting, and performing other operations, all within a single statement—occasionally a rather long statement but always a single statement. As we wrote these statements, we received IntelliSense help, allowing us to select our operations, see the available properties, and invoke .NET methods or our own methods as we went. In all, we used LINQ to define properties, bind data, transform objects in preparation for XML serialization, and locate unsynchronized data.

Business objects made our application more functional, modular, and representative of the business; and although LINQ to Objects made much of that code easier to write, we still needed to write significantly more code in the LINQ to Objects version than we did in the LINQ to Datasets version, especially in the area of change tracking and data synchronization. What makes the writing of this additional code even more disconcerting is the fact that we know this code has already been written by others.

Whoever designed and authored the `System.Data`, and related namespace, classes wrote change tracking and data synchronization logic, for we can see it in the `DataRow` and `DataAdapter` classes. Whoever wrote the `DataContext` class for LINQ to SQL on the desktop wrote change tracking and data synchronization logic. We would like to avoid reinventing the wheel and instead have an application that provides the benefits of business objects with already proven change tracking and data synchronization capability, thus producing the most functional application while writing the smallest amount of code. And that leads us to the hybrid version.

7.2.4 The Hybrid Version

We begin with the realization that each of our business objects was created from a row in the database and that each object's data consisted of individual

Download at WoweBook.Com

fields that were exposed as properties. Why, then, not load the data into a data set and use the column values of the rows as the backing fields for the object's read-write properties?

In this scenario, each object would contain the reference to its corresponding data row and nothing more. To put it another way, the object would be, in effect, data-bound to the data row. Any value assigned to a property of the object would be stored in the matching field of the data row, from whence it would be retrieved whenever the property value was accessed. No data would be duplicated, as the business objects would be holding a reference to the data row, not a copy of data. And the data rows would provide change tracking on behalf of the objects.

In addition, the application's data could be written to a file by serializing the data set rather than the application's data object. The application's data object, containing its collections of customers, orders, and so on, would still be needed, for we want the benefit of an object's read-only properties, such as `Order.Details` and `Order.parentCustomer`, and the data validation benefit that comes with the read-write properties. But this new application data object would no longer need to be serializable. Having an application data object that does not need to be serialized provides greater design flexibility and requires less code.

In this scenario, our business objects would provide data binding for the UI and the source of all business logic. The data set would provide serialization of application data, change tracking, and the uploading of data changes back to the remote data source, thus achieving our original application goals, shown in the following list, while requiring the least amount of programming on our part.

1. Retrieve all of the employee's orders from the host SQL Server.
2. Add, delete, and modify orders.
3. Save/restore all application data to a local file.
4. Upload all entered information to the host SQL Server.

Figure 7.11 shows the updated schematic diagram, representing this hybrid version of our application.

So, let us now examine the code reductions in this hybrid version of our application. The data would be read into a data set, saved to a file, and returned to the source database, using the same code that was in the LINQ to Datasets version. The definition of the business objects and the application data object remains mostly the same. The constructors of the business

Download at WoweBook.Com

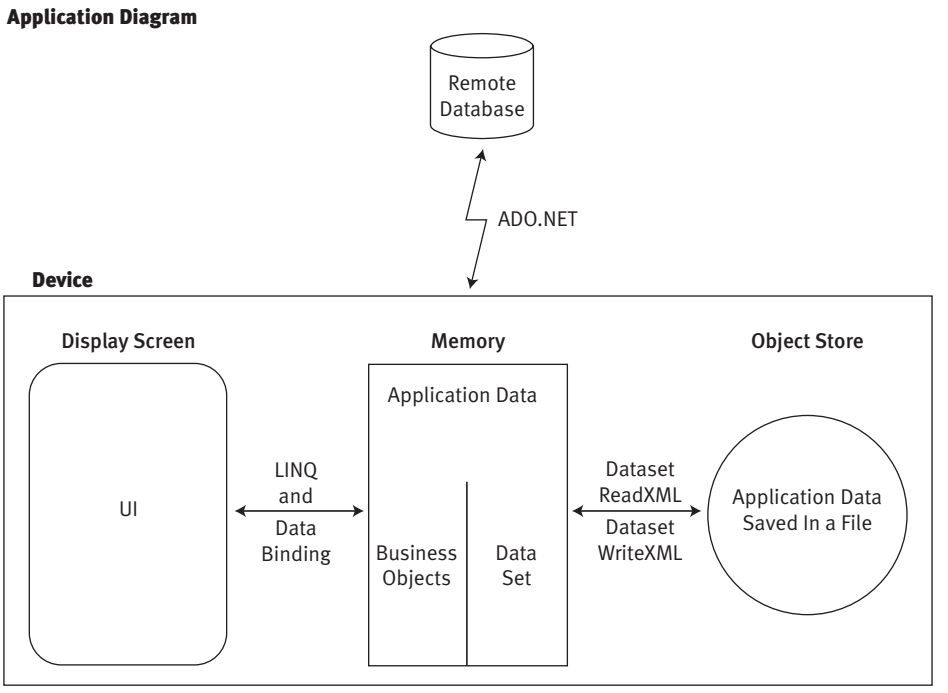


FIGURE 7.11: Sample Application Diagram: Hybrid Version

objects change, as they are now created from data rows rather than from a data reader, and they store a reference to the row rather than its data. For instance, the `Order` constructor for creating an `Order` from a data row is as follows:

```
public Order(DataRow rowThis)
{
   this.rowThis = rowThis;
}
```

And the constructor creating a new order, which must create a new data row and then fill it with default values, is as follows:

```
public Order()
{
   DataTable myTable =
      AppData.Current.dsetEmployee.Tables["Orders"];
   rowThis = myTable.NewRow();
   this.OrderID =
      AppData.Current.Orders.Values
         .Max(order => order.OrderID) + 1;
   this.EmployeeID = AppData.Current.EmployeeID;
```

Download at WoweBook.Com


```
    this.CustomerID = string.Empty;
    this.OrderDate = Order.defaultOrderDate;
    this.RequiredDate = Order.defaultRequiredDate;
    this.ShippedDate = Order.defaultShippedDate;
}
```

And the `OrderDate` read-write property, for instance, backed by the matching field in the data row, becomes the following:

```
public DateTime OrderDate
{
    get
    {
        return (DateTime)rowThis["OrderDate"];
    }
    set
    {
        AppDataUtil.ValidateValueProperty(
                                "OrderDate",
                                value,
                                DateTime.Today.AddYears(-100),
                                DateTime.Today);
        rowThis["OrderDate"] = value;
    }
}
```

The read-only properties of the `Order` class, which use LINQ to select information from the application's collections, such as `Order.NetValue`, remain unchanged, as shown here:

```
public decimal NetValue
{
    get
    {
        return this.Details.Sum(detail => detail.NetValue);
    }
}
```

All serialization code, and all change tracking/data source update code, is gone, replaced by the capabilities inherent in the data set, data table, and data adapter classes.

Certainly, we have been able to achieve the best of both worlds: ADO.NET for retrieving, tracking, and persisting data, and business objects for validation, encapsulation, and use of the user interface with that data. All we need now is a nice, flexible way to send, receive, and store this information as XML to, from, and on our device.

Download at WoweBook.Com

7.3 LINQ to XML

Since the earliest planning stages of .NET, Microsoft has been authoring classes to provide XML support for developers, introducing new classes, and refining old ones to keep pace with the ever-evolving World Wide Web Consortium specifications. With the release of .NET 2.0, developers were presented with an enormous amount of XML support, as Table 7.3 illustrates. This raises the obvious question: Why add even more XML support?

The answer is this: Microsoft is committed to providing LINQ support for collections of objects, including collections of XML objects; that is, support for XML documents. And since an XML document is a collection of XML objects—attributes, elements, directives, comments, and so on—LINQ to XML became a highly desirable feature, and highly desirable features make it into the released product.

TABLE 7.3: XML Support in .NET 2.0

Feature	Classes	Comments
Readers and writers	`XmlReader`, `XmlWriter`	Reads/writes XML to and from a stream one node at a time. Tedious to program for. Used by other classes to retrieve and store XML
Schema validation	`XmlReader`	Warns of, or rejects, incoming XML that does not comply with the specified schema
Document objects	`XmlDocument` or `XPathDocument`, plus various node classes and other supporting classes	Holds an entire XML document or fragment in memory. Provides XPath querying of the document. May provide for modification of document contents
XSLT transformations	`XslTransform`	Provides for the transformation of an XML document into XML of a different format or into HTML or into any text format
XML serialization	Generic classes	Provides for object content to be serialized/deserialized to and from a stream

Download at WoweBook.Com

Unlike LINQ to Datasets, which did not require the development of new ADO.NET classes, LINQ to XML did require that new XML classes be written. If you think back to our discussion of LINQ to Datasets, you will remember that our LINQ queries were performed against data tables, the same `DataTable` class that existed in .NET 2.0. The simple rectangular structure of a `DataTable` combined with its already existing implementation of `IEnumerable` (actually, its implementation of the even more LINQ-able `IListSource` interface) made it a natural for LINQ implementation.

The same cannot be said for the `XmlDocument` and `XPathDocument` classes. First, their hierarchical structure is more complex than relational, preventing them from providing a generic enumeration. And second, their specific enumerations have already been defined and not by Microsoft. These enumerations have been defined by the World Wide Web Consortium, and developers know them as *axes,* such as `Descendant`, `Child`, and `Ancestor-or-self`. Therefore, new classes that paralleled the old XML document and XPath classes and that provided LINQ syntax for making XPath selections were added to .NET.

These classes, referred to as the X classes, duplicate some capabilities that already existed in .NET 2.0, such as loading an XML document from a stream into a document object, saving an XML document from a document object to a stream, and querying/modifying the nodes of a document. What the X classes add to .NET is LINQ syntax for specifying XPath queries and very flexible constructors. That might not seem like much of an addition, but it is. To see why, consider that `XDocuments` and `XElements` can be constructed from the following:

- An incoming XML stream
- Literal values
- An object
- A collection of objects
- A collection of objects that each contains collections of objects
- The output from a LINQ to Objects query

It is a somewhat deliberately redundant list, but it all comes down to this: It is easier than it has ever been for a smart-device application to selectively gather and arrange any information that exists anywhere in that application and deliver it to a remote recipient, or to the device store as XML in any format. It is a strong claim, so we best support it with some code.

Download at WoweBook.Com

The first capability we want to illustrate is constructing an `XElement` object from literal values. It is not a common need, but the code to do it most clearly illustrates the creation of inner elements and attributes as part of the total construction. Listing 7.15 shows the code for creating an `Employees` element that contains three child `Employee` elements, each of which contains one attribute and three child elements.

Listing 7.15: Creating an `Employees` Element

```
XElement xmlEmployees =
   new XElement("Employees",
      new XElement("Employee",
         new XAttribute("EmployeeId", "10"),
         new XElement("FirstName", "John"),
         new XElement("MiddleInitial", "B"),
         new XElement("LastName", "Durant")),
      new XElement("Employee",
         new XAttribute("EmployeeId", "11"),
         new XElement("FirstName", "Susan"),
         new XElement("LastName", "Shank")),
      new XElement("Employee",
         new XAttribute("EmployeeId", "12"),
         new XElement("FirstName", "John"),
         new XElement("LastName", "Basilone"))
      );
```

If we subsequently wished to expand our `xmlEmployees XElement` object into an `XDocument` object, perhaps because we were about to send it to a remote recipient and wanted to add a declaration, we would simply do this:

```
XDocument docEmployees =
   new XDocument(
      new XDeclaration("1.0", "utf-8", "yes"),
      xmlEmployees);
```

If we want to create a new `XElement` object whose child elements consist of only those employees who have a middle initial, we execute a LINQ to XML query against `docEmployees` and construct a new document from the result. The LINQ to XML query to do this, and place the result into an `XDocument` named `xmlEwMI`, is as follows:

```
XElement xmlEwMI = new XElement("EmployeesWithMIs",
   xmlEmployees
      .Descendants("Employee")
         .Where(EwMI =>
            EwMI.Element("MiddleInitial") != null));
```

Download at WoweBook.Com

The corresponding XPath syntax, `"//Employee[MiddleInitial]"`, is certainly more concise but not necessarily easier to understand or maintain. In general, LINQ to XML is easier to read, and XPath is more concise.

Now let's change our requirement a little bit. Instead of creating new XML from already-existing XML, as we have just done, we'll create an `XElement` from the output of a LINQ to Objects query. Think back to our application data, where we have an `Employee` object whose `Orders` property contains the collection of that employee's orders. We want to query that collection to obtain all the orders that shipped late, and store the result in an X object. In doing so, we want to specify which `Order` properties are to be captured and also specify what their XML types, attribute or element, and names will be. In addition, we want the orders to be sequenced by descending value. Once we have captured the information as XML, we wish to write it to a file in device storage. Listing 7.16 shows the code for doing this.

Listing 7.16: Selecting and Persisting Late Orders as XML

```
Employee employee = AppData.Current.Employees[EmployeeID];
string XmlFileName = @"My Documents\LateOrders.xml";

XElement xmlLateOrders;
xmlLateOrders =
   new XElement(
      "LateOrders",
      employee.Orders
         .Where(order => order.ShippedDate > order.RequiredDate)
         .OrderByDescending(order => order.NetValue)
         .Select(order => new XElement(
            "Order",
            new XAttribute("OrderID", order.OrderID),
            new XAttribute("CustomerID", order.CustomerID),
            new XAttribute("Value", order.NetValue),
            new XElement("Ordered", order.OrderDate),
            new XElement("Required", order.RequiredDate),
            new XElement("Shipped", order.ShippedDate))
            )
         );

XDocument docLateOrders =
      new XDocument(
         new XDeclaration("1.0", "utf-8", "yes"),
         xmlLateOrders);

docLateOrders.Save(XmlFileName);
```

Download at WoweBook.Com

When we run the application, we set a break point at the last line of code shown in Listing 7.16. When the application enters break mode, we use the XML visualizer to look into `docLateOrders` and see the results, as shown in Figure 7.12.

Of all the features provided by LINQ to XML, the one that is illustrated in Listing 7.16 is the most beneficial and the greatest leap forward. No matter what application data must be exported, and no matter what XML format it must be in, you can provide it. And that is a nice capability indeed.

So, given the list of features mentioned in Table 7.1 plus the new features that we just covered, which ones should you focus on? Which ones are you must likely to use in your application? Our recommendation is this: If the XML that your application will be sending and receiving is of a known and unchanging syntax, use XML serialization and, if helpful, the supporting tools, such as XSD.EXE. If the schema of the data is unknown or unpredictable at design time, or if the subset of data that will be used cannot be determined until runtime, use LINQ to XML and LINQ to Objects. One way or the other, your application will be able to send, receive, and persist the XML data that other applications provide or require.

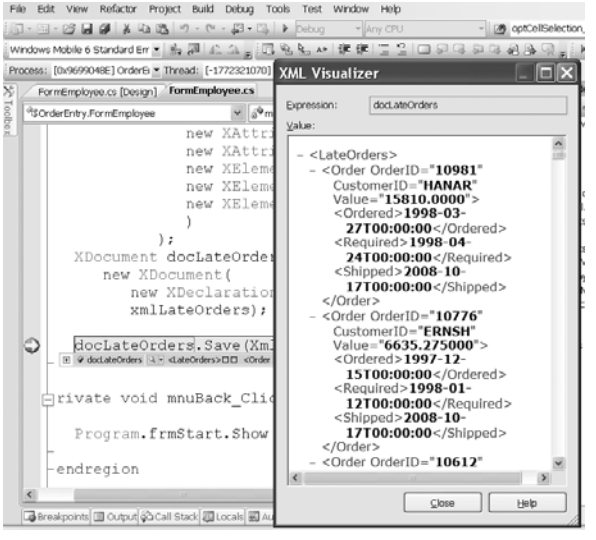


FIGURE 7.12: XML Generated from LINQ to Objects Query

Download at WoweBook.Com

7.4 Conclusion

In this chapter, we looked at LINQ to Datasets, LINQ to Objects, and LINQ to XML, but really we looked at LINQ for the Compact Framework application. We saw how LINQ provides for the manipulation of collections, including filtering, ordering, transforming, merging, and aggregating objects. We saw its benefit in preparing data for data binding, generating property values on the fly, searching for objects whose state has changed, transforming collections in preparation for serialization, and handling application data as XML. LINQ, like the collections that it works on, is everywhere in .NET. Taking advantage of LINQ can help you produce the maximum amount of application functionality from the minimum amount of code.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

8
Synchronizing Mobile Data

In Chapter 6, ADO.NET Programming, we covered managing and displaying data on a Windows CE device using ADO.NET and SQL Server CE. In this chapter, we expand outward to retrieve that data from a desktop SQL Server and post it back to the desktop after it has been modified on the device.[1]

THREE THINGS MAKE UP THE CORE of a typical SQL Server CE application scenario: a Windows CE device, some data to be stored, and a user who is most likely—though not necessarily—mobile. While scaled down from its desktop and server counterparts, SQL Server CE supports a very capable database engine that can easily support a single stand-alone database.

In addition, the SQL Server CE database engine has built-in support for sharing data between a locally stored database and a remote SQL Server database. This support, and how you can use it to move data between a SQL Server CE database and a SQL Server database, is the subject of this chapter.

By itself, SQL Server CE is a single-user database engine. But it can participate in schemes to share a central database with other users through support of the server-side database engine, SQL Server. For example, a company might equip each member of its sales force with a Windows CE device. When a salesperson is visiting customers, a SQL

1. Some material in this chapter first appeared in an article we wrote, "SQL Server CE: New Version Lets You Store and Update Data on Handheld Devices," *MSDN Magazine* 16(6): June 2001; available at http://msdn.microsoft.com/msdnmag in the Back Issue Archive. Reprinted with permission of *MSDN Magazine.*

Download at WoweBook.Com

Server CE database could operate as a stand-alone database engine. When a salesperson comes back to the office (or dials into the central system via modem or remote Internet connection) the device-based database can share updates with the central database while also receiving updates back from the central server.

In this chapter, we look at synchronizing mobile data in general and at three specific solutions available to .NET Compact Framework applications: Remote Data Access (RDA), Merge Replication, and Data Synchronization Services. The first two of these solutions, RDA and Merge Replication, synchronize SQL Server CE data with SQL Server data only, while Synchronization Services are more flexible, allowing for a wider variety of server-side data stores.

Applications that connect to a server, retrieve data, disconnect, modify data locally, reconnect, and synchronize the modifications made at the device and at the server are often referred to as *Occasionally Connected Applications* (OCAs). Two elements are central to these types of applications: a central data store maintained external to the device and a transport mechanism for moving the data between the Windows CE device and the central data store.

Regardless of which of the three synchronization solutions that are covered in this chapter is used, the connection between SQL Server CE and the desktop SQL Server is established using Internet-standard protocols. The server side of the connection requires that you configure a Web site and install whatever additional components are necessary for the solution that you choose. Those components send and receive data via HTTP, translate that data into SQL Server commands, connect to the SQL Server, submit the commands, and return the results to SQL Server CE on the device.

The device side of the connection can be established using any of several methods, provided that the end result is a TCP/IP connection to the Web site that hosts the connection to the central database. For example, a device could be connected to a desktop system using ActiveSync 4.5, which has built-in support for providing network access to a connected device. A device can also connect to a server using a dedicated wired or wireless network card. A connection can be accomplished using a modem for servers that are visible through the Internet or a modem with a virtual private network (VPN) connection for servers that are protected behind a network firewall.

Download at WoweBook.Com

Terminology: SQL Server CE and SQL Server

For the remainder of this chapter, we use "SQL Server CE" to designate SQL Server Windows CE Edition. Unless otherwise noted, we use "SQL Server" to refer to the versions of SQL Server that run on the Windows desktop and server operating systems. (Older versions of SQL Server, versions 6.5 and 7.0, have some limitations when interfacing with devices. We will mention these limitations as we encounter them in this chapter.)

8.1 Understanding SQL Server CE Synchronization

All synchronization solutions share common characteristics and problems. It is worth thinking about these for a moment for their very universality influences your development environment and your code. They show up in questions that wizards will ask you, in generated code, and in the code you write yourself. These characteristics are as follows.

1. Changes to data must be tracked at both the device and the server.
2. Changes to data will be made one row at a time. Multiple changes may be submitted within one batch and they may be applied within one transaction, but there will be one SQL statement for each row being modified.
3. A corollary of statement 2: Every table must have a key, be it the primary key specified by the database designer or some other key, such as a GUID column added by Merge Replication setup.
4. Normally, a subset of server data, not the entire set of server data, is maintained at the application.
5. A corollary of statement 4: Data synchronization statements can fail, even if your database design and application logic are sound. See Table 8.3 for some reasons and examples.
6. Data on the device is unprotected on the server while it resides on the device; that is, other applications can alter the server-side version of a row that is concurrently being modified on the device.
7. A corollary of statement 6: Data synchronization statements can fail, even if your database design and application logic are sound. See Table 8.4 for reasons.

Download at WoweBook.Com

As statements 5 and 7 imply, things can go wrong, even if you do things right. You can take steps to minimize the probability of synchronization failure—good design of data ownership, for instance—but you can never eliminate the inevitable. As you evaluate possible data synchronization solutions for your application, examine how each one detects conflicts, notifies your application of those conflicts, provides for automatic resolution of conflicts, and provides for manual resolution of conflicts. Merge Replication and Synchronization Services, for instance, differ greatly in this regard; not in what they can do but in how they do it.

Throughout this book, we have talked about handling errors. But in the remote data environment, it is always best to avoid errors rather than handle them. If the data store on either side of a synchronization rejects a change made at the other side, your application must decide whether it wants to impose that change, make a new change, or live with the rejection. Either application intelligence or human intelligence, or both, must be invoked. Both have a cost associated with them. We address the topic of avoiding synchronization errors later in this chapter, in the Using Good Design to Avoid Synchronization Failures subsection.

8.1.1 Three Synchronization Mechanisms

All three mechanisms—RDA, Merge Replication, and Data Synchronization Services—are provided to move data and schema between SQL Server CE and SQL Server. All are used in conjunction with a Web server based on Microsoft's Web server technology, Internet Information Services (IIS). The complex part of any SQL Server CE application is the transfer of data between SQL Server CE and SQL Server, especially if you have never used synchronization in your development or have never developed IIS-based applications.

One of the design benefits of all three solutions is that they move both data and schemas. This transfer of data and schemas can be as broad as an entire database or as narrow as a subset of a table. This control of schemas resides on the SQL Server, and in most cases eliminates the need for explicit data definition statements in SQL Server CE applications altogether.

The primary benefits of RDA are its smaller footprint on the device and its somewhat faster speed due to the reduced functionality and tracking that it provides. The key disadvantages of RDA are that it requires more code, both ADO.NET code as well as SQL code; it provides little support for handling conflicts; it is not an established database interface, being

Download at WoweBook.Com

used only in SQL Server CE; it is old; and it is scheduled for deprecation. Its major benefit is its simplicity.

Merge Replication, on the other hand, can be set up by a database administrator, requires very little code, and allows for the establishment of database update rules, including rules for conflict resolution when synchronizing new data into an existing SQL Server database. While RDA is supported for all versions of SQL Server, Merge Replication requires SQL Server 2000 or later.

Data Synchronization Services is more developer-oriented and less administrator-oriented. It can synchronize your SQL Server CE data with data stores other than SQL Server. Being a programmable service, it can provide the greatest flexibility error handling and the application of business logic but at the cost of potentially greater coding effort.

8.1.2 IIS Connectivity

The SQL Server CE team chose to use IIS as its primary conduit for several reasons. A very important segment of the user base consists of mobile users, so the connectivity options needed to be as widespread as possible. By building SQL Server CE around HTTP connectivity, wherever users can gain access to an Internet connection to browse the Web they can also connect to their SQL Server home world. In addition to its widespread availability, an IIS connection also provides authentication and authorization so that a user's data can get through a corporate firewall. And finally, IIS provides the ability to encrypt data when using a certificate (i.e., HTTPS access) so that private data can remain private.

ActiveSync and SQL Server CE

No discussion of connectivity to a Windows CE device would be complete without some mention of ActiveSync. ActiveSync describes a mechanism for moving data between a Windows workstation and a Windows CE device. For example, ActiveSync is the primary mechanism for moving e-mail messages, appointments, task lists, and contact information between a Windows CE device and a desktop. It's important to be aware that SQL Server CE does not use ActiveSync for the transmission of data. But when installing SQL Server CE on a device such as a Pocket PC or a handheld PC, ActiveSync must be present to assist in this operation.

Download at WoweBook.Com

When data is transmitted between SQL Server CE and SQL Server, it is automatically compressed for its trip between the CE device and the HTTP site. In one benchmark, using the Northwind database, the data was compressed to approximately one-eighth of its original size.

While compression is always enabled, encryption is not. However, you can enable encryption of the data being transferred between SQL Server CE and the IIS Server by using the standard encryption capability of IIS. This encryption is entirely transparent to the two database engines.

8.1.3 Database Connectivity

While IIS is used for data transmission, SQL Server CE components are used at both ends of the transmission for database connectivity. SQL Server CE has two components for this: the SQL Server CE Client Agent runs on the device, and the SQL Server CE Server Agent runs on the desktop. Thus, five components are involved in the transfer of data between SQL Server and SQL Server CE (see Figure 8.1):

- On the device:
  - SQL Server CE
  - The SQL Server CE Client Agent (RDA and Merge Replication)
- On the server(s):
  - The SQL Server CE Server Agent (RDA and Merge Replication)
  - The IIS Server
  - SQL Server

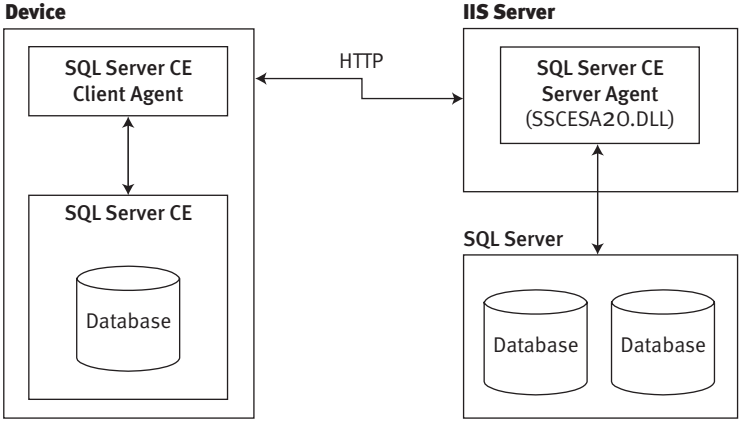


FIGURE 8.1: The Five Components of Remote Data Connectivity

Download at WoweBook.Com

The three server-side components can be deployed with some amount of flexibility. In particular, IIS and SQL Server can be deployed either on the same server system or on separate machines. The SQL Server CE Server Agent component, however, must be deployed on the same system with IIS.

The SQL Server CE Client Agent, which is needed for RDA and Merge Replication only, is a device-side component that tracks changes made to data on the device and, when instructed to do so, submits them to the SQL Server CE Server Agent. Note that the track-change support for SQL Server CE is not the same as the track-change support for ADO.NET data sets that we mentioned in Chapter 6.

The SQL Server CE Server Agent is a server-side component that receives changes from the Client Agent and connects to the SQL Server database on behalf of the device. The Server Agent combines three different roles within one component:

- As an ISAPI[2] extension dynamic link library (DLL) to a Web server running Microsoft's IIS
- As part of a URL (i.e., its name), which provides the means by which the Client Agent and Server Agent are able to find each other
- As a SQL Server client program because it connects to and interacts with a SQL Server database

Because it is an ISAPI extension DLL, the SQL Server CE Server Agent must be copied to a Web site and registered with that Web site. Because the name of the Server Agent is part of a URL, you need to specify that URL in your application. Because it is a SQL Server client program, you may need to provide SQL Server logon information in your application. And because the transfer of data to and from SQL Server is done via IIS, your request must pass through both IIS security and SQL Server security. All of this impacts how you configure the components on the server side.

In this chapter, we cover how to install and configure the various components required to synchronize mobile data. We do not intend to make you an IIS administrator or a SQL Server administrator. Instead, the intent of this chapter is to provide you with enough information that you can set up a development environment on your desktop machine at work or at home.

2. *ISAPI* stands for Internet Server Applications Programming Interface, a mechanism for extending the functionality of IIS.

Download at WoweBook.Com

It might seem strange that you need to install SQL Server CE server-side components to run the sample applications for this chapter, but you did not need to do any SQL CE install for Chapter 6. You do not need to install SQL Server CE for local data access because SQL Server CE is automatically installed on a device along with an application whenever an application contains a reference to the `System.Data.SqlServerCe.dll` assembly. But the SQL Server CE Server Agent is not automatically installed on your IIS Web server system. You or an administrator must do that installation. We cover that topic next.

8.2 Installing Remote Data Connectivity

If you will be using Data Synchronization Services only, you can skip this section and proceed directly to the Using Data Synchronization Services section. People familiar with IIS recognize that the device-to-IIS connection can be made in Anonymous, Basic Authentication, or Integrated Windows Authentication mode; people familiar with SQL Server know that the IIS-to-SQL-Server connection can be made in Windows Authentication or SQL Server Authentication mode. All of this results in a connectivity/security chain that involves your SQL Server CE application, the network, the operating system, IIS, NTFS, SQL Server, and, in the case of Merge Replication, a shared directory.

8.2.1 Creating the Virtual Directory

As we stated, three security contexts exposed by IIS can be used with SQL Server CE (Anonymous, Basic, and Integrated). Of the three, Integrated is the most restrictive when using SQL Server CE because it requires having IIS and SQL Server on the same machine. This is because NTLM does not support proxying security credentials through machines. Kerberos Authentication does support this in Windows, but support for Kerberos is not available on Windows CE devices. As with any Web-based application, if you use Basic Authentication, it is recommended that you also use SSL (i.e., Web connections should use HTTPS instead of HTTP as the transfer protocol) to prevent the username and password from being deciphered between the device and the IIS box.

Before you begin to code the application, you have to address a number of standard Web development issues. You must decide the following: Where will the ISAPI extension .dll file be located? Where will the SQL Server be located? Will there be a firewall or proxy server between them?

Download at WoweBook.Com

What operating system user should the Server Agent execute as? How do you ensure that that operating system user is a valid SQL Server login? Should there be multiple sites, each with its own security? Or should there be just one site, one connection point, for all the applications?

These questions include the typical setup procedures for doing any Web-based application that also has back-end database support (setting up a virtual directory, installing the server-side components in the virtual directory, choosing the authentication modes for the virtual directory, assigning user rights, and setting up the necessary security and access rights for gaining access to SQL Server). It is not our intent to repeat the details for doing this; you can find that information in the SQL Server CE Books Online, located at http://msdn.microsoft.com/en-us/library/bb734673.aspx. Rather, we'll take a high-level look at the SQL Server CE application to IIS to SQL Server chain.

Fortunately, SQL Server CE provides a tool for creating and configuring the Web server to be accessed by client devices. That means you must install the server-side component of SQL Server CE and then use it to install the IIS site(s). Books Online refers to this as "Installing SQL Server CE," but it is really installing the server-side components of SQL Server CE.

You can find the software and installation programs at one of two locations. The root location for device software on your machine is `Program Files\Microsoft.NET\SDK\CompactFramework\vN.N\WindowsCE`. If you do not have the software already on your machine, you can go to www.microsoft.com/sql/editions/sqlmobile/connectivity-tools.mspx and obtain the necessary connectivity tool. This site contains the tools and setup instructions for every common combination of SQL Server CE version and SQL Server version, including both the *SQL Server and IIS on the same machine* scenario and the *SQL Server and IIS on different machines* scenario.

But installing the Server Agent does not yet mean you can transfer data between the device and the desktop. Because this transfer is done via the Internet, you must establish one or more sites on your IIS machine for use by SQL Server CE. Into each site you must install the sscesaNN.dll file, and you must set the appropriate permissions for that site.

You can reach the program that helps you do this, the SQL Server Connectivity Management program, from the Start menu entry shown in Figure 8.2. This management application allows you to reach the Virtual Directory Creation Wizard, which helps you configure the site. To

Download at WoweBook.Com

configure a new site from the SQL Server Connectivity Management program, select Microsoft SQL Server CE in the left-hand panel and click on Create a Virtual Directory in the right-hand panel, as shown in Figure 8.3. To modify an existing directory configuration, select the directory from the right-hand panel.

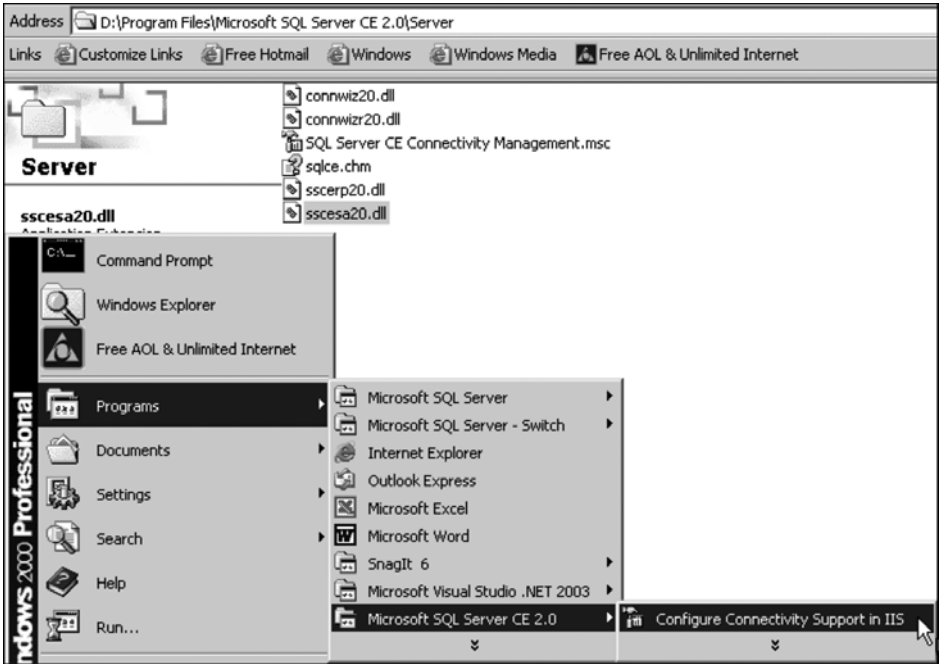


FIGURE 8.2: After Server-Side Installation of SQL Server CE

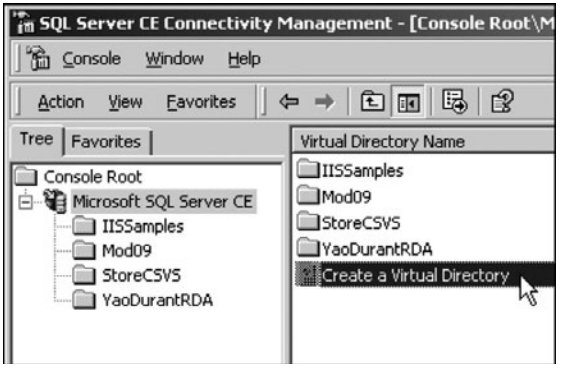


FIGURE 8.3: Launching the Virtual Directory Creation Wizard

Download at WoweBook.Com

The wizard walks you through five self-explanatory dialog boxes. Through the wizard you specify the name and location of the virtual directory, choose the authentication method, and optionally configure and share the snapshots directory needed for Merge Replication. The wizard creates the Web site and virtual directory, copies the sscesaNN.dll file to it, and registers sscesaNN.dll. It does not configure NTFS permissions, add domain users and groups, add SQL Server logons, or generate code. You or an appropriate administrator must perform those tasks after analyzing the needs of the site.

For instance, the need to provide read-only access to public data usually results in the following:

- A site that allows anonymous access
- A SQL Server logon for the `IUSR_MachineName` user
- A SQL Server that allows SQL Server logon authentication
- Application code that does not specify any of the change-tracking options

Authenticated access to corporate data might be accomplished with the following:

- A site that uses Integrated Windows Authentication and SSL
- A new SQL Server logons group

When you have finished configuring a Web site, you can test it by opening your browser and navigating to http://<server name>/<virtual directory name>/sscesaNN.dll, as shown in Figure 8.4. If you receive the response shown in the figure, your site installation was successful.

If you specified a security model that required authentication and if you were not currently logged on when you navigated to the site, you were asked to enter your username and password. Whatever URL provides access to the Server Agent is the URL to specify in your application code. Whatever username and password you specified to log on is the username and password to specify in your application code.

If you are unable to browse to the site from your device, try browsing from the server machine, using `localhost` as the server name. If this local access attempt fails, the problem lies in the Web server configuration.

Download at WoweBook.Com

Return to the SQL Server Connectivity Management program and check the properties and spellings of your configuration.

If local access succeeds, the problem lies in configuring your device. From your device, try to browse to a known site, such as www.microsoft.com. If that fails, you have a general network connectivity problem. If it succeeds, the problem lies in connecting your device specifically to your server.

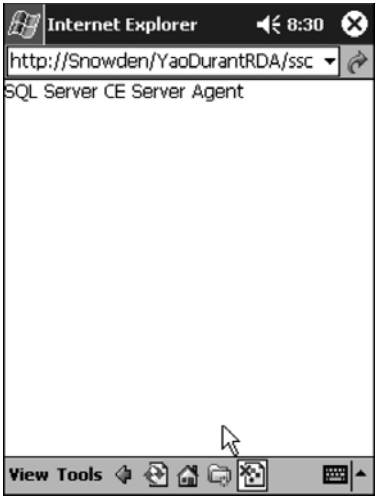


FIGURE 8.4: Configuration Test Showing Successful Site Installation

8.2.2 Configuring Additional Components

The IIS authentication configuration that you specify determines the username that the SQL Server CE Server Agent uses when it connects to SQL Server and when it accesses the shared directory required by Merge Replication. This user must be able to log on to SQL Server and access the directory, which in turn must be shared. Your application is ready to connect when you have accomplished all of the following tasks.

- All operating system users and groups, if any, have been added.
- All SQL Server logons and groups have been added.
- The directory has been shared.
- Share permissions have been granted.

Download at WoweBook.Com

8.3 Using RDA

Once you have completed configuring the components, you can develop and test your RDA and Merge Replication applications. We begin with RDA, the easier to administer but less capable of the two.

8.3.1 RDA Capabilities and Overhead

RDA enables a SQL Server CE application to do the following:

- Retrieve data and schemas from a SQL Server database into a SQL Server CE database on a device
- Add to, remove, or modify that data at the device and have SQL Server CE track those changes
- Have SQL Server CE submit the changed data to the SQL Server database and receive any exceptions resulting from failed updates on the server
- Submit SQL statements directly from the .NET Compact Framework application to a SQL Server database

RDA is simpler than Merge Replication, but it requires that you write more code, both C# code and SQL code. It also requires that you use an ADO.NET class that we did not need in Chapter 6 when we accessed a SQL Server database: the `SqlCeRemoteDataAccess` class.

RDA uses a concurrency model that the online documentation calls *optimistic concurrency.* The meaning of optimistic concurrency in RDA is different from the meaning known by many veteran SQL Server programmers. In RDA, optimistic concurrency means the following activities take place.

1. Pull data from the server to the device.
2. Modify the data on the device.
3. Submit data changes to the server.
4. The server applies the changes even if the data on the server has been modified since it was pulled to the device.

So, you need to adopt a new mindset if you are used to optimistic concurrency meaning the following.

Download at WoweBook.Com

1. Retrieve data into the application.
2. Modify data within the application.
3. Submit data changes to the server.
4. The server does not apply the changes if the data on the server has been modified since it was read by the application.

For instance, the following RDA scenario causes one update to overlay another.

08:00	Client A pulls data from the server.
08:00	Client B pulls data from the server.
12:00	Client A updates row X on device A.
12:00	Client B updates row X on device B.
18:00	Client A pushes updates to the server.
19:00	Client B pushes updates to the server.

Client B's update overlays Client A's update. No warning, error, exception, or conflict occurs.

The following change to the scenario causes B's update to fail.

12:00	Client A deletes row X.

In the new scenario, there is no row X on the server at 19:00, and B's update fails. When this failure occurs, RDA logs the failure in the local database. It is up to the client application to make whatever changes to local data that it feels are appropriate and to notify the user of the failure.

8.3.2 Programming for RDA

The primary ADO.NET class for RDA programming is `SqlCeRemoteDataAccess`. This class has three methods, corresponding to the three capabilities mentioned earlier (`SubmitSQL`, `Pull`, and `Push`), plus additional methods inherited from its base classes. It also has a set of properties for providing connection information. Table 8.1 lists the main methods and properties of the `SqlCeRemoteDataAccess` class.

Download at WoweBook.Com

TABLE 8.1: Methods and Properties of the `SqlCeRemoteDataAccess` Class

Name	Comment
Methods	
`Pull`	Submits a `SELECT` statement to a SQL Server database. Creates a table and indexes in a SQL Server CE database to hold the selected data. Retrieves the selected rows and inserts them into the table. Initiates tracking of changes made to the selected data
`Push`	Submits changes made to previously pulled data back to the SQL Server. Receives errors from the SQL Server and stores them in an application-specified error table
`SubmitSQL`	Submits SQL statements to a SQL Server
Properties	
`InternetLogon`	Internet username
`InternetPassword`	Internet user password
`InternetProxyLogon`	Proxy username
`InternetProxyPassword`	Proxy password
`InternetProxyServer`	Proxy server name
`InternetURL`	URL of the sscesa20.dll file at a Web site (i.e., the virtual directory specified in the Virtual Directory Alias and Content Folder dialog box of the Virtual Directory Creation Wizard
`LocalConnectionString`	Connection string for the SQL Server CE database

8.3.2.1 *Pulling Data*

The `Pull` method submits a SQL `SELECT` statement to the SQL Server for processing. The method creates the table to hold the returned rows; that table must not already exist. Some of the information the `Pull` method needs is specified by setting properties before invoking the method; some is supplied through the parameters, which are shown in Table 8.2.

For example, the code shown in Listing 8.1 pulls information from the Northwind `Categories` table and stores it in a newly created table that is

Download at WoweBook.Com

also to be named Categories. Listings 8.1 through 8.5 show the code that you can find in the RDA project at the book's Web site.

Listing 8.1: An RDA Pull

```
//  The database file
private string  strDBFile = @"My Documents\Northwind.sdf";

//  The local connection string
private string  strConnLocal =
        "Data Source=" + @"My Documents\Northwind.sdf";

//  The remote connection string
private string strConnRemote = "Provider=sqloledb; "
                             + "Data Source=Snowden; "
                             + "Initial Catalog=Northwind; "
                             + "Integrated Security=SSPI;";

//  The URL
private string strURL =
        "http://207.202.168.30/YaoDurantRDA/sscesa20.dll";


private void mitemPull_Click(object sender, EventArgs e)
{
//  Create an RDA object.
SqlCeRemoteDataAccess  rdaNW = new SqlCeRemoteDataAccess();

try
{
   //  Have RDA:
   //     Create local tables named Categories and
   //        ErrorCategories.
   //     Connect to the remote server and submit the
   //        SELECT statement.
   //     Place the results in the local Categories table.
   rdaNW.LocalConnectionString = strConnLocal;
   rdaNW.InternetUrl = strURL;
   rdaNW.InternetLogin = "";
   rdaNW.InternetPassword = "";
   rdaNW.Pull("Categories",
               "SELECT CategoryID, CategoryName " +
               "  FROM Categories",
               strConnRemote,
               RdaTrackOption.TrackingOnWithIndexes,
               "ErrorCategories");
}
catch(SqlCeException exSQL)
{
   HandleSQLException(exSQL);
```

Download at WoweBook.Com


```
}
finally
{
   rdaNW.Dispose();
}
```

TABLE 8.2: Parameters of the `Pull` Method

Parameter	Comment
`localTableName`	Name of the SQL Server CE table to be created to receive the output of the `SELECT` statement
`sqlSelectString`	The `SELECT` statement
`oledbConnectionString`	The connection string used to access the SQL Server database
`trackOptions`	An enumerated indicator specifying whether changes to the data in the table should be tracked so that they can be subsequently pushed back to the SQL Server. Also specifies whether indexes should be created on the local table
`errorTable`	The name of a table to be created in the local database that holds errors returned by subsequent pushes

8.3.2.2 *Fixing the* `IDENTITY` *Property*

When you pull a table from SQL Server down to your device using RDA, the schema of that table is brought with the data so that the table can be created on the device. That schema might be incorrect regarding the `IDENTITY` property. If the SQL Server table has an `IDENTITY` property column, the new SQL Server CE table also has that column designated as an `IDENTITY` property column, but its seed and increment are always set to `(1,1)`. You must initially correct them to the values you want, as shown in Listing 8.2.

LISTING 8.2: Setting the `IDENTITY` Property Column's Seed Value

```
//  The IDENTITY property seed value of the new table
//     is at 1, even after the retrieved rows have
//     been added to the table.  "Fix" it.
SqlCeConnection connLocal = new SqlCeConnection(strConnLocal);
connLocal.Open();
```

Download at WoweBook.Com


```
SqlCeCommand  cmndLocal = new SqlCeCommand();
int intMaxCategoryID;
try
{
   cmndLocal.Connection = connLocal;
   //  Retrieve the highest CategoryID in the table.
   cmndLocal.CommandText =
      "SELECT max(CategoryID) FROM Categories";
   string strMaxCategoryID =
      cmndLocal.ExecuteScalar().ToString();
   intMaxCategoryID = int.Parse(strMaxCategoryID);

   //  Set the seed one higher.
   cmndLocal.CommandText =
      "ALTER TABLE Categories " +
      "ALTER COLUMN CategoryID IDENTITY (" +
      (intMaxCategoryID + 1).ToString() + ",1)";
   cmndLocal.ExecuteNonQuery();
}
catch( SqlCeException exSQL )
{
   HandleSQLException(exSQL);
}
finally
{
   connLocal.Close();
}
```

Be aware that the auto-generated `CategoryID` value of any rows that you insert into the local `Categories` table is always overridden with a new value when those rows are pushed to the host server.

8.3.2.3 *Viewing the Pulled Schema*

If you use RDA to retrieve data from the desktop and you plan to propagate local changes to that data back to the desktop server, you must specify one of the change-tracking enumerations in the fourth parameter when you pull the data.

Two of the tracking options also specify that indexes are to be created on the local table. The `TrackingOnWithIndexes` option adds yet one more index to your table. Do your own performance testing to determine whether your application benefits from the indexes. Using either tracking option adds the two extra columns shown in Figure 8.5 to your database. Using the `TrackingOnWithIndexes` option adds the extra index also shown in Figure 8.5.

Download at WoweBook.Com

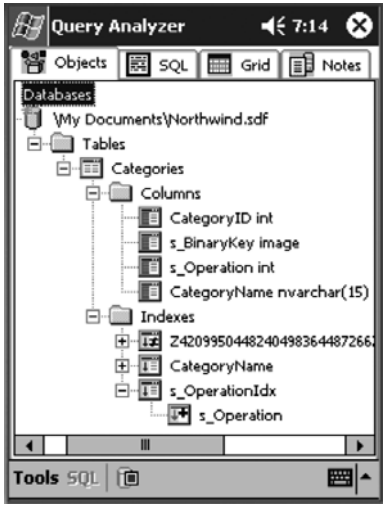


FIGURE 8.5: A Table Pulled with Tracking and Indexes

8.3.2.4 *Modifying Pulled Data Locally*

The connection to SQL Server is open only while the data is being pulled. When the data has been transferred to the SQL Server CE data table, the SQL Server connection is closed. Once you have pulled the data and, if necessary, altered the `IDENTITY` property, you can modify the data locally using the techniques covered in Chapter 6 and illustrated in Listing 8.3.

LISTING 8.3: Modifying Pulled Data at the Device

```
private void mitemUpdate_Click(object sender, EventArgs e)
{
   SqlCeConnection  connLocal =
      new SqlCeConnection(strConnLocal);
   connLocal.Open();

   SqlCeCommand  cmndLocal = new SqlCeCommand();
   try
   {
      cmndLocal.Connection = connLocal;
      cmndLocal.CommandText =
                  "UPDATE Categories " +
                  "   SET CategoryName = 'new Name'   " +
                  " WHERE CategoryID = 2";
      cmndLocal.ExecuteNonQuery();

      cmndLocal.CommandText =
```

Download at WoweBook.Com


```
            "DELETE Categories WHERE CategoryID = 3";
         cmndLocal.ExecuteNonQuery();

         cmndLocal.CommandText =
            "INSERT Categories (CategoryName) " +
            "       VALUES ('new Category I') ";
         cmndLocal.ExecuteNonQuery();
      }
      catch( SqlCeException exSQL )
      {
         HandleSQLException(exSQL);
      }
      finally
      {
         cmndLocal.Dispose();
         connLocal.Close();
      }
}
```

Figure 8.6 shows the `Categories` data after modification. The row for `CategoryID = 3` has been deleted, the row for `CategoryID = 2` has been modified, and a new row with `CategoryID = 10` has been added.

The inserted and modified rows are marked in the `s_Operation` column; the deleted row is represented in the `MSysRDATombstone` table.

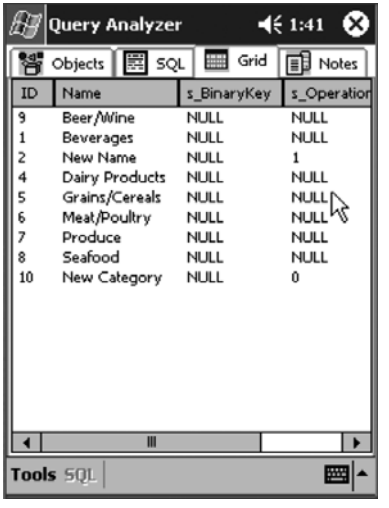


FIGURE 8.6: The `Categories` Table after Updates

Download at WoweBook.Com

8.3.2.5 *Pushing Data*

To propagate the table's changes back to the desktop SQL Server, you use the `Push` method of the `SqlCeRemoteDataAccess` class. This is shown in Listing 8.4.

Listing 8.4: An RDA Push

```
private void mitemPush_Click(object sender, EventArgs e)
{
   //  Create an RDA object.
   SqlCeRemoteDataAccess  rdaNW =
      new SqlCeRemoteDataAccess();
   try
   {
      //  Have RDA:
      //     Create local tables named Categories and
      //        ErrorCategories.
      //     Connect to the remote server and submit
      //        the changes.
      rdaNW.LocalConnectionString = strConnLocal;
      rdaNW.InternetUrl = strURL;
      rdaNW.InternetLogin = "";
      rdaNW.InternetPassword = "";
      rdaNW.Push("Categories", strConnRemote);
   }
   catch( SqlCeException exSQL )
   {
      HandleSQLException(exSQL);
   }
   finally
   {
      rdaNW.Dispose();
   }
}
```

After the `Push` method completes, you can detect the rows that failed to update at the server due to concurrency conflicts by accessing the local table that was specified as a `Pull` method parameter. In our case, this was the `ErrorCategories` table, shown in Figure 8.10, specified as the last parameter in the following code snippet extracted from Listing 8.1.

```
rdaNW.Pull("Categories",
           "SELECT CategoryID, CategoryName " +
           "  FROM Categories",
           strConnRemote,
           RdaTrackOption.TrackingOnWithIndexes,
           "ErrorCategories");
```

Download at WoweBook.Com

Figure 8.7 shows the contents of the `Categories` table that we pulled to the local database.

When we push the changes back to the SQL Server database, using the code in Listing 8.4, the following activities occur.

1. The modified row from our local database is modified on the server.
2. The deleted row in our local database fails to delete because of a foreign key violation; there are some `Confections` products in the remote database, so the category cannot be deleted.
3. The inserted row from our local database fails to insert because a `CategoryId = 10` row already exists in the remote database. Perhaps some other user added this row; perhaps we added it with a `SubmitSQL` method call.

This can be verified by viewing the contents of the `Categories` table on the server after the push has completed, as shown in Figure 8.8.

When the `Push` method executed, it raised the exception shown in Figure 8.9. As a result of the exception, the rows shown in Figure 8.10 were added to the SQL Server CE `ErrorCategories` table.

	CategoryID	CategoryName	Description	Picture
▶	1	Beverages	Soft drinks, coffees	<Binary>
	2	Condiments	Sweet and savory s	<Binary>
	3	Confections	Desserts, candies,	<Binary>
	4	Dairy Products	Cheeses	<Binary>
	5	Grains/Cereals	Breads, crackers, p	<Binary>
	6	Meat/Poultry	Prepared meats	<Binary>
	7	Produce	Dried fruit and bea	<Binary>
	8	Seafood	Seaweed and fish	<Binary>
	9	Beer/Wine	Beer and Wine	<Binary>
*				

FIGURE 8.7: Remote Data before the Push

	CategoryID	CategoryName	Description	Picture
▶	1	Beverages	Soft drinks, coffees	<Binary>
	2	New Name	Sweet and savory s	<Binary>
	3	Confections	Desserts, candies,	<Binary>
	4	Dairy Products	Cheeses	<Binary>
	5	Grains/Cereals	Breads, crackers, p	<Binary>
	6	Meat/Poultry	Prepared meats	<Binary>
	7	Produce	Dried fruit and bea	<Binary>
	8	Seafood	Seaweed and fish	<Binary>
	9	Beer/Wine	Beer and Wine	<Binary>
	10	New Category II	From SubmitSQL	<Binary>
*				

FIGURE 8.8: Remote Data after the Push

Download at WoweBook.Com

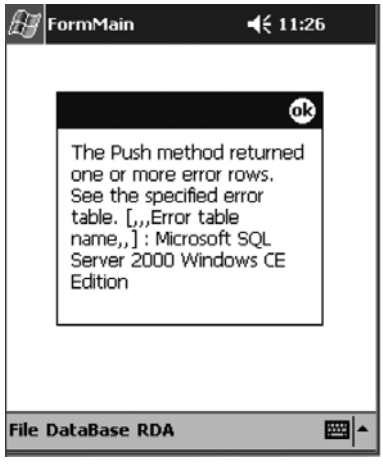


FIGURE 8.9: The `Push` Method Error Message

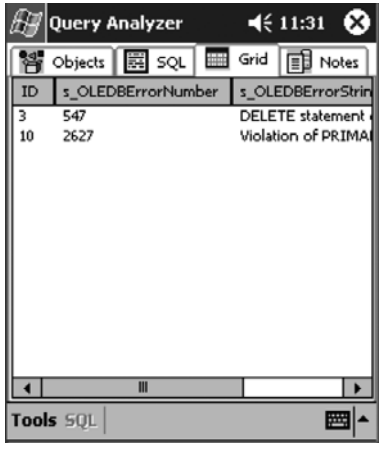


FIGURE 8.10: The `ErrorCategories` Table after the Push

There is no automatic conflict resolution in RDA. You must examine the failed rows placed in the error table and decide what changes to make to the data on your device as a result of the failed server updates.

8.3.2.6 *The* `SubmitSQL` *Method*

The `SubmitSQL` method is pretty self-explanatory. It bypasses the SQL Server CE database completely and simply submits a SQL statement to the SQL Server for processing. This method is used for submitting updates to the server. (If you submit a `SELECT` statement, no exception will be thrown, but you will not accomplish anything.) For example, Listing 8.5 shows the

Download at WoweBook.Com

code used to insert a row into the remote `Categories` table; in fact, it was one of the rows that caused our subsequent push to fail.

Listing 8.5: Executing the `SubmitSQL` Method

```
private void mitemSubmitSQL_Click(object sender, EventArgs e)
{
   //  Create an RDA object.
   SqlCeRemoteDataAccess  rdaNW = new SqlCeRemoteDataAccess();
   try
   {
      //  Have RDA:
      //     Create local tables named Categories and
      //        ErrorCategories.
      //     Connect to the remote server and submit
      //        the changes.
      rdaNW.LocalConnectionString = strConnLocal;
      rdaNW.InternetUrl = strURL;
      rdaNW.InternetLogin = "";
      rdaNW.InternetPassword = "";
      rdaNW.SubmitSql(
         "INSERT Categories (CategoryName, Description)" +
            "VALUES ('New Category II', 'From SubmitSQL')",
         strConnRemote);
   }
   catch( SqlCeException exSQL )
   {
      HandleSQLException(exSQL);
   }
   finally
   {
      rdaNW.Dispose();
   }
}
```

As is the case with the `Push` and `Pull` methods, the `SubmitSQL` method requires that you set property values with the information appropriate to your Web site configuration and that you correctly code your SQL statements. Once you have done that, transferring the data is a matter of a single method call.

As we are about to see, this emphasis on providing connection information through property values and the need to understand database technology continues into Merge Replication. But when you understand how your application connects to the server and how to request data from the server, you will be able to transfer data with even less code than RDA requires.

Download at WoweBook.Com

8.4 Using Merge Replication

If you have worked with replication, you know that prior planning and design are important. First-time replicators often think that having the same data in two places and needing to keep that data in sync is an easy task. But consider just two examples of the unexpected complexity involved in replication.

- SQL Server provides a capability called *triggers;* a change in `TableX` automatically causes a change to occur in `TableY`. But SQL Server CE does not support triggers. If `TableX` and `TableY` are replicated down to a SQL Server CE database, a change in `TableX` does not cause an automatic change to `TableY`. Thus, the same statement executed on the "same" table, `TableX`, produces different results in `TableY`, and the two copies of `TableY` are not in sync.
- If you replicate a table that has a foreign key, you need to replicate the primary key table as well to validate the rows inserted into the foreign key table. Perhaps you wish to replicate only some of the rows in the foreign key table—say, only the rows for delivery route West92. You want to replicate only those rows from the primary key table (the `Products` table) that are relevant to delivery route West92. But delivery route code probably is not a column in the `Products` table, so how do you specify which `Products` table rows should be replicated? (As we will see later in this chapter, you can accomplish this by publishing all the tables involved in enforced relationships or by adding a horizontal filter that extends outward to include related rows from other tables.)

At first, such considerations may make you wish for the perceived simplicity of RDA. But after you examine some key differences between Merge Replication and RDA, you will realize that Merge Replication is quite attractive. Merge Replication requires less code than RDA. Merge Replication provides the ability to have the server control much of the application logic, instead of having the logic distributed to the device. For example, Ranged Identity columns and dynamic horizontal partitions are two key areas where the logic resides on the server and not on the device. This can substantially reduce the amount of code on the device and allow application maintenance to be performed on the server rather than on the device.

Download at WoweBook.Com

Any replication schema must be well planned and well designed, including the ones you plan to use with your SQL Server CE applications. Therefore, we take a moment to review the basic concepts of SQL Server's Merge Replication and then follow up with a detailed examination of design considerations.

In Merge Replication, as it is used with a SQL Server CE application, the SQL Server is referred to as the *publisher* with one or more defined publications on one or more databases, and the SQL Server CE databases are referred to as the *subscribers.* Publications are made up of a defined set of tables, columns, and filters. These chosen tables are referred to as the *articles of a publication;* and the definition of a subset, such as "only `columnA`, `columnB`, `columnC` of `TableX`" or "only those rows of `TableY` where route code equals `'West92'`" are referred to as *filters*. Filters allow you to replicate a vertical subset of a table (such as the first example in the previous sentence) or replicate a horizontal subset of a table (such as the second example) or both. They also can be dynamic, such as "only rows in the `Orders` table where `EmployeeID` equals the employee ID of the user performing the synchronization".

With this flexibility in publication design comes the need to ensure that changes made to this subset of the data residing on the device will synchronize correctly when propagated back to the server.

8.4.1 Using Good Design to Avoid Synchronization Failures

Most mobile applications that have a Merge Replication component consist of three repetitively executed steps.

1. Subscribe to a publication located at the server and synchronize the data to the device.
2. Make updates to that data as it resides on the device.
3. Synchronize with the server to submit the changes made locally back to the server database.

All mobile application designers hope that step 3 succeeds so that modifications are applied at the server without errors and that those modifications do not overwrite any modifications made by other users. Many mobile application designers end up wondering why step 3 failed. The purpose of this subsection is to explain why publication design decisions can cause synchronization failures and to provide a blueprint on how to avoid those failures.

Download at WoweBook.Com

A mobile application that uses Merge Replication for the transfer of data between SQL Server and SQL Server CE does so by subscribing to a publication that resides on the server. Since SQL Server CE provides only for anonymous subscriptions, the entire publication must be subscribed to. But mobile applications usually do not want to work with large amounts of data; rather, they want just the data that applies to the subject at hand and to the user of the device. They almost always want less than the entire server database. This means that almost all publications are defined as subsets of the database and therefore that all mobile applications work with a subset of the database.

Working with a subset of data always creates the possibility of data integrity problems occurring in the subset that will not be detected until the data is synchronized with the full set of data residing at the server. For example, the Northwind database that comes with SQL Server contains two tables that are related by a foreign key constraint: `Products`, which contains the foreign key, and `Categories`, which contains the primary key. Now suppose that you define a publication that includes the `Products` table but not the `Categories` table. When you synchronize, the `Products` table is brought to the device. That table has a `CategoryID` column both on the server and on the device. At the server, the list of possible values that can be entered into that column is known; by definition it is the list of values in the `Categories` table's primary key column. But on the device, that list is not known. The local application cannot insert rows into the `Products` table or update the `CategoryID` field in rows that are already there because the application does not know what values will be accepted or rejected when the data is synchronized to the server.

Not all developers understand that each decision made during the design of the publication results in a specific and definable limitation that must be placed on the application for local data modifications to synchronize successfully at the server. As we look at publication design decisions and the resulting restrictions that those decisions place on your local application, let us be clear about something: The design of a publication does not place automatic restrictions on your application; rather, it creates situations that force you to design limitations into your application as you develop it.

To illustrate this "choice and impact" world in which you must develop applications based on Merge Replication, we develop a sample publication and mobile application design. For our server database, we pick the Northwind database. Our Merge Replication–based mobile

Download at WoweBook.Com

application synchronizes order data for one employee with the Northwind database. In the publication definition, a row filter specifies that only order information for the employee who is performing the synchronization should be delivered to the device. Our definition of "order information" is illustrated by the SQL Server data diagram in Figure 8.11. Thus, our publication includes the employee row, all orders related to that employee, all customers and order details connected to those orders, and all products for the order details. We included only those columns we determined to be essential to our application.

Our publication (and therefore the data delivered to the device) is a four-way subset of the host database; it contains some of the tables, some of the columns, some of the rows, and some of the constraints of the Northwind database. Each of these reductions in content places limitations on the modifications that the application can make to that data on the device.

For the sake of illustration, the employee named Nancy Davolio is performing the synchronization, unless otherwise stated.

Table 8.3 relates publication design decisions with their impacts on the application.

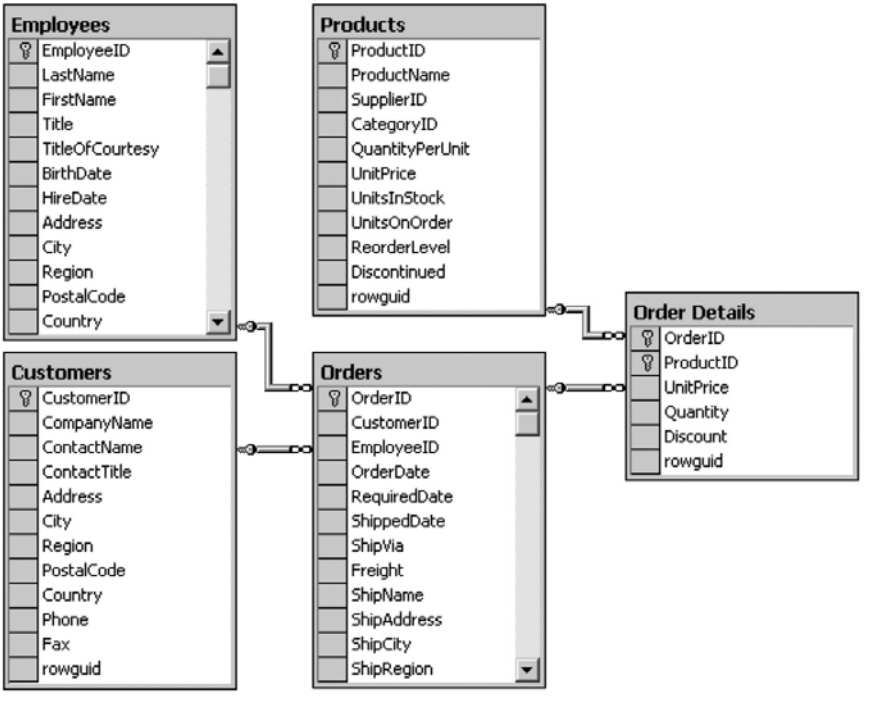


FIGURE 8.11: Tables to Be Included in the Sample Publication

Download at WoweBook.Com

TABLE 8.3: Impacts of Publication Design

Type of Subset	Impact on Local Application	Example
Not all columns of a table are included	We cannot insert rows unless the database at the server contains default value definitions or permits nulls for the missing columns Deleting rows will result in the loss of unknown information	The sample application should not add new customers or delete existing customers
Not all rows of a table are included	Inserted rows must adhere to the row filter specified by the publication	The sample application should not add a new employee or any orders for that new employee. The new rows would successfully update to the server, but they would disappear from the device during the next synchronization because they would not be Nancy Davolio's data
A table that has a foreign key is included, but the table containing the matching primary key is not	New rows cannot be inserted into the table, nor can the foreign key column be updated	The sample application should not insert rows into the `Products` table or update its `CategoryID` column because the domain of allowable values is not known
A table that has a primary key column is included, but the table with the corresponding foreign key column is not	The primary key column cannot be updated (which isn't allowed anyway). Rows cannot be deleted from the table	The sample application should not delete a customer because that customer may have data in the `CustomerCustomerDemo` table on the server

Continues

Download at WoweBook.Com

TABLE 8.3: Impacts of Publication Design *(Continued)*

Type of Subset	Impact on Local Application	Example
Some rows are not unique to a specific synchronization (i.e., they will be propagated to more than one device). This is an inevitable result of a many-to-many relationship, such as employees and customers. For instance, a customer might have placed orders with several employees	The table whose rows can synchronize to more than one device cannot be updated unless ownership of rows and columns can be assigned to a single subscriber. Rows cannot be deleted from the table Updating the row on more than one device will result in conflicts during synchronization. The last update synchronized will be the winner	The sample application should not update customer information (however, if the `Customers` table had a `Responsible-Employee` foreign key column, the sample application could permit only that employee to make modifications)
A column that has the `IDENTITY` property is included in the publication	SQL Server's Ranged Identity capability must be specified in the publication definition. The application should not try to solve this problem	If, for example, an `Identity` column were added to the `Orders` table on the server, Ranged Identity would assign a range of possible identity values to each subscriber, ensuring that two devices did not give different orders the same identifying number
Derived (calculated) columns are included in the publication definition	These columns normally should not be updated. Their values are derived from other values in the database. They can be updated only if all their source values are included in the publication	The sample application should not recalculate `Product.UnitsIn-Stock` based on subscriber activity. Any such updates would reflect only the activity of the local user. During synchronization, each subscriber's value would overwrite the previous subscriber's value. All values would be wrong; the last subscriber's value would persist and propagate

Download at WoweBook.Com

TABLE 8.3: Impacts of Publication Design *(Continued)*

Type of Subset	Impact on Local Application	Example
A recursive relationship on a table with a row filter specified is included in the publication	The table will not synchronize to the device	When Nancy Davolio synchronizes, her row is the only employee row brought to the device. In the `ReportsTo` column of that table is the employee ID of her manager. The foreign key constraint mandates that her manager's row be on the device before her row can be inserted. But her manager's row is not on her device because it was filtered out by the publication

Many of the restrictions in Table 8.3 sound like nothing more than common sense, but they are grounded on the basic principles of relational algebra. These restrictions are not arbitrary or optional; they cannot be circumvented by clever application code or expensive hardware. Common sense is not always common. When you are involved in a design-time discussion, it's nice to know that your recommendations are based on solid ground, especially when those of others are not.

Having studied the restrictions that will be placed on our application by our publication, we decide that the restrictions are acceptable. It is time now to configure the publication.

8.4.2 Configuring Merge Replication

We have completed the necessary steps that lead up to configuring Merge Replication, as summarized here.

1. The configuration of an IIS site (covered earlier in this chapter) is complete.
2. The necessary domain users and groups have been added.
3. The necessary SQL Server logins and users have been added.
   a. All Northwind employees have been added as SQL Server logins. (An alternative scenario would have been to add them as domain

Download at WoweBook.Com

groups and users and use Windows Authentication, but we wanted to keep the focus on SQL Server, not on domain management.)

   b. The logins have been added as users of the Northwind database and given the necessary permissions.

4. The publication has been designed but not implemented.
   a. It consists of the tables and columns shown earlier in Figure 8.11.
   b. It contains a dynamic filter to limit the data to that which is appropriate for the user executing the synchronization.

We are ready to define our publication. Fortunately, SQL Server's Enterprise Manager console snap-in has the Create Publication Wizard to help us. We run it by selecting <Server Name> | Databases | Northwind | Publications in the Enterprise Manager's Tree window, right-clicking on it, and choosing New Publication from the pop-up menu. After working through about twenty dialog boxes, including the specification of dynamic filters and expanded (join) filters, we have defined our publication.

If we need to modify the definition of our publication once we have completed the Create Publication Wizard, we can do so by right-clicking on the publication name in the Enterprise Manager's Tree window and selecting Properties. When we make changes to a publication, we need to resynchronize all devices. Additionally, some selections cannot be changed, requiring that we delete the publication and create a new one with the desired selections. As always, having a solid publication design before beginning development is most important.

Once we have completed the publication, we are two-thirds of the way home. Configuring our Internet connectivity was the first third, defining our publication was the second third, and writing the .NET Compact Framework application that does synchronizations and modifies data is the final third.

8.4.3 Programming for Merge Replication

Programming a .NET Compact Framework application to work with Merge Replication consists of using the `SqlCeReplication` class located in the `System.Data.SqlServerCe` namespace to initially subscribe to a publication and to periodically synchronize with that publication. Between synchronizations, the application updates the data in the SQL Server CE database using the classes and techniques covered in the Microsoft SQL Server CE section of Chapter 6.

Download at WoweBook.Com

Our sample application will subscribe to the publication we just created, synchronize the data to the device, modify the data on the device, and resynchronize to transmit the modifications back to the server and to receive new data (other users' modifications) from the server. Along the way, we use some utility routines to view the schema created by the subscription and the data delivered by the synchronization. The application is predominately menu-driven, with a single `DataGrid` control for displaying information. Figure 8.12 shows the application. The code for this application is shown in Listings 8.6 through 8.9 and can be found in the `Repl` project at the book's Web site.

The code begins with the same references we used in Chapter 6:

```
using System;
using System.IO;
using System.Windows.Forms;
using System.Data;
using System.Data.Common;
using System.Data.SqlServerCe;
```

The `FormMain` class begins with some generated code followed by the local connection information and the remote connection information. The remote information defines the connection to the Web site but not the SQL Server connection string. That string will be built by the sscesaNN.dll code from parameters that will be passed in the `AddSubscription` and `Synchronize` calls, parameters that are declared in Listing 8.6.

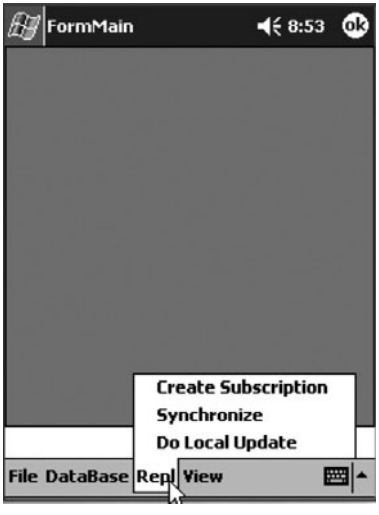


FIGURE 8.12: The Merge Replication Sample Application

Download at WoweBook.Com

Listing 8.6: Declaring Connection Parameters

```
/// <summary>
/// Summary description for Form1
/// </summary>
public class FormMain : System.Windows.Forms.Form
{
   internal System.Windows.Forms.MainMenu MenuMain;
   internal System.Windows.Forms.MenuItem mitemFile;
        :
        :
   private System.Windows.Forms.DataGrid dgridOutput;

   public FormMain()
   {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
   }
   /// <summary>
   /// Clean up any resources being used.
   /// </summary>
   protected override void Dispose( bool disposing )
   {
      base.Dispose( disposing );
   }

   #region Windows Form Designer generated code

   /// <summary>
   /// The main entry point for the application
   /// </summary>

   static void Main()
   {
      Application.Run(new FormMain());
   }

   //  The database file
   private string  strDBFile = @"My Documents\Northwind.sdf";

   //  The local connection string
   private string  strConnLocal =
      "Data Source=" + @"My Documents\Northwind.sdf";

   //  The URL
   private string strURL =
      "http://207.202.168.30/YaoDurantRDA/sscesa20.dll";
```

Download at WoweBook.Com

8.4.3.1 *Subscribing to a Publication*

A .NET Compact Framework application needs to subscribe to a publication only once. Doing so causes information about the subscription to be stored at the server. Thereafter, each time the application attempts to synchronize, the server verifies that the subscription does exist, that the publication has not be modified or re-created since the previous subscription, and that valid credentials are being presented by the application.

To subscribe, the application must create a `SqlCeReplication` object, set its properties, invoke its `AddSubscription` method, and dispose of it, as shown in Listing 8.7. Subscribing does not cause data or schemas to be transmitted to the device.

Listing 8.7: Subscribing to a Publication

```
private void mitemCreateSubscription_Click(object sender,
                                           EventArgs e)
{
   SqlCeReplication  replNW = new SqlCeReplication();
   try
   {
      replNW.ExchangeType = ExchangeType.BiDirectional;
      replNW.InternetUrl = strURL;
      replNW.InternetLogin = "";
      replNW.InternetPassword = "";
      replNW.Publisher = "SNOWDEN";
      replNW.PublisherDatabase = "Northwind";
      replNW.Publication = "EmployeeOrderInfo";
      replNW.PublisherSecurityMode =
         SecurityType.DBAuthentication;
      replNW.PublisherLogin = "Davolio";
      replNW.PublisherPassword = "Nancy";
      replNW.Subscriber = "YaoDurant";
      replNW.SubscriberConnectionString = strConnLocal;

      replNW.AddSubscription(AddOption.CreateDatabase);
   }
   catch( SqlCeException exSQL )
   {
      HandleSQLException(exSQL);
   }
   finally
   {
      replNW.Dispose();
   }
}
```

Download at WoweBook.Com

The properties whose names begin with `Pub` are used by the sscesa20.dll file at the `strUrl` location to connect to the specified server, find the database, and subscribe to the publication. Because we specified SQL Server Authentication, both at the server and here in our code, we must specify a publisher login and password. That login will be used by sscesa20.dll as it subscribes to the publication on the application's behalf; that is, it will subscribe as `"Davolio"`. `"Davolio"`, in turn, will become the `user_name` value that the publication filter uses. Thus, information about Nancy Davolio's orders will be synchronized to the device.

8.4.3.2 *Synchronizing the Data*

Once we have subscribed to a publication and once the snapshot agent for that publication has run at least once, we can synchronize with the publication, resulting in the two-way exchange of data between the server and the device.

Listing 8.8 shows the code. It is identical to the subscription code in Listing 8.7 except that the `Synchronize` method is called instead of the `AddSubscription` method.

LISTING 8.8: Synchronizing the Device with the Remote Server

```
private void mitemSynchronize_Click(object sender,
                                System.EventArgs e)
{
   SqlCeReplication  replNW = new SqlCeReplication();
   try
   {
      replNW.ExchangeType = ExchangeType.BiDirectional;
      replNW.InternetUrl = strURL;
      replNW.InternetLogin = "";
      replNW.InternetPassword = "";
      replNW.Publisher = "SNOWDEN";
      replNW.PublisherDatabase = "Northwind";
      replNW.Publication = "EmployeeOrderInfo";
      replNW.PublisherSecurityMode =
         SecurityType.DBAuthentication;
      replNW.PublisherLogin = "Davolio";
      replNW.PublisherPassword = "Nancy";
      replNW.Subscriber = "YaoDurant";
      replNW.SubscriberConnectionString = strConnLocal;

      replNW.Synchronize();
   }
   catch( SqlCeException exSQL )
   {
```

Download at WoweBook.Com


```
            HandleSQLException(exSQL);
        }
        finally
        {
            replNW.Dispose();
        }
    }
```

Synchronization brings new data and schemas, if necessary, to the device and closes the connection to the server. Because we have not previously synchronized, there is no data already on our device that came from the remote server and therefore no data that will be returned to the server during this synchronization.

Our sample application has methods that select all rows from `msysObjects` and `msysConstraints` and display the results in the `DataGrid` control. Figures 8.13 and 8.14 show the information presented by these routines, respectively. We can see that both tables and constraints have been propagated from the server to the device.

8.4.3.3 *Modifying the Data at the Device*

Once the data is on our device, we can add, update, and delete as covered in Chapter 6 and in accordance with the restrictions described in Table 8.3 earlier in this chapter. Listing 8.9 shows a simple example.

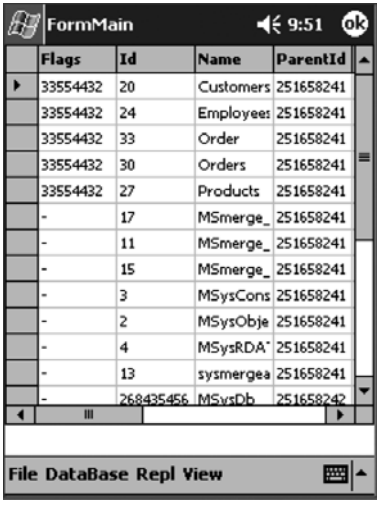


FIGURE 8.13: Objects after Synchronization

Download at WoweBook.Com

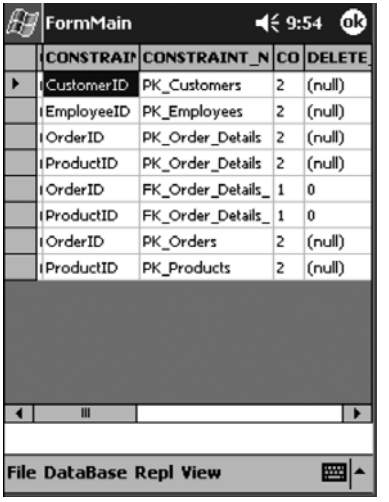


FIGURE 8.14: Constraints after Synchronization

LISTING 8.9: Updating the Data Locally

```
private void mitemLocalUpdate_Click(object sender,
                              System.EventArgs e)
{
   SqlCeConnection  connLocal =
      new SqlCeConnection(strConnLocal);
   connLocal.Open();

   SqlCeCommand  cmndLocal = new SqlCeCommand();
   try {
      cmndLocal.Connection = connLocal;
      cmndLocal.CommandText =
         "INSERT \"Order Details\" " +
         " (OrderID, ProductID, UnitPrice, " +
         "  Quantity, Discount) " +
         " VALUES (10258, 1, 19.95, 44, 0.0)";
      cmndLocal.ExecuteNonQuery();
   }
   catch( SqlCeException exSQL )
   {
      HandleSQLException(exSQL);
   }
   finally
   {
      cmndLocal.Dispose();
      connLocal.Close();
   }
}
```

Download at WoweBook.Com

After executing this statement, we examine the `Order Details` table on the device to verify that the row is there, as shown in Figure 8.15.

When we synchronize again, we use Enterprise Manager to view the row on the server (see Figure 8.16), thus verifying that it was received.

This completes the walkthrough of our sample application. During the walkthrough, we subscribed, synchronized, modified data, and resynchronized—the steps that every Merge Replication application must perform.

A Merge Replication application requires a small amount of code to subscribe and synchronize because it is drawing on components defined

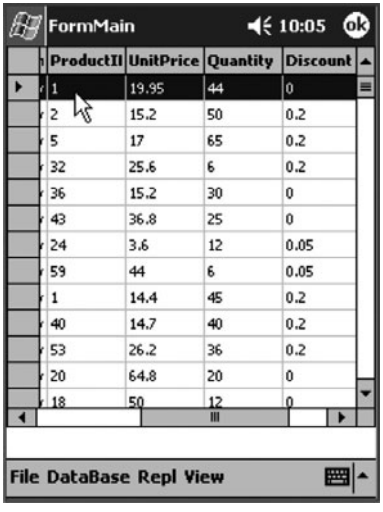


FIGURE 8.15: `Order Detail` **Row Added for Merge Replication**

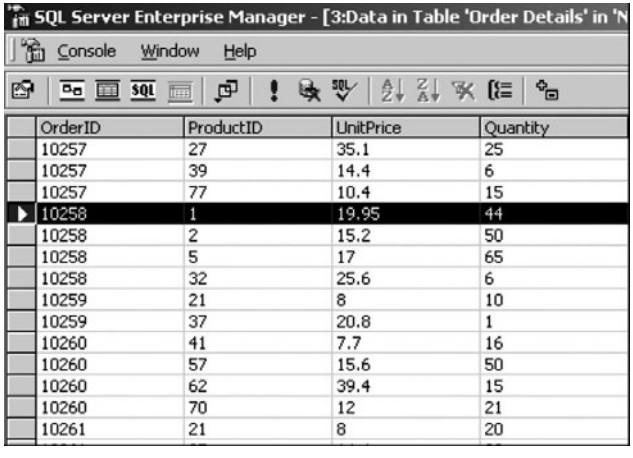


FIGURE 8.16: `Order Detail` **Row Viewed at the Server**

Download at WoweBook.Com

outside the program, mainly the virtual directory configuration and the publication definition. Good planning and design of these outside components is essential for success.

8.5 Choosing between Merge Replication and RDA

An important design issue that you have to address when you start to think about using SQL Server CE involves selecting which of the two connectivity solutions to use. If you must work with the much older versions of SQL Server (versions 6.5 and 7.0), the choice is easy: You must use RDA because Merge Replication is not supported. With the current versions, however, you can use either method.

The biggest difference is that Merge Replication lets you specify the relationship between the desktop data and the Windows CE data declaratively, through the definition publications, while RDA requires that you write SQL code to define the data to be transferred. Thus, with Merge Replication, the central data store controls data movement, while RDA allows the Windows CE devices to run the show.

Initially, this might make RDA seem to be the better choice. You just code a `SELECT` statement, use the `Pull` method, and, *voilà,* SQL Server CE creates a table and transfers the selected data into the table. No need to bother with the administrative effort of setting up replication.

For a simple or "one-time" application, this might be true. But for a major application, Merge Replication probably is the way to go. Merge Replication provides capabilities that are not matched in RDA (e.g., conflict resolution, Ranged Identity columns, and dynamic partitions). These declarative capabilities, which are not built into RDA, are extremely difficult to hand-code into an RDA application. The deeper you go into enterprise application development, the more you need Merge Replication. Although a good replication scenario requires up-front thought, planning, and design, the resulting reduction in code necessary to complete the application makes it worth the effort in the development of full-scale applications.

8.6 Using Data Synchronization Services

Another option for data synchronization is to use a Data Synchronization Service. Data Synchronization Services are new to .NET Framework 3.5. They provide more flexibility than Merge Replication, and they do so by

Download at WoweBook.Com

shifting much of the development workload from the database administrator to the application developer.

A discussion of Data Synchronization Services could easily fill an entire chapter, perhaps even an entire book. What we will do here is provide some basic background information and then walk through the development of a simple Data Synchronization Service; one that is specific to .NET Compact Framework clients.

8.6.1 Understanding Data Synchronization Service Requirements

A Data Synchronization Service is a Web service. Like any Web service, it is accessible via a URL and contains a class of functionality whose methods can be invoked via a Web protocol. The standard for Data Synchronization Services, that is, the interface definition specified by the .NET Framework, mandates only four methods.

1. `ApplyChanges`: Receive all changes that have occurred on the client since the last exchange of data (i.e., since the last synchronization) and apply them on the server.
2. `GetChanges`: Retrieve all changes that have been made by other uses on the server since the last synchronization and deliver them to the client.
3. `GetSchema`: Retrieve schema information regarding the tables that participate in the synchronization from the server and deliver it to the client.
4. `GetServerInfo`: Retrieve the name and description of the tables that participate in the synchronization from the server and deliver them to the client.

Normally, the service combines the first two methods in the preceding list into a single method call named `Synchronize`.

Although few in number, these methods represent an extensive amount of functionality, for implied in `ApplyChanges` and `GetChanges` is concurrency conflict detection. That is, the service must work in conjunction with the client application and the server-side data store to detect whether the client is attempting change data that was modified on the server by some other user as it was being modified on the client. Furthermore, the service should give the client a choice between the following.

Download at WoweBook.Com

- The modification of the row does not occur and the client is so notified.
- The modification of the row does occur at the server, regardless of the impact on the changes made by the other user.

The second option is often referred to as "Last updater wins," but "Forced update" is a more correct phrase.

For a data store to be able to provide concurrency conflict detection, it must be able to track changes. Any data store that is to be used by a Data Synchronization Service must be able to do this. SQL Server CE 3.5 SP1 and SQL Server 2008 have a built-in change tracking capability that was designed with Data Synchronization Services support as a requirement. SQL Server 2005, on the other hand, has no built-in change tracking support. As will become apparent when we develop our sample application later in this chapter, the Configure Data Synchronization Wizard will generate, execute, and store SQL scripts that will add this capability to selected tables of a SQL Server 2005 database.

This is a less than ideal solution. Moving the application's back-end component to a new server means running the scripts at the new server (and possibly running the undo scripts at the old server). Many of you have probably added some form of change tracking to a SQL Server database in support of an application that you have developed and have encountered similar database maintenance overhead. In our upcoming sample, we will use SQL Server 2005 as our server-side data store. This will allow us to see the Designer-generated SQL code and the impact that it has on the Northwind database.

Interestingly, the change tracking needed for Data Synchronization Services does not require that a *before* and *after* image of a modified row be maintained; only that the time and user of the modification be tracked. The image of a deleted row, however, must be maintained in the database. If the client submits, and is determined to force acceptance of, an update of a row that was concurrently deleted at the server, that row first must be undeleted at the server and then the client's update applied. This can happen only if the deleted row is still available at the server. Rows that have been deleted from one table are normally maintained in a separate table that is referred to as the *tombstone* version of the original table.

Given this background information, we can begin development of our own Data Synchronization Service and a .NET Compact Framework application that will be a client of that service.

Download at WoweBook.Com

8.6.2 Building a Data Synchronization Service

For our Data Synchronization Service we'll implement a very simple example; one that updates some of the `Customers` of the Northwind database. We develop our service, and its client application, through a specific sequence of steps. Not only is this sequence probably the easiest way to do the development, but it also helps to illustrate the nature of a Data Synchronization Service and its relationship to the .NET Compact Framework client application. Specifically, we will do the following.

1. Create a Visual Studio 2008 solution that contains a Windows Communication Foundation (WCF) Service Library project and a Smart Device project. The former will evolve into our Data Synchronization Service; the latter will be our client application.
2. Use the Sync Services Designer Wizard to generate code, some of which will be inserted in the service application and some into the client application. Most importantly, the code that is inserted into the client application will talk to two synchronization providers: one that connects to SQL Server CE on the device and one that connects to the server-side data store.
3. Modify and complete the service application. Some of the changes we must make, such as specifying the URL, are applicable to any Data Synchronization Service; others, such as the Web binding to be used, will make the service accessible to a mobile device client.
4. Once the service definition is complete, turn to our client project and add a Web reference to that service, thus generating proxy code in our client application. This code, also, will need to be modified, mainly because it will contain class definitions that we do not want; class definitions that conflict with definitions that are already in the Framework Class Library.
5. Once our client has the correct code for accessing the service, add the core functionality of our application.

A simple diagram of the completed application is shown in Figures 8.34 and 8.35 located near the end of this chapter. Readers are encouraged to view these figures as we proceed through the development of the application.

Let us begin.

Download at WoweBook.Com

> **NOTE Required Software**
>
> Before starting development of a Data Synchronization Service and its consuming Smart Device client application, ensure that you have the necessary software on your development machine. You'll need Visual Studio 2008 SP1 or later for development. You will also need Microsoft Synchronization Services SP1 (Devices) or later. (Support for desktop clients comes with Visual Studio 2008, but for .NET Compact Framework clients you will need this separate download.) For the server-side data store, SQL Server 2005 or later is needed; SQL Server 2008 makes life easier; an Express version of either will do just fine. And lastly, for the client-side data store you'll need SQL Server CE 3.5 SP1 or later.

8.6.2.1 *Beginning Development of Both Projects*

We begin the actual development by creating a solution, named `DataSyncDemo`, containing two projects: a `Visual C# / WCF / WCF Service Library` project named `SyncClnt` and a `Visual C# / Smart Device / Smart Device Project / Device Application` named `SyncServ`. Figure 8.17 shows the resulting Solution Explorer window.

As we can see from Figure 8.17, we have a device application with a default form defined and a service application with a default WCF Service and contract defined. These last two are of no use to us, and we will delete them shortly. What we want is not a WCF Service but rather a Data Synchronization Service. To have Visual Studio generate the starter code for us, we need to invoke the Configure Data Synchronization Wizard. To do that, we need to add a Local Data Cache item to our project.

"Which project?" you ask. Normally, it would not matter which project, for the wizard will generate code into both projects, which is why we

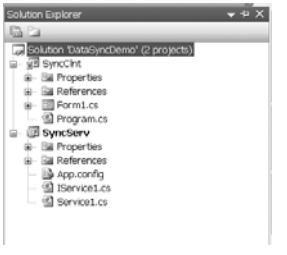


FIGURE 8.17: Server and Client Projects in the Same Solution

Download at WoweBook.Com

needed both projects to be in the same solution. Whenever you are adding a Local Data Cache item, keep the following in mind: A Local Data Cache item consists of both server-side and client-side components. It is easiest if you think of yourself as adding the Local Data Cache to the entire solution, not to a single project within that solution.

In actuality, the term *Local Data Cache* is misleading. It is not a memory-resident cache such as the Web application `Cache` object is, for its data is maintained in a SQL Server CE database. It is not just data, for it includes the components necessary to synchronize that data with the back-end data store. And although it resides on the local device, it is frequently communicating with the server and bringing data that originated on the server down to the device. Perhaps the best name would be the class name of its keystone object, `SyncAgent`.

In our situation it does matter to which project we try to add the Local Data Cache, for Smart Device projects do not provide you the option of adding a Local Data Cache item. Therefore, we must add a new Local Data Cache item to the service project, as shown in Figure 8.18.

This will result in the appearance of the Configure Data Synchronization Wizard, shown in Figure 8.19. First, we make our Server Connection selection to choose the connection to the server database. You probably have a data source connection already defined within your development environment for the target database. If so, select it; if not, select New and create one. Once we have made our selection, the Client Connection drop down and the Add button become enabled. We can either select a client connection or have a new one created for us. If we select New, the wizard will not ask us for further information; it knows where the client-side database will be located, on the device, and it knows that it will be a SQL Server

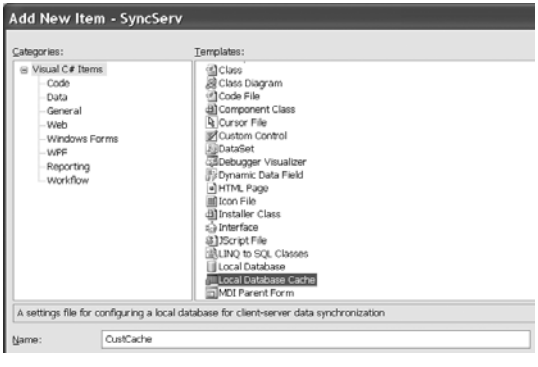


FIGURE 8.18: Adding a Local Data Cache Item

Download at WoweBook.Com

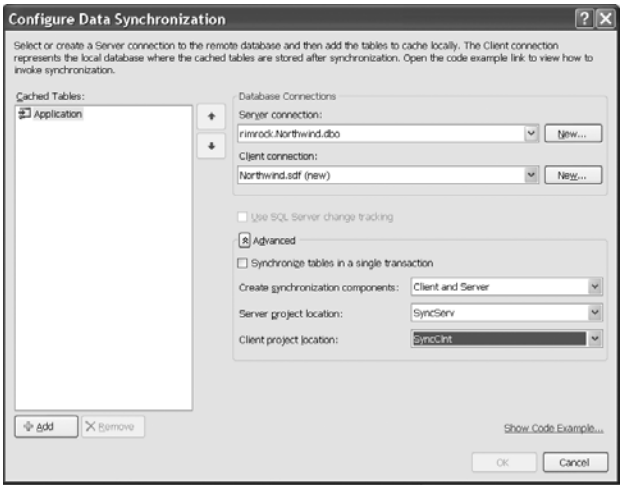


FIGURE 8.19: Configuring Data Synchronization

CE database. The wizard will simply generate the code to create and use a SQL Server CE database file on the device.

The "Use SQL Server change tracking" option is disabled because we chose a SQL Server 2005 database rather than a SQL Server 2008 database. Since SQL Server 2005 does not have built-in change tracking, the option is disabled.

If the selected SQL Server 2005 database has not had the necessary scripts run against it to create a change tracking capability, the word "(New)" will appear next to the choices in the Server Location drop downs. If this is the case, when we finish using the wizard, it will generate SQL scripts and display a dialog asking us whether we want the scripts run now and where we want them stored.

If asked, the wizard will modify the selected Northwind database, adding/altering the tables and triggers necessary to provide a change tracking capability sufficient for use with a Data Synchronization Service.

Click on the Advanced chevron and you will be able to tell the wizard that you want code generated for both the service and the client. Be sure to select the correct projects in the Server Project and Client Project drop downs. When you have done this you should be in the state shown in Figure 8.19.

At this point, we can click on the Add button and begin to specify which server-side tables will participate in the synchronization and provide some information about how the rows of each are to be synchronized, as shown in Figure 8.20.

Download at WoweBook.Com

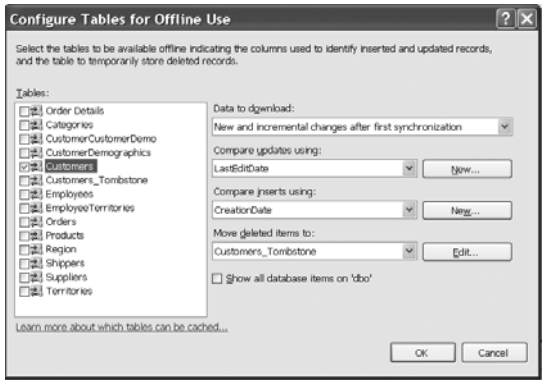


FIGURE 8.20: Specifying Tables

When we click OK twice to dismiss both dialog boxes, the Designer adds some references and writes some code into both projects. We cannot see what that code is yet, for a new dialog box has appeared; but some of the newly added code relates to the addition of the earlier mentioned SQL Server CE database file onto our device store. Since the wizard is providing for the storage of data on the client, why is the new dialog box (see Figure 8.21) offering to include a data set on the client as well? The answer is, because data tables are data-bindable, whereas a database is not. (See Chapter 4, Data Binding to Controls, and Chapter 6.) The wizard is merely offering to include an ADO.NET typed data set in your client project so that you can more easily display synchronized data to the user by moving it from the SQL Server CE database into a data set and then binding the data tables to controls.

FIGURE 8.21: Providing a Client-Side Data Set

Download at WoweBook.Com

The fact that this dialog is presented as part of adding a new Local Data Cache item to the Data Synchronization Services project hides the fact that the data set will be added to the client project, not to the Data Synchronization Services project. Always remember, a Local Data Cache item consists of both service components and client components.

We'll accept the offer of a typed data set in our device client application and dismiss the dialog. A look at the Solution Explorer window, shown in Figure 8.22, reveals that much has been added to our projects.

On the client side, references to two `Microsoft.Synchronization` libraries have been added. The `SynchProvider` and `SynchAgent` classes (more on these in a minute) have been added into `CustCache.Client.sync`. Also added to the client project is the SQL Server CE database file and the ADO.NET typed data set. What has not been added is the proxy code necessary for calling the service. This cannot be done at this time because we have not yet finished developing the service; therefore, a Web reference to the service cannot be set, and therefore the proxy cannot be generated.

The `SynchProvider` and `SynchAgent` classes that are defined in the newly added CustCache.Client.Designer.cs file know how the data is to be synchronized and where it is located on the device, for we provided that information to the wizard. What they do not know is what Data Synchronization Service will be used. That will not be known until we have completed the service project, run it, and set a client project Web reference to it. In other words, the `SynchProvider` and `SynchAgent` classes will work in conjunction

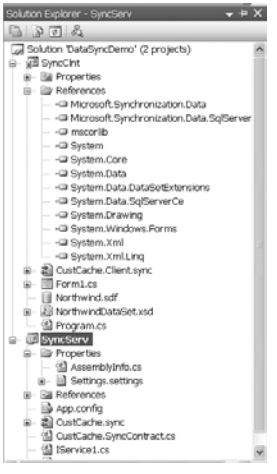


FIGURE 8.22: Viewing the Designer-Added Components

Download at WoweBook.Com

with the to-be-written `Proxy` class to synchronize data between device and server. In production, if the service is moved to a different server, the `SynchProvider` and `SynchAgent` classes will remain unchanged; the `Proxy` class, or at least its `URL` property, will need to be updated.

At this point, the Configure Data Synchronization Wizard has written as much skeleton code as it can. We need to finish developing the service and then complete the client.

8.6.2.2 *Completing the Service*

Our application is being developed, in part, as a learning tool. To help us present the completion of each individual project as clearly as possible, we are going to separate each project into its own Visual Studio 2008 solution. There is no technical need for doing this, but it does make it easier to focus in on the Data Synchronization Services project first and then the Smart Device client project second.

We begin our work on the service by removing two files that came with the WCF Service project template, as they are not needed for a Data Synchronization Service. We mentioned at the time we added the project that we would be removing them. So, we now delete the IService1.cs and Service1.cs files from the project.

What we now are interested in is the code in the CustCache.SyncContract.cs file. In writing this file, the Configure Data Synchronization Wizard did three things: It made one wrong assumption about our consuming client application, it tried to help us transition the service from a WCF Service to a Data Synchronization Service, and it had to specify a default URL. To complete our service application, we need to address all three.

First, the wrong assumption: The wizard did not know that our client would be a Smart Device client, and that a Smart Device client must use XML serialization when passing objects to/from the service. We need to specify XML serialization, and we do that by adding an `[XmlSerializerFormat()]` code attribute to the `ICustCacheSyncContract` definition that is located at the end of the CustCache.SyncContract.cs code file, as shown in Listing 8.10.

Listing 8.10: The Attributed `ICustCacheSyncContract` Class

```
[ServiceContractAttribute()]
[XmlSerializerFormat()]
public interface ICustCacheSyncContract
{
```

Download at WoweBook.Com


```
    [OperationContract()]
    SyncContext ApplyChanges(
                  SyncGroupMetadata groupMetadata,
                  DataSet dataSet,
                  SyncSession syncSession);

    [OperationContract()]
    SyncContext GetChanges(
                  SyncGroupMetadata groupMetadata,
                  SyncSession syncSession);

    [OperationContract()]
    SyncSchema GetSchema(
                 Collection<string> tableNames,
                 SyncSession syncSession);

    [OperationContract()]
    SyncServerInfo GetServerInfo(
                     SyncSession syncSession);
  }
```

Second, the generated code to help transition the service from a WCF Service to a Data Synchronization Service: The project's App.Config file contains elements that are applicable to a WCF Service, not to a Data Synchronization Service. At the top of the CustCache.SyncContract.cs code file, as comments, are the suggested replacements that the Configure Data Synchronization Wizard has provided for us. The instructions in the comments tell us to uncomment two elements and replace the corresponding elements in the App.Config file with the uncommented elements.

As we do so, we'll need to make two changes. We will need to replace the default URL with our service's actual URL. We must do this, for `localhost` is meaningless within a Smart Device application (the wizard's third miscalculation). In our case,

```
http://localhost:8080/CustCacheSyncService/
```

becomes

```
http://192.168.254.2:1824/CustCacheSyncService/
```

And because we must support Smart Device clients, we must use basic HTTP binding rather than Web Services HTTP binding. Thus,

```
binding="wsHttpBinding"
```

Download at WoweBook.Com

becomes

```
binding="basicHttpBinding"
```

Once we do so, we have a valid App.Config file.

We also have completed the Data Synchronization Service. We can now turn our attention to our Smart Device client project.

8.6.2.3 *Completing the Client*

When we changed the binding specification of our service to `basicHttpBinding`, it became a conventional Web service, something that .NET Compact Framework applications have been communicating with for some time now. And almost always, we enable that communication in our Smart Device application by adding a Web reference to the Web service. To add a Web reference, the service must be running, so we go to our service project and run it.

> **NOTE**
>
> It is important that your client program not be running at this time. If it is, you will be unable to add a Web reference to your client project. If you are still developing both projects within the same solution, running the service will cause the client to be deployed. If that happens, you will need to cancel that deployment.

Once the service is up and running, a default client program is automatically started, a portion of which is shown in Figure 8.23.

We are interested in the URL displayed on the second line in the window. It should be exactly what we specified in the `baseAddress` element of the App.Config file, suffixed with "mex" (which indicates the location for metadata exchange and which you need to ignore when setting the reference). As

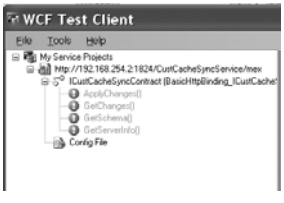


FIGURE 8.23: The Test Client Program Displayed

Download at WoweBook.Com

we saw earlier, in our case this URL is http://192.168.254.2:1824/CustCacheSyncService/.

Thus, we turn to our client project and add a Web reference. In the Add Web Reference dialog, we specify the URL exactly as we specified it in the service's `baseAddress` element, beginning with "http" and ending with the trailing "/". Then we click on the Go arrow, browsing to the service and receiving the service description page, shown in Figure 8.24, in return.

We choose a meaningful name for our Web reference and add it to our project. Once it has been added, we expand the files in our Solution Explorer window, as seen in Figure 8.25, to show the generated proxy code file, Reference.cs.

Like other files before it, this file also needs to be modified.

When the proxy code within Reference.cs was generated, one class definition that we need was written, as were several that we do not need. To make a long story as short as we can: The methods of our Data Synchronization Service pass parameters of data types that are defined in the `Microsoft.Synchronization` namespaces. As an inevitable result of proxy code generation, these classes are redefined as they are added to the proxy class definition; specifically, they are redefined as carriers of data only. That is, all method definitions are lost. Since we wish to use the `Microsoft.Synchronization` versions and their defined methods, we must delete the generated versions from the proxy code, now and anytime we regenerate the proxy code!

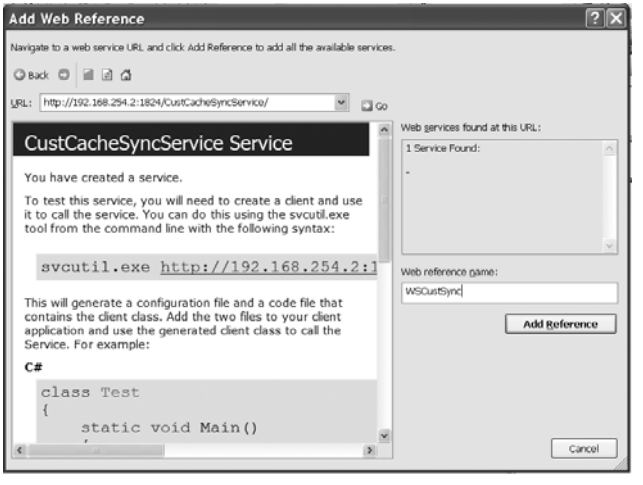


FIGURE 8.24: Setting the Web Reference

Download at WoweBook.Com

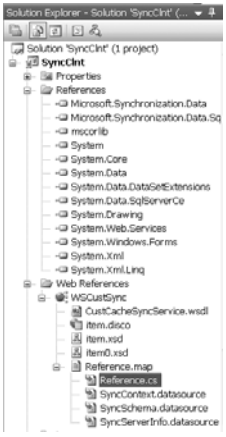


FIGURE 8.25: Solution Explorer Showing Generated Proxy Files

The one class that we do not want to delete should be easy to find. It inherits from `SoapHttpClientProtocol`, should be the first class in the code file, and should have a name that ends with `SyncService`. So, open Reference.cs and look for the following line of code:

```
public partial class CustCacheSyncService
```

We delete *all other* classes and add the following `using` statement at the top of the file:

```
using Microsoft.Synchronization.Data;
```

We do not need to add references to the `Microsoft.Synchronization` libraries, for the Configure Data Synchronization Wizard has already done that for us. When it comes to dealing directly with Visual Studio, its templates, and its wizards, we are done. What is left for us to do is write the functionality of our application.

We'll add just enough functionality so that we can walk through the following scenario.

1. Synchronize data with the server.
2. Make some nonconflicting changes at the device and on the server.
3. Synchronize data with the server.
4. Make some conflicting changes at the device and on the server.
5. Synchronize data with the server.
6. After each synchronization, examine results, data, and errors.

Download at WoweBook.Com

FIGURE 8.26: The Main Form and Its Controls

We begin by opening the form in design mode and then adding a full-screen docked `DataGrid` control plus a two-entry menu to it, as shown in Figure 8.26.

Then we turn to the form's code file. We'll need to add the following: namespace declarations for synchronization, for our proxy classes, and for our typed data set; a routine to modify some customer rows; a routine to synchronize our changes with the server; and code to display the results of that synchronization.

We add the namespace declarations at the start of the code thusly:

```
using Microsoft.Synchronization.Data;
using SyncClnt.WSCustSync;
using SyncClnt.NorthwindDataSetTableAdapters;

namespace SyncClnt
{
   public partial class FormData : Form
```

In the form's `Load` event we create a typed data set and a table adapter for later use, like so:

```
private void FormData_Load(object sender,
                           EventArgs e)
{
   dsetNorthwind = new NorthwindDataSet();
   daptCust = new CustomersTableAdapter();
}
```

To modify some data at the device, modifications that will then need to be synchronized with the server, we iterate through the rows of the

Download at WoweBook.Com

Customers data table. Every customer whose CustomerId starts with the letter *A* will have "** " injected into the start of his CustomerName field. This code appears in Listing 8.11.

Listing 8.11: The Data Modification Routine

```
private void mitemModifyData_Click(object sender,
                                   EventArgs e)
{
   Cursor.Current = Cursors.WaitCursor;
   foreach (NorthwindDataSet.CustomersRow rowCustomer
            in dsetNorthwind.Customers)
   {
      if (rowCustomer.CustomerID.StartsWith("A"))
      {
         rowCustomer.CompanyName =
            "** " + rowCustomer.CompanyName;
      }
   }
   daptCust.Update(dsetNorthwind.Customers);
   Cursor.Current = Cursors.Default;
}
```

Listing 8.12 shows the routine for doing the synchronization of our changes with the server's data store, via the service. It is well annotated. Any class name that contains the word Cust was written by one of the wizards that we walked through. Any class name that contains the word Sync, but not the word Cust, is defined in one of the Microsoft.Synchronization libraries that the project references.

Listing 8.12: The Data Synchronization Routine

```
private SyncStatistics SyncCustomers()
{
   SyncStatistics syncStats;
   try
   {  // Create a proxy object for the synch service.
      //    Use it to create a generic proxy object.
      ServerSyncProviderProxy prxCustSync =
          new ServerSyncProviderProxy(
             new CustCacheSyncService());

     // Create the sync agent
     //    for the Customer table.
      //    Assign the proxy object to it.
      //    Set it for bi-directional sync.
      CustCacheSyncAgent syncAgent =
         new CustCacheSyncAgent();
```

Download at WoweBook.Com


```
        syncAgent.RemoteProvider = prxCustSync;
        syncAgent.Customers.SyncDirection =
           SyncDirection.Bidirectional;

        // Sync changes with the server.
        syncStats = syncAgent.Synchronize();

        // The above call to syncAgent.Synchronize
        //    retrieved all Customer table changes
        //    that had were made at the server
        //    between the previous call and this
        //    call and placed them in our
        //    SQL Server CE Northwind database here
        //    on the device.
        // We need to display that table to
        //    our user.  We will use a data bound
        //    data table to do so.
        daptCust.Fill(dsetNorthwind.Customers);
        dgridCustomers.DataSource =
           dsetNorthwind.Customers;
     }
     catch (Exception exSync)
     {
        MessageBox.Show(exSync.Message);
        throw;
     }
     return syncStats;
}
```

Note that the routine first creates a proxy object for our service and then uses that proxy object in the creation of another, more generic, proxy object. Then the `SyncAgent` is created and is passed a reference to the generic proxy object. Once any other `SynchAgent` properties have been set, the agent knows what needs to be synchronized and what service to use to do it. The agent's `Synchronize` method then does the synchronization and returns the results in a `SyncStats` structure.

The `SynchAgent` object is the keystone object of data synchronization. Although it is located on the device, you want to think of it as sitting between the server-side data store and the client-side SQL Server CE database. Anything that your application needs to do at runtime to influence or alter the behavior of a synchronization will be done by invoking methods of, or setting properties of, or responding to events raised by the `SynchAgent` object or one of its contained objects.

For example, later in this demo we will need to obtain information about rows that failed to synchronize. We will do that by handling an

Download at WoweBook.Com

event that is raised by an object that is contained within the `SynchAgent` object.

One last comment on our synchronization code: Immediately after the synchronization, we load the rows of the SQL Server CE's Northwind `Customers` table into the typed data set's `Customers` table. Normally, this is not a good idea. The data set contains the data that we attempted to upload to the server. After synchronization, the SQL Server CE database contains the data that was received from the server. If the synchronization of a row failed, the client-side version will be in the data set table; the server-side version will be in the SQL Server CE table. We might want access to both as we are reconciling the differences.

In the menu selection that initiates the synchronization, we do some UI setup, call our synchronizing routine, and display the results. Listing 8.13 shows this code.

Listing 8.13: The Synchronization Menu Handler

```
private void mitemSynchronize_Click(object sender,
                                    EventArgs e)
{
   Cursor.Current = Cursors.WaitCursor;
   SyncStatistics syncStats = SyncCustomers();
   MessageBox.Show(
      string.Format(
              "{1} {2}.{0}" +
              "{3} {4}.{0}" +
              "{5} {6}.{0}" +
              "{7} {8}.{0}",
              Environment.NewLine,
              "Rows sent - total",
              syncStats.TotalChangesUploaded,
              "Rows sent - succeeded",
              syncStats.UploadChangesApplied,
              "Rows received - total",
              syncStats.TotalChangesDownloaded,
              "Rows received - succeeded",
              syncStats.DownloadChangesApplied));
   Cursor.Current = Cursors.Default;
}
```

That's it for adding code to our application. Now it's time to run the application and see what happens.

Download at WoweBook.Com

FIGURE 8.27: The Device Synced with the Server

We start by running the application and selecting Synchronize from the menu, twice if necessary, to ensure that the data on the device and on the server are identical, as shown in Figure 8.27.

Now we use our data modification routine to make changes to our client version of the data; see Figure 8.28. And we use SQL Server's Management Studio to make nonconflicting changes to the data on the server. We insert a few $s into the `CompanyName` column of the first two `B` customers, as shown in Figure 8.29.

Now we can synchronize. Once again we select our Synchronize menu selection. This time we receive the response shown in Figure 8.30.

From Figure 8.30, we can see that our four modified rows were sent to the server, modified at the server, and *echoed back to us.* Why were our modified rows returned to us? For the same reason that so many of you have written code that immediately retrieved newly inserted/updated rows. Inserted/updated rows often contain values that were supplied

FIGURE 8.28: Pending Client-Side Changes

Download at WoweBook.Com

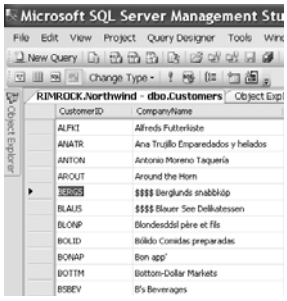


FIGURE 8.29: Pending Server-Side Changes

FIGURE 8.30: Viewing the Nonconflicting Results

not by the client but rather by the database engine. System-assigned primary keys, GUID columns, default values, and trigger-generated values are all examples of this. Rather than the client having to ask for the newly inserted/modified row, the service returns the row to the client automatically.

Also, we can see that the rows that were added by some other client, in this case Management Studio, are also delivered to our client, as we would expect.

Now let's try some conflicting updates. First, we set the data back to what it was when we began our demo. Again we synchronize, twice if necessary, to ensure that the data in both locations is identical. (Refer back to Figure 8.27.) As before, we modify data on the device by running the data modification routine. (Refer back to Figure 8.28.) But this time, in Management Studio, we modify two of the same rows that we are concurrently modifying on the device, as shown in Figure 8.31. When we synchronize the device, we receive the results shown in Figure 8.32.

Download at WoweBook.Com

FIGURE 8.31: Modifying the Two Rows

FIGURE 8.32: Viewing the Conflicting Results

Of the four rows that were returned to our device, two are the two rows that we sent to the server and were accepted at the server, including any values that were added on the server as the row was being inserted/updated. The other two are the two rows that we sent to the server that were not accepted at the server. What is sent back to the device is the row *as it now exists on the server*. That is, we now have the server-side version of the row, which we can present to the user to see whether she wants to resubmit the same update.

8.6.2.4 *Handling Concurrency Errors*

As we have just seen, rows may fail to synchronize; changes made at either end might not be acceptable at the other end. Therefore, we add one last bit of functionality to our demo. For each row that fails to synchronize, we need to determine which row it was, where it was rejected (server or client), and why it was rejected.

The possible reasons for why a row might fail to synchronize are specified by Microsoft in the `ConflictType` enumerator; see Table 8.4.

Download at WoweBook.Com

TABLE 8.4: Synchronization Errors

ConflictType	Meaning
ClientDeleteServerUpdate	The client deleted a row that the server updated
ClientInsertServerInsert	The client and server both inserted a row that has the same primary key value
ClientUpdateServerDelete	The server deleted a row that the client updated
ClientUpdateServerUpdate	The client and the server updated the same row

Note that ClientDeleteServerDelete is not an error. Any scenario in which the client wanted the row deleted and the row is deleted is not an error. The fact that the row was already deleted, and may have had different values in it from what the client thought it had, does not matter.

We start with the SyncAgent object; the object whose Synchronize method executes and monitors the entire synchronization. The SyncAgent object holds references to two SyncProvider objects: one that it uses to communicate with the server and one that it uses to communicate with the client. In our application, one is a SQL Server 2005 provider and one is a SQL Server CE provider. During synchronization, whenever a row cannot be inserted, updated, or deleted, at either the client or the server, because of a currency conflict, the appropriate provider raises an ApplyChangeFailed event. To process conflicts, we need to handle that event in our application. To illustrate, we will detect failures that occur at the client; highly similar code would be used to detect server-side failures.

It is the SyncAgent's LocalProvider property object that will raise the ApplyChangeFailed event. To make this event visible to IntelliSense, we need to cast the LocalProvider to the specific class of provider that will reside in the property at runtime; normally either SqlCeClientSyncProvider or DbServerClientSyncProvider. Thus, our event signup code is:

```
((SqlCeClientSyncProvider)
   (syncAgent.LocalProvider))
      .ApplyChangeFailed +=
         new EventHandler
            <ApplyChangeFailedEventArgs>
               (FormData_ApplyChangeFailed);
```

Download at WoweBook.Com

The actual handler is shown in Listing 8.14. The `EventArgs` that we are handed contains a `Conflict` object, which, in turn, contains the reason for the row being rejected and also the row itself. The `Conflict.ConflictType` property contains the reason, while the `Conflict.ClientChange` property contains an untyped data table that holds the rejected row.

Listing 8.14: The Synchronization Error Handler

```
void FormData_ApplyChangeFailed(
              object sender,
              ApplyChangeFailedEventArgs e)
{
   MessageBox.Show(
     string.Format(
             "{1}{0}{2}.{0}{0}" +
             "{3}   {4}.{0}{0}" +
             "{5}{0}{6}.{0}",
             Environment.NewLine,
             "Conflict Type:",
             e.Conflict.ConflictType.ToString(),
             "CustomerID:",
             e.Conflict.ClientChange.Rows[0]
                       ["CustomerID"].ToString(),
             "CompanyName:",
             e.Conflict.ClientChange.Rows[0]
                       ["CompanyName"].ToString()));
}
```

Figure 8.33 shows the result from a scenario in which a row was inserted at the server concurrently with a row of the same primary key being inserted at the device.

Figure 8.33: Displaying the Failed Row

Download at WoweBook.Com

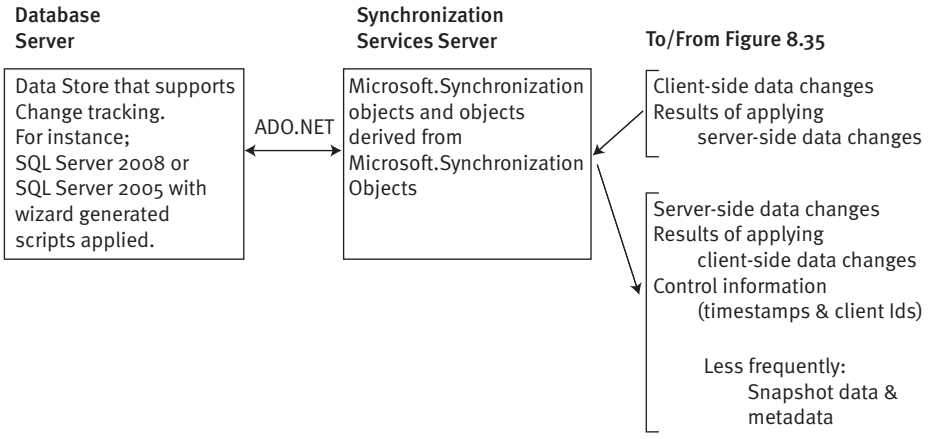


FIGURE 8.34: The Smart Device Application

This concludes our brief demonstration of developing a Data Synchronization Service. Figures 8.34 and 8.35 diagram our resulting Smart Device application and Data Synchronization Service, respectively.

8.6.2.5 *Some Closing Notes on Data Synchronization Services*

As the need arises to add functionality to your solution, such as automatic conflict resolution or business logic, you can do so in any of several ways, three of which are mentioned here.

- Add logic to the client program, such as we did with our simple conflict detection logic. This is a good solution for logic that is applicable to this client only.
- Add logic to the service program. This is a slightly unstable solution, as it may need to be redone if future modifications are generated by rerunning the Configure Data Synchronization Wizard.
- Derive classes that contain your functionality from the `Microsoft.Synchronization` classes and use these classes in your service. This is a good solution for logic that must be applied to the data regardless of the application that is manipulating that data.

In summary, Data Synchronization Services do the following:

- Provide a complete solution for the synchronization of data that is gathered by Occasionally Connected Applications

Download at WoweBook.Com

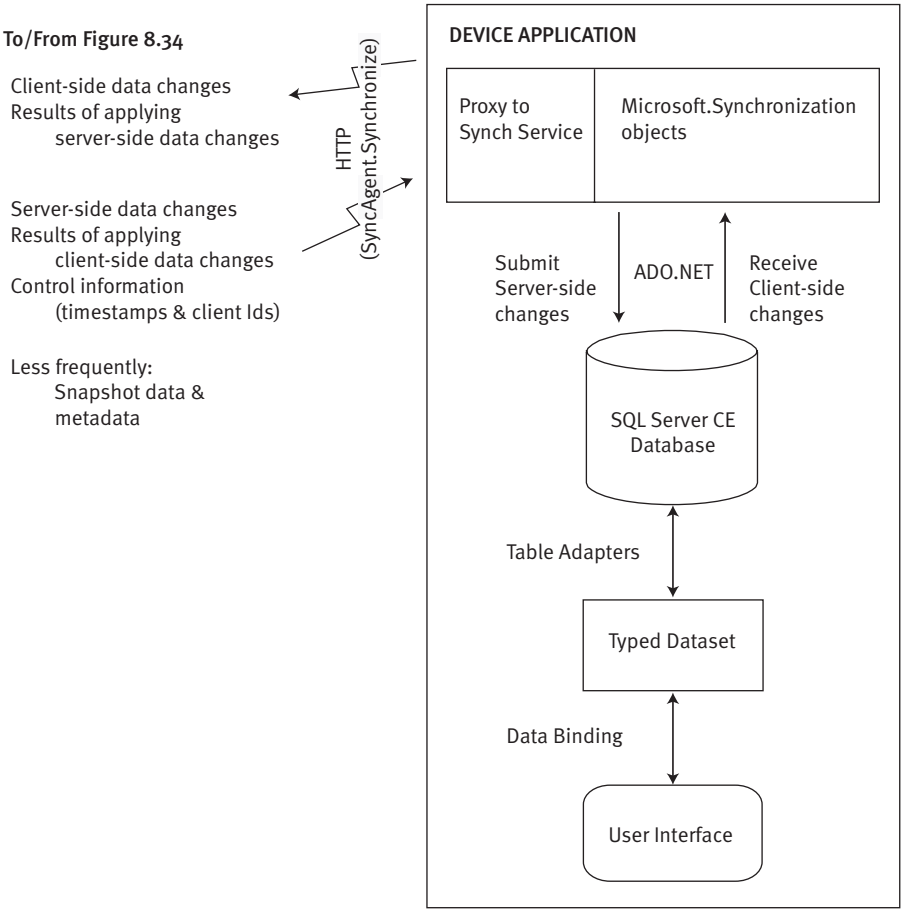


FIGURE 8.35: The Data Synchronization Service

- Allow the Smart Device application to handle synchronization failures
- Encapsulate the synchronization infrastructure functionality within a `SyncAgent` object and its contained objects
- Provide more capability than RDA (albeit by being more complex)
- Differ from Merge Replication in that they are controlled by the developer rather than the database administrator, and this allows for more application of business logic

8.7 Conclusion

Developers who are experienced with building SQL applications on the Windows desktop will find much that is familiar with SQL Server Windows

Download at WoweBook.Com

CE Edition. That is, you have a host of old standbys, such as the SQL database statements as well as the ADOCE and the OLEDB CE programming interfaces, to work with. What is new is the connectivity model that SQL Server CE provides for the merging of mobile data into a central data store.

Developers who have built Web-based applications will also find much that is familiar here. SQL Server CE provides a high-powered, client-oriented database engine that lets you coordinate the movement of data between mobile, occasionally connected devices and a central data store. And of course, it also can run as a stand-alone database quite nicely.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

9
The Remote API

The preceding chapter discussed options for connecting mobile systems to desktop and server systems. This chapter covers the Remote API (RAPI), another option for connecting devices to desktop systems.

THIS CHAPTER COVERS RAPI, a Win32 API that runs on desktop systems for accessing Windows CE–powered devices. Most RAPI functions access a device's object store, while a few others let you connect to provide interesting ways to access a Windows CE system from a desktop system.

RAPI relies on one of two desktop-based extensions to Windows: ActiveSync (for Windows XP and earlier) and Windows Device Center (on Windows Vista and later). We assume that you have one of these two packages installed.

9.1 RAPI Fundamentals

Windows CE RAPI is a remote procedure call (RPC) facility. Desktop programs (both native and managed) call RAPI functions to cause some result to occur on a device. It is not a general-purpose RPC mechanism but instead is specialized because it supports a subset of all the device-side functions that might be available.

Many RAPI functions access stored data. Using RAPI, for example, a desktop-side program can open a file and read and write its contents as though that file were on the desktop. RAPI lets a desktop program open nodes in the system registry and create keys, as well as read and write values on existing keys. You have complete access to the property databases

Download at WoweBook.Com

in the object store, so you can create a database, delete a database, add or remove database records, and read or write individual property values.

RAPI function names have a `Ce` prefix to distinguish them from non-RAPI functions. For example, the Win32 file open function is `CreateFile`; its RAPI equivalent is `CeCreateFile`. Once opened, the RAPI function to read a file is `CeReadFile` and the RAPI close file function is `CeCloseHandle`.

Some RAPI functions have identical names as their device-side counterpart. This is true for CE-specific functions, which already have a `Ce` prefix. This means that, for example, `CeReadRecordProps` is the name of both a desktop-side RAPI function and a device-side function.

9.1.1 Available RAPI Functions

RAPI functions reside on Windows XP and on Windows Vista in the file rapi.dll, which gets installed to a desktop system when ActiveSync is installed. On Windows Vista, this library gets installed when the Windows Mobile Device Manager has been installed.

We divide the 78 RAPI functions into the following eight groups:

1. RAPI Support
2. Run Program/Load DLL
3. File System
4. Registry
5. CE Property Database
6. Object Store Queries
7. System Information
8. Remote Windowing

Table 9.1 provides details of each group. Although most RAPI functions have an equivalent Win32 function in Windows CE, a few RAPI functions have no Win32 equivalent, such as those used for RAPI management and support. In two cases—`CeFindAllFiles` and `CeFindAllDatabases`—functions were created specifically to boost performance. These two functions access a user-defined subset in a query, allowing for faster queries of available files or databases.

9.1.2 Building .NET ActiveSync Applications

Because ActiveSync applications run on the desktop, when you build them in Visual Studio .NET you do not create a smart-device application. Instead, you pick any of the various desktop project types. You can build

Download at WoweBook.Com

ActiveSync applications either as managed (.NET) applications or as unmanaged (Win32) applications. This book does not cover the Win32 approach because it is covered elsewhere.[1] This chapter focuses instead on building managed-code ActiveSync applications.

TABLE 9.1: Available RAPI Functions

RAPI Function	Description
Group 1: RAPI Support	
`CeRapiInit`	Is a blocking initialization function
`CeRapiInitEx`	Is a nonblocking initialization function that returns a Win32 event handle that gets signaled when a connection is complete
`CeRapiUninit`	Cleans up for `CeRapiInit` and `CeRapiInitEx`
`CeRapiGetError`	Queries for RAPI-specific errors
`CeGetLastError`	Queries for non-RAPI device-side errors (equivalent to `GetLastError`)
`CeRapiFreeBuffer`	Is a free buffer allocated by various functions, including `CeFindAllFiles` and `CeFindAllDatabases`
Group 2: Run Program/Load DLL	
`CeCreateProcess`	Starts a program running on the Windows CE device
`CeRapiInvoke`	Calls a named function in a named Win32 dynamic link library (DLL) on a Windows CE device. Does not support generic RPCs to DLLs because the function called must take a specific set of parameters
Group 3: File System	
`CeCloseHandle`	Closes a file opened with a call to `CeCreateFile` (or a database opened with `CeOpenDatabaseEx`)
`CeCopyFile`	Copies a file on the device to another location on the device (does not copy between the device and the desktop system)

Continues

1. For example, see Doug Boling's book, *Programming Windows Embedded CE 6.0 Developer Reference* (Redmond, WA: Microsoft Press, 2008).

Download at WoweBook.Com

TABLE 9.1: Available RAPI Functions *(Continued)*

RAPI Function	Description
`CeCreateDirectory`	Creates a directory in the file system of a device
`CeCreateFile`	Opens a file on the device, optionally creating it at the same time
`CeDeleteFile`	Deletes a file on the device file system
`CeFindAllFiles`	Searches for all files in a given directory that match a specific criterion, returning a subset of the data returned by the normal file enumeration functions. This reduction in data causes this function to run faster than the generic file enumeration functions
`CeFindClose`	Cleans up file enumeration begun with a call to `CeFindFirstFile`
`CeFindFirstFile`	Starts file enumeration in a given directory for specific search criteria
`CeFindNextFile`	Continues file enumeration begun with a call to `CeFindFirstFile`
`CeGetFileAttributes`	Queries the attribute of a file in the device file system
`CeGetFileSize`	Queries file size
`CeGetFileTime`	Queries creation, last accessed, and last modified date and time
`CeGetSpecialFolderPath`	Gets the pathname for shell file system folders
`CeGetTempPath`	Gets the path to the temporary directory
`CeMoveFile`	Renames an existing file or directory on a device. Files can be moved between different drives; directories can be moved only to a different location on the existing drive
`CeReadFile`	Reads a number of bytes from a file opened with a call to `CeCreateFile`
`CeRemoveDirectory`	Deletes an existing empty directory
`CeSetEndOfFile`	Truncates the file at the current position of the file pointer
`CeSetFileAttributes`	Modifies the file attribute flags to enable or disable the following bits: archive, hidden, normal, read-only, system, and temporary

Download at WoweBook.Com

TABLE 9.1: Available RAPI Functions *(Continued)*

RAPI Function	Description
`CeSetFilePointer`	Moves the file pointer in a file opened with `CeCreateFile`
`CeSetFileTime`	Modifies the directory entry for a file on a device to change the associated created, last accessed, or modified time
`CeSHCreateShortcut`	Creates a program shortcut on the remote device
`CeSHGetShortcutTarget`	Queries the path stored in an existing shortcut
`CeWriteFile`	Writes a number of bytes to a file opened with a call to `CeCreateFile`
Group 4: Registry	
`CeRegCloseKey`	Closes a registry key opened with a call to either `CeRegOpenKeyEx` or `CeRegCreateKeyEx`
`CeRegCreateKeyEx`	Creates a new registry key or opens an existing registry key
`CeRegDeleteKey`	Deletes a named subkey and the associated value in the registry. If the named subkey has subkeys under it, the call fails
`CeRegDeleteValue`	Deletes one value from the specified registry node
`CeRegEnumKeyEx`	Enumerates the subkeys of an indicated registry node
`CeRegEnumValue`	Enumerates the values in an indicated registry node
`CeRegOpenKeyEx`	Opens a registry node
`CeRegQueryInfoKey`	Queries details about a given registry node, including the number of subkeys and the length of the longest subkey name
`CeRegQueryValueEx`	Fetches a value from a registry node
`CeRegSetValueEx`	Writes a value to a registry node
Group 5: CE Property Database	
`CeCloseHandle`	Closes a database opened with `CeOpenDatabaseEx` (or a file opened with `CeCreateFile`)

Continues

Download at WoweBook.Com

TABLE 9.1: Available RAPI Functions *(Continued)*

RAPI Function	Description
`CeCreateDatabase`	Creates a new property database (obsolete—use `CeCreateDatabaseEx` instead)
`CeCreateDatabaseEx`	Creates a new property database with an associated GUID
`CeDeleteDatabase`	Deletes a database by object ID value
`CeDeleteDatabaseEx`	Deletes a database by object ID and GUID
`CeDeleteRecord`	Deletes a database record
`CeEnumDBVolumes`	Enumerates all mounted property database volumes
`CeFindAllDatabases`	Returns a subset of the data returned by the general-purpose database enumeration functions. The reduction in requested data causes this to run faster than the general-purpose enumeration functions
`CeFindFirstDatabase`	Starts enumerating all property databases in the object store
`CeFindFirstDatabaseEx`	Starts enumerating all property databases in the system or on the indicated database volume
`CeFindNextDatabase`	Continues enumeration started with `CeFindFirstDatabase`
`CeFindNextDatabaseEx`	Continues enumeration started with `CeFindFirstDatabaseEx`
`CeFlushDBVol`	Forces a save of database changes from program memory to storage
`CeMountDBVol`	Issues a mount request for a database volume
`CeOpenDatabase`	Opens a database (obsolete—use `CeOpenDatabaseEx` instead)
`CeOpenDatabaseEx`	Opens a database by name or by GUID
`CeReadRecordProps`	Reads a database record
`CeReadRecordPropsEx`	Reads a database record, optionally allocating memory from a provided Win32 heap
`CeSeekDatabase`	Sets the current database record based on selection criteria

Download at WoweBook.Com

TABLE 9.1: Available RAPI Functions *(Continued)*

RAPI Function	Description
`CeSetDatabaseInfo`	Modifies various database attributes, including name, type, and sort order
`CeSetDatabaseInfoEx`	Modifies various database attributes, including name, type, and sort order, as well as the GUID
`CeUnmountDBVol`	Unmounts a database volume mounted with an earlier call to `CeMountDBVol`
`CeWriteRecordProps`	Writes a record to the database
Group 6: Object Store Queries	
`CeOidGetInfo`	Queries the type of record located within the object store for a given object ID value
`CeOidGetInfoEx`	Queries the type of record in the object store, or for any mounted database volume, for a given object ID value
`CeGetStoreInformation`	Queries the total size and free space for the object store
Group 7: System Information	
`CeCheckPassword`	Compares an entered password to a mobile device password, as set in the Windows CE Control Panel. A password is not passed as a parameter in the normal RAPI startup functions because—when password protection is enabled on a device—ActiveSync automatically prompts for a valid password before allowing the RAPI connection to be established. You would likely check the password only for an extra measure of security before performing an especially sensitive operation
`CeGetDesktopDeviceCaps`	Queries the graphics capabilities of a remote device (this is equivalent to the `GetDeviceCaps` graphics capability query function)
`CeGetSystemInfo`	Queries the CPU and memory architecture of a remote device
`CeGetSystemMetrics`	Queries the size of elements on the user interface of the remote system, including the width and height of window borders, icons, scroll bars, etc.

Continues

Download at WoweBook.Com

TABLE 9.1: Available RAPI Functions *(Continued)*

RAPI Function	Description
`CeGetSystemPowerStatusEx`	Queries the battery status and whether wall power is available
`CeGetVersionEx`	Queries Windows CE operating system version information
`CeGlobalMemoryStatus`	Queries available memory load, amount of physical memory, and available virtual memory
Group 8: Remote Windowing	
`CeGetClassName`	Queries the window class name for a given window handle
`CeGetWindow`	Walks the window hierarchy of the remote device
`CeGetWindowLong`	Queries the state information for an indicated window
`CeGetWindowText`	Queries the text of an indicated window

If you plan to call ActiveSync from managed code, you need P/Invoke wrappers to access the Win32 ActiveSync libraries. We built two sets of wrappers—one in C# and the other in Visual Basic—that are included with the source code you can download from the book's source code Web site. We used the P/Invoke Wizard introduced in Chapter 3, Platform Invoke, and then fine-tuned the output by hand. As we mentioned earlier, you can download the P/Invoke Wizard from The Paul Yao Company Web site.[2]

9.1.3 RAPI Startup and Shutdown

Like other Win32 API function sets, RAPI has startup and shutdown functions. Before calling any other RAPI function, you must call a startup function: either `CeRapiInit` or `CeRapiInitEx`. The difference is that the first function is a blocking function and the second is not.

When startup is a success, your program does its work and then calls the shutdown function, `CeRapiUninit`. If you call a startup function, you must always call the shutdown function. This is true even if the startup fails. When we have run programs that did not follow this guideline, we've noticed that ActiveSync itself stops running properly until the offending program terminates.

2. Details are available at www.paulyao.com/pinvoke.

Download at WoweBook.Com

9.1.3.1 CeRapiInit *versus* CeRapiInitEx

At first glance, the CeRapiInit function seems the simplest way to establish a RAPI connection. But this is a blocking function, so a call at the wrong time causes a thread to hang, keeping a program in memory when it was supposed to have ended. By contrast, CeRapiInitEx is a nonblocking function.

The CeRapiInitEx function notifies the caller that a connection has been established through a Win32 event (you can wrap a .NET event object around a Win32 event). This function accepts a pointer to a RAPIINIT structure for input, and it returns an HRESULT result code. Here are the C# declarations for the CeRapiInitEx function and the required RAPIINIT structure:

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int CeRapiInitEx (ref RAPIINIT pRapiInit);

[StructLayout(LayoutKind.Sequential,Pack=4)]
public struct RAPIINIT
{
   public int cbSize;
   public IntPtr heRapiInit;
   public int hrRapiInit;
};
```

Handling Win32 HRESULT Values

Treat Win32 HRESULT values as int values to stay compatible with the Common Language Specification. Avoid an overflow in C# with the unchecked keyword as shown here:

```
int E_FAIL = unchecked((int)0x80004005);
```

To call the CeRapiInitEx function, first allocate a RAPIINIT structure and set the cbSize member to the structure size. If you set the incorrect size, the CeRapiInitEx function returns an error (like many other Win32 functions). On return from the function call, the other two structure members hold return values. A Win32 event handle is returned in the location referenced by the second parameter, heRapiInit. After that event becomes signaled, hrRapiInit holds the initialization result code.

9.1.3.2 *Shutdown with* CeRapiUninit

After any call to a RAPI initialization function—whether or not a connection was established—you must call the CeRapiUninit function. The C# P/Invoke wrapper for this function is shown here:

Download at WoweBook.Com


```
[DllImport("rapi.dll")]
public static extern int CeRapiUninit();
```

9.1.3.3 *Two Approaches to Startup*

So far, we have looked at two functions and one structure that provide the first two functions you will need for all RAPI code you write. Now it's time to look at these elements in the context of real working code. We are going to show two different approaches to handling RAPI startup. The first is short and self-contained, suitable for showing all the startup details in one place. The second is a more real-world approach that involves creating a worker thread to handle the startup activity.

Approach 1: Simple, Single-Threaded RAPI Startup Listing 9.1 shows the simplest possible RAPI startup and shutdown. This program includes all needed P/Invoke declarations for the two required rapi.dll functions: `CeRapiInitEx` and `CeRapiUninit`. This program provides a quick starting point for experimenting with other RAPI functions.

Listing 9.1: RapiStartup.cs with Simplest Possible RAPI Startup and Shutdown

```
// RapiStartup.cs - Simple thread-free RAPI startup
// with all P/Invoke declarations included
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//

using System;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace RapiStartup
{
   public class RapiStartup
   {
      const string m_strAppName = "RapiStartup";

      public RapiStartup()
      {
      }

      // ------------------------------------------------------
      // rapi.dll Definitions
      // ------------------------------------------------------
      public struct RAPIINIT
```

Download at WoweBook.Com


```
{
   public int cbSize;
   public IntPtr heRapiInit;
   public int hrRapiInit;
};

[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int CeRapiInitEx (ref RAPIINIT p);
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int CeRapiUninit ();

public const int S_OK = 0;

// -------------------------------------------------------
// Main -- Program entry point
// -------------------------------------------------------
public static void Main()
{
   // Allocate structure for call to CeRapiInitEx.
   RAPIINIT ri = new RAPIINIT();
   ri.cbSize = Marshal.SizeOf(ri);

   // Call the init function.
   int hr = CeRapiInitEx(ref ri);

   // Wrap event handle in the corresponding .NET object.
   ManualResetEvent mrev = new ManualResetEvent(false);
   mrev.Handle = ri.heRapiInit;

   // Wait five seconds, then fail.
   if (mrev.WaitOne(5000, false) && ri.hrRapiInit == S_OK)
   {
      // Connection is established.
      MessageBox.Show("Connection Established",
         m_strAppName);
   }
   else
   {
      // On failure, disconnect from RAPI.
      CeRapiUninit();

      MessageBox.Show("Timeout - No Device", m_strAppName);
      return;
   }

   // If we get here, we have established a RAPI connection.

   // ToDo: Put your RAPI calls here...

   // Clean up.
```

Download at WoweBook.Com


```
            CeRapiUninit();
        }

    } // class RapiStartup
} // namespace RapiStartup
```

Approach 2: Multithreaded RAPI Startup Our second method for starting RAPI involves a background thread. At the same time, this sample shows how to connect to the ActiveSync wrapper library. We have two such libraries—a C# version and a Visual Basic version—both of which are named `YaoDurant.Win32.Rapi.dll`. There are at least two ways to use these libraries. You can copy the source-code declarations directly to your source code, or you can call into one of these libraries from your code.

Three pieces are needed to allow an application to access a DLL, no matter what language or API you are using: (1) a piece for the compiler, (2) a piece for the linker, and (3) a piece for the loader. Win32 programmers recognize these pieces as (1) the include (.h) file for the compiler, (2) the library (.lib) file for the linker, and (3) the DLL itself for the loader. Here is how to provide these three pieces for a .NET DLL.

1. For the compiler (and to help IntelliSense), add a `using` statement to your source file to identify the desired namespace(s):
   ```
   using YaoDurant.Win32;
   ```
2. For the linker, add a reference to the assembly (or DLL filename) from the Visual Studio .NET IDE.
   a. Open the Solution Explorer (select Menu Items View | Solution Explorer).
   b. Right-click on the `References` entry and select Add Reference.
   c. Click on the Browse button.
   d. Select the desired library from the list, then click the Open button.
3. For the loader, copy the DLL to the directory of the program that needs to use it. (Visual Studio .NET does this automatically if you add the DLL to the project.)

Once you have completed these steps, you can access the RAPI types using the wrapper class name, `Rapi`. In a Visual Studio .NET editor window, when you type "Rapi" IntelliSense shows the types defined within the `Rapi` class.

Download at WoweBook.Com

Listing 9.2 shows RapiStartupThread.cs, a source file taken from the FindPrograms sample. It performs RAPI initialization on a background worker thread, which is more typical of what a production RAPI program would do.

Listing 9.2: RapiStartupThread.cs with Thread-Based RAPI Startup

```
// Program Name: FindPrograms.exe
//
// File Name: RapiStartupThread.cs - Creates a background thread
// for the purpose of starting RAPI.
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//

using System;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using System.Diagnostics;
using YaoDurant.Win32;

namespace FindPrograms
{
   // Table of reasons that WorkerThread calls into the
   // user interface thread
   public enum INVOKE_STARTUP
   {
      STARTUP_SUCCESS,
      STARTUP_FAILED,
      STATUS_MESSAGE
   }

   /// <summary>
   /// StartupThread - Wrapper class that spins a thread
   /// to initialize RAPI. Calls a delegate to report status.
   /// </summary>
   public class StartupThread
   {
      public string strBuffer;       // Interthread buffer
      public INVOKE_STARTUP itReason;   // Interthread reason

      private Thread m_thrd = null;    // The contained thread
      private Control m_ctlInvokeTarget; // Interthread control
      private EventHandler m_deleCallback; // Interthread delegate
      private bool m_bContinue; // Continue flag
```

Download at WoweBook.Com


```
public bool bThreadContinue // Continue property
{
   get { return m_bContinue; }
   set { m_bContinue = value; }
}

/// <summary>
/// StartupThread - Constructor
/// </summary>
/// <param name="ctl">Owner control</param>
/// <param name="dele">Delegate to invoke</param>
public StartupThread(Control ctl, EventHandler dele)
{
   bThreadContinue = true;
   m_ctlInvokeTarget = ctl;  // Who to call.
   m_deleCallback = dele;    // How to call.
}

/// <summary>
/// Run - Init function for startup thread
/// </summary>
/// <returns></returns>
public bool Run()
{
   ThreadStart ts = null;
   ts = new ThreadStart(ThreadMainStartup);
   if (ts == null)
      return false;

   m_thrd = new Thread(ts);
   m_thrd.Start();
   return true;
}

/// <summary>
/// ThreadMainStartup - Start RAPI connection.
/// </summary>
private void ThreadMainStartup()
{
   // Allocate structure for call to CeRapiInitEx.
   Rapi.RAPIINIT ri = new Rapi.RAPIINIT();
   ri.cbSize = Marshal.SizeOf(ri);

   // Call init function.
   int hr = Rapi.CeRapiInitEx(ref ri);

   // Wrap event handle in corresponding .NET object.
   ManualResetEvent mrev = new ManualResetEvent(false);
   mrev.Handle = ri.heRapiInit;
```

Download at WoweBook.Com


```
            // Wait five seconds, then fail.
            if (mrev.WaitOne(5000, false) &&
               ri.hrRapiInit == Rapi.S_OK)
            {
               // Notify caller that connection is established.
               itReason = INVOKE_STARTUP.STARTUP_SUCCESS;
               m_ctlInvokeTarget.Invoke(m_deleCallback);
            }
            else
            {
               // On failure, disconnect from RAPI.
               Rapi.CeRapiUninit();

               strBuffer = "Timeout - no device present.";
               itReason = INVOKE_STARTUP.STATUS_MESSAGE;
               m_ctlInvokeTarget.Invoke(m_deleCallback);

               // Notify caller that connection failed.
               itReason = INVOKE_STARTUP.STARTUP_FAILED;
               m_ctlInvokeTarget.Invoke(m_deleCallback);
            }

            // Trigger that thread has ended.
            m_thrd = null;
         }
      } // class StartupThread
} // namespace FindPrograms
```

The `StartupThread` class provides a wrapper around the startup thread. We put this class into its own file to make it easy for you to reuse. This class uses the `Invoke` function to communicate from the background thread to the user interface thread. This class depends on having a control (or a form) and on that control having a delegate to receive the interthread calls. Our startup wrapper has three methods: a constructor, `Run`, and `ThreadMainStartup`.

The `ThreadMainStartup` method is the thread entry point, and it does the actual work of calling the `CeRapiInitEx` function to initiate the RAPI connection and calling the `CeRapiUninit` function if the attempt to connect fails.

A background thread allows the user interface thread to be available to receive user input. A downside to multithreaded programming is that the two threads need to be carefully crafted to avoid conflicts, which are usually very hard to reproduce because they are often related to timing and thus are hard to find. Our approach to this problem is for each thread to run in a separate class and not interact with others, with two exceptions: thread startup code and thread-safe communication code. In this code,

Download at WoweBook.Com

there is a one-way communication from the worker thread to the main thread. The communication is handled by the only function in the `Control` class guaranteed to be thread-safe: `Control.Invoke`.

Why Control.Invoke Is Thread-Safe

The `Control.Invoke` method is thread-safe (in both the desktop .NET Framework and the .NET Compact Framework) because it relies on an underlying Win32 feature that is guaranteed to be thread-safe. That feature is the `SendMessage` function, a Win32 function that has been around since the first version of Windows. This function sends a Win32 message to a window, waiting for the message to be delivered before returning. If the recipient is busy with another message, the `SendMessage` function blocks until the previous message has been handled. Over the years, this function has been the backbone for interthread and interprocess communication in a variety of Windows technologies, including Dynamic Data Exchange (DDE), Object Linking and Embedding (OLE), and Component Object Model (COM). It is not surprising, then, that this time-tested mechanism is also used by the .NET `Control` class.

The desktop .NET Framework provides two overloads for the `Invoke` method:

```
public object Invoke(Delegate);
public virtual object Invoke(Delegate, object[]);
```

Each overload accepts a parameter of type `Delegate` (similar to a C function pointer). The second overload, which accepts an array of values to be used as parameters to the called function, is the version most often used in the desktop .NET Framework. The .NET Compact Framework, however, supports only the first type of overload, and because this is a .NET Compact Framework book we are going to resist the temptation to do things the desktop .NET Framework way. Here is a call to the `Invoke` method to notify the main thread that a successful RAPI connection has been established:

```
// Notify caller that connection is established.
itReason = INVOKE_STARTUP.STARTUP_SUCCESS;
ctlInvokeTarget.Invoke(deleCallback);
```

Download at WoweBook.Com

The recipient of this function call is a function named `StartupCallback`, a member of the `MainForm` class in our `FindPrograms` sample. We cannot just pass a function pointer like we do in unmanaged C, however. Instead, we pass the .NET equivalent, a delegate of type `EventHandler`. Here is the declaration for the delegate, along with the other declarations needed to support the startup thread:

```
// Startup thread definitions
StartupThread thrdStartup = null;
private EventHandler deleStartup;
private bool bRapiConnected = false;
```

We initialize the delegate as follows:

```
deleStartup = new EventHandler(this.StartupCallback);
```

and then pass it to the constructor for our startup thread wrapper object:

```
// Create thread to connect to RAPI.
thrdStartup = new StartupThread(this, deleStartup);
if (!thrdStartup.Run())
   thrdStartup = null;
```

After creating the `StartupThread` object, we call the `Run` member to start it. This two-step approach—create the object, then initialize the object—may be familiar to C++ programmers from other object-oriented APIs. A two-step approach works because a constructor cannot easily provide a return value, nor can it report a reason when it fails. A stand-alone initialization function—`Run` in this example—can do both.

Our interthread delegate function, `StartupCallback`, takes two parameters: `sender` and `e`. The parameters have names that suggest they might be useful, but what each name suggests does not quite match with what each parameter actually contains. The first, `sender`, identifies the *recipient* of the function call—the control that is receiving the `Invoke` call—and not the sender as the name implies. The second parameter, `e`, is of type `EventArgs`. Because the first parameter gives us no useful information, you might suppose that the second parameter—which seems to want to contain event arguments—would yield a useful hint or two. And while `EventArgs` is the base class for every other event argument class, this type contains no event data. So, where does that leave us? With two parameters that are essentially empty.

Download at WoweBook.Com

Because neither parameter is useful, we structure our code so that this function has only one possible caller, the thread wrapper object, `thrdStartup`. The recipient of the call then accesses that caller's public data to receive any parameters needed for the interthread function call and responds accordingly. Because of the way the `Invoke` method works, our background thread is blocked while the main thread runs the `Invoke` target function. In this way, our code is thread-safe.

Listing 9.3 shows the code for the `StartupCallback` function, which receives an interthread call courtesy of the `Invoke` method in the `Control` class.

LISTING 9.3: `StartupCallback` **Accepting** `Invoke` **Calls from the Startup Thread**

```
private void
StartupCallback(object sender, System.EventArgs e)
{
   INVOKE_STARTUP it = this.m_thrdStartup.itReason;
   switch(it)
   {
      case INVOKE_STARTUP.STARTUP_SUCCESS:
         m_bRapiConnected = true;
         EnableUI();
         break;
      case INVOKE_STARTUP.STARTUP_FAILED:
         ResetUI();
         break;
      case INVOKE_STARTUP.STATUS_MESSAGE:
         sbarMain.Text = m_thrdStartup.strBuffer;
         break;
   }
}
```

A key element connecting the main thread to the startup thread is the connection status flag, `m_bRapiConnected`. When the startup thread establishes the RAPI connection, it sends the main thread a `STARTUP_SUCCESS` code, and the main thread sets the connected flag to `true`:

```
m_bRapiConnected = true;
```

This flag reminds us to shut down our RAPI connection at the appropriate time. For example, as shown in Listing 9.4, when the main form closes, we also shut down any running worker threads.

Download at WoweBook.Com

Listing 9.4: Cleanup of Background Threads and RAPI Connection

```
private void
MainForm_Closed(object sender, System.EventArgs e)
{
   // If threads are running, trigger shutdown.
   if (this.m_thrdStartup != null)
      this.m_thrdStartup.bThreadContinue = false;
   if (this.m_thrdFindFiles != null)
      this.m_thrdFindFiles.bThreadContinue = false;

   if (m_bRapiConnected)
   {
      Rapi.CeRapiUninit();
      m_bRapiConnected = false;
   }
}
```

9.1.3.4 *Defensive Coding*

RAPI is a communications protocol, which means you must code defensively. When a connection has been established, your program must be able to handle lengthy operations—reading or writing data—that may take a long time because either the connection is slow (USB or serial), or the volume of data is large, or both. Also, when communicating between two machines, the connection may get interrupted. For example, ActiveSync is interrupted when a user removes a device from its docking cradle. In addition to interrupted connections, your code must handle the possibility that no connection—meaning no device—is available.

Lengthy operations can be caused by blocking functions, by a slow connection, or simply by the need to move a lot of data. Blocking functions return only when the requested action is finished. For example, the `CeRapiInit` function returns only after a RAPI connection is established, and the function hangs when no device is present. Most RAPI functions, in fact, are blocking (also called *synchronous*) and finish their work before returning. Functions that return a large amount of data might take several seconds or even several minutes to finish.

While these issues require some effort to address, none are entirely daunting. To avoid hanging a program's user interface, use background threads. Without threads, an application can become unresponsive to a user's input and perhaps make a user worry that the program has hung. Threads—and feedback to the user with progress information—help ease such user concerns.

Download at WoweBook.Com

The final challenge, a disconnected connection, can be a little more difficult to address. For individual function calls, you must use vigilance in checking return values. In addition, you will want to use exception handling (the `try` and `catch` keywords) so that when something fails deep within your code, you can escape the problem by raising an exception and recovering from the failure higher up the call stack. Exception handling simplifies the handling of unexpected failures.

9.2 Accessing the Object Store

The majority of RAPI functions involve reading from or writing to the object store. As detailed in Chapter 5, Storage, the object store is the RAM-based storage area with three parts: a file system, the registry, and property databases.

The Windows CE file system is very similar to the desktop file system. Both are hierarchical, supporting nested subdirectories (sometimes called *folders*) and long file paths that can be up to 260 characters long. The file system in the object store provides the primary storage area for a Windows CE device. Additional file systems known as *installable file systems* can be added to extend available storage space. Two common installable file systems are Compact Flash memory cards and Secure Digital (SD) storage cards. Both are sometimes referred to as *memory cards* because they contain nonvolatile flash memory, but both are used for file storage and not as program memory. You can access both object store files as well as installable file system files by using the RAPI functions.

The registry provides a hierarchy for storing application and system settings. The registry is as important to Windows CE as it is to desktop Windows. The registry contains settings that the operating system uses for a wide range of tasks; examples include device drivers to load and applications to launch at system boot time. Using RAPI functions, you can access any part of the registry of a connected Windows CE device.

The third part of the object store, property databases, provides a memory-efficient way to store and access sparsely populated record information. These databases are used by the built-in PDA applications—the calendar, inbox, task list, and contact list—for their respective storage needs. In general, a property database provides an easy way to store small, relatively static data. As the number of records increases beyond 1,000 records or as the number of changes increases, the performance of CE property databases drops. The SQL Server CE database meets the needs of

Download at WoweBook.Com

large, dynamic databases that need to reside on a Windows CE device. Using RAPI functions, you have full access to any CE property database on a connected device. RAPI does not provide any direct help for SQL Server CE databases, however, and if you needed to remotely update a SQL Server CE database, you would have to implement your own custom solution.

This book has RAPI sample programs for each of the three parts of the object store. Two programs, `FindPrograms` and `FindProgramsFaster`, access the file system. These programs search the device file system for executable programs. You will see how a special RAPI function, `CeFindAllFiles`, helps provide fast directory searches. Our registry sample program, `RegShowStartup`, provides details about the Windows CE system boot process. This program lists the registry entries for programs and DLLs loaded at system boot time. The property database sample program, `ShowDatabases`, provides a list of database volumes and databases that reside on a remotely connected device.

9.2.1 Using RAPI to Access Device Files

When you look at the RAPI file system functions presented earlier in Table 9.1, you see that you can create, delete, and query directories. You can also create and delete files and open, read, and write individual files. With the aid of the RAPI file system functions, your desktop-based programs can access a device file system as though it were a local file system because the RAPI file functions have parameters identical to functions used to access a local file system.

And there are other similarities as well. As mentioned earlier, Windows CE file systems and desktop Windows file systems are hierarchical and support long file paths. Files and directories have attributes, so a given file or directory—on either the desktop or a device—could be marked *hidden, normal, system,* or *read-only.* Such similarities mean you can move entire directory trees from the desktop to a device and back again.

But the two types of file systems are not exactly the same. The desktop shows its MS-DOS (and CP/M[3]) legacy with the use of letters—`C:`, `D:`, and so on—for disk partitions that Windows CE file systems do not support. Instead, a single root directory holds all mounted file systems. For example, when you plug a Compact Flash memory card into the appropriate

3. Old-timers, especially "software archeologists," may remember CP/M—the Control Program for Microcomputers—created by Gary Kildall, founder of Digital Research. It was a predecessor to IBM's PC-DOS and Microsoft's MS-DOS.

Download at WoweBook.Com

slot on a device, a directory with a name such as `Storage Card` appears in the root directory.

Detecting the Names of Installed File Systems

While you might be tempted to hard-code a name for removable media such as `Storage Card`, there are variations for different Compact Flash and Smart Media storage cards. On the Pocket PC 2002, the Win32 functions `FindFirstFlashCard` and `FindNextFlashCard` enumerate the root directory name for installed file systems. These functions, unfortunately, have no RAPI equivalent.

RAPI functions make it easy, but not fast, to access a device file system. At issue here are limitations in the physical connections that do not involve what ActiveSync is doing or not doing. For example, file I/O through a USB port or a serial port connection is slow when compared with accessing files in a local file system. When designing a solution that moves data with RAPI, you need to think about minimizing the amount of data you move. When something runs too slowly, you need to measure the time taken by various parts of the operation, a process sometimes referred to as **profiling** (i.e., formally measuring the time taken), to help you devise strategies to identify and fix whatever is slowing down your code.

There are two basic ways to profile. The simplest involves the `TickCount` property in the `Environment` class, which returns the milliseconds since system startup. You can measure time in seconds with code such as the following:

```
int cTicks = Environment.TickCount;
// Do something.
int cSeconds = (Environment.TickCount – cTicks + 500) / 1000;
```

You can measure milliseconds by eliminating the division operation in the equation, or you can call a more precise profiling function: `QueryPerformanceCounter`. This Win32 function requires a P/Invoke declaration, and returns the current value for a continually running counter. A second function, `QueryPerformanceFrequency`, returns the rate of the counter, in units of counts per second.

Download at WoweBook.Com

We wrote two sample programs—`FindPrograms` and `FindPrograms-Faster`—to show how to access a device file system. Both programs search for executable files, adding to a `ListBox` control the full pathname for programs found. Both sample programs report the search time in the status bar. Figure 9.1 shows `FindPrograms` after searching for all programs in all directories. You can run a program on a device by selecting the program from the list and clicking on the Run button.

Our two sample programs are identical except in how they access the device file system. `FindPrograms` uses a standard Win32 approach: the `CeFindFirstFile` and `CeFindNextFile` functions. With this approach, it took five minutes to build a complete list of program files from a Pocket PC 2002 (on a Pocket PC connected via USB). As its name suggests, `FindProgramsFaster` performs a faster search by using another approach to search the file system: the `CeFindAllFiles` function. This function is faster because it lets you pick a subset of the per-file data that you want to query. By reducing the amount of data that has to be moved from the device to the desktop, you get the results faster.

So, how much faster is the `CeFindAllFiles` function? When connected via USB, the search time was reduced to 15 seconds from 300+ seconds in the original program. We then did the same tests using a 10Mbps network connection, and the search time dropped further still. Our first program did a full search in six seconds. The second program again dropped the time, finishing its search in just two seconds.

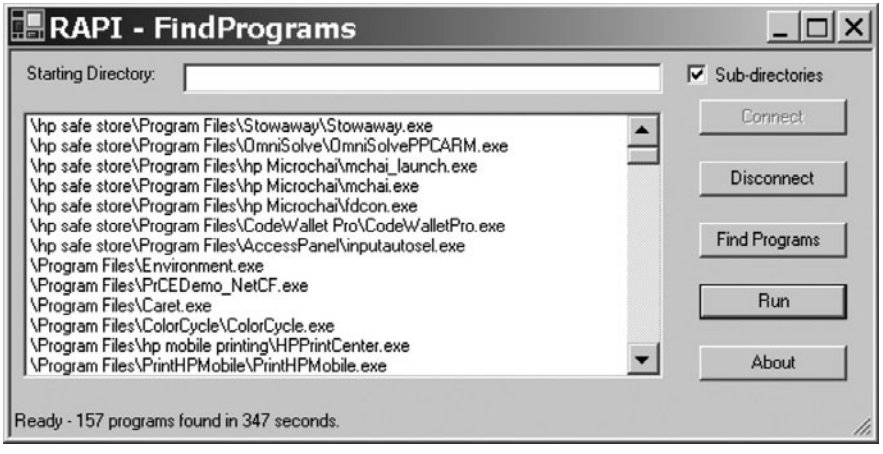


FIGURE 9.1: `FindPrograms`, Which Uses Remote File Functions to Locate Executable Programs

Download at WoweBook.Com

9.2.1.1 FindPrograms: *Using RAPI to Find Files in the Device File System*

This program uses RAPI functions to search for executable files and allows a user to start a program running. This program has three source files: MainForm.cs, RapiStartupThread.cs, and RapiFindFilesThread.cs. As usual, MainForm.cs has the user interface code. The RAPI startup code can be found in RapiStartupThread.cs (which appears earlier in this chapter in Listing 9.2).

The third file, RapiFindFilesThread.cs, contains all of the code for the actual file system search. As with the RAPI startup we discussed earlier, our RAPI file search is done on a background thread. Results get sent to user interface threads using the thread-safe `Control.Invoke` mechanism. The source code for the `FindFilesThread` class contains the following member functions:

- `FindFilesThread` (class constructor)
- `Run`
- `ThreadMainFindFiles`
- `AddProgramsInDirectory`
- `FetchAndDisplayError`

The constructor makes a local copy of startup data and nothing more, once again leaving the real startup work to an initialization function. In this case, that means the `Run` function.

The `Run` function creates a thread that runs on the `ThreadMainFind-Files` thread entry point. That function, in turn, calls back to the main form with a `FINDFILE_QUERYSTARTPATH` request through the interthread invoke delegate, `deleCallback` (as shown in the following code excerpt) and then calls the `AddProgramsInDirectory` function:

```
itReason = INVOKE_FINDFILES.FINDFILE_QUERYSTARTPATH;
ctlInvokeTarget.Invoke(deleCallback);
string strPath = strBuffer;
AddProgramsInDirectory(strPath);
```

The `AddProgramsInDirectory` function does the actual work of calling RAPI file search functions. This function is recursive, to handle situations when the user clicks the Sub-directories checkbox. When checked, the function drills down into other subdirectories in search of more program files. As directories are encountered, they are added to a list—actually an

Download at WoweBook.Com

ArrayList—and when the search in the current directory is complete, a subdirectory search is started.

The last function, FetchAndDisplayError, sends error information to the user interface thread in case of failure.

Listing 9.5 contains the code for the RapiFindFilesThread.cs source file, which provides the support for the background thread that actually does the search for a file.

Listing 9.5: RapiFindFilesThread.cs Managing the Background Thread That Searches for a File

```
// Program Name: FindPrograms.exe
//
// File Name: RapiFindFilesThread.cs - Creates a background
// thread to retrieve file names from the device
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//

using System;
using System.Threading;
using System.Collections;
using System.Windows.Forms;
using System.Diagnostics;
using YaoDurant.Win32;

namespace FindPrograms
{
   // Reasons our thread invokes user interface thread
   public enum INVOKE_FINDFILES
   {
      FINDFILE_QUERYSTARTPATH,
      FINDFILE_NEWFILE,
      FINDFILE_COMPLETE,
      STATUS_MESSAGE
   }

   /// <summary>
   /// FindFilesThread wraps a thread that supports
   /// RAPI search of device directory.
   /// </summary>
   public class FindFilesThread
   {
      public string strBuffer;       // Interthread buffer
      public INVOKE_FINDFILES itReason;   // Interthread reason

      private Thread m_thrd = null;    // The contained thread
```

Download at WoweBook.Com


```
private Control m_ctlInvokeTarget; // Interthread control
private EventHandler m_deleCallback; // Interthread delegate
private bool m_bContinue; // Continue flag
private bool m_bIncludeSubDirs = false; // Search subdirs
private int m_cFiles = 0;   // File find counter

public bool bThreadContinue // Continue property
{
   get { return m_bContinue; }
   set { m_bContinue = value; }
}

/// <summary>
/// FindFilesThread - Constructor
/// </summary>
/// <param name="ctl">Owner control</param>
/// <param name="dele">Delegate to invoke</param>
public FindFilesThread(Control ctl, EventHandler dele)
{
   bThreadContinue = true;
   m_ctlInvokeTarget = ctl;  // Who to call
   m_deleCallback = dele;    // How to call
}

/// <summary>
/// Run - Init function for file find thread
/// </summary>
/// <param name="bSubDirs"></param>
/// <returns></returns>
public bool Run(bool bSubDirs)
{
   ThreadStart ts = null;
   ts = new ThreadStart(ThreadMainFindFiles);
   if (ts == null)
      return false;

   m_bIncludeSubDirs = bSubDirs;

   m_thrd = new Thread(ts);
   m_thrd.Start();
   return true;
}

/// <summary>
/// ThreadMainFindFiles - Main thread for file find thread
/// </summary>
private void ThreadMainFindFiles()
{
   int cTicks = Environment.TickCount;
```

Download at WoweBook.Com


```
   itReason = INVOKE_FINDFILES.FINDFILE_QUERYSTARTPATH;
   m_ctlInvokeTarget.Invoke(m_deleCallback);
   string strPath = strBuffer;
   AddProgramsInDirectory(strPath);

   int cSeconds =
      (Environment.TickCount - cTicks + 500) / 1000;
   if (bThreadContinue)
   {
      // Send message for search time.
      strBuffer = "Ready - " + m_cFiles +
         " programs found in " + cSeconds + " seconds.";
      itReason = INVOKE_FINDFILES.STATUS_MESSAGE;
      m_ctlInvokeTarget.Invoke(m_deleCallback);

      // Trigger that search is done.
      itReason = INVOKE_FINDFILES.FINDFILE_COMPLETE;
      m_ctlInvokeTarget.Invoke(m_deleCallback);
   }
}
/// <summary>

/// AddProgramsInDirectory - Recursive function to search
/// into directory tree.
/// </summary>
/// <param name="strDir">Starting directory</param>
/// <returns></returns>
private bool AddProgramsInDirectory(string strDir)
{
   Trace.WriteLine("FindPrograms: " +
      "AddProgramsInDirectory (" + strDir + ")");

   // Update status bar through delegate function.
   strBuffer = "Searching in " + strDir + "...";
   itReason = INVOKE_FINDFILES.STATUS_MESSAGE;
   m_ctlInvokeTarget.Invoke(m_deleCallback);

   // As we add programs, store directory names.
   ArrayList alDirectories = new ArrayList();
   Rapi.CE_FIND_DATA fd = new Rapi.CE_FIND_DATA();

   // Start our search.
   string strSearch = strDir + "*.*";
   IntPtr hff = Rapi.CeFindFirstFile(strSearch, ref fd);
   if ((int)hff == -1)
   {
      FetchAndDisplayError();
   }
   else
   {
```

Download at WoweBook.Com


```
      do
      {
         string strFileName = fd.cFileName;
         int iFlag = (int)
            Rapi.FILE_ATTRIBUTE.FILE_ATTRIBUTE_DIRECTORY;
         if ((fd.dwFileAttributes & iFlag) == iFlag)
         {
            alDirectories.Add(strDir+fd.cFileName);
         }
         else
         {
            if (strFileName.EndsWith(".EXE") ||
               strFileName.EndsWith(".exe"))
            {
               m_cFiles++;

               strBuffer = strDir + fd.cFileName;
               itReason = INVOKE_FINDFILES.FINDFILE_NEWFILE;
               m_ctlInvokeTarget.Invoke(m_deleCallback);
            }
         }
      } while (bThreadContinue &&
         Rapi.CeFindNextFile(hff, ref fd) != 0);

      if (bThreadContinue && m_bIncludeSubDirs)
      {
         foreach (string str in alDirectories)
         {
            AddProgramsInDirectory(str + "\\");
         }
      }
   }

   Rapi.CeFindClose(hff);

   return true;
}

/// <summary>
/// FetchAndDisplayError - Displays error in status bar
/// </summary>
public void FetchAndDisplayError()
{
   strBuffer = string.Empty;

   // Is this a RAPI error?
   int err = Rapi.CeRapiGetError();
   if (err != Rapi.S_OK)
   {
      strBuffer = "RAPI Error (0x" +
```

Download at WoweBook.Com


```
                ((int)err).ToString("x") +")";
            }
            else
            {
               // Check for CE error.
               err = Rapi.CeGetLastError();
               if (err != (int)Rapi.ERROR.ERROR_FILE_NOT_FOUND)
               {
                  strBuffer = "CE Error (code = " +
                     err.ToString("x") + ")";
                }
            }
            if (strBuffer != string.Empty)
            {
               itReason = INVOKE_FINDFILES.STATUS_MESSAGE;
               m_ctlInvokeTarget.Invoke(m_deleCallback);
            }

            // Trigger that thread has ended.
            m_thrd = null;
         }
      } // class FindFilesThread
} // namespace FindPrograms
```

For most purposes, the `CeFindFirstFile` and `CeFindNextFile` functions do a reasonable job of checking whether a given file exists on a device. For searching an entire file system—or even all the files in a directory—a faster approach is needed. That is why RAPI adds a faster search function not otherwise found in the regular Win32 file functions: `CeFindAllFiles`.

9.2.1.2 `FindProgramsFaster`: *Speeding Up a File Search*

How much faster is the `CeFindAllFiles` function? As we mentioned, a 300-second search done over a USB connection took only 15 seconds, and a 6-second search over a network link got cut to a mere 2 seconds. Obviously, using the `CeFindAllFiles` function is faster than using the `CeFindFirstFile` and `CeFindNextFile` functions.

Calling the `CeFindAllFiles` function stretches the limits of P/Invoke support and requires us to take some steps not required for simpler functions. The Win32 definition for this function appears here:

```
// Win32 function definition
BOOL CeFindAllFiles(LPCWSTR szPath, DWORD dwFlags,
LPDWORD lpdwFoundCount, LPLPCE_FIND_DATA ppFindDataArray);
```

Download at WoweBook.Com

It takes the following parameters:

- `LPCWSTR szPath`: the path to search
- `DWORD dwFlags`: flags indicating which fields to return
- `LPDWORD lpdwFoundCount`: a pointer to an integer to return the number of items found
- `LPLPCE_FIND_DATA ppFindDataArray`: a pointer to a pointer to an array of `CE_FIND_DATA` structures

From the perspective of the function's work, the important parameters are the first and the last. The first, `szPath`, identifies the directory to search. The last, `ppFindDataArray`, is the address of a pointer variable—what C programmers refer to as a "pointer to a pointer." This function allocates and fills a buffer with an array of found file information and returns a pointer to that buffer in the last parameter.

The `CE_FIND_DATA` structure occupies 300 bytes, which is a lot if you consider that this amount is downloaded for each file with the find first/find next query method. `CeFindAllFiles` speeds up the search by being clever about how it downloads this data. The second parameter, `dwFlags`, lets you pass flags to define a subset of downloaded fields. In addition, a minimum number of characters—and not the 260 defined for the array—are downloaded for the filename. The third parameter, `lpdwFoundCount`, receives the number of files in the array.

Let us take a look, then, at the required P/Invoke declarations for this function, shown here:

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int CeFindAllFiles (string szPath,
   FAF dwFlags, ref int lpdwFoundCount,
   ref IntPtr ppFindDataArray);
```

Table 9.2 compares Win32 parameters for the `CeFindAllFiles` function to the comparable .NET parameters. Each parameter to the `CeFindAllFiles` function shows different aspects of P/Invoke.

The first parameter, `szPath`, is a regular C# string, which—because of the `CharSet=CharSet.Unicode` attribute—becomes a pointer to a Unicode string.

The second parameter, `dwFlags`, provides the desired filter and retrieval flags for the search. In our P/Invoke declaration, we put these values within the `FAF` enumeration, which is defined in `YaoDurant.Win32.Rapi` (and which appears in Listing 9.6). We could have

Download at WoweBook.Com

defined the second parameter as an integer and then defined a set of integer flag values. But we like using enumerations, which allows IntelliSense to provide a list of possible values from within the code editing window.

Listing 9.6: Enumeration for File Filter and Retrieval Flags

```
public enum FAF
{
   FAF_ATTRIBUTES = 0x01,
   FAF_CREATION_TIME = 0x02,
   FAF_LASTACCESS_TIME = 0x04,
   FAF_LASTWRITE_TIME = 0x08,
   FAF_SIZE_HIGH = 0x10,
   FAF_SIZE_LOW = 0x20,
   FAF_OID = 0x40,
   FAF_NAME = 0x80,
   FAF_FLAG_COUNT = 8,
   FAF_ATTRIB_CHILDREN = 0x01000,
   FAF_ATTRIB_NO_HIDDEN = 0x02000,
   FAF_FOLDERS_ONLY = 0x04000,
   FAF_NO_HIDDEN_SYS_ROMMODULES = 0x08000,
   FAF_GETTARGET = 0x10000,
}
```

The third parameter, `lpdwFoundCount`, is a pointer to an integer. Passing pointers into Win32 functions is quite common, and the `ref` keyword provides the means in managed code to ask the compiler to generate a pointer when the function is called.

The fourth parameter, `ppFindDataArray`, is the trickiest: a pointer to a pointer. C# can allocate pointers but only at a cost: You must declare a function `unsafe` and use the `/unsafe` compiler switch to mark the entire program as unsafe. In some settings, your program would not be allowed to run because of the local security policy. Rather than creating an explicit pointer, we turn to the `Marshal` class for help.

Table 9.2: Comparison between Win32 and .NET Parameter Types

Win32 Parameter	.NET Parameter
`LPCWSTR szPath`	`string szPath`
`DWORD dwFlags`	`FAF dwFlags`
`LPDWORD lpdwFoundCount`	`ref int lpdwFoundCount`
`LPLPCD_FIND_DATA ppFindDataArray`	`ref IntPtr ppFindDataArray`

Download at WoweBook.Com

Actually, all of the managed-to-unmanaged code relies on the `Marshal` class. All of the various declarations and attributes we create for Win32 functions are consumed by this class. With this fourth parameter, however, we need to directly call some of the static functions exported by the `Marshal` class.

We define the fourth parameter as `IntPtr`, which is not so much a type as a placeholder for a variety of different things. When calling Win32 functions, this type is used for handles (e.g., window handles, file handles) and for pointers. Because of the two levels of indirection—we are passing a pointer to a pointer—we use the `ref` keyword for this pointer declaration. With one level of indirection, we would use `IntPtr` without `ref`.

The `ref` keyword is needed both in the function declaration and in the actual function call:

```
// Start our search.
string strSearch = strDir + "*.*";

IntPtr pfdAllFiles = IntPtr.Zero;  // Return pointer.
int cFound = 0;          // Return count of files.

// This call looks for all files in current directory.
Rapi.CeFindAllFiles(strSearch,
    Rapi.FAF.FAF_ATTRIBUTES |
    Rapi.FAF.FAF_NAME,
    ref cFound,
    ref pfdAllFiles);
```

After this call to the `CeFindAllFiles` function, the fourth parameter, `pfdAllFiles`, holds a pointer to an array of `CE_FIND_DATA` values. The third parameter, `cFound`, holds the count of array members. The .NET definition for the `CE_FIND_DATA` structure appears in Listing 9.7.

Listing 9.7: Return Values Occupying an Array of `CE_FIND_DATA` Structures

```
[StructLayout(LayoutKind.Sequential,Pack=4,
   CharSet=CharSet.Unicode)]
public struct CE_FIND_DATA
{
   public int dwFileAttributes;
   public FILETIME ftCreationTime;
   public FILETIME ftLastAccessTime;
   public FILETIME ftLastWriteTime;
   public int nFileSizeHigh;
   public int nFileSizeLow;
   public int dwOID;
```

Download at WoweBook.Com


```
   [ MarshalAs( UnmanagedType.ByValTStr, SizeConst=MAX_PATH)]
   public string cFileName;
};

public const int MAX_PATH = 260;
```

We have a pointer to the data, but because of the strict type checking of C#—and indeed of all .NET languages—we cannot directly access the data referenced by an `IntPtr`. Instead, we need help from the `Marshal` class.

The `Marshal` class resides in the `System.Runtime.InteropServices` namespace and is supported in both the desktop .NET Framework and the .NET Compact Framework. This class provides static functions for moving data between managed memory and unmanaged memory. We call the `Marshal.PtrToStructure` function, defined in the following code, to create a managed object of type `CE_FIND_DATA` from a block of unmanaged memory:

```
public static object PtrToStructure(
   IntPtr ptr, Type structureType );
```

The `PtrToStructure` function accepts an `IntPtr` pointer and the type of the object to create, then allocates and initializes the requested object. This function provides the "secret sauce" needed to create a .NET object from unmanaged memory. Here is how we call the `PtrToStructure` function in the `FindProgramsFaster` sample program:

```
IntPtr pfd = pfdAllFiles;
// . . .
Rapi.CE_FIND_DATA fd = // Output .NET object
   (Rapi.CE_FIND_DATA) // Always cast this
   Marshal.PtrToStructure(  // The function
   pfd,                          // Input Win32 ptr
   typeof(Rapi.CE_FIND_DATA));  // Output type
```

As written, this function reads a single member of the array to provide data on a single file. We read all of the elements in the array by putting the following call into a loop and increment the pointer to access each element of the array as shown here:

```
// Get ready for next loop.
pfd = (IntPtr)((int)pfd + Marshal.SizeOf(fd));
```

Download at WoweBook.Com

The `Marshal.SizeOf` Method

This code shows the `Marshal.SizeOf` method, which is similar to the C/C++ `sizeof` operator.

After looping through all of the array elements, we must free the memory allocated by the `CeFindAllFiles` function. We do not call a `Marshal` class method; instead, we call a RAPI function provided especially for freeing memory allocated by `CeFindAllFiles` (and `CeFindAllDatabases`), the `CeRapiFreeBuffer` function, as shown here:

```
// Free memory returned by CeFindAllFiles.
Rapi.CeRapiFreeBuffer(pfdAllFiles);
```

Listing 9.8 shows these calls in the context of the `FindProgramsFaster` sample program.

Listing 9.8: The `AddProgramsInDirectory` Function from `FindProgramsFaster`

```
/// <summary>
/// AddProgramsInDirectory - Recursive function to search
/// into directory tree
/// </summary>
/// <param name="strDir">Starting directory</param>
/// <returns></returns>
private bool AddProgramsInDirectory(string strDir)
{
   Trace.WriteLine("FindPrograms: " +
      "AddProgramsInDirectory (" + strDir + ")");

   // Update status bar through delegate function.
   strBuffer = "Searching in " + strDir + "...";
   itReason = INVOKE_FINDFILES.STATUS_MESSAGE;
   m_ctlInvokeTarget.Invoke(m_deleCallback);

   // As we add programs, store directory names.
   ArrayList alDirectories = new ArrayList();

   // Start our search.
   string strSearch = strDir + "*.*";

   IntPtr pfdAllFiles = IntPtr.Zero;  // Return pointer.
   int cFound = 0;          // Return count of files.

   // This call looks for all files in current directory.
```

Download at WoweBook.Com


```
Rapi.CeFindAllFiles(strSearch,
    Rapi.FAF.FAF_ATTRIBUTES |
    Rapi.FAF.FAF_NAME,
    ref cFound,
    ref pfdAllFiles);

// Loop through all files found.
IntPtr pfd = pfdAllFiles;
while (cFound > 0)
{
   //
   // Here is the secret sauce. This function uses a
   // Win32 pointer to create a .NET object.
   //
   Rapi.CE_FIND_DATA fd = // Output .NET object
      (Rapi.CE_FIND_DATA) // Always cast this
      Marshal.PtrToStructure(  // The function
         pfd,                          // Input Win32 ptr
         typeof(Rapi.CE_FIND_DATA));  // Output type

   string strFileName = fd.cFileName;
   int iFlag = (int)
      Rapi.FILE_ATTRIBUTE.FILE_ATTRIBUTE_DIRECTORY;
   if ((fd.dwFileAttributes & iFlag) == iFlag)
   {
      alDirectories.Add(strDir+fd.cFileName);
   }
   else
   {
      if (strFileName.EndsWith(".EXE") ||
         strFileName.EndsWith(".exe"))
      {
         m_cFiles++;

         strBuffer = strDir + fd.cFileName;
         itReason = INVOKE_FINDFILES.FINDFILE_NEWFILE;
         m_ctlInvokeTarget.Invoke(m_deleCallback);
      }
   }

   // Get ready for next loop.
   pfd = (IntPtr)((int)pfd + Marshal.SizeOf(fd));
   cFound--;
}

// Free memory returned by CeFindAllFiles.
Rapi.CeRapiFreeBuffer(pfdAllFiles);

if (bThreadContinue && m_bIncludeSubDirs)
{
```

Download at WoweBook.Com


```
      foreach (string str in alDirectories)
      {
         AddProgramsInDirectory(str + "\\");
      }
   }

   return true;
}
```

9.2.2 Remote Access to Device Registry Entries

A Windows CE object store contains a registry in addition to and outside of the file system. Table 9.1 earlier in this chapter summarizes the ten registry functions found in rapi.h. As on desktop Windows, the registry provides a hierarchical structure for storing key name/value pairs. The registry has many uses and many users, including the operating system, device drivers, and applications. The registry serves as a central storehouse for tiny bits of data that might otherwise get lost if put in a file.

If you have not worked with registry functions before, you can get a good idea of the basic layout of the registry with a registry editing tool. One such tool, regedit.exe, ships with each 32-bit version of Microsoft Windows. Run this tool by typing its name into the Run dialog box. Another registry tool, the Remote Registry Editor, lets you view the registry of Windows CE devices. This second tool ships with Visual Studio.

Figure 9.2 shows the Remote Registry Editor viewing some registry contents of a Windows CE device.

The left panel of the Remote Registry Editor shows the hierarchical structure of registry keys. You identify a specific registry key by connecting key names together, separating them from each other with a backslash (\) character just as you do for the file system hierarchy. The status bar of the Remote Registry Editor in Figure 9.2 gives an example—`HKEY_LOCAL_MACHINE\ init`. The right panel of the Remote Registry Editor shows all of the values associated with the selected registry key. To access a value in the registry, you open a key (in the left panel) with one set of functions. Then you read and write values (the contents of the right panel) using a second set of functions. When you are done, you must close any key you have opened.

As the Remote Registry Editor shows, the registry holds several hierarchies. You identify the top of each hierarchy by using a special "magic number." The C# definitions for these magic numbers are shown here:

```
public static IntPtr HKEY_CLASSES_ROOT  =
   new IntPtr(unchecked((int)0x80000000));
```

Download at WoweBook.Com

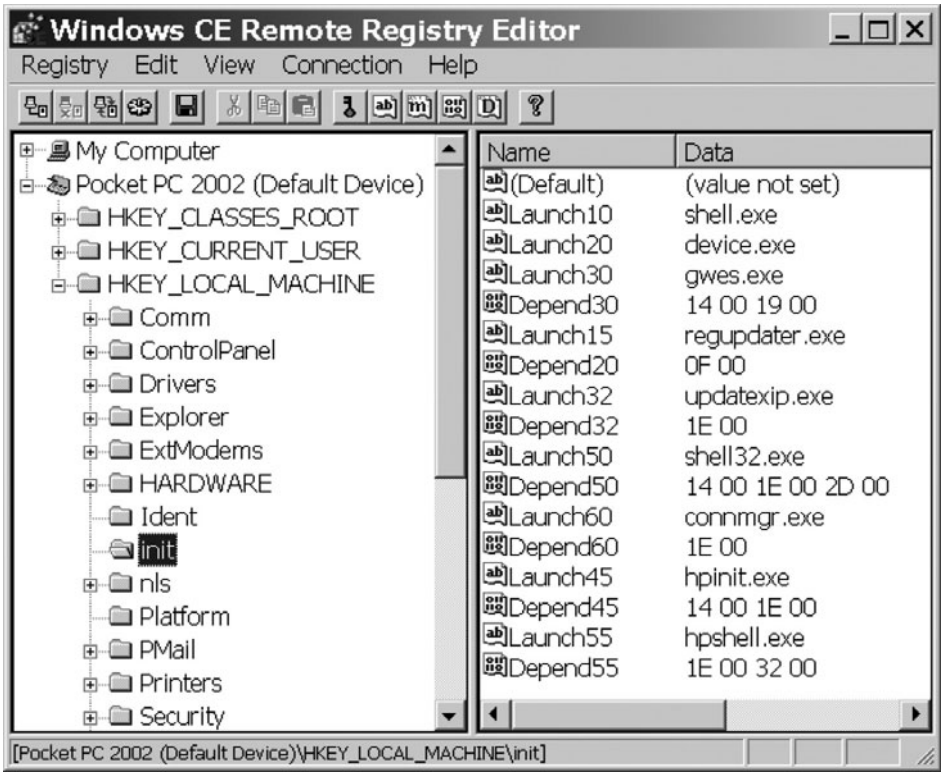


FIGURE 9.2: Remote Registry Editor Viewing the Contents of a Windows Mobile Registry

```
public static IntPtr HKEY_CURRENT_USER  =
   new IntPtr(unchecked((int)0x80000001));
public static IntPtr HKEY_LOCAL_MACHINE =
   new IntPtr(unchecked((int)0x80000002));
public static IntPtr HKEY_USERS         =
   new IntPtr(unchecked((int)0x80000003));
```

The `unchecked` keyword is needed in C# to prevent an overflow exception, which occurs because the initialization values are outside the valid range of values for a signed integer.

Open an existing registry key by calling the `CeRegOpenKeyEx` RAPI function. The P/Invoke definition for this function is shown here:

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegOpenKeyEx (IntPtr hKey, string lpszSubKey,
int Reserved1, int Reserved2, ref IntPtr phkResult);
```

Download at WoweBook.Com

Values that can be set to `0` are named `Reserved`. The `CeRegOpenKeyEx` function's return value is the last input parameter, `phkResult`. Here is how to call this function to get the key to the `HKEY_LOCAL_MACHINE\init` registry node:

```
IntPtr hkeyInit = IntPtr.Zero;

int iResult = Rapi.CeRegOpenKeyEx(Rapi.HKEY_LOCAL_MACHINE, "init",
   0, 0, ref hkeyInit);
```

Another way to open a registry key is to call the `CeRegCreateKeyEx` function. As the name suggests, this function also creates a key if the named node does not exist.

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegCreateKeyEx (IntPtr hKey, string lpszSubKey,
   int Reserved, int Reserved2, int Reserved3, int Reserved4,
   int Reserved5, ref IntPtr phkResult,
   ref REGDISP lpdwDisposition);
```

The final parameter, `lpdwDisposition`, provides a return value with information about whether a key already exists or was created. Our library, `YaoDurant.Win32.Rapi.dll`, has an enumeration that contains the two possible values for this parameter:

```
public enum REGDISP
{
   REG_CREATED_NEW_KEY = 0x00000001,
   REG_OPENED_EXISTING_KEY = 0x00000002,
}
```

Once a key has been opened, you can read registry values by calling the `CeRegQueryValueEx` function and write values with the `CeRegSetValueEx` function. We have several definitions for each function to accommodate different types—integer, string, and binary—that can be written to the registry (these definitions are for the most common types). Listing 9.9 shows the P/Invoke wrappers for various overloads that we define to call the `CeRegQueryValueEx` function.

Listing 9.9: P/Invoke Wrappers for RAPI Registry Query Functions

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegQueryValueEx (IntPtr hKey, string lpValueName,
```

Download at WoweBook.Com


```
   int lpReserved, ref REGTYPE lpType, IntPtr iptrData,
   ref int lpcbData);

[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegQueryValueEx (IntPtr hKey, string lpValueName,
   int lpReserved, ref REGTYPE lpType, string strData,
   ref int lpcbData);

[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegQueryValueEx (IntPtr hKey, string lpValueName,
   int lpReserved, ref REGTYPE lpType, ref int piData,
   ref int lpcbData);
```

As before, set parameters with `Reserved` in the name to `0`. The fourth parameter to `CeRegQueryValueEx` is a `REGTYPE` reference, a value from the following enumeration:

```
public enum REGTYPE
{
   REG_NONE = 0,
   REG_SZ = 1,
   REG_EXPAND_SZ = 2,
   REG_BINARY = 3,
   REG_DWORD = 4,
   REG_DWORD_LITTLE_ENDIAN = 4,
   REG_DWORD_BIG_ENDIAN = 5,
   REG_LINK = 6,
   REG_MULTI_SZ = 7,
   REG_RESOURCE_LIST = 8
}
```

Assuming that you have already opened a registry key—using a function described earlier—here is how to read a string value named `Launch10`:

```
int cbData=32;
Rapi.REGTYPE iType;
string str = new string(new char[cbData]);
Rapi.CeRegQueryValueEx(hkey, "Launch10", 0, ref iType, str,
   ref cbData);
char [] ach = {'\0'};
str = str.Trim(ach);
```

We trim away the extra `null` values to get the string by itself. In addition to returning the value, the `CeRegQueryValueEx` function returns two

Download at WoweBook.Com

other pieces of information: the registry key type (`iType`) and the number of bytes in the value (`cbData`).

Set a registry value by calling the `CeRegSetValueEx` function. Here are three overloaded versions of this function in our library:

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegSetValueEx (IntPtr hKey, string lpValueName,
   int Reserved, REGTYPE dwType, IntPtr lpData, int cbData);

[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegSetValueEx (IntPtr hKey, string lpValueName,
   int Reserved, REGTYPE dwType, string lpData, int cbData);

[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeRegSetValueEx (IntPtr hKey, string lpValueName,
   int Reserved, REGTYPE dwType, ref int lpData, int cbData);
```

The parameters to the `CeRegSetValueEx` function are similar to those passed to the `CeRegQueryValueEx` function. One difference is that there are fewer reference values (the `ref` keyword) because this function does not have to tell us the data type or the size of the input data. Here is how to call this function to write a string value named `Launch90`:

```
string strPath = @"\windows\pword.exe";
int cbData = (strPath.Length + 1) * 2;
Rapi.CeRegSetValueEx(hkeyInit, "Launch90", 0,
   Rapi.REGTYPE.REG_SZ, strPath, cbData);
```

Before shutting down, be sure that your program closes any registry key you opened, using the `CeRegCloseKey` function as defined here:

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int CeRegCloseKey (IntPtr hKey);
```

Listing 9.10, which we present at the end of this discussion, shows the registry enumeration code from `RegShowStartup`. This RAPI sample program shows two different enumeration functions: `CeRegEnumKeyEx` enumerates registry keys, and `CeRegEnumValue` enumerates the values in a given key. This sample program lists the startup programs in Windows CE and the startup DLLs. Figure 9.3 shows the startup programs for a Pocket PC as listed by `RegShowStartup`.

Download at WoweBook.Com

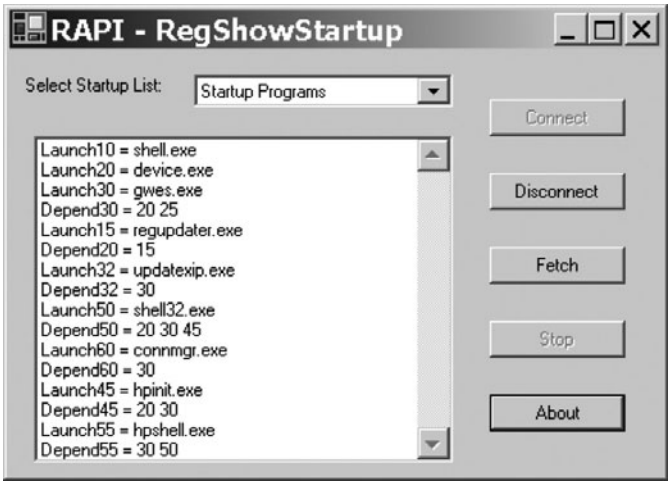


FIGURE 9.3: `RegShowStartup` **with the Startup List from a Windows Mobile Device**

Just as on desktop Windows, you can set a registry setting in Windows CE to start a program at system boot time. The registry key is `HKEY_LOCAL_MACHINE\init`. There is an important difference between the way Windows CE and desktop Windows use their startup lists. When you turn off the power for a desktop Windows system, the system shuts down completely. Then when the power comes back on, the items in the various startup lists get run.

A Windows CE device goes into a sleep state when the power switch is turned off. When the power is turned on, the system wakes up but does not completely reboot. For the programs in a startup list to run on a Windows CE device, you must do more than just turn the power off and on. You must either warm-boot the device (a reset button is commonly provided), or cold-boot the device by removing and then replacing the batteries.

The startup DLLs in Windows CE are really a special kind of device driver. These *stream interface drivers* are used for a wide range of devices. Such device drivers are outside the scope of this book, but it is worth knowing about them when you want to create an installable device driver. Such drivers must be written as Win32 DLLs.

Listing 9.10 shows the code for RapiEnumRegistryThread.cs.

LISTING 9.10: RapiEnumRegistryThread.cs from the `RegShowStartup` Program

```
// Program Name: RegShowStartup.exe
//
// File Name: RapiEnumRegistryThread.cs - Creates a thread to
```

Download at WoweBook.Com


```
// enumerate both registry keys and registry values
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//

using System;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using YaoDurant.Win32;

namespace RegShowStartup
{
   // Reasons our thread invokes user interface thread
   public enum INVOKE_ENUMREG
   {
      ENUMREG_NEWKEY,
      ENUMREG_NEWVALUE,
      ENUMREG_ENDED,
      STATUS_MESSAGE
   }

   /// <summary>
   /// RapiConnection - Manages ActiveSync RAPI connections
   /// </summary>
   public class RapiEnumRegistryThread
   {
      public string strBuffer;        // Interthread buffer
      public INVOKE_ENUMREG itReason; // Interthread reason

      // Public thread to allow monitoring by UI thread
      public Thread thrd = null;    // The contained thread
      private Control m_ctlInvokeTarget; // Interthread control
      private EventHandler m_deleCallback; // Interthread delegate

      // Private data.
      private bool m_bContinue;    // Continue flag
      private bool m_bKeys;        // Enumerate keys or values
      private IntPtr m_hkeyRoot;   // Root enumeration key
      private string m_strRegNode; // Root enumeration node

      public bool bThreadContinue  // Continue property
      {
         get { return m_bContinue; }
         set { m_bContinue = value; }
      }

      public RapiEnumRegistryThread(
```

Download at WoweBook.Com


```
    Control ctl,          // Who to call
    EventHandler dele,    // Notification delegate
    bool bKeys,           // Enum keys or values
    IntPtr hkeyRoot,      // Root of search
    string strRegNode)    // Node to search
{
    // Make private copies of init data.
    m_ctlInvokeTarget = ctl;
    m_deleCallback = dele;
    m_bKeys = bKeys;
    m_hkeyRoot = hkeyRoot;
    m_strRegNode = strRegNode;

    bThreadContinue = true;
}

public bool Run()
{
    ThreadStart ts = null;
    ts = new ThreadStart(ThreadMainEnumReg);
    if (ts == null)
        return false;

    thrd = new Thread(ts);
    thrd.Start();

    return true;
}


/// <summary>
/// ThreadMainEnumReg - Enumerate registry values.
/// </summary>
private void ThreadMainEnumReg()
{
    // Open registry key.
    IntPtr hkeySearchNode = IntPtr.Zero;
    int iResult = Rapi.CeRegOpenKeyEx(
        this.m_hkeyRoot, this.m_strRegNode,
        0, 0, ref hkeySearchNode);
    if (iResult != Rapi.ERROR_SUCCESS && m_bContinue)
    {
        // Send error message.
        itReason = INVOKE_ENUMREG.STATUS_MESSAGE;
        strBuffer = "Error accessing registry key";
        this.m_ctlInvokeTarget.Invoke(m_deleCallback);

        // Trigger end of enumeration.
        itReason = INVOKE_ENUMREG.ENUMREG_ENDED;
        this.m_ctlInvokeTarget.Invoke(m_deleCallback);
```

Download at WoweBook.Com


```
   // Trigger that shutdown is complete.
   thrd = null;
   return;
}

// Keys or values?
if (this.m_bKeys)  // Enumerate keys.
{
   int iIndex = 0;
   while (iResult == Rapi.ERROR_SUCCESS && m_bContinue)
   {
      string strKeyName = new string('\0', 32);
      int cbLength = 32;
      iResult = Rapi.CeRegEnumKeyEx(
         hkeySearchNode, iIndex++,
         strKeyName, ref cbLength,
         0, 0, 0, 0);
      if (iResult == Rapi.ERROR_SUCCESS && m_bContinue)
      {
         itReason = INVOKE_ENUMREG.ENUMREG_NEWKEY;
         strBuffer = strKeyName;
         this.m_ctlInvokeTarget.Invoke(m_deleCallback);
      }
   } // while
}
else               // Enumerate values.
{
   int iIndex;
   for (iIndex = 0; iResult == Rapi.ERROR_SUCCESS &&
        m_bContinue; iIndex++)
   {
      int cbName = 32;
      string strName = new string('\0', 16);
      Rapi.REGTYPE iType = 0;
      int cbLength = 0;

      // Enumerate key names only (not values).
      iResult = Rapi.CeRegEnumValue(hkeySearchNode,
         iIndex, strName, ref cbName,
         IntPtr.Zero, ref iType, IntPtr.Zero,
         ref cbLength);

      if (iResult == Rapi.ERROR_SUCCESS && m_bContinue)
      {
         if (iType == Rapi.REGTYPE.REG_SZ) // string
         {
            int cbData=32;
            string str = new string(new char[cbData]);
            Rapi.CeRegQueryValueEx(hkeySearchNode,
                strName, 0, ref iType, str,
```

Download at WoweBook.Com


```
                    ref cbData);
                char [] ach = {'\0'};
                strBuffer = strName.Trim(ach) + " = "
                   + str.Trim(ach);
             }
             else if (iType == Rapi.REGTYPE.REG_BINARY)
             {
                // Fetch binary array of short values.
                char [] ach = {'\0'};
                strBuffer = strName.Trim(ach) + " = ";

                // Allocate buffer of short values.
                Int16 sh = 0;
                IntPtr iptr;
                int cbSizeOfShort = Marshal.SizeOf(sh);
                int cbData =  cbSizeOfShort * 5;
                iptr = Marshal.AllocCoTaskMem(cbData);

                // Fetch array of short values.
                Rapi.CeRegQueryValueEx(hkeySearchNode,
                   strName, 0, ref iType, iptr,
                   ref cbData);

                // Copy array to managed array.
                int cElements = cbData / cbSizeOfShort;
                Int16 [] ash = new Int16 [cElements];
                Marshal.Copy(iptr, ash, 0, cElements);
                Marshal.FreeCoTaskMem(iptr);

                // Add values to string for display.
                for (int i = 0; i < cElements; i++)
                {
                   strBuffer = strBuffer + ash[i] + " ";
                }
             }
             else
             {
                strBuffer = strName + " not expected type";
             }
             itReason = INVOKE_ENUMREG.ENUMREG_NEWVALUE;
             this.m_ctlInvokeTarget.Invoke(m_deleCallback);
          }
       } // for
    } // if

    Rapi.CeRegCloseKey(hkeySearchNode);

    // Trigger end of enumeration.
    itReason = INVOKE_ENUMREG.ENUMREG_ENDED;
    this.m_ctlInvokeTarget.Invoke(m_deleCallback);
```

Download at WoweBook.Com


```
            // Trigger that shutdown is complete.
            thrd = null;
        }
    } // class RapiEnumRegistryThread
} // namespace RegShowStartup
```

9.2.3 Remote Access to Device Property Databases

The third kind of entry in the Windows CE object store is the Windows CE property database. This is really a flat-file manager that stores data as records in a memory-efficient way—suitable for small devices with small storage space. Desktop Windows has no comparable support.

Neither the .NET Compact Framework nor the .NET Framework provides any support for accessing Windows CE property databases. Such databases are not the same as SQL Server CE databases or Pocket Access databases; in short, such database files cannot be manipulated using ADO.NET classes. The primary benefits of using property databases are their very small size and their database format in common with non–.NET Compact Framework programs. Access to property databases occurs through a set of Win32 API functions, a subject addressed in existing Windows CE programming books.[4] Another set of database access functions is provided through RAPI, which offers the same level of support for reading and writing database records from a remotely connected PC.

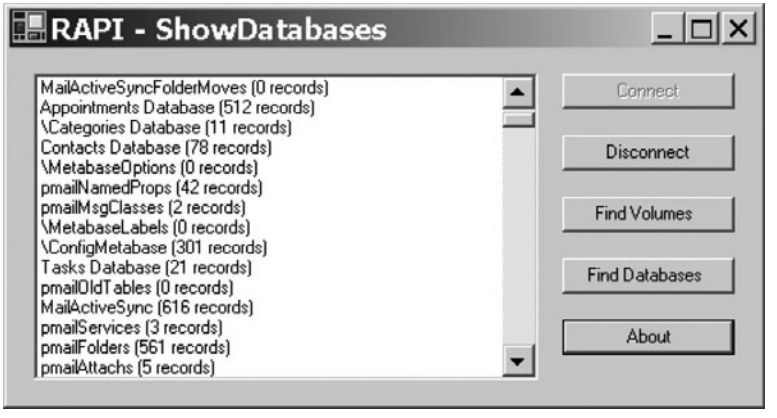


FIGURE 9.4: `ShowDatabases` **with Names of Databases Available on a Currently Connected Device**

4. For example, Doug Boling covers this topic in his book, *Programming Windows Embedded CE 6.0 Developer Reference* (Redmond, WA: Microsoft Press, 2008).

Download at WoweBook.Com

Our sample program, ShowDatabases, enumerates the available database volumes as well as the available databases on the connected Windows CE device. Figure 9.4 shows this program with the list of available databases. The code for the database enumeration portion of this program appears in Listing 9.11.

Listing 9.11: RapiEnumDBThread.cs from the ShowDatabases Program

```
// Program Name: ShowDatabases.exe
//
// File Name: RapiEnumDBThread.cs - Creates a background
// thread to names of database volumes and names of databases
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//
using System;
using System.Threading;
using System.Collections;
using System.Windows.Forms;
using System.Diagnostics;
using System.Runtime.InteropServices;
using YaoDurant.Win32;

namespace ShowDatabases
{
   // Reasons our thread invokes user interface thread
   public enum INVOKE_ENUMDB
   {
      ENUMDB_NEWVOLUME,
      ENUMDB_NEWDATABASE,
      ENUMDB_COMPLETE,
      STATUS_MESSAGE
   }

   /// <summary>
   /// Summary description for RapiEnumDBThread
   /// </summary>
   public class RapiEnumDBThread
   {
      public string strBuffer;       // Interthread buffer
      public INVOKE_ENUMDB itReason;   // Interthread reason

      private Thread m_thrd = null;    // The contained thread
      private Control m_ctlInvokeTarget; // Interthread control
      private EventHandler m_deleCallback; // Interthread delegate
      private bool m_bContinue; // Continue flag
      private bool m_bVolume;   // Enum volumes or databases?
```

Download at WoweBook.Com


```
public bool bThreadContinue // Continue property
{
   get { return m_bContinue; }
   set { m_bContinue = value; }
}

public RapiEnumDBThread(Control ctl, EventHandler dele,
   bool bVolume)
{
   m_bContinue = true;
   m_ctlInvokeTarget = ctl;  // Who to call
   m_deleCallback = dele;    // How to call
   m_bVolume = bVolume;
}
/// <summary>
/// Run - Init function for enum thread
/// </summary>
/// <param name="bSubDirs"></param>
/// <returns></returns>
public bool Run()
{
   ThreadStart ts = null;
   ts = new ThreadStart(ThreadMainEnumDB);
   if (ts == null)
      return false;

   m_thrd = new Thread(ts);
   m_thrd.Start();
   return true;
}

private void ThreadMainEnumDB()
{
   if (m_bVolume)  // Enumerate volumes.
   {
      Guid guid =
         new Guid("ffffffffffffffffffffffffffffffff");
      int cch = 32;
      string str = new String('\0', cch);
      while (Rapi.CeEnumDBVolumes(ref guid, str, cch) ==
         Rapi.TRUE && m_bContinue)
      {
         strBuffer = str;
         itReason = INVOKE_ENUMDB.ENUMDB_NEWVOLUME;
         m_ctlInvokeTarget.Invoke(m_deleCallback);
      }
   }
   else            // Enumerate databases.
   {
      Int16 cRecords = 0;
```

Download at WoweBook.Com


```
            IntPtr pfdbAll = IntPtr.Zero;
            Rapi.CeFindAllDatabases(0,
               Rapi.FAD.FAD_NAME | Rapi.FAD.FAD_NUM_RECORDS,
               ref cRecords,
               ref pfdbAll);
            IntPtr pfdb = pfdbAll;
            while (cRecords > 0)
            {
               // Set pointer to next record.
               Rapi.CEDB_FIND_DATA dbfd =
                  (Rapi.CEDB_FIND_DATA)
                  Marshal.PtrToStructure(
                     pfdb, typeof(Rapi.CEDB_FIND_DATA));

               // Post name to listbox.
               strBuffer = dbfd.DbInfo.szDbaseName + " (" +
                  dbfd.DbInfo.wNumRecords + " records)";
               itReason = INVOKE_ENUMDB.ENUMDB_NEWDATABASE;
               m_ctlInvokeTarget.Invoke(m_deleCallback);

               // Get ready for next loop.
               pfdb = (IntPtr)((int)pfdb + Marshal.SizeOf(dbfd));
               cRecords--;
            } // while

            // Free memory returned by CeFindAllDatabases.
            Rapi.CeRapiFreeBuffer(pfdbAll);
         } // if

         // Notify main thread that we are done.
         itReason = INVOKE_ENUMDB.ENUMDB_COMPLETE;
         m_ctlInvokeTarget.Invoke(m_deleCallback);

         // Mark thread as done.
         m_thrd = null;
      }

   } // RapiEnumDBThread
} // ShowDatabases
```

9.3 Detecting Changes in Device Connection State

ActiveSync provides two ways to let you know that a device has created a connection. We call one method the *auto-start method* and the other the *callback method.* The first is simple but crude, while the second involves more code but is also more elegant. You install both with registry entries in the desktop registry.

Download at WoweBook.Com

The auto-start method relies on registry keys that contain command lines to start programs running. When a connection begins, ActiveSync runs all of the programs in one key. When a connection ends, ActiveSync runs all of the programs in the other key. You could put the same program file in each registry key with a different command-line argument to differentiate when you're connecting from when you're disconnecting.

The callback method uses a callback mechanism—a COM interface—to let you know when the connection state changes. Your program must be running to get this notification. You could rely on the auto-start method to start your program running and then use the callback method to detect specific changes to the connection state.

Let us look at each approach in turn.

9.3.1 The Auto-Start Approach

When ActiveSync establishes a connection to a Windows CE device, it starts all of the programs listed in its `AutoStartOnConnect` registry key. When ActiveSync detects that a connection has been dropped, it starts another set of programs, this time the ones listed in the `AutoStartOnDisconnect` registry key. You can start as many programs as you like, with one registry value entry per program.

An auto-start registry entry is a program command line. Valid command lines include program names, program names with parameters, and data filenames with extensions of registered file types. You can test a command line by typing it into the Run dialog box from the Windows Start menu. Examples of valid auto-start registry entries include the following:

- `explorer.exe`
- `"explorer" c:\Temp`
- `"mspaint.exe" MyImage.bmp`
- `MyImage.bmp`

As these examples show, you can pass parameters to the program by putting the program name between quotation marks.

Here is the registry key for start connection command lines: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows CE Services\AutoStartOnConnect`. The registry key for end connection commands is identical except that the final part is `AutoStartOnDisconnect`.

The auto-start approach is easy to set up. A disadvantage is that each event starts a program running, which you have to shut down sooner or

Download at WoweBook.Com

later. Another disadvantage is that your auto-start program usually has to let another program know of the start or stop event, which leaves you having to pick an interprocess communication technique. The callback approach, which we discuss next, involves more code but ends up providing a more elegant solution than the auto-start approach.

9.3.2 The Callback Approach

ActiveSync supports a second notification mechanism, an Advise Sink. This COM-oriented approach involves creating an `IDccMan` COM object provided by the RAPI libraries. To get a callback, you must create an object that exposes the `IDccManSink` interface, which gets called when a connection has started or stopped.

ActiveSync uses TCP/IP for its connections. IP networking is so versatile that the IP connection can run on a wide range of physical media, including USB, serial, infrared, and Bluetooth—and, of course, over both wired and wireless networks. For most of the other RAPI operations, the use of TCP/IP is transparent to your application. When you use the callback method for notifications, however, ActiveSync sends you the IP address for the connected device. You can use this IP address to establish a socket connection to a device if you want to establish a direct link to a running program on your device.

Listing 9.12 shows the RapiConnectDetect.cs source file, which has the C# declarations needed for the two interfaces used to detect changes to an ActiveSync connection. This source file is part of `FindProgramsOnConnect`, the latest installment in our ongoing saga of running programs on a device from a desktop using RAPI. In this episode, our hero automatically starts a search when a device connection is detected. It uses the faster of the two search methods and so provides a complete list of executable programs whenever a device is connected.

Listing 9.12: RapiConnectDetect.cs for Detecting ActiveSync Connections

```
// Program Name: FindProgramsOnConnect.exe
//
// File Name: RapiConnectDetect.cs - Creates an ActiveSync COM
// object of type IDccMan and an Advise Sink of type IDccManSink
// to detect the start and end of ActiveSync connections
//
// Code from _.NET Compact Framework Programming with C#_
// and _.NET Compact Framework Programming with Visual Basic .NET_
// Authored by Paul Yao and David Durant.
//
```

Download at WoweBook.Com


```
using System;
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace FindPrograms
{
   // Reason we are calling
   public enum INVOKE_CONNECT
   {
      START,
      STOP,
   }

   /// <summary>
   /// Provides notifications of ActiveSync connection
   /// </summary>
   public class RapiConnectDetect : IDccManSink
   {
      public INVOKE_CONNECT itReason;   // Interthread reason

      private EventHandler m_deleCallback; // Callback delegate
      private Control m_ctlTarget; // Target for Invoke calls

      private IDccMan m_pDccMan = null;  // RAPI-provided object
      private bool m_Connected = false;   // Connection state
      private int m_iAdviseSinkContext = 0; // Advise Sink context

      [DllImport("Ole32.dll"), PreserveSig]
      private static extern int
         CoCreateInstance(
         [In] ref Guid clsid,
         [MarshalAs(UnmanagedType.Interface)]
         object punkOuter,
         [MarshalAs(UnmanagedType.I4)] int context,
         [In] ref Guid iid,
         [In, Out, MarshalAs(UnmanagedType.LPArray)]
         IDccMan[] pDccMan);

      public const int CLSCTX_INPROC_SERVER = 1;

      /// <summary>
      /// RapiConnectDetect -- class constructor
      /// </summary>
      /// <param name="dele"></param>
      public RapiConnectDetect(Control ctl, EventHandler dele)
      {
         m_deleCallback = dele; // Callback for notifications
         m_ctlTarget = ctl;     // Control to notify
      }
```

Download at WoweBook.Com


```
public bool Init()
{
   System.Guid IID_IDccMan = new
      System.Guid("A7B88841-A812-11CF-8011-00A0C90A8F78");
   System.Guid CLSID_DccMan = new
      System.Guid("499C0C20-A766-11CF-8011-00A0C90A8F78");

   IDccMan [] pTempArray = new IDccMan[1];
   int iErr = CoCreateInstance(ref CLSID_DccMan, null,
      CLSCTX_INPROC_SERVER, ref IID_IDccMan, pTempArray);

   if (pTempArray[0] == null)
      return false;

   m_pDccMan = pTempArray[0];
   return true;
}

/// <summary>
/// Enable - Toggle Advise Sink operation
/// </summary>
/// <param name="bEnable"></param>
public void Enable(bool bEnable)
{
   if(bEnable && m_iAdviseSinkContext == 0)
   {
      IDccManSink idcc = (IDccManSink)this;
      m_pDccMan.Advise(idcc, ref m_iAdviseSinkContext);
   }

   if (!bEnable && m_iAdviseSinkContext != 0)
   {
      m_pDccMan.Unadvise(m_iAdviseSinkContext);
      m_iAdviseSinkContext = 0;
   }
}

//
// IDccManSink interface functions
//

public int OnLogAnswered()  { return 0; } // Line detected
public int OnLogActive()    { return 0; } // Line active

/// <summary>
/// OnLogIpAddr - First event when it makes sense to
/// try to do any real work with target device
/// </summary>
/// <param name="dwIpAddr"></param>
/// <returns></returns>
```

Download at WoweBook.Com


```
    public int OnLogIpAddr(int dwIpAddr) // Link established.
    {
       if (!m_Connected) // Notify only if not connected.
       {
          this.itReason = INVOKE_CONNECT.START;
          this.m_ctlTarget.Invoke(this.m_deleCallback);
          m_Connected = true;
       }
       return 0;
    }


    /// <summary>
    /// OnLogDisconnection - Connection ended.
    /// </summary>
    /// <returns></returns>
    public int OnLogDisconnection()
    {
       if (m_Connected) // Notify only if connected.
       {
          this.itReason = INVOKE_CONNECT.STOP;
          this.m_ctlTarget.Invoke(this.m_deleCallback);
          m_Connected = false;
       }

       return 0;
    }

    public int OnLogListen()      { return 0; }
    public int OnLogTerminated()  { return 0; }
    public int OnLogInactive()    { return 0; }
    public int OnLogError()  { return 0; }

} // class RapiConnectDetect


#region IDccMan Definitions
/// <summary>
/// IDccMan - Interface of COM component provided by RAPI
/// </summary>
[Guid("a7b88841-a812-11cf-8011-00a0c90a8f78"),
InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
interface IDccMan   // ActiveSync notification interface
{
    void Advise (IDccManSink pDccSink, ref int pdwContext);
    void Unadvise(int dwContext);
    void ShowCommSettings ();
    void AutoconnectEnable ();
    void AutoconnectDisable ();
    void ConnectNow ();
```

Download at WoweBook.Com


```
      void DisconnectNow ();
      void SetIconDataTransferring ();
      void SetIconNoDataTransferring ();
      void SetIconError ();
   }

   /// <summary>
   /// IDccManSink - Interface we implement to grab notifications
   /// </summary>
   [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
   interface IDccManSink
   {
      [PreserveSig] int OnLogIpAddr (int dwIpAddr);
      [PreserveSig] int OnLogTerminated ();
      [PreserveSig] int OnLogActive ();
      [PreserveSig] int OnLogInactive ();
      [PreserveSig] int OnLogAnswered ();
      [PreserveSig] int OnLogListen ();
      [PreserveSig] int OnLogDisconnection ();
      [PreserveSig] int OnLogError ();
   }

#endregion

} // namespace FindPrograms
```

One of the most interesting aspects of the connection detection callback mechanism is that we do not create a separate thread for the callback mechanism. Instead, we run on the program's user interface thread. At first glance, this might seem like the wrong approach to take. After all, any work on the user interface thread can cause unwanted delays in the responsiveness of a program's user interface.

But the work of an Advise Sink, generally speaking, can be done very quickly. What is more important, however, is that an Advise Sink is an interprocess communication that relies on the time-tested Win32 message delivery mechanism (the `GetMessage` or `PeekMessage` "message pump" function). If we did decide to put the callback on a background thread, we would need to tell the .NET Framework to run the message pump. The desktop .NET Framework provides us with some help to do just that, namely a static method named `DoEvents` in the `System.Windows.Forms.Application` class. A property of that same class, `MessageLoop`, would even let us know whether a message loop already existed in the thread (in which case we would not have to call the `DoEvents` method). Without a message loop on its thread, an Advise Sink would

Download at WoweBook.Com

never work. In our example, we rely on the message loop in the program's main thread for this. You can do the same in your production code, or you can create a background thread—if you make sure there is a message queue and active polling of that queue in whichever thread owns the callback function. Our connection detection callback object gets created in the constructor of our program's main form with the following code:

```
// Create detect connect object.
m_rapicd = new RapiConnectDetect(this, this.m_deleConnect);
m_rapicd.Init();
m_rapicd.Enable(true);
```

Our connection detection mechanism runs on the user interface thread, which means we get free synchronization with the program's user interface. Yet we filter all calls from the callback to our controls using the thread-safe `Invoke` function. By giving a single point of contact between the two classes, we make our code easier to support (and, we hope, to reuse!), and we also make it easy to put the callback on a background thread if that becomes necessary. This example shows that calls to the `Invoke` function—which are required for interthread calls to controls—are quite flexible. `Invoke` can be used to call between threads (as shown earlier); it can also be used when the caller and the target control are on the same thread. Newcomers to .NET programming often make the mistake of concluding that the `Invoke` function can be used only for interthread calls, but that is not the case.

As with other callback functions, we first need a declaration for a delegate variable, such as this one:

```
private EventHandler m_deleConnect;
```

We initialize this value with a pointer to our target function:

```
m_deleConnect = new EventHandler(this.DetectCallback);
```

Listing 9.13 shows our callback function.

Listing 9.13: `DetectCallback` Detecting the Starting and Stopping of Connections

```
/// <summary>
/// DetectCallback - Called to notify start and end of
/// ActiveSync connections
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

Download at WoweBook.Com


```
private void
DetectCallback(object sender, System.EventArgs e)
{
   INVOKE_CONNECT it = this.m_rapicd.itReason;
   switch(it)
   {
      case INVOKE_CONNECT.START:
         this.cmdConnect_Click(this, new EventArgs());
         this.txtStartDir.Text = @"\";
         this.chkSubs.Checked = true;
         this.cmdFind_Click(this, new EventArgs());
         break;
      case INVOKE_CONNECT.STOP:
         this.cmdDisconnect_Click(this, new EventArgs());
         break;
   }
}
```

When a connection is detected, we simulate the actions a user would perform for a manual connection: Click on the Connect button, fill in the search criteria, then click on the Find button. When a disconnect event is received, we once again simulate the user action, which is to click on the Disconnect button.

9.4 Loading Programs and DLLs

Most of the discussion in this chapter has been about using RAPI functions to access the various parts of a device's object store. In this section, we look at starting programs running and also at loading DLLs. You can run any type of program, including Win32 programs and .NET Compact Framework programs. When it comes to loading DLLs, you can load only Win32 DLLs. The reason is that RAPI calls into DLLs at exported functions, and .NET Compact Framework libraries cannot export functions.

Of course, you can access programs and DLLs that are already present in the Windows CE file system. The really interesting scenario, however, involves downloading a program or a DLL to a device and starting it up. For that to work for Win32 executables, you must know what CPU is present on the target system. We are going to spend some time looking at how to retrieve device-side information to know what CPU is present before you download a program.

The requirements for running .NET Compact Framework programs are different. You do not need to know what CPU is present because .NET Compact Framework programs are portable between different CPUs.

Download at WoweBook.Com

Before you download and run a .NET Compact Framework program, however, you want to be sure that the .NET Compact Framework itself is present on a system. We will show one way to accomplish that.

9.4.1 Running Device-Side Programs

You start a program running by calling the `CeCreateProcess` function. You can call this function by using either of two P/Invoke wrappers, as shown here:

```
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeCreateProcess (string lpApplicationName,
   string lpCommandLine, int Res1, int Res2, int Res3,
   int dwCreationFlags, int Res4, int Res5, int Res6,
   ref PROCESS_INFORMATION lpProcessInformation);

[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern
int CeCreateProcess (string lpApplicationName, int Zero,
   int Res1, int Res2, int Res3, int dwCreationFlags, int Res4,
   int Res5, int Res6,
   ref PROCESS_INFORMATION lpProcessInformation);
```

We define two wrappers to give you a choice of providing arguments for the program or not. The first and last parameters for both wrappers are the most important ones—the rest can be set to some form of zero (either `0` or `string.Empty`). The first parameter is the fully qualified path of the program. If a program is not in the `\windows` directory, you must specify the full path because Windows CE does not have a `PATH` environment variable as desktop Windows has.

The last parameter is a pointer to a `PROCESS_INFORMATION` structure, whose P/Invoke definition is as follows:

```
[StructLayout(LayoutKind.Sequential,Pack=4)]
public struct PROCESS_INFORMATION
{
   public IntPtr hProcess;
   public IntPtr hThread;
   public int dwProcessId;
   public int dwThreadId;
};
```

Here is how to run the calculator program, `calc.exe`, using this data structure and the `CeCreateProcess` function:

Download at WoweBook.Com


```
string strProg = @"\windows\calc.exe"
Rapi.PROCESS_INFORMATION pi = new Rapi.PROCESS_INFORMATION();
Rapi.CeCreateProcess(strProg, 0, 0, 0, 0, 0, 0, 0, 0, ref pi);

Rapi.CeCloseHandle(pi.hProcess);
Rapi.CeCloseHandle(pi.hThread);
```

Notice that we close the handles to the create process and thread after starting the program. Closing the handles does not cause the running program to terminate but only releases the connection we have to that program. Doing so allows Windows CE to clean up properly after the process has terminated.

9.4.2 Loading Device-Side DLLs

Call the `CeRapiInvoke` function from your desktop program to load a device-side DLL and call an entry point in that DLL. The two P/Invoke wrappers for this function are shown here:

```
// Block mode – Set ppIRAPIStream to zero.
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int
CeRapiInvoke (string pDllPath, string pFunctionName, int cbInput,
   IntPtr pInput, ref int pcbOutput, ref IntPtr ppOutput,
   int Res1, int dwReserved);

// Stream mode – Set ppIRAPIStream to valid IRAPIStream interface.
[DllImport("rapi.dll", CharSet=CharSet.Unicode)]
public static extern int
CeRapiInvoke (string pDllPath, string pFunctionName, int cbInput,
   IntPtr pInput, int Res1, int Res2,
   ref IRAPIStream ppIRAPIStream, int dwReserved);
```

This function has two overloads because it has two modes: block and stream. You can tell what mode is selected by looking at the next to last parameter (`Res1`), which is `0` for block mode or a valid COM interface pointer for stream mode. The first function overload supports block mode because it lets you set this key parameter to `0`. The second function overload is for stream mode, for which the `ppIRAPIStream` value must be a valid `IRAPIStream` interface pointer.

Whichever of the two modes you select, when you call the `CeRapiInvoke` function, it loads the requested DLL and calls the function you name. Version 1.0 of the .NET Compact Framework does not let you create a DLL that exports C-callable functions. The target DLL, then, cannot

Download at WoweBook.Com

Reserved Parameters

When using P/Invoke to call into Win32 libraries, you sometimes run into situations when you need to pass a `NULL` pointer to a Win32 function, as in the call to the `CeRapiInvoke` function to request block mode. To keep things simple, we overload the function and set parameter names to `Reserved` (or when there are several reserved parameters, `ResN` where `N` = 1, 2, 3, etc.). Set reserved values to `0`.

be a managed-code DLL. It must instead be an unmanaged DLL, and the function you call must be a C or C++ function.

Whether you are an experienced C++ programmer or not, the Win32 declarations for exporting a DLL function contain elements that are specific to Windows programming. Here is how to declare an exported function named `InvokeFunction` for a device-side DLL so that the function gets accessed using a call to the `CeRapiInvoke` function in a desktop-side application:

```
// Win32 declaration for device-side DLL function
extern "C" HRESULT __declspec(dllexport)
InvokeFunction( DWORD cbInput, BYTE  *pInput,
   DWORD *pcbOutput, BYTE  **ppOutput,
   IRAPIStream *pIRAPIStream)
```

Even experienced C/C++ programmers (particularly those who are new to Win32 programming) might be puzzled by this function declaration. Here are details on what each part of this function declaration means.

- `extern "C"`: Retain C-style function names. A C++ style function name gets "decorated" to support function overloading; we disable that feature with this declaration.
- `HRESULT`: This is the function return value, a COM "result code." The .NET approach to handling such values is to treat them as 32-bit integers (a C# `int`).
- `__declspec(dllexport)`: This function is exported to make it visible outside the DLL. This or a comparable entry in the `EXPORTS` section of a module definition (.def) file is needed because DLL functions are private (and therefore hidden) by default in Windows.

Download at WoweBook.Com

- `InvokeFunction`: This is the name of our function. You can choose any name you like, and you must pass this same function name in the call to the `CeRapiInvoke` function as the `pFunctionName` parameter.

The five parameters to the device-side function map directly to five of the eight parameters that get passed to the `CeRapiInvoke` function. How these parameters are interpreted depends on whether the call is being made in block mode or in stream mode. The final parameter, `pIRAPIStream`, is `NULL` for block mode and other than `NULL` for stream mode.

In block mode, the first two parameters refer to data being sent to the device: `cbInput` says how many bytes are in the buffer of data pointed to by `pInput`. The next two parameters are used to send data back to the desktop: `pcbOutput` refers to a buffer for how many bytes are being sent back, and `ppOutput` retrieves a pointer to the buffer holding the return data.

In stream mode (i.e., when `pIRAPIStream` is not `null`), `pIRAPIStream` is a valid pointer to an `IRAPIStream` interface. Stream mode provides an open data pipe, driven by the device-based DLL. This interface contains 14 member functions; the most noteworthy are the `Read` and `Write` member functions. With these two functions, the device-side DLL can exchange data with the calling desktop application.

For example, Listing 9.14 shows a DLL function that accepts a Unicode string and converts it to uppercase letters before sending it back to the caller. There are, of course, easier ways to convert a string to uppercase. When you get this function to work, you know that your device-side DLL is functioning properly. This function was taken from the `SimpleBlockModeInvoke` sample program (available online with the rest of the book's source code, with the other C++ source projects in the `CPP` directory tree).

Listing 9.14: `UpperCaseInvoke`, **a Device-Side Win32 Function**

```
//------------------------------------------------------------
// UpperCaseInvoke -- Win32 device-side function
//------------------------------------------------------------
extern "C" HRESULT __declspec(dllexport)
UpperCaseInvoke( DWORD cbInput, BYTE  *pInput,
   DWORD *pcbOutput, BYTE  **ppOutput,
   IRAPIStream *pIRAPIStream )
{
   // Init output values
   *pcbOutput = 0;
   *ppOutput = (BYTE *)NULL;

   if (cbInput != 0)
```

Download at WoweBook.Com


```
   {
      // Check for a valid input parameter.
      if (IsBadReadPtr(pInput, cbInput))
         return E_INVALIDARG;

      // Allocate a buffer.
      BYTE * pData = (BYTE *)LocalAlloc(LPTR, cbInput);
      if (pData == NULL)
         return E_OUTOFMEMORY;

      // Copy bytes.
      memcpy(pData, pInput, cbInput);

      // Make uppercase.
      LPWSTR lpwstr = (LPWSTR)pData;
      wcsupr(lpwstr);

      // Send return data to caller.
      *pcbOutput = cbInput;
      *ppOutput = pData;
   }

   return S_OK;

} // UpperCaseInvoke
```

9.4.2.1 *Calling a DLL Function with* `CeRapiInvoke`

The companion program that calls our DLL is named `CallDeviceDll`. Like the other programs in this chapter, `CallDeviceDll` uses our ActiveSync wrapper library, `YaoDurant.Win32.Rapi.dll`, to access the service of rapi.dll.

There are two basic challenges to calling the `CeRapiInvoke` function in block mode. One is getting the data formatted to send down to the device; the second is retrieving the data when we get it back from the device. For both of these, we rely on the `IntPtr` type and the members of the `Marshal` class to do the heavy lifting for us.

To create the unmanaged Unicode string that we send to the device, we call the `Marshal.StringToHGlobalUni` function, which accepts a string as input, allocates an object in the local (unmanaged) heap of the current process, and copies the Unicode string to the memory. We get back a pointer to the memory, which is the parameter we pass as input to the `CeRapiInvoke` function. We must allocate from the process heap because the `CeRapiInvoke` function frees the memory after it is done with it.

Download at WoweBook.Com

Local and Global Memory

The Win32 API contains many sets of memory allocation functions. Two sets of functions trace their roots back to the very first version of Windows: the local allocator (`LocalAlloc`) and the global allocator (`GlobalAlloc`). In the original Win16 API, these memory types were very different. The `GlobalAlloc` set of functions was the segment allocator, and the `LocalAlloc` set of functions was the subsegment allocator. Today, the two sets of functions are supported for backward compatibility, and calls to each get sent to the native Win32 heap allocator function, `HeapAlloc`. There is, in short, no difference between the two.

This is important to know because the RAPI documentation describes how you must call `LocalAlloc` with data you send into `CeRapiInit`, and later call `LocalFree` with the buffer you get back from the function. The .NET P/Invoke helper class, `Marshal`, does not provide a `LocalAlloc` equivalent, although it does support `GlobalAlloc`. The two, however, are interchangeable, as you can see by studying our sample program.

Listing 9.15 shows the managed-code method from `CallDeviceDll` that calls the `CeRapiInvoke` function.

Listing 9.15: Calling `CeRapiInvoke` from Managed Code

```
private void
cmdInvoke_Click(object sender, System.EventArgs e)
{
   // Fetch string from textbox.
   string strInput = textInput.Text;

   // Set up input values - count of bytes.
   int cbInput = (strInput.Length + 1) * 2;
   IntPtr ipInput = Marshal.StringToHGlobalUni(strInput);

   // Set up output values.
   IntPtr ipOutput = IntPtr.Zero;
   int cbOutput = 0;

   // Call device-side DLL.
   Rapi.CeRapiInvoke(@"\windows\SimpleBlockModeInvoke.dll",
      "UpperCaseInvoke",
      cbInput,
      ipInput,
```

Download at WoweBook.Com


```
      ref cbOutput,
      ref ipOutput,
      0, 0);

   // Calculate output length (count of characters).
   int cchOut = (cbOutput / 2) - 1;

   // Convert return value to a string.
   string strOutput =
      Marshal.PtrToStringUni(ipOutput, cchOut);

   // Free memory allocated by caller
   Marshal.FreeHGlobal(ipOutput);

   // Display resulting string.
   MessageBox.Show(strOutput, "CallDeviceDll");

} // cmdInvoke_Click
```

9.5 Conclusion

In this chapter, we looked at using RAPI from managed code. When a connection has been established through ActiveSync, RAPI lets a Windows desktop program connect to and control a Windows CE device. In addition to accessing a device's object store, RAPI also lets you start programs running and load Win32 DLLs. RAPI's various features allow desktop-side code to connect to and manage a Windows CE device while the device is in its home port—its docking cradle.

The next chapter begins the final section of this book, which covers the creation of graphical output from a .NET Compact Framework program. Your need for creating graphical output depends on the type of programming you expect to do. Some programmers rely on existing controls to display information to the user, so they have little need for graphical output. Other programmers want to take full control of the visual display provided by their programs and create custom controls, custom graphical output, and hard-copy output. This second group of programmers is going to find helpful hints in the next section, starting with Chapter 11, Creating Graphical Output, which provides the basics for creating graphical output in a .NET Compact Framework program.

Download at WoweBook.Com

10 Windows Communication Foundation

This chapter describes the basic WCF capabilities as implemented in the full .NET Framework to provide context for understanding WCF.[1] Our main focus is on understanding how to implement WCF clients on Windows Mobile devices. Out of necessity, this chapter also describes implementing a WCF service. The .NET Compact Framework does not support the creation of WCF services, so instead we rely on the full .NET Framework running on a desktop version of Microsoft Windows—Windows XP or Windows Vista, for example—to provide a service for our device-side client to connect with.

10.1 What Is WCF?

WINDOWS COMMUNICATION FOUNDATION (WCF) provides a unified programming model for linking software modules that may be deployed across a network boundary but that are made to appear as local procedure calls. WCF could easily be called "Common Communication Foundation," like other .NET parts that have a "Common" first name. The word *common* can mean "plain" or "ordinary," but we prefer "standard"—

1. For a deeper understanding of WCF, we recommend the following book, another title in the Microsoft .NET Development Series: *Essential Windows Communication Foundation for .NET Framework 3.5,* by Steve Resnick, Richard Crane, and Chris Bowen (Boston: Addison-Wesley, 2008).

Download at WoweBook.Com

or perhaps "universal"—because each "common" technology plays a role in the set of standard elements for .NET software development.[2]

WCF could just as easily have been called "Distributed Service Foundation," since it is about enabling distributed services using whatever supporting technologies are required. Our point is that WCF is not just about communication, which its name seems at first to suggest. Instead, WCF connects service providers and service consumers. Perhaps we quibble too much; in WCF, after all, both the service and the underlying communications channels play important roles.

WCF provides a rich and extensible set of capabilities for **service-oriented computing (SOC)**. This is part of a long-term trend toward enabling and enhancing distributed computing. This trend is about breaking down barriers and opening new avenues toward giving users access to their data—and the services that manage that data—from a disparate set of clients, whether those clients are running as a thick client or in a browser, or are deployed on a PC or mobile phone. This trend creates more choices for system architects, who can look to deploy software components on a broad range of machines and connect them together using whatever communications transport happens to be available. Some industry pundits see service orientation as the next "big thing" in computing; for our perspective, see the Service-Oriented Computing sidebar.

We start our discussion of WCF with a description of the support that is available in the full .NET Framework. Perhaps the most important reason to cover the full .NET Framework is that the .NET Compact Framework supports only the creation of WCF clients. Even if your only interest is in working with device-side development, you will need some understanding of the full framework to create a WCF service—that runs on a desktop or server system—to provide the other end of the connection that your Windows Mobile WCF client will be accessing.

If you are already familiar with WCF, we suggest that you jump ahead to the discussion of WCF in the Compact Framework. Otherwise, we provide this overview because the technology is new enough that few have had a chance to dig into it and learn about it. In addition, the support provided in the first version of the Compact Framework is likely to grow over time, and so an understanding of the full implementation can only help to prepare you for future enhancements.

---

2. This list includes Common Type System, Common Language Infrastructure, Common Intermediate Language, Common Language Runtime, and Common Language Specification.

Download at WoweBook.Com

Service-Oriented Computing

The meaning of *service-oriented computing* depends in large part on the specific vendor you are talking with. For Microsoft, it means "Software plus Service," since the company fully expects that client-oriented computing will continue to be important and that client/server computing enhances it but does not replace it. For IBM, with its focus on consulting services, it means providing a business orientation to everything it offers: hardware, software, and professional services. Web-centric companies (such as Google and Salesforce.com) see service-oriented computing as presaging the end of thick client computing and the eventual dominance of Web-centric systems. Which is right? The market will decide in the long run, and all of us will have to wait to find out.

We believe service-oriented computing is just another step in the evolution of computing. Business computing started in the 1950s with large, centralized systems. The 1960s and 1970s brought minicomputers and the start of decentralization. The personal computers of the 1980s and 1990s brought the pendulum more fully toward a decentralized approach. Along the way, the World Wide Web of the 1990s started the pull toward a more centralized way of doing things. A service-oriented model of computing is not an "either/or" question but rather is about finding a way to have "both/and." There will continue to be decentralized personal systems—witness the success of mobile phones—along with centralized enterprise systems. The world of service-oriented computing is about enabling the ability to tie service producers with service consumers. And that is just what WCF enables.

WCF was designed to overcome some of the shortcomings of earlier technologies. At the same time, WCF was designed to interoperate with earlier technologies to allow a smooth migration into a WCF-enabled world. A WCF client can be built to connect to and interoperate with the remote procedure calling mechanism supported in the Component Object Model (COM) world (including COM+ and DCOM), as well as distributed applications built on .NET Remoting and with ASP.NET Web Services. WCF provides support for some very old technologies, such as native Win32 Named Pipes. WCF also supports very new technologies, including Representational State Transfer (REST). REST extends the standard HTTP

Download at WoweBook.Com

protocol to enable data exchange—sometimes called "hypermedia"—using nothing more than URIs. By supporting this wide range of technologies, WCF is a good choice for creating highly interoperable services.

WCF was also built to be extensible. If you do not find what you need in the basic set of services, you can always build your own custom extensions. The Windows Mobile team created one such extension, a set of custom transports for SMTP (the Internet standard for e-mail exchange). This new transport—also known as the Mail Transport—ships with the .NET Compact Framework. It enables the use of e-mail to communicate in a peer-to-peer fashion between two WCF-enabled applications. Before we dig into more of the WCF programming details, we need to explain WCF terminology. There are few new concepts here but rather precise ways of describing the world from the WCF perspective.

10.1.1 WCF Terminology

Perhaps the most important WCF term to understand is **endpoint.** An endpoint provides a destination for a message and is analogous to a telephone handset for voice communication. When a client talks to a service, both the client and the service have an endpoint. To a software developer, the most interesting kinds of messages are those for method calls from a client into a service. And those are the kinds we are going to focus on.

Enabling an endpoint requires three basic pieces, which you can remember with the letters *ABC.* The letter *A* stands for the **address** of a service, the letter *B* stands for **binding,** and the letter *C* stands for **contract.**

An endpoint address is defined using a URI. Table 10.1 has sample URIs for different WCF-compatible transport protocols. Each transport protocol has a unique prefix for the URIs that you use in forming the service address. The .NET Compact Framework supports two of the available transports, as indicated in the rightmost column of Table 10.1.

TABLE 10.1: Transports and Sample URIs

Transport Protocol	Sample URIs	.NET Compact Framework Supported
HTTP	www.paulyao.com/power/rates http://localhost:1108/power/rates	Yes
HTTPS	https://www.paulyao.com/folder/Service https://localhost:8080/some/OtherService	Yes

Download at WoweBook.Com

TABLE 10.1: Transports and Sample URIs *(Continued)*

Transport Protocol	Sample URIs	.NET Compact Framework Supported
Microsoft Message Queue (MSMQ)	net.msmq://paulyao.com/power/rates net.msmg://10.0.0.200/Still/Other/Service	No
Named Pipes	net.pipe://paulyao.com/power/rates net.pipe://10.0.0.101/power/rates	No
TCP	net.tcp://www.paulyao.com:8123/power/rates net.tcp://10.0.0.125:8246/power/rates	No

From the URI, a client can locate the server on a network and the service on the server. But two other details are still needed. One of these, the binding, tells our client system how to communicate with the server system. The third and final item, the contract, provides the details on how our client code can interact with the specific service.

The binding describes all that goes on to pack messages at the client and unpack them correctly at the service (and back again, as needed, for the return trip home). A predefined set of 13 bindings are defined with WCF, with a wide range of settings to help customize the packing and transport. Table 10.2 lists five predefined bindings (for a complete list of the 13 predefined bindings, see the MSDN documentation).

So, what goes into a binding? A binding is like a network stack; in WCF-speak, a binding is a collection of binding elements. The elements are invoked in a precise order, with the transport binding element being at the bottom of the stack. There are several types of binding elements, including ones for transactions, reliability, security, message encoding, and transport. Of the various types, most are optional. Only two types of binding elements are required: a message encoding binding element and a transport binding element.

Of the 13 predefined bindings, the .NET Compact Framework supports only: `basicHttpBinding`. This predefined binding enables basic Web services over HTTP with a message encoding of text/XML and a transport binding of HTTP. With the Compact Framework, security is supported in the form of an X.509 certificate that must be configured in code.

Download at WoweBook.Com

One other transport might interest you: the Mail Transport, which ships with the .NET Compact Framework. For Windows Mobile devices that are connected to an Exchange Server, the e-mail transport provides a way to build either client/server or peer-to-peer topologies. We exclude that transport from the accompanying table because there is no corresponding entry in the predefined system bindings.

We have covered two of the three items needed to establish connections: an address and the binding. The address indicates where to find a service and the binding provides details on formatting and transporting message data. The third remaining item, the contract, spells out the supported programmable details. These details include the available method calls (**service contracts**), the layout of data structures that are passed (**data contracts**), and how exceptions are to be handled (**fault contracts**). The .NET Compact Framework does support service contracts and data contracts, but it does not support fault contracts.

In code, contracts are defined using managed-code attributes. Available methods are defined by applying two attributes, `ServiceContract` and `OperationContract`, to an interface and the interface methods. The layout of passed data structures is defined by applying two other attributes, `DataContract` and `DataMember`, to a data structure definition and the

TABLE 10.2: System Bindings and .NET Compact Framework Support

System Binding	Description	.NET Compact Framework Supported
`basicHttpBinding`	HTTP for transport and text/XML for message encoding. Suitable for communicating with ASMX-based Web services	Yes
`mexHttpBinding`	Metadata exchange bindings, suitable for enabling discovery of a service's additional endpoints	No
`wsHttpBinding`	HTTPS secure interoperable binding	No
`netTCPBinding`	Secure, reliable binding using TCP for message delivery and binary message encoding	No
`netMsmqBinding`	Queued binding suitable for communicating between WCF-enabled applications	No

Download at WoweBook.Com

individual data members contained therein. Like most programming keywords, those that apply to WCF contracts are best understood in the context of actual code (see Listing 10.1).

Listing 10.1: An Interface Marked with WCF Attributes

```
[ServiceContract]
public interface IProvider
{
    [OperationContract]
    string[] GetAllProviderCodes();

    [OperationContract]
    string GetProviderName(string strProviderCode);

    [OperationContract]
    Rate GetProviderRate(string strProviderCode, DateTime dtWhen);
}
```

Listing 10.1 shows code taken from the WCF service sample that appears later in this chapter. In the example, the `IProvider` interface has been decorated with the appropriate service contract attributes so that the marked methods are available to clients. A WCF service is defined in terms of methods from a specific interface. The use of interfaces maps well to the ideas around a service contract, which describe the basic syntax of the service contract without including a specific implementation. As shown in the example, an interface is marked with the `ServiceContract` attribute. Within such an interface, methods are marked with the `OperationContract` attribute—and at least one method must be so marked. This opt-in approach means you can hide methods that are defined in an interface but that you decide not to expose in a specific WCF service.

Data contracts are defined in a manner similar to the way service contracts are defined, as shown in Listing 10.2. The `DataContract` attribute identifies a class (or structure) that contains one or more data elements. As with service contracts, data contracts also use an opt-in approach, as data elements without the `DataMember` attribute are excluded from the wire format. Those that have the attribute, on the other hand, are included when an instance of the structure is included as either an input parameter or a return value.

Listing 10.2: A Data Structure Marked with WCF Attributes

```
[DataContract]
public class Rate
```

Download at WoweBook.Com


```
{
    [DataMember] public Single BaseRate;
    [DataMember] public Int32 OveruseThreshold;
    [DataMember] public Single OveruseRate;
}
```

Describing the service contracts and the data contracts for a WCF service is straightforward. Service contracts involve defining an interface; an interface that contains the specific methods that the service is to provide. You must then write the code to implement the interface definitions, giving substance to the promises made by the various contracts.

There is more to creating a WCF service than defining contracts. We will return to that subject, once we have summarized some of the limitations that you need to understand about WCF support in the .NET Compact Framework. As we already stated, one of the main limitations in WCF support in the .NET Compact Framework is that you cannot implement WCF services. In other words, this client platform supports only clientlike actions. You must deploy WCF services on desktop and server systems, and call those services from your Windows Mobile device. Let us look at the details of some other limitations.

10.1.2 WCF in the .NET Compact Framework

As we stated elsewhere in this book, among the strengths of the Microsoft strategy with regard to Windows Mobile (and Windows Embedded CE) is how desktop technologies are leveraged in the context of mobile and embedded platforms. Given the limitations on system resources, the mobile and embedded systems typically provide a subset of what is found in the desktop. The same is true of WCF.

A general rule in a resource-constrained environment is that if there are two or more ways to do something, all but one is eliminated. In the case of WCF support in the .NET Compact Framework, this "eliminate all but one" rule can be seen in the set of predefined systems bindings that are present. While the full .NET Framework supports more than a dozen, the .NET Compact Framework supports one: `basicHttpBinding`.

Calls made to WCF services map directly to the way that regular method calls are made: Calls are synchronous with parameters going in and return values coming out. WCF in the .NET Framework enables enhancements to this basic call/return model, including support for asynchronous calls and one-way calls. But the .NET Compact Framework supports neither of these two enhanced calling patterns.

Download at WoweBook.Com

And finally, fault contracts are not supported in the .NET Compact Framework. Fault contracts enable exception handling to pass over the WCF wire from service to client. This means that WCF services that may have Windows Mobile clients cannot allow unhandled exceptions.

This concludes this brief introduction to the basic concepts of WCF. With these concepts in mind, we now turn our attention to adding WCF support to Windows Mobile applications. In the sections that follow, we show the steps to implementing a WCF service and the steps for accessing such a service from a.NET Compact Framework program running on a Windows Mobile device.

10.2 Creating a WCF Service

Most WCF code will probably be deployed in a client/server topology, although there is nothing about WCF itself to prevent other topologies including token-passing ring, peer-to-peer, or *N*-tier. The server side of the equation is referred to as a **WCF service.** As is common with server-side software, the startup of a server component must occur chronologically before the startup of a client; once started, the server sits and waits for client connections.

The .NET Compact Framework does not support the creation of WCF services. Among other missing elements is the `ServiceHost` class, which enables a process to host a WCF service. For this reason, we build our WCF service using the .NET Framework 3.5. Once built, our WCF service can run on a desktop (or server) version of Microsoft Windows. Any computer suitable for hosting a Windows Mobile development system is also suitable for hosting a WCF service.

10.2.1 Generating the Code

In Visual Studio 2008 you can create a WCF service with a few mouse clicks. The service has both a service contract and a data contract. The resulting code provides a straightforward example to help you learn to define your own WCF contracts. To create a WCF service, follow these steps in Visual Studio 2008.

1. Summon the New Project Wizard (File | New | Project).
2. In the "Project type" window, expand your favorite language (e.g., C#) and click WCF.
3. In the "Installed templates" window, select WCF Service Library.
4. Provide a name for the library and click OK.

Download at WoweBook.Com

This sample provides an easy-to-understand example of both a service contract and a data contract. The auto-generated WCF service library consists of two C# source files, which are named according to whatever library name you provide (I<ProjectName>1.cs and <ProjectName>1.cs). An application configuration file (App.config) is also generated, to enable service configuration by IT personnel in the field.

When you build the service, you create a dynamic link library (DLL). As you probably know, a Windows DLL cannot run by itself. Instead, DLLs must be loaded into a process that has been created by an executable program file. Visual Studio provides such an executable program to enable you to load and test your newly created WCF service: the **WCF Service Host** utility, `WcfSvcHost.exe`. This program can load and test any WCF service DLL because service libraries have a built-in mechanism[3] to enumerate service details. After you create and build a WCF service library, run the WCF Service Host program from within Visual Studio by selecting the Debug | Start Without Debugging menu command.

The WCF Service Host utility calls a companion helper: the **WCF Test Client** program (`WcfTestClient.exe`). The benefit of this test client is that you do not have to write any of your own client code. As long as a metadata exchange (MEX) endpoint is exposed, this utility can query the service contracts and data contracts and provide a platform for testing your new service. Figure 10.1 shows the WCF Test Client calling our WCF service sample.

The WCF Service Host program is a very useful utility. It lets you test a new service without having to create a custom client, thus solving a standard problem of client/server technologies. This class of problems is especially irksome when something breaks and you try to figure out whether the problem is on the client, on the server, or somewhere in between. You can waste a lot of time looking for such bugs, which seem to vanish when you turn on the lights. The WCF Service Host provides a stable, working client so that WCF service developers can get a service working before tackling client-side code.

The WCF Service Host is especially useful in the early stages of WCF service development. When setting up the basic service configuration, it is easy to create invalid, unloadable configurations. We suggest that you plan to use this tool to validate each and every change to interfaces and to your service configuration. The main problem is that few configuration

3. The Metadata Exchange—or MEX—endpoint, described in the MSDN documentation.

Download at WoweBook.Com

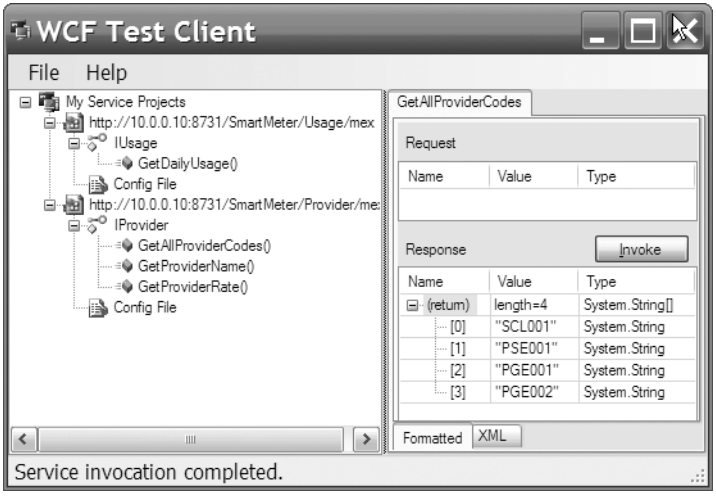


FIGURE 10.1: The WCF Test Client Running the Default WCF Service Library

changes are validated by the build tools, which are focused on language and type consistency rather than runtime issues. It is quite possible to create a WCF service that builds correctly, but because of configuration problems is not able to load. When a load problem does occur, the WCF Service Host utility lets you know about error conditions and other detectable problems. But what about services and interfaces that you expect to see that do not appear? By using this tool to test every single change that is made, you assure yourself that you are never more than one or two Undos away from working code.

10.2.2 Making a Windows Mobile–Compatible WCF Service

When Visual Studio provides its auto-generated WCF service library, it creates code for the full desktop framework. To be compatible with the .NET Compact Framework we must remove any incompatibilities; and it just so happens that one—and only one—incompatibility exists. The auto-generated service uses a binding, `wsHttpBinding`, that is not supported by the .NET Compact Framework. By changing the binding to `basicHttpBinding` in the file App.config (see Listing 10.3), we enable the WCF service to be accessed from a Windows Mobile client.

The App.config file is an application configuration file, with which you may already be familiar. It is an XML file, full of the angle brackets and well-formed markup that XML fans have grown to know and love. Within the outermost `<configuration>` block, the auto-generated sample provides a `<system.serviceModel>` block for WCF service configuration details.

Download at WoweBook.Com

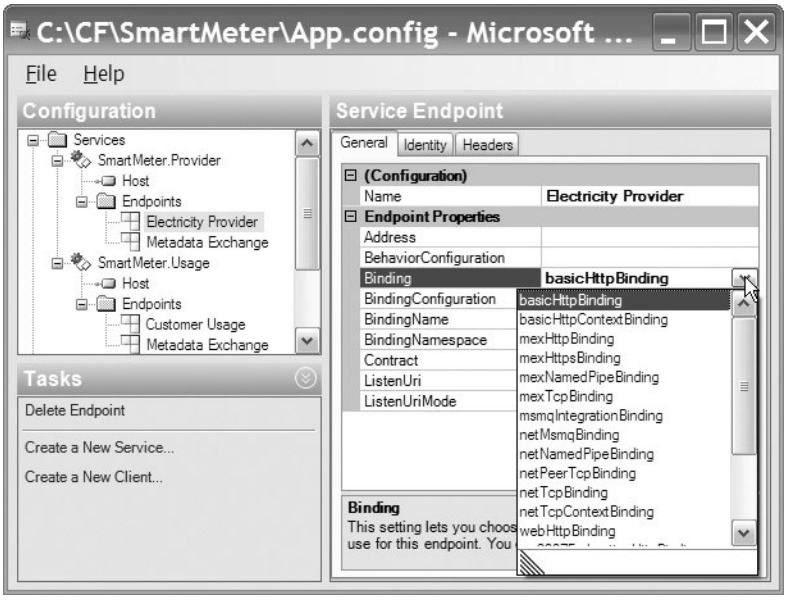


FIGURE 10.2: WCF Service Configuration Editor

There are at least two ways to modify the App.config file. Since the file contains XML, it can be edited with any text editor. In addition, Visual Studio provides a special editor: the **WCF Service Configuration Editor,** which you can run from the Visual Studio Tools menu. Figure 10.2 shows the WCF Service Configuration Editor changing the service binding.

The App.config file is optional because WCF services are entirely configurable in code. If you choose to take this route, you create a problem for yourself because even the smallest change requires source code changes and the resulting need to rebuild and redeploy your binaries. The architects of WCF designed the system to support both code configurations and config file configurations. Config files allow WCF services to be field-tweaked to meet site-specific requirements. In most cases, config file changes take precedence over code configuration changes; the one exception, for security settings, allows developers to hard-code the minimum security settings for a WCF service.

10.2.3 Setting the Host Address

Another change is needed to the auto-generated WCF service. This time, network addressability is the issue and not fixing an incompatibility. The auto-generated App.config file has a host address with http://localhost.

Download at WoweBook.Com

The network stack resolves this address to an IP address of 127.0.0.1, which is the Internet host loopback address.

The loopback address makes a lot of sense for an auto-generated service sample. It is particularly convenient when a developer plans to run both client and service on the same host machine, because no service configuration change is needed. Eventually, of course, the host address must be updated because the main point of WCF is to enable network communication between client and server systems.

The loopback address makes less sense for a Windows Mobile developer because client and host can never run on the same system. Even when testing your code on an emulator, the emulator is physically running on your host but from a network point of view is logically separate. A network-enabled emulator has a network address that is different from the host system on which it is running. To allow another network machine to access the WCF service, the configuration file needs an address other than the loopback address.

The solution is plain: Each device needs a unique network address. For LANs, which are the typical development environment, we use a hard-coded IP address (10.0.0.100) plus subnet mask (255.255.255.0). Internet-deployed services need an address that can be resolved correctly by the DNS servers. (Listing 10.3 shows the auto-generated App.config file; the two items needing to be changed are highlighted in bold text.) See the sidebar Setting Up No-Network Hosts for tips on enabling networks when no LAN or WAN is available (e.g., on a plane or when giving a demo at a trade show or locked-down security sites such as the Microsoft main campus).

Setting Up No-Network Hosts

There are times when you need a network but lack a network. For example, in an airplane (as a passenger, presumably) you want to debug a WCF service. Another scenario is for a demo to an audience, maybe your boss to justify your job; maybe to a user group.[a] As you start, you discover you have no network—what to do? Here is one approach.

Continues

a. Maybe even WE-DIG, the Windows Embedded Developers Interest Group (www.we-dig.org).

Download at WoweBook.Com

Add the Microsoft Loopback Adapter on your desktop system to simulate a network adapter. From the Control Panel, run the Add Hardware Wizard. Do not let the wizard perform any auto-detections since there is nothing to detect. Instead, manually select the hardware. For the manufacturer, choose Microsoft and select Microsoft Loopback Adapter.

Next, set the loopback adapter's IP address. We use an IP address of 10.0.0.10 with a subnet mask of 255.255.255.0 in this chapter's sample WCF service. With the host side ready, you next prepare your device. We often use the emulator when teaching a class or giving a demo.

The best way to connect with the emulator is through a high-speed memory channel known as DMA (short for direct-memory access). If you do not already have it installed, you may need to download and install Microsoft Virtual PC 2007. Among its other features, this product installs the network support that is needed to connect to the emulator.

We establish a connection to the emulator by using ActiveSync on Windows XP (Windows Mobile Device Center—WMDC—on Windows Vista and later). In the Connection Settings dialog, set the value for "Allow connections to one of the following" to DMA. Note that only one device can be connected to ActiveSync, so if you have another device—whether emulator or physical device—you must detach that device before connecting to the emulator.

To establish a connection to the emulator, you use the Device Emulator Manager. You can start this tool from the Tools menu in Visual Studio 2008. From within the Device Emulator Manager, select your target emulator (e.g., Windows Mobile 6 Professional Emulator) and then select Cradle. You should see the status within ActiveSync/WMDC change to show that the emulator is connected.

Listing 10.3: App.config Holding WCF Configuration Details

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <!-- When deploying the service library project, the content
       of the config file must be added to the host's app.config
       file. System.Configuration does not support config files
       for libraries.
       -->
  <system.serviceModel>
    <services>
```

Download at WoweBook.Com


```
  <service name="SmartMeter.Service1"
        behaviorConfiguration="SmartMeter.Service1Behavior">
    <host>
      <baseAddresses>
        <add baseAddress =
             "http://localhost:8731/SmartMeter/Service1/" />
      </baseAddresses>
    </host>
    <!-- Service Endpoints -->
    <!-- Unless fully qualified, address is relative to base
         address supplied above -->
    <endpoint address =""
              binding="wsHttpBinding"
              contract="SmartMeter.IService1">
      <!--
          Upon deployment, the following identity element
          should be removed or replaced to reflect the
          identity under which the deployed service runs.
          If removed, WCF will infer an appropriate identity
          automatically.
          -->
      <identity>
        <dns value="localhost"/>
      </identity>
    </endpoint>
    <!-- Metadata Endpoints -->
    <!-- The Metadata Exchange endpoint is used by the
         service to describe itself to clients. -->
    <!-- This endpoint does not use a secure binding and
         should be secured or removed before deployment -->
    <endpoint address="mex" binding="mexHttpBinding"
              contract="IMetadataExchange"/>
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name="SmartMeter.Service1Behavior">
      <!-- To avoid disclosing metadata information,
           set the value below to false and remove the
           metadata endpoint above before deployment
           -->
      <serviceMetadata httpGetEnabled="True"/>
      <!-- To receive exception details in faults for
           debugging purposes, set the value below to
           true.  Set to false before deployment to
           avoid disclosing exception information
           -->
      <serviceDebug includeExceptionDetailInFaults="False" />
    </behavior>
  </serviceBehaviors>
```

Download at WoweBook.Com


```
    </behaviors>
  </system.serviceModel>
</configuration>
```

The auto-generated WCF service is a good learning tool. It provides a working WCF service library and helps get you started with the various tools that support WCF service development. Eventually, you will need to modify the starting point to create a service that actually does what you want it to do. We have such a sample in the next section.

10.2.4 SmartMeter: A Sample WCF Service

Our sample WCF service is called `SmartMeter`. This service enables customers to download electricity usage data such as might be collected by a smart electrical meter. With today's environmental concerns and related worries around "peak oil," a "smart grid" provides a partial solution by adding a digital information system in parallel to the analog electricity distribution system. Among its other problems, the old electrical system cannot even detect power outages except when customers report those outages by notifying the electric company.

One small part of making the electrical system smarter involves collecting electricity usage data for individual customers and then providing that data back to the customers. Real-time usage data can help customers become aware of how individual actions affect their electricity consumption. Our sample shows how a WCF service could provide historical usage information for individual customers. A service such as this could be deployed on a central server, by an individual electric company. Alternatively, the service could be operated for several electric companies in "the cloud" running a server-side system, such as Windows Azure (Microsoft's cloud operating system).

However such a service is ultimately deployed, we built our service on the assumption that a single WCF service could provide usage information for customers from several electric companies. This assumption gave us the motivation to show how multiple service contracts might be made available from a single WCF service. To understand any WCF service code, it helps to start by studying the configuration file; the one for our sample appears in Listing 10.4.

Listing 10.4: App.config: `SmartMeter` Application Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
```

Download at WoweBook.Com


```
  <compilation debug="true" />
</system.web>
<!-- When deploying the service library project, the content
     of the config file must be added to the host's app.config
     file. System.Configuration does not support config files
     for libraries. -->
<system.serviceModel>
  <services>
    <service name="SmartMeter.Provider"
         behaviorConfiguration="SmartMeter.ProviderBehavior">
      <host>
        <baseAddresses>
          <add baseAddress =
               "http://10.0.0.10:8731/SmartMeter/Provider" />
        </baseAddresses>
      </host>
      <!-- Service Endpoints -->
      <!-- Unless fully qualified, address is relative to
           base address supplied above -->
      <endpoint address =""
                binding="basicHttpBinding"
                contract="SmartMeter.IProvider" />
        <!--
            Upon deployment, the following identity element
            should be removed or replaced.
            -->
        <!-- Metadata Endpoints -->
      <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange"/>
    </service>
      <service name="SmartMeter.Usage"
       behaviorConfiguration="SmartMeter.UsageBehavior">
          <host>
              <baseAddresses>
                  <add baseAddress =
                   "http://10.0.0.10:8731/SmartMeter/Usage" />
              </baseAddresses>
          </host>
          <!-- Service Endpoints -->
          <endpoint address =""
                    binding="basicHttpBinding"
                    contract="SmartMeter.IUsage" />
          <!--
            Upon deployment, the following identity element
            should be removed or replaced.
            -->
          <!-- Metadata Endpoints -->
          <endpoint address="mex" binding="mexHttpBinding"
                    contract="IMetadataExchange"/>
```

Download at WoweBook.Com


```
        </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="SmartMeter.ProviderBehavior">
          <!-- To avoid disclosing metadata information,
          set the value below to false and remove the metadata
          endpoint above before deployment -->
          <serviceMetadata httpGetEnabled="True"/>
          <!-- To receive exception details in faults for
          debugging purposes, set the value below to true.
          Set to false before deployment to avoid disclosing
          exception information -->
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
          <behavior name="SmartMeter.UsageBehavior">
              <!-- To avoid disclosing metadata information, set
              the value below to false and remove the metadata
              endpoint above before deployment -->
              <serviceMetadata httpGetEnabled="True"/>
              <!-- To receive exception details in faults for
               debugging purposes, set the value below to true.
               Set to false before deployment to avoid disclosing
               exception information -->
              <serviceDebug includeExceptionDetailInFaults="False" />
          </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

The `SmartMeter` service described by the `App.config` file in Listing 10.4 provides two WCF services: `SmartMeter.Provider` and `SmartMeter.Usage`. These correspond to the `<namespace>.<class>` for the implementation of the two services. Each has an associated contract, as defined by `IProvider` and `IUsage`.

The `IProvider` interface (see Listing 10.5) defines a contract for electricity provider information, including the provider name and a short rate table. The interface returns electricity rate information for each provider. We greatly simplified the rate table into something that could be hard-coded in a short piece of code. If you spend any time looking at the rate table for electricity providers, you will understand how complex they can be. Different rates are set for home users, businesses, and industrial concerns. In some places, time of usage can affect the rates. To avoid all of these complexities, these tables have a base rate for usage below a given

Download at WoweBook.Com

threshold and a secondary rate once that threshold has been passed. This one feature is common to many rate tables we reviewed.

Listing 10.5: IProvider.cs: Electricity Provider Interface

```
using System;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace SmartMeter
{
    [ServiceContract]
    public interface IProvider
    {
        [OperationContract]
        string[] GetAllProviderCodes();

        [OperationContract]
        string GetProviderName(string strProviderCode);

        [OperationContract]
        Rate GetProviderRate(string strProviderCode,
                             DateTime dtWhen);
    }

    [DataContract]
    public class Rate
    {
        [DataMember] public Single BaseRate;
        [DataMember] public Int32  OveruseThreshold;
        [DataMember] public Single OveruseRate;
    }
}
```

Our implementation for this interface is the `SmartMeter.Provider` class, which appears in Listing 10.6. We include provider information for four electricity providers located on the West Coast of the United States. The six-digit coding scheme shown here is easily adequate to enable a database for all of the public electric utilities in the United States. To scale properly, however, we would probably use a database for the thousands of actual electric companies.

Listing 10.6: Provider.cs: Electricity Provider Implementation

```
using System;
using System.Runtime.Serialization;
using System.ServiceModel;
```

Download at WoweBook.Com


```
namespace SmartMeter
{
    public class Provider : IProvider
    {
        public string[] GetAllProviderCodes()
        {
            string[] astr = new string[4];
            astr[0] = "SCL001";
            astr[1] = "PSE001";
            astr[2] = "PGE001";
            astr[3] = "PGE002";

            return astr;
        }

        public string GetProviderName(string strProviderCode)
        {
            string strRet = "Unknown Provider";

            // Parameter test - trap the empty string
            if (strProviderCode == null)
                return strRet;

            // Select from hard-coded list.
            string strCode = strProviderCode.ToUpper();
            if (strCode == "SCL001")
                strRet = "Seattle City Light";
            if (strCode == "PSE001")
                strRet = "Puget Sound Energy";
            if (strCode == "PGE001")
                strRet = "Pacific Gas & Electric";
            if (strCode == "PGE002")
                strRet = "Portland General Electric";

            return strRet;
        }

        public Rate
        GetProviderRate(string strProviderCode, DateTime dtWhen)
        {
            var r = new Rate();
            switch (strProviderCode)
            {
                case "SCL001": // Seattle City Light
                    r.BaseRate = .0376F;
                    r.OveruseThreshold = 10;
                    r.OveruseRate = 0.0973F;
                    break;
                case "PSE001": // Puget Sound Energy
                    r.BaseRate = .084772F;
```

Download at WoweBook.Com


```
                r.OveruseThreshold = 20;
                r.OveruseRate = 0.102581F;
                break;
            case "PGE001": // Pacific Gas & Electric
                r.BaseRate = .11921F;
                r.OveruseThreshold = 30;
                r.OveruseRate = 0.17102F;
                break;
            case "PGE002": // Portland General Electric
                // Peak rates are 3pm - 8pm
                if (dtWhen.Hour > 15 && dtWhen.Hour < 20)
                {
                    r.BaseRate = .12155F;  // On-Peak
                }
                // Mid-Peak rates are 6am - 3pm, and 8pm - 10pm
                else if (dtWhen.Hour > 6 && dtWhen.Hour < 15 ||
                         dtWhen.Hour > 20 && dtWhen.Hour < 22)
                {
                    r.BaseRate = .06899F;      // Mid-Peak
                }

                // Off-Peak rates are 10pm until 6am
                else
                    r.BaseRate = .04052F;      // Off-Peak

                r.OveruseThreshold = 0;
                r.OveruseRate = r.BaseRate;
                break;
        }

        return r;
    }
  }
}
```

The contract that describes how usage information gets reported is defined in the `IUsage` interface, which appears in Listing 10.7. The world of smart electrical metering does not exist on a large scale today, but it is coming. As it is currently being planned, electricity users will be able to ask electricity providers for details on their power usage. An open question at this time involves the right granularity for the data. In other words, should the smart meters collect one sample every second, one sample every minute, or one sample every hour? Almost anything is an improvement over what came before, when the electric companies needed only one sample every month (or sometimes every two months) to generate a utility bill. As defined in this interface, the customer can request details about usage

Download at WoweBook.Com

information for a given day. The number of samples for that day is not specified. Presumably the day starts at midnight and runs through the following midnight, but another start time could certainly be specified.

Listing 10.7: IUsage.cs: Customer Usage Interface

```
using System;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace SmartMeter
{
    [ServiceContract]
    public interface IUsage
    {
        [OperationContract]
        DailyUsage GetDailyUsage(string strCustID,
                                 DateTime dtUsage);
    }

    [DataContract]
    public class DailyUsage
    {
        [DataMember] public bool bValid;
        [DataMember] public DateTime dtDay;
        [DataMember] public Int32 [] kWh;
    }
}
```

Customer usage data, as provided by the implementation of `SmartMeter.Usage` (see Listing 10.8), consists of 24 numbers with one number providing the average watts used by the customer. Our choice of one sample every hour is not meant to imply that this is the right choice for a utility. It is, instead, the right choice for a sample in a programming book. In the interest of brevity, our usage information is limited to three days: June 1–3, 2010. Our sample does not even validate user credentials or check for a valid customer ID.

Listing 10.8: Usage.cs: Customer Usage Implementation

```
using System;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace SmartMeter
{
    public class Usage : IUsage
```

Download at WoweBook.Com


```
{
    public DailyUsage GetDailyUsage(string strCustID,
                                    DateTime dtUsage)
    {
        var u = new DailyUsage();
        u.bValid = false;

        if (dtUsage == new DateTime(2010,06,01))
        {
            u.kWh = new Int32[24];
            u.bValid = true;
            u.kWh[0] = 925;   u.kWh[1] = 1027;
            u.kWh[2]  = 894;  u.kWh[3]  = 966;
            u.kWh[4]  = 985;  u.kWh[5]  = 1070;
            u.kWh[6]  = 1421; u.kWh[7]  = 1956;
            u.kWh[8]  = 2154; u.kWh[9]  = 2475;
            u.kWh[10] = 1273; u.kWh[11] = 1305;
            u.kWh[12] = 1211; u.kWh[13] = 1478;
            u.kWh[14] = 1320; u.kWh[15] = 1246;
            u.kWh[16] = 1173; u.kWh[17] = 1442;
            u.kWh[18] = 2242; u.kWh[19] = 2481;
            u.kWh[20] = 1984; u.kWh[21] = 1404;
            u.kWh[22] = 1265; u.kWh[23] = 981;
            u.bValid = true;
        }
        else if (dtUsage == new DateTime(2010, 06, 02))
        {
            u.kWh = new Int32[24];
            u.bValid = true;
            u.kWh[0] = 959;   u.kWh[1] = 982;
            u.kWh[2] = 858;   u.kWh[3] = 901;
            u.kWh[4] = 905;   u.kWh[5] = 926;
            u.kWh[6] = 1522;  u.kWh[7] = 1806;
            u.kWh[8] = 1545;  u.kWh[9] = 1401;
            u.kWh[10] = 1161; u.kWh[11] = 1251;
            u.kWh[12] = 1071; u.kWh[13] = 1278;
            u.kWh[14] = 1210; u.kWh[15] = 1204;
            u.kWh[16] = 1137; u.kWh[17] = 1671;
            u.kWh[18] = 1961; u.kWh[19] = 2251;
            u.kWh[20] = 2101; u.kWh[21] = 1541;
            u.kWh[22] = 1306; u.kWh[23] = 1211;
        }
        else if (dtUsage == new DateTime(2010, 06, 03))
        {
            u.kWh = new Int32[24];
            u.bValid = true;
            u.kWh[0] = 1185;  u.kWh[1] = 1075;
            u.kWh[2] = 1211;  u.kWh[3] = 1143;
            u.kWh[4] = 1168;  u.kWh[5] = 1209;
            u.kWh[6] = 1741;  u.kWh[7] = 1948;
```

Download at WoweBook.Com


```
                u.kWh[8] = 1631;  u.kWh[9] = 1539;
                u.kWh[10] = 1271; u.kWh[11] = 1281;
                u.kWh[12] = 1141; u.kWh[13] = 1048;
                u.kWh[14] = 1150; u.kWh[15] = 1174;
                u.kWh[16] = 1217; u.kWh[17] = 1421;
                u.kWh[18] = 1751; u.kWh[19] = 2101;
                u.kWh[20] = 2424; u.kWh[21] = 2536;
                u.kWh[22] = 1763; u.kWh[23] = 1439;
            }
            return u;
        }
    }
}
```

A lot of things can be fine-tuned in this WCF service. Our aim was not to define the ultimate set of electricity provider and electricity usage interfaces but rather to create an example that was somewhat interesting that could be accessed from Windows Mobile. At a minimum, we suggest that you make sure your service can be accessed from the WCF Service Host before you proceed with the creation of a WCF client under Windows Mobile.

10.3 Creating a WCF Client in Windows Mobile

The reason for having WCF becomes readily apparent when you start to work on the client side of the equation. With so much work already done for you, you can query whatever data is needed. This leaves more time to do client-side things that make your users happy.

Four steps are required.

1. Reference required assemblies and namespaces.
2. Create the WCF client proxy and add the proxy files to your Visual Studio project.
3. Create WCF client code.
4. Build and run.

Working through the details of these four steps is the focus of the rest of this chapter.

10.3.1 WCF Client Namespaces and Assemblies

To build a WCF client, you must start with a Visual Studio 2008 project that you create for Windows Mobile. Because WCF is supported only in

Download at WoweBook.Com

Version 3.5 (or later) of the .NET Compact Framework, you must have specified that version when you created your project. If you did not, you must create a new Visual Studio 2008 project and move your code into the project.

Add a reference to the `System.ServiceModel` and `System.Runtime.Serialization` namespaces with code as follows:

```
using System.ServiceModel;
using System.Runtime.Serialization;
```

From the Visual Studio menu, select the Project | Add Reference menu item. If you do not find this menu item, you must open the Solution Explorer and make sure a project item is selected (and that the top-level solution item is not selected). From the Add Reference dialog, add the following two assemblies to your project:

1. `System.ServiceModel.dll`
2. `System.Runtime.Serialization.dll`

10.3.2 Generating the WCF Client Proxy

Programmers who have worked with WCF support in the full .NET Framework may be familiar with the Service Model Metadata Utility (`SvcUtil.exe`). This tool generates the WCF client proxy code. You cannot use this tool for Windows Mobile clients, because it references classes that are not supported in the .NET Compact Framework. You need another version of this tool, created especially for the .NET Compact Framework: `NETCFSvcUtil.exe`.

As of this writing, the special version of the metadata utility does not ship with Visual Studio 2008. Your next task, then, is to locate and install the metadata utility. It is included in the Power Toys for .NET Compact Framework 3.5 and is located on the Microsoft corporate Web site. Use your favorite Internet search engine to locate the install package; download and install the package to your development system. We need only one file: `NETCFSvcUtil.exe`.

This is a character-based tool, which you run inside a command prompt window. The simplest way to run this tool is to specify the URI for each base address. For the two services provided in our sample, we run the following commands:

```
C> NETCFSvcUtil.exe http://10.0.0.10:8731/SmartMeter/Provider
```

Download at WoweBook.Com


```
C> NETCFSvcUtil.exe http://10.0.0.10:8731/SmartMeter/Usage
```

If you get error messages, check for a correct address. Also make sure the service is up and running (running in the WCF Service Host, `WcfSvcHost.exe`, is sufficient). If the target service is not running, you cannot generate the client proxy code using this method.

There is another way to generate the client proxy code. Copy the WCF service library (e.g., `SmartMeter.dll`) into the same directory as the `NETCFSvcUtil.exe` metadata utility. Run the service metadata utility using the name of the service library as the parameter, as shown here:

```
C> NETCFSvcUtil.exe SmartMeter.dll
```

This operation outputs a set of files. One file has an extension of .wsdl, and the other files have an extension of .xsd. You use wildcards to specify both sets of files in a single call to this utility:

```
C> NETCFSvcUtil.exe *.wsdl *.xsd
```

Whichever approach you take, the result is two or more C# source files. With the first approach, the following three files are generated:

- CFClientBase.cs
- Provider.cs
- Usage.cs

Whereas the second approach produces the following two source files

- CFClientBase.cs
- schemas.microsoft.com.2003.10.Serialization.Arrays.cs

Move these files into the directory that contains your Windows Mobile source code. Add all of the generated source files to your Visual Studio project (with the Project | Add Exiting Item menu item).

Of these two approaches, we prefer the first. That is, we prefer calling into the running service. One reason we prefer the first approach is that the resulting source files have friendlier names. The second reason is that when you call into the running service, the resulting code has a constructor that makes it simpler to create a WCF client.

Download at WoweBook.Com

10.3.3 Instantiating a WCF Client

As we discussed earlier in this chapter, you need three things to access a WCF endpoint: an address, a binding, and a contract. If you want, you can create each of these items yourself in your WCF client. In other words, you can take control of individual elements and modify them as needed. For example, a WCF client might know several different addresses for a service and may decide which one to use based on time of day or geographical location (one for Asia, another for Europe, and a third for the United States). Or there might be different security configurations—different certificates, for example—that you need to dynamically alter.

In addition to enabling greater flexibility in configuration, there is one situation when you are required to use the more verbose approach. That occurs for client proxy code that is generated from a local service DLL (the second method described in the preceding section). Here is code that creates each required element of a WCF endpoint—address, binding, and contract—in connecting to the WCF service:

```
var address = new EndpointAddress(
       "http://10.0.0.10:8731/SmartMeter/Provider");
var binding = new BasicHttpBinding();
var Provider = new ProviderClient(binding, address);

var addrUsage = new EndpointAddress(
       "http://10.0.0.10:8731/SmartMeter/Usage");
var bindUsage = new BasicHttpBinding();
var clientUsage = new UsageClient(bindUsage, addrUsage);
```

The auto-generated proxy code supports your doing that. It also supports your creating a connection with a single line of code. That code involves creating an instance of a class that starts with the service name and ends with the word `Client`. The `Provider` proxy provides a class named `ProviderClient`; the `Usage` proxy provides a class named `UsageClient`. In these two lines of code, we create an instance of each:

```
var pc = new ProviderClient();
var uc = new UsageClient();
```

When you develop desktop WCF clients, the proxy classes contain a `Close` method. This is not the case on the proxies that are created for Windows Mobile clients. (You get a reminder about the need to call the `Close` method when you enter the service URI into a browser.) Since your proxies

Download at WoweBook.Com

have no `Close` method, you do not (and cannot) call this method with a Windows Mobile WCF client.

10.3.4 Accessing the WCF Service

After you instantiate a WCF client connection, you access the WCF service through methods in the client instance. The IntelliSense support that is built into Visual Studio 2008 makes it very simple. When you type the name of a client instance (`pc` or `uc` in the last example) you are shown a list of available methods. To make a call to get the name of the electric company that is represented by a provider code of SCL001, we make a call such as the following:

```
string strMyProvider = pc.GetProviderName("SCL001");
```

To query the electricity usage on June 2, 2010 for customer number 12-34-56-78, we make a call such as this:

```
DailyUsage du = uc.GetDailyUsage("12-34-56-78",
                new DateTime(2010, 06, 2));
```

The real beauty of WCF can be seen in the client code. The "real work" that we delegate to the WCF service is made available to our client through a combination of the physical network and the WCF plumbing code. The result is that to the WCF client, the service looks and feels like a local procedure call. To the developer, it is an easy thing.

Even though calls to a WCF service may be as easy as calling local methods, the calls are in fact more expensive. Sending a function call across a network takes time, and that can make your application appear sluggish. Better to have fewer calls that each returns an array of data than to have many calls that each returns a single value. You can get a sense for this when you load and run our WCF client sample, `ReadSmartMeter`.

10.3.5 WCF Client Sample: `ReadSmartMeter`

Our sample WCF client, `ReadSmartMeter`, calls into both the `IProvider` interface and the `IUsage` interfaces that are provided by our WCF service. Figure 10.3 shows our client running and displaying the electricity costs for one day in the life of an electricity user. The total amount of WCF plumbing code in our client is very short: one line of code for instantiating the client interface object. At least, that is what it looks like on the surface. When you explore the three source files that we had to add to our client, however, it is a different story.

Download at WoweBook.Com

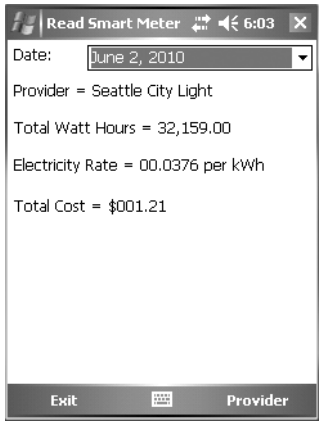


FIGURE 10.3: `ReadSmartMeter` **Displaying Customer Cost and Usage Details**

One source file in our project is CFClientBase.cs. This file contains about six hundred lines of source code. Four classes are defined within this file: `CFClientBase`, `CFContractSerializer`, `CFInvokeInfo`, and `CFFaultException`. This file was generated by the metadata utility, `NETCFSvcUtil.exe`. These four classes are the same for every .NET Compact Framework WFC client.

You might be wondering what the classes are used for. We certainly did not create any of these four types of objects, at least not explicitly. And breakpoints set in these classes never get hit. You might wonder whether they are ever even used. In fact, they are used. The debugger ignores these classes because most of the classes in the source file are tagged with the following attribute:

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
```

If you remove (or, better, comment out) this line of code, you can start to see how these classes enable the plumbing for our WCF client.

To get a complete picture of what is happening, make sure to comment out the debugger-disabling line of code in all the other auto-generated source files (in our sample that would mean these two files: Usage.cs and Provider.cs). You can trace through the creation of the WCF client instances, and also into method calls. You get all this nice plumbing code for free, and you get to enjoy the benefits. Listings 10.9 and 10.10 show the two main source files from our WCF client.

Download at WoweBook.Com

LISTING 10.9: The Main Form in ReadSmartMeter

```
public partial class ReadSmartMeter : Form
{
    ProviderClient g_pc;
    UsageClient g_usage;
    Cost g_CurrentCost = null;

    private string p_ProviderCode;
    public string g_CurrentProvider
    {
        get { return p_ProviderCode;}
        set
        {
            p_ProviderCode = value;
            string strProvider = g_pc.GetProviderName(value);
            labelProvider.Text = "Provider = " + strProvider;
        }
    }

    public ReadSmartMeter()
    {
        InitializeComponent();

        // Create connections to WCF services.
        g_pc = new ProviderClient();
        g_usage = new UsageClient();

        // Fetch list of provider codes.
        string[] astrPCodes = g_pc.GetAllProviderCodes();

        // Populate provider menu with provider codes.
        foreach (string str in astrPCodes)
        {
            MenuItem mi = new MenuItem();
            mi.Text = str;
            mi.Click += new
                System.EventHandler(this.menuProvider_Click);
            menuProvider.MenuItems.Add(mi);
            if (g_CurrentProvider == null)
                g_CurrentProvider = str;
        }
    }

    private void Recalc()
    {
        string str = (string)comboDate.SelectedItem;
        DateTime dt = DateTime.Parse(str);
        g_CurrentCost = new Cost(dt, g_CurrentProvider, "Customer");
```

Download at WoweBook.Com


```
        labelUsage.Text = g_CurrentCost.strUsageSummary;
        labelRate.Text = g_CurrentCost.strRateSummary;
        labelCost.Text = g_CurrentCost.strCostSummary;
    }

    private void menuProvider_Click(object sender, EventArgs e)
    {
        var mi = (MenuItem)sender;
        g_CurrentProvider = mi.Text;
        Recalc();
    }

    private void menuExit_Click(object sender, EventArgs e)
    {
        Close();
    }

    private void
    comboDate_SelectedIndexChanged(object sender, EventArgs e)
    {
        Recalc();
    }

}
```

LISTING 10.10: Cost Calculations Done by the `Cost` Class

```
class Cost
{
    public string strCostSummary;
    public string strUsageSummary;
    public string strRateSummary;
    private DailyUsage m_usage;
    private float [] m_HourlyRate;
    private float m_BaseRate;
    private float [] m_HourlyCost;

    public Cost(DateTime dt, string strProv,
                string Customer)
    {
        // Fetch the customer usage information.
        UsageClient usage = new UsageClient();
        m_usage = usage.GetDailyUsage(Customer, dt);

        // Create a provider client instance.
        ProviderClient provider = new ProviderClient();
        Rate r = null;

        // Initialize various arrays.
        m_HourlyRate = new float[24];
```

Download at WoweBook.Com


```
        m_HourlyCost = new float[24];


        // Loop through 24 hours, calculating
        // hourly and total cost.
        int year, month, day;
        year = dt.Year;
        month = dt.Month;
        day = dt.Day;
        DateTime dtNow  = new DateTime(year, month, day,0,0,0);

        // Fetch base rate
        // Ignoring time-of-day benefits for now
        // Sorry Portland!
        r = provider.GetProviderRate(strProv, dtNow);
        m_BaseRate = r.BaseRate;

        // Set summary values to zero.
        double dTotalCost = 0;
        double dTotalWatts = 0;

        for (int i = 0; i < 24; i++)
        {
            m_HourlyRate[i] = m_BaseRate;
            m_HourlyCost[i] = m_BaseRate * m_usage.Wh[i];

            dTotalCost += (double)m_HourlyCost[i];
            dTotalWatts += (double)m_usage.Wh[i];
            dtNow.AddHours(1);
        }

        strUsageSummary = "Total Watt Hours = " +
            dTotalWatts.ToString("#,000.00");

        strRateSummary = "Electricity Rate = " +
            m_BaseRate.ToString("00.0000") + " per kWh";

        dTotalCost = dTotalCost / 1000.00;
        strCostSummary = "Total Cost = $" +
           dTotalCost.ToString("#,000.00");
    }

}
```

10.4 Conclusion

If you have been a developer for any length of time, you have probably seen a lot of different ways to package and ship code. From static libraries

Download at WoweBook.Com

to DLLs to COM libraries, you might wonder whether we need yet another way to package services.

We hope this coverage of WCF programming for Windows Mobile has convinced you that the architecture of WCF is clean and nicely put together. Whether you spend your time on the server side of WCF or on the client side, you can focus your time and energy on adding real value and letting the WCF infrastructure do what needs to be done to connect clients and servers in a transparent manner.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

11
Creating Graphical Output

This chapter covers the fundamentals of creating graphical output.

THIS CHAPTER DESCRIBES the process of drawing in a .NET Compact Framework program. We prefer using .NET Compact Framework classes whenever possible, but from time to time we may use P/Invoke to drill down to the underlying Win32 API substrate. This chapter and the one that follows discuss the fundamentals for creating graphical output.

11.1 An Introduction to .NET Compact Framework Graphics

In general, programs do not directly touch hardware to draw. Instead, programs use graphical libraries for drawing. Those libraries, in turn, rely on device drivers to touch the underlying hardware. And so, graphical output typically involves these software layers:

- Drawing program
- Graphics function library
- Graphics device driver (display driver or printer driver)

The core graphics library on desktop Windows is the Graphics Device Interface (GDI, `gdi32.dll`). With .NET, Microsoft added a second library (GDI+, `gdiplus.dll`[1]) to add new drawing support. While GDI+ was intended to support managed-code drawing, the library is also accessible

1. GDI+ is a native-mode, unmanaged-code library.

Download at WoweBook.Com

from native-mode application programmers: The library can be called from unmanaged (native-mode) C++ programs. On the desktop, these two graphics libraries—GDI and GDI+—provide the underpinnings for all of the .NET graphics classes. And so, with .NET Framework programs running on the Windows desktop, the architecture of graphical output involves the following elements:

- Managed-code program
- Shared managed-code library (`System.Drawing.dll`)
- GDI+ native-code library (`gdiplus.dll`)
- GDI native-code library (`gdi32.dll`)
- Graphics device driver (display driver or printer driver)

Windows CE supports a select set of GDI drawing functions. There is no library explicitly named "GDI" in Windows CE. Instead, the graphical output functions reside in the `COREDLL.DLL` library. These functions are exactly like their desktop counterparts, so even if there is no library named "GDI" in Windows CE, we refer to these functions as "GDI functions."

Of the desktop's 400 or so GDI functions, only 85 exist in Windows CE. Windows CE has no GDI+ drawing functions. This limits the extent to which Windows CE can support .NET drawing functions.

With these limits, you might wonder whether Windows CE can create interesting graphical output. The answer is a resounding "yes!" The functions that are present were hand-picked as the ones that programs tend to use most. It represents a good set of text, raster, and vector functions so that programs can draw rich text output, display bitmaps in a wide range of formats, and draw complex vector objects.

On both desktop and compact versions, the main .NET graphical library is `System.Drawing.dll`. At 59KB, the .NET Compact Framework library is significantly smaller than its 616KB desktop counterpart. While the desktop library supports five namespaces, the .NET Compact Framework version supports one: `System.Drawing` (plus tiny fragments of two other namespaces). The architecture for drawing from a .NET Compact Framework program is as follows:

- Managed-code program
- Managed-code library (`System.Drawing.dll`)
- GDI functions in the native-code library (`COREDLL.DLL`)
- Graphics device driver (display or printer)

Download at WoweBook.Com

From the arrangement of these software layers, a savvy .NET Compact Framework programmer can divine two interesting points: (1) The managed-code library depends on the built-in GDI drawing functions, and managed-code programs can do the same; and (2) as on the desktop, display screens and printers require a dedicated graphics driver to operate.

11.1.1 Drawing Surfaces

On the Windows desktop, there are four types of drawing surfaces:

1. Display screens
2. Printers
3. Bitmaps
4. Metafiles

When we use the term *drawing surface,* we mean either a physical drawing surface or a logical drawing surface. Two of the four drawing surfaces in the list are physical drawing surfaces, which require dedicated device drivers: display screens and printers. The other two drawing surfaces are logical drawing surfaces: bitmaps and metafiles. These latter two store pictures for eventual output to a device.

Bitmaps and metafiles are similar enough that they share a common base class in the desktop .NET Framework: the `Image`[2] class. Metafiles are not officially supported in Windows CE, however, and so their wrapper, the `Metafile`[3] class, does not exist in the current version of the .NET Compact Framework. Because metafiles might someday be supported in a future version of the .NET Compact Framework, they are worth a brief mention here.

11.1.1.1 *Display Screens*

The display screen plays a central role in all GUI environments because it is on the display screen that a user interacts with the various GUI applications. The real stars of the display screen are the windows after which the operating system gets its name. A window acts as a *virtual console*[4] for interacting with a user. The physical console for a desktop PC consists of a

2. `System.Drawing.Image`.
3. `System.Drawing.Imaging.Metafile`.
4. A term we first heard from Marlin Eller, a member of the GDI team for Windows 1.x.

Download at WoweBook.Com

display screen, a mouse, and a keyboard. On a Pocket PC, the physical console is made up of a display screen, a stylus and a touch-sensitive screen for pointing, and hardware buttons for input (supported, of course, by the on-screen keyboard).

All graphical output on the display screen is directed to one window or another. Enforcement of window boundaries relies on **clipping.** Clipping is the establishment and enforcement of drawing boundaries; a program can draw inside clipping boundaries but not outside them. The simplest clipping boundaries are a rectangle. The area inside a window where a program may draw is referred to as the window's **client area.**

11.1.1.2 *Printers*

Printers are the best-established and most-connected peripherals in the world of computers. While some industry pundits still rant about the soon-to-arrive paperless office, just the opposite has occurred. Demand for printed output has continued to go up, not down. Perhaps the world of computers—with its flashing LCD displays, volatile RAM, and ever-shrinking silicon—makes a person want something that is more real.

Printing from Windows CE–powered devices is still in its infancy, which is a nice way to say that it is not well supported. The official story is that users have not asked for it—in other words, there is not a good enough business case for adding better printing support. Someday that may change, but until then you are on your own when it comes to creating hard-copy output from a Windows Mobile device.

11.1.1.3 *Bitmaps*

Bitmaps provide a way to store a picture. Like its desktop counterparts, Windows CE supports device-independent bitmaps (DIBs) as first-class citizens. In-memory bitmaps can be created of any size[5] and treated like any other drawing surface. After a program has drawn to a bitmap, that image can be put on the display screen.

If you look closely, you can see that Windows CE and the .NET Compact Framework support other raster formats. Supported formats include GIF, PNG, and JPEG. When Visual Studio .NET reads files with these formats (which it uses for inclusion in image lists, for example), it converts the raster data to a bitmap. The same occurs when a PNG or JPEG file is read

5. The amount of available system memory limits the bitmap size.

Download at WoweBook.Com

from the object store into a .NET Compact Framework program. Whatever external format is used for raster data, Windows CE prefers bitmaps. In this chapter, we show how to create a bitmap from a variety of sources and how to draw those bitmaps onto the display screen from a .NET Compact Framework program.

11.1.1.4 *Metafiles*

A second picture-storing mechanism supported by desktop Windows consists of metafiles. A metafile is a record-and-playback mechanism that stores the details of GDI drawing calls. The 32-bit version of Windows metafiles are known as **Enhanced Metafiles (EMFs)**. The following Win32 native metafile functions are exported from `COREDLL.DLL` but are not officially supported in Windows CE, although they might gain official support in some future version of Windows CE:

- `CreateEnhMetaFile`
- `PlayEnhMetaFile`
- `CloseEnhMetaFile`
- `DeleteEnhMetaFile`

11.1.1.5 *Supported Drawing Surfaces*

Of these four types of drawing surfaces, three have official support in Windows CE: display screens, printers, and bitmaps. Only two are supported by the .NET Compact Framework: display screens and bitmaps. Support for bitmaps centers on the `Bitmap`[6] class, which we discuss later in this chapter. We start this discussion of graphical output with the drawing surface that is the focus in all GUI systems: the display screen.

11.1.2 Drawing Function Families

All of the graphical output functions can be organized into one of three drawing function families:

- Text
- Raster
- Vector

6. `System.Drawing.Bitmap`.

Download at WoweBook.Com

Each family has its own set of drawing attributes and its own logic for how its drawing is done. The distinction between these three kinds of output extends from the drawing program into the graphics device drivers. Each family is complex enough for a programmer to spend many years mastering the details and intricacies of each type of drawing. The drawing support is rich enough, however, so that you do not have to be an expert to take advantage of what is offered.

11.1.2.1 *Text Output*

For drawing text, the most important issue involves selection of the font because all text drawing requires a font, and the font choice has the greatest impact on the visual display of text. The only other drawing attribute that affects text drawing is color—both the foreground text and the color of the background area. We touch on text briefly in this chapter but provide more details and examples in Chapter 12, Text and Fonts.

11.1.2.2 *Raster Output*

Raster data involves working with arrays of pixels, sometimes known as bitmaps or image data. Internally, raster data is stored as a DIB. As we discuss in detail later in this chapter, six basic DIB formats are supported in the various versions of Windows: 1, 4, 8, 16, 24, and 32 bits per pixel. Windows CE adds a seventh DIB format to this set: 2 bits per pixel.

Windows CE provides very good support for raster data. You can dynamically create bitmaps, draw on bitmaps, display them for the user to see, and store them on disk. A bitmap, in fact, has the same rights and privileges as the display screen. By this we mean you use the same set of drawing functions both for the screen and for bitmaps. This means you can use bitmaps to achieve interesting effects by first drawing to a bitmap and subsequently copying that image to the display screen. An important difference from desktop versions of Windows is that Windows CE does not support any type of coordinate transformations, and in particular there is no support for the rotation of bitmaps; the .NET Compact Framework inherits these limitations because it relies on native Win32 API functions for all of its graphics support.

11.1.2.3 *Vector Output*

Vector drawing involves drawing geometric figures such as ellipses, rectangles, and polygons. There are, in fact, two sets of drawing functions for

Download at WoweBook.Com

each type of figure. One set draws the border of geometric figures with a *pen.* The other set of functions fills the interiors of geometric figures using a *brush.* You'll find more details on vector drawing later in this chapter.

11.1.3 .NET Compact Framework Graphics

The .NET Framework has six namespaces that support the various graphical output classes. In the .NET Compact Framework, just one namespace has made the cut: `System.Drawing`. This namespace and its various classes are packaged in the `System.Drawing.dll` assembly. For a detailed comparison between the graphics support in the .NET Framework and in the .NET Compact Framework, see the sidebar Comparing Supported Desktop and Smart-Device Drawing.

Comparing Supported Desktop and Smart-Device Drawing

The `System.Drawing` namespace in the .NET Compact Framework holds the primary elements used to draw on a device screen from managed code. The desktop .NET Framework provides five namespaces for creating graphical output, but in the .NET Compact Framework this has been pared back to two: `System.Drawing` and `System.Drawing.Design` (plus some fragments from two other namespaces).

Table 11.1 summarizes the .NET namespaces supported in the desktop .NET Framework, along with details of how these features are supported in the .NET Compact Framework. The `System.Drawing` namespace supports drawing on a device screen. A second namespace, `System.Drawing.Design`, helps when building a custom control. In particular, this namespace contains elements used to support design-time drawing of controls (i.e., drawing controls while they are being laid out inside the Designer). The elements of this namespace reside in the `System.CF.Design.dll` assembly, a different name from the assembly name used for the desktop. The change in the filename makes it clear that this file supports .NET Compact Framework programming.

On the surface, it would be easy to conclude that Microsoft gutted the desktop `System.Drawing.dll` library in creating the .NET Compact Framework edition. For one thing, the desktop version is a whopping 456KB, while the compact version is a scant 38KB. What's more, the

Continues

Download at WoweBook.Com

TABLE 11.1: Desktop .NET Framework Drawing Namespaces in the .NET Compact Framework

Namespace	Description	Support in the .NET Compact Framework
`System.Drawing`	Core drawing objects, data structures, and functions	A minimal set that allows for the drawing of text, raster, and vector objects with no built-in coordinate transformation
`System.Drawing.Design`	Support for the Designer and the various graphics editors of Visual Studio .NET	Support provided by a .NET Compact Framework–specific alternative library named `System.CF.Design.dll`
`System.Drawing.Drawing2D`	Support for advanced graphics features, including blends, line caps, line joins, paths, coordinate transforms, and regions	Not supported in the .NET Compact Framework (except for the `CombineMode` enumeration)
`System.Drawing.Imaging`	Support for storage of pictures in metafiles and bitmaps; bitmap conversion; and management of metadata in image files	Not supported in the .NET Compact Framework (except for the `ImageAttributes` class)
`System.Drawing.Printing`	Rich support for printing and the user interface for printing	Not supported in the .NET Compact Framework
`System.Drawing.Text`	Font management	Not supported in the .NET Compact Framework

desktop version supports 159 classes, while the compact version has a mere 17 classes. A more specific example of the difference between the desktop .NET Framework and the .NET Compact Framework—from a drawing perspective—is best appreciated by examining the `Graphics` class (a member of the `System.Drawing` namespace). The desktop .NET Framework version of this class supports 244 methods and 18 properties; the .NET Compact Framework version supports only 26 methods and 2 properties. By this accounting, it appears that the prognosis of "gutted" is correct. Yet, as any thinking person knows, looks can be deceiving.

Download at WoweBook.Com

To better understand the differences between the desktop .NET Framework and the .NET Compact Framework, we have to dig deeper into the `Graphics` class. To really see the differences between the desktop and compact versions, we must study the overloaded methods. If we do, we see that the desktop .NET Framework provides many overloaded methods for each drawing call, while the .NET Compact Framework provides far fewer. For example, the desktop .NET Framework provides six different ways to call `DrawString` (the text drawing function), while there is only one in the .NET Compact Framework. And there are 30 versions of `DrawImage` (the function for drawing a bitmap) in the desktop .NET Framework but only 4 in the .NET Compact Framework.

We have, in short, fewer ways to draw objects—but in general we can draw most of the same things with the .NET Compact Framework that we can draw on the desktop. This supports a central design goal of Windows CE, which is to be a small, compact operating system. Win32 programmers who have worked in Windows CE will recognize that a similar trimming has been done to define the Windows CE support for the Win32 API. Instead of calling this a "subset," we prefer to take a cue from the music recording industry and use the term *greatest hits.* The .NET Compact Framework implementation of the `System.Drawing` namespace is, we believe, the greatest hits of the desktop `System.Drawing` namespace.

In comparing the desktop .NET Framework to the .NET Compact Framework, an interesting pattern emerges that involves floating-point numbers. In the desktop .NET Framework, most of the overloaded methods take floating-point coordinates. For all of the overloaded versions of the `DrawString` methods, you can use *only* floating-point coordinates. In the .NET Compact Framework, few drawing functions have floating-point parameters—most take either `int32` or a `Rectangle` to specify drawing coordinates. A notable exception is the `DrawString` function, which never takes integer coordinates in the desktop .NET Framework; in the .NET Compact Framework, it is the sole drawing method that accepts floating-point values.

It is worth noting that the underlying drawing functions (both in the operating system and at the device driver level) use exclusively integer coordinates. The reason is more an accident of history than anything else. The Win32 API and its supporting operating systems trace their origins back to the late 1980s, when the majority of systems did not have built-in floating-point hardware. Such support is taken for granted today, which is no doubt why the .NET Framework has such rich support for floating-point values.

Continues

Download at WoweBook.Com

A fundamental part of any graphics software is the coordinate system used to specify the location of objects drawn on a drawing surface. The desktop .NET Framework supports seven distinct drawing coordinate systems in the `GraphicsUnit` enumeration. Among the supported coordinate systems are `Pixel`, `Inch`, and `Millimeter`. While the .NET Compact Framework supports this same enumeration, it has only one member: `Pixel`. This means that when you draw on a device screen, you are limited to using pixel coordinates. One exception involves fonts, whose height is always specified in `Point` units.

This brings up another difference between the desktop .NET Framework and the .NET Compact Framework: available coordinate transformations. The desktop provides a rich set of coordinate transformations—scrolling, scaling, and rotating—through the `Matrix` class and the 3×3 geometric transform provided in the `System.Drawing.Drawing2D` namespace. The .NET Compact Framework, by contrast, supports no coordinate mapping. That means that, on handheld devices, application software that wants to scale, scroll, or rotate must handle the arithmetic itself because neither the .NET Compact Framework nor the underlying operating system provides any coordinate transformation helpers. What the .NET Compact Framework provides, as far as coordinates go, is actually the same thing that the underlying Windows CE system provides: pixels, more pixels, and only pixels.

While it might be lean, the set of drawing services provided in the .NET Compact Framework is surprisingly complete. That is, almost anything you can draw with the desktop .NET Framework can be drawn with the .NET Compact Framework. The key difference between the two implementations is that the desktop provides a far wider array of tools and helpers for drawing. Programmers of the desktop .NET Framework are likely to have little trouble getting comfortable in the .NET Compact Framework, once they get used to the fact that there are far fewer features. But those same programmers are likely to be a bit frustrated when porting desktop .NET Framework code to the .NET Compact Framework world and are likely to have to rewrite and retrofit quite a few of their applications' drawing elements.

11.1.3.1 *The Role of the* `Graphics` *Class*

The most important class for creating graphical output is the `Graphics`[7] class. It is not the only class in the `System.Drawing` namespace, but only

7. `System.Drawing.Graphics`.

Download at WoweBook.Com

the Graphics class has drawing methods. This class holds methods such as DrawString for drawing a string of text, DrawImage for displaying a bitmap onto the display screen,[8] and DrawRectangle for drawing the outline of a rectangle. Here is a list of the other classes in the System.Drawing namespace for the .NET Compact Framework:

- Bitmap
- Brush
- Color
- Font
- FontFamily
- Icon
- Image
- Pen
- Region
- SolidBrush
- SystemColors

These other classes support objects that aid in the creation of graphical output, but none have any methods that actually cause graphical output to appear anywhere. So, while you are going to need these other classes and will use these other classes, they play a secondary role to the primary graphical output class in the .NET Compact Framework: Graphics.

11.1.3.2 *Drawing Support for Text Output*

Table 11.2 summarizes the methods of the Graphics class that support text drawing. The DrawString method draws text, while the MeasureString method calculates the bounding box of a text string. This calculation is needed because graphical output involves putting different types of graphical objects on a sea of pixels. When dealing with a lot of text, it is important to measure the size of each text box to make sure the spacing matches the spacing as defined by the font designer. Failure to use proper spacing creates a poor result. In the worst cases, it makes the output of your program unattractive to users. Even if a user does not immediately notice minor spacing problems, the human eye is very finicky about what text it considers acceptable. Poor spacing makes text harder to read

8. DrawImage can also be used to draw bitmaps onto other bitmaps.

Download at WoweBook.Com

because readers must strain their eyes to read the text. Properly spaced text makes readers—and their eyes—happier than poorly spaced text does.

TABLE 11.2: `System.Drawing.Graphics` Methods for Text Drawing

Method	Comment
`DrawString`	Draws a single line of text using a specified font and text color
`MeasureString`	Calculates the width and height of a specific character string using a specific font

11.1.3.3 *Drawing Support for Raster Output*

Table 11.3 summarizes the methods of the `Graphics` class that draw raster data. We define raster graphics as those functions that operate on an array of pixels. Two of the listed functions copy an icon (`DrawIcon`) or a bitmap (`DrawImage`) to a drawing surface. The other two methods fill a rectangular area with the color of an indicated brush. We discuss the details of creating and drawing with bitmaps later in this chapter.

TABLE 11.3: `System.Drawing.Graphics` Methods for Raster Drawing

Method	Comment
`Clear`	Accepts a color value and uses that value to fill the entire surface of a window or the entire surface of a bitmap
`DrawIcon`	Draws an icon at a specified location. An icon is a raster image created from two rectangular bitmap masks. The `DrawIcon` method draws an icon by applying one of the masks to the drawing surface, using a Boolean `AND` operator, followed by the use of the `XOR` operator to apply the second mask to the drawing surface. The benefit of icons is that they allow portions of an otherwise rectangular image to display the screen behind the icon. The disadvantage of icons is that they are larger than comparable bitmaps and also slower to draw
`DrawImage`	Draws a bitmap onto the display screen or draws a bitmap onto the surface of another bitmap
`FillRegion`	Fills a region with the color specified in a brush. A region is defined as a set of one or more rectangles joined by Boolean operations

Download at WoweBook.Com

11.1.3.4 *Drawing Support for Vector Output*

Table 11.4 summarizes the seven methods in the `Graphics` class that draw vector `Graphics` objects in the .NET Compact Framework. There are substantially fewer supported vector methods than in the desktop .NET Framework. The vector methods whose names start with `Draw` draw lines. The vector methods whose names start with `Fill` fill areas.

TABLE 11.4: `System.Drawing.Graphics` Methods for Vector Drawing

Method	Comment
`DrawEllipse`	Draws the outline of an ellipse using a pen
`DrawLine`	Draws a straight line using a pen
`DrawPolygon`	Draws the outline of a polygon using a pen
`DrawRectangle`	Draws the outline of a rectangle using a pen
`FillEllipse`	Fills the interior of an ellipse using a brush
`FillPolygon`	Fills the interior of a polygon using a brush
`FillRectangle`	Fills the interior of a rectangle using a brush

11.2 Drawing on the Display Screen

The various `System.Drawing` classes in the .NET Compact Framework exist for two reasons. The first and most important reason is for output to the display screen. The second reason, which exists to support the first reason, is to enable drawing to bitmaps, which can later be displayed on the display screen.

Taken together, the various classes in the `System.Drawing` namespace support all three families of graphical output: text, raster, and vector. You can draw text onto the display screen using a variety of sizes and styles of fonts. You can draw with raster functions, including functions that draw icons, functions that draw bitmaps, and functions that fill regions[9] or the entire display screen. The third family of graphical functions, vector functions, supports the drawing of lines, polygons, rectangles, and ellipses on the display screen.

9. A region is a set of rectangles. Regions exist primarily to support clipping but can also be used to define an area into which one can draw.

Download at WoweBook.Com

11.2.1 Accessing a `Graphics` Object

For a .NET Compact Framework program to draw on the display screen, it must have an instance of the `Graphics` class—meaning, of course, a `Graphics` object. A quick visit to the online documentation in the MSDN Library shows two interesting things about the `Graphics` class. First, this class provides no public constructors. Second, this class cannot be inherited by other classes. Thus, you might wonder how to access a `Graphics` object.

Close study of the .NET Compact Framework classes reveals three ways to access a `Graphics` object: two for a display screen and one for bitmaps. Table 11.5 summarizes three methods that are needed to gain access to a `Graphics` object. We include a fourth method in the table, `Dispose`, because you need to call that method to properly dispose of a `Graphics` object in some circumstances.

The display screen is a shared resource. A multitasking, multithreaded operating system such as Windows CE needs to share the display screen and avoid conflicts between programs. For that reason, Windows CE uses the same mechanism used by Windows on the desktop: Drawing on a display screen is allowed only in a window (i.e., in a form or a control).

To draw on the display screen, a program draws in a control. You get access to a `Graphics` object for the display screen, then, through controls. Not just any control class can provide this access, however—only the control classes that derive from the `Control` class can.

TABLE 11.5: .NET Compact Framework Methods for Accessing a `Graphics` Object

Namespace	Class	Method	Comment
`System.Drawing`	`Graphics`	`FromImage`	Creates a `Graphics` object for drawing a bitmap. Clean up by calling `Dispose`
`System.Drawing`	`Graphics`	`Dispose`	Reclaims a `Graphics` object's memory
`System.Windows.Forms`	`Control`	`CreateGraphics`	Creates a `Graphics` object for drawing in a control's client area. Clean up by calling `Dispose`
`System.Windows.Forms`	`Control`	`Paint` event handler	Obtains a `Graphics` object far a `Paint` event. Do not call the `Dispose` method

Download at WoweBook.Com

One way to get a Graphics object for the display screen involves the Paint event. The Paint event plays a very important role in the design of the Windows CE user interface, a topic we discuss later in this chapter. Access to a Graphics object is provided to a Paint event handler method as a property of its PaintEventArgs parameter. Incidentally, when you get a Paint event, you are allowed to use the Graphics object while responding to the event. You are not allowed to hold on to a reference to the Graphics object because the .NET Compact Framework needs to recycle the contents of that Graphics object for other controls to use.[10]

A second way to get a Graphics object is by calling the CreateGraphics method, a method defined in the Control class (and therefore available to classes derived from the Control class). Using the Graphics object returned by this call, your program can draw inside a control's client area. Although the method name suggests that it is creating a Graphics object, this is not what happens. Instead, like the Graphics object that arrives with the Paint event, the Graphics object that is provided by the CreateGraphics method is loaned to you from a supply created and owned by the Windows CE window manager. Therefore, you are required to return this object when you are done by calling the Graphics object's Dispose method. Failure to make this call results in a program hanging.

Calling the Dispose Method for a Graphics Object

There are two ways to get a Graphics object, but you need to call the Dispose method for only one of those ways. You must call the Dispose method for Graphics objects that are returned by the CreateGraphics method. But you do not call Dispose for Graphics objects that are provided as a parameter to the Paint event handler.

The third way to get a Graphics object is by calling the static FromImage method in the Graphics class. On the desktop, the Image class is an abstract class that serves as the base class for the Bitmap and Metafile classes. Because metafiles are not supported in the .NET Compact

10. Ultimately, the window manager reuses the device context contained within the Graphics object.

Download at WoweBook.Com

Framework, the `FromImage` method can return only a `Graphics` object for a bitmap. You can use the resulting `Graphics` object to draw onto a bitmap in the same way that the `Graphics` object described earlier is used to draw on a display screen. We are going to discuss drawing to bitmaps later in this chapter; for now, we explore the subject of drawing in controls.

A main theme for .NET Compact Framework controls is that "inherited does not mean supported." Of the 28 available .NET Compact Framework control classes, only 5 support drawing. To help understand what drawing types are supported, we start by identifying the specific controls onto which you can draw. We then cover the most important control event for drawing, the `Paint` event. We then discuss how non-`Paint` event drawing differs from `Paint` event handling.

11.2.2 Drawing in Controls

In the desktop .NET Framework, a program can draw onto any type of control (including onto forms). This feature is sometimes referred to as **owner-draw support**, a feature first seen in native-code programming for just a few of the Win32 API controls. The implementers of the .NET Framework for the desktop seem to think this feature is something that every control should support. On the desktop, every control supports the owner-draw feature. In other words, you can get a `Graphics` object for every type of control[11] and use that object to draw inside the client area of any control. Owner-draw support is widely available because it allows programmers to inherit from existing control classes and change the behavior and appearance of those classes. This support allows the creation of custom control classes from existing control classes.

Things are different in the .NET Compact Framework, for reasons that are directly attributable to the .NET Compact Framework design goals. The .NET Compact Framework itself was built to be as small as possible and also to allow .NET Compact Framework programs to run with reasonable performance. The result is a set of controls with the following qualities.

11. All desktop control classes that we tested support the `CreateGraphics` method. However, a few desktop control classes do not support the overriding of the `Paint` event: the `ComboBox`, `HScrollbar`, `ListBox`, `ListView`, `ProgressBar`, `StatusBar`, `TabControl`, `TextBox`, `Toolbar`, `TrackBar`, `TreeView`, and `VScrollBar` classes.

Download at WoweBook.Com

- .NET Compact Framework controls rely heavily on the built-in, Win32 API control classes.
- .NET Compact Framework controls do not support every Property, Method, and Event (a.k.a. "PME") inherited from the base `Control`[12] class.

There are two types of drawing: `Paint` event drawing and `CreateGraphics` method drawing. The clearest way to describe the difference is relative to events because of the unique role played by the `Paint` event and its associated `Paint` event handler method. From this perspective, the two types of drawing are better stated as `Paint` event drawing and drawing for other events. We turn our attention now to the subject of the `Paint` event and its role in the Windows CE user interface.

Anywhere, Anytime Control Drawing

An early definition of .NET talked about "anywhere, anytime access to information." Arbitrary boundaries are annoying. It is odd, then, that you cannot draw onto your controls anywhere at any time. But wait—maybe you can.

If you are willing to step outside the managed-code box, you can draw on any control at any time. The .NET Compact Framework team did a great job of giving us a small-footprint set of libraries with very good performance. That is why owner-draw support is so limited—not because of any belief on the part of the .NET Compact Framework team that you should not be allowed to draw inside controls.

Native-code drawing means using GDI calls, each of which requires you to have a handle to a *device context* (hdc). There are two types of device contexts: those used to draw inside windows and those that can draw anywhere on a display screen. To draw in a window, you first must get the window handle (set focus to a control and then call the native `GetFocus` function). Call the native `GetDC` function to retrieve a device context handle, and call the `ReleaseDC` function when you are done.

Continues

12. `System.Windows.Forms.Control`.

Download at WoweBook.Com

The second method for accessing a device context is by using this call: `hdc = CreateDC(NULL, NULL, NULL, NULL)`. The device context that is returned provides access to the entire display screen, not just inside windows. Among its other uses, this type of device context is useful for taking screen shots of the display screen, which can be useful for creating documentation. When done with the device context, be sure to clean up after yourself by calling the `DeleteDC` function.

The `hdc` returned by either of these functions—`GetDC` or `CreateDC`—can be used as a parameter to any GDI drawing function. When done drawing, be sure to provide your own manual garbage collection. In other words, be sure to call the `ReleaseDC` or `DeleteDC` function.

11.2.3 The `Paint` Event

To draw in a window—that is, in a form or in a control—you handle the `Paint`[13] event. The system sends this event to notify a window that the contents of the window need to be redrawn. In the parlance of Windows programmers, a window needs to be redrawn when some portion of its client area becomes *invalid.* To fix an invalid window, a control draws everything that it thinks ought to be displayed in the window.

Generating a `Paint` Event

The purpose of the `Paint` event is to centralize all the drawing for a window in one place. Before we look at more of the details of how to handle the `Paint` event, we need to discuss the circumstances under which a `Paint` event gets generated. A `Paint` event gets generated when the contents of a window become invalid. (We use the term *window* to mean a form or any control derived from the `Control` class.) But what causes a window to become invalid? There are several causes.

When a window is first created, its contents are invalid. When a form first appears, every control on the form is invalid. A `Paint` event is delivered to each control (which, in some cases, is handled by the native-code control that sits behind the managed-code control).

13. This is what Win32 programmers know as a `WM_PAINT` message, which Microsoft Foundation Class (MFC) programmers handle by overriding the `CWnd::OnPaint` method.

Download at WoweBook.Com

A window can also become invalid when it gets hidden. Actually, a hidden window is not invalid; it is just hidden. But when it gets uncovered, the window also becomes invalid. At that moment, a `Paint` event is generated by the system so that the window can repair itself.

A window can also become invalid when it gets scrolled. Every scroll operation causes three possible changes to the contents of a window. Some portion of the contents might disappear, which occurs when something scrolls off the screen. Nothing is required for that portion. Another portion might move because it has been scrolled up (or down, left, or right). Here again, nothing is required. The system moves that portion to the correct location. The third portion is the new content that is now visible to be viewed. This third portion must be drawn in response to a `Paint` event.

Finally, a `Paint` event is triggered when something in the logic of your program recognizes that the graphical display of a window does not match the program's internal state. Perhaps a new file was opened, or the user picked an option to zoom in (or out) of a view. Maybe the network went down (or came up), or the search for something ended.

To generate a `Paint` event for any window, a program calls one of the various versions of the `Invalidate` method for any `Control`-derived class. This method lets you request a `Paint` event for a portion of a window or for the entire window and optionally allows you to request that the background be erased prior to the `Paint` event.

This approach to graphical window drawing is not new to the .NET Compact Framework or even to the .NET environment. All GUI systems have a `Paint` event—from the first Apple Macintosh and the earliest versions of desktop Windows up to the current GUI systems shipping today. A window holds some data and displays a view of that data.

In one sense, drawing is simple: A window draws on itself using the data that it holds. And what happens if the data changes? In that case, a window must declare its contents to be invalid, which causes a `Paint` event to be generated. A control requests a `Paint` event by calling the `Invalidate` method. Two basic problems can be observed with the `Paint` event:

1. Failing to request `Paint` events (this causes cold windows with stale contents)
2. Requesting `Paint` events too often (this causes hot window flickers that annoy users)

Download at WoweBook.Com

These are different problems, but both involve calling the `Invalidate` method the wrong number of times. The first problem arises from not invalidating a window often enough. The second problem arises from invalidating the window too often. A happy medium is needed: invalidating a window the right number of times and at just the right times.

To draw in response to a `Paint` event, a program adds a `Paint` event handler to a control. You can add a `Paint` event handler to any `Control`-derived class. Here is an empty `Paint` event handler:

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // draw
}
```

The second parameter to the `Paint` event handler is an instance of `PaintEventArgs`.[14] A property of this class is a `Graphics` object, which provides the connection that we need to draw in the form. There is more to be said about the `Graphics` object, but first let us look at the case of drawing for events besides the `Paint` event.

11.2.4 Non-`Paint` Event Drawing

A window that contains any graphical output must handle the `Paint` event. Often, the only drawing that a window requires is the drawing for the `Paint` event. This is especially true if the contents of the window are somewhat static. For example, `Label` controls are often used to display text that does not change. For a `Label` control, drawing for the `Paint` event is all that is required. However, windows whose contents must change quickly might need to draw in response to events other than the `Paint` event. A program that displays some type of animation, for example, might draw in response to a `Timer` event. A program that echoes user input might draw in response to keyboard or mouse events.

Figure 11.1 shows the `DrawRectangles` program, a sample vector drawing program. This program draws rectangles in the program's main form, using a pair of (x,y) coordinates. One coordinate pair is collected for the `MouseDown` event, and a second coordinate pair is collected for the `MouseUp` event. As the user moves the mouse (or a stylus on a Pocket PC), the program draws a stretchable rubber rectangle as the mouse/stylus is moved

14. `System.Windows.Forms.PaintEventArgs`.

Download at WoweBook.Com

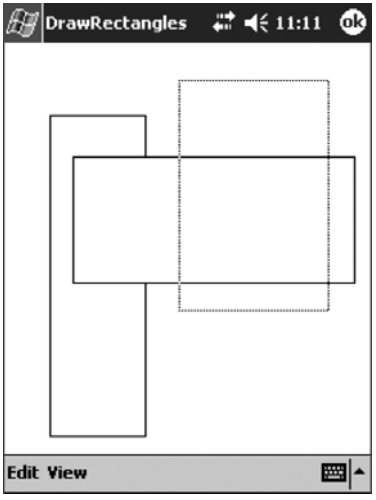


FIGURE 11.1: A Stretchable Rubber Rectangle Created in the `DrawRectangles` Program

from the `MouseDown` point to the `MouseUp` point. The program accumulates rectangles as the user draws them.

The `DrawRectangles` program uses both `Paint` and non-`Paint` event drawing. In response to the `Paint` event, the program draws each of the accumulated rectangles. In response to the `MouseMove` event, the stretchable rectangle is drawn to allow the user to preview the result before committing to a specific location.

The basic template for the code used in non-`Paint` event drawing appears here:

```
Graphics g = CreateGraphics();
// Draw
g.Dispose();
```

This follows a programming pattern familiar to some as the *Windows sandwich.*[15] The top and bottom lines of code make up the two pieces of bread—these are always the same. The filling in between the two slices of bread consists of the drawing, which is accomplished with the drawing methods from the `Graphics` class.

15. This is what Eric Maffei of *MSDN Magazine* used to refer to as the *Windows Hoagie.*

Download at WoweBook.Com

11.3 Raster Graphics

We define raster graphics as those functions that operate on an array of pixels. The simplest raster operation is to fill a rectangle with a single color. On a display screen, this is one of the most common operations. If you study any window on any display screen, you are likely to see that the background is a single color, often white or sometimes gray. You can use three methods in the `Graphics` class to fill a rectangular area:

- `Clear`: fills a window with a specified color
- `FillRectangle`: fills a specified rectangle using a brush
- `FillRegion`: fills a specified region using a brush

The `Clear` method accepts a single parameter, a structure of type `Color`.[16] The other two methods accept a `Brush`[17] object as the parameter that identifies the color to use to fill the area. Before we can fill an area with any of these functions, then, we need to know how to define colors and how to create brushes.

11.3.1 Specifying Colors

The most basic type of drawing attribute is color, yet few drawing methods accept colors directly as parameters. Most drawing methods require other drawing attributes that have a built-in color. For example, for filling areas, the color to use for filling the area is the color that is part of a brush. When drawing lines, the line color is the color that is part of a pen. Brushes are also used to specify text color. So, even though color parameters are not directly used as parameters to methods, they are indirectly specified through a pen or brush.

There are three ways to specify a color in a .NET Compact Framework program:

1. With a system color
2. With a named color
3. With an RGB value

16. `System.Drawing.Color`.
17. `System.Drawing.Brush`.

Download at WoweBook.Com

System colors are a set of colors used to draw the elements of the user interface. The use of system colors helps create consistency between different programs. For example, a given system might be set up with black text on a white background, or it could be set up with the opposite, white text on a black background. System colors are made available as properties of the `SystemColors`[18] class. Available system colors are listed in Table 11.6 in the System Colors subsection.

Named colors provide access to colors that use human-readable names such as `Red`, `Green`, `Blue`, `White`, and `Black`. There are also a large number of colors with less common names such as `SeaShell` and `PeachPuff`. Whether or not you like all the names of the colors you encounter, they provide a way to specify colors that is easy to remember. Color names are made available as static properties of the `Color` structure.

When you specify a color using an RGB value, you specify an amount of red, an amount of green, and an amount of blue. Each is defined with a byte, meaning that values can range from `0` to `255`. It is sometimes helpful to remember that RGB is a video-oriented color scheme often used for display screens and televisions. When the energy for all three colors is `0`, the color you see is black; when the energy for all three colors is at 100% (`255`), the resulting color is white.

11.3.1.1 *System Colors*

System colors let you connect your program's graphical output to current system settings. This allows a program to blend in with the current system configuration. On some platforms, users can change system colors from the system Control Panel (such as on desktop versions of Microsoft Windows). Other platforms, such as the Pocket PC, do not provide the user with an easy way to modify system color settings. A custom embedded smart device could easily be created with a unique system color scheme—say, to match corporate logo colors or to meet unique environmental requirements such as usage in low-light conditions or in sunlight. For all of these cases, the safest approach to selecting text colors involves using system colors.

System colors are available as read-only properties in the `SystemColors` class, the contents of which are summarized in Table 11.7. If you study this table, you may notice that several entries have the word `Text` in the name—such as `ControlText` and `WindowText`. There are, after all, many

18. `System.Drawing.SystemColors`.

Download at WoweBook.Com

uses of text in the user interface. When specifying the color for drawing text, these system colors provide your best choice.

TABLE 11.6: System Colors in the .NET Compact Framework

Color	Description
`ActiveBorder`	Border color of a window when the window is active
`ActiveCaption`	Color of the background in the caption when a window is active
`ActiveCaptionText`	Color of the text in the caption when a window is active
`AppWorkspace`	Color of the unused area in an MDI[a] application
`Control`	Background color for a three-dimensional control
`ControlDark`	Color of the middle of the shadow for a three-dimensional control
`ControlDarkDark`	Color of the darkest shadow for a three-dimensional control
`ControlLight`	Color of the lightest element in a three-dimensional control
`ControlLightLight`	Color of the lightest edge for a three-dimensional control
`ControlText`	Color for drawing text in controls
`Desktop`	Color of the desktop background
`GrayText`	Color for drawing grayed text (e.g., for disabled controls)
`Highlight`	Background color of highlighted areas for menus, `ListBox` controls, and `TextBox` controls
`HighlightText`	Text color for highlighted text
`HotTrack`	Color of hot-tracked items
`InactiveBorder`	Border color of a top-level window when the window is inactive
`InactiveCaption`	Color of the background in the caption when a window is inactive
`InactiveCaptionText`	Color of the text in the caption when a window is inactive
`Info`	Background color of a tool tip

Download at WoweBook.Com

TABLE 11.6: System Colors in the .NET Compact Framework *(Continued)*

Color	Description
InfoText	Text color of a tool tip
Menu	Menu background color
MenuText	Menu text color
ScrollBar	Background color of a scroll bar
Window	Background color of a window
WindowFrame	Color of a window border
WindowText	Color of text in a window

a. MDI (Multiple Document Interface) is not supported on Windows CE or in the .NET Compact Framework.

In some cases, there is a pair of system color names: one with and one without the word `Text` in the name (e.g., `Control` and `ControlText`, `Window` and `WindowText`). One color in the pair defines a system color for text, and the other defines the color of the background. For example, when drawing in a form or dialog box, use the `Window` color for the background and `WindowText` for the text color. When you create a custom control, use the `Control` color for the control's background area and `ControlText` for the color of text drawn in a control. In the following code, the background is filled with the default window background color:

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.Clear(SystemColors.Window);
}
```

11.3.1.2 *Named Colors*

The `System.Drawing.Color` class defines 142 named colors as read-only properties. The names include old favorites such as `Red`, `Green`, `Blue`, `Cyan`, `Magenta`, `Yellow`, `Brown`, and `Black`. It also includes some new colors such as `AliceBlue`, `AntiqueWhite`, `Aqua`, and `Aquamarine`. With names such as `Chocolate`, `Honeydew`, and `PapayaWhip`, you may get hungry just picking a color. The named colors appear in Table 11.7.

Download at WoweBook.Com

TABLE 11.7: Named Colors in the .NET Compact Framework

AliceBlue	AntiqueWhite	Aqua	Aquamarine
Azure	Beige	Bisque	Black
BlanchedAlmond	Blue	BlueViolet	Brown
BurlyWood	CadetBlue	Chartreuse	Chocolate
Coral	CornflowerBlue	Cornsilk	Crimson
Cyan	DarkBlue	DarkCyan	DarkGoldenrod
DarkGray	DarkGreen	DarkKhaki	DarkMagenta
DarkOliveGreen	DarkOrange	DarkOrchid	DarkRed
DarkSalmon	DarkSeaGreen	DarkSlateBlue	DarkSlateGray
DarkTurquoise	DarkViolet	DeepPink	DeepSkyBlue
DimGray	DodgerBlue	Firebrick	FloralWhite
ForestGreen	Fuchsia	Gainsboro	GhostWhite
Gold	Goldenrod	Gray	Green
GreenYellow	Honeydew	HotPink	IndianRed
Indigo	Ivory	Khaki	Lavender
LavenderBlush	LawnGreen	LemonChiffon	LightBlue
LightCoral	LightCyan	LightGolden-rodYellow	LightGray
LightGreen	LightPink	LightSalmon	LightSeaGreen
LightSkyBlue	LightSlateGray	LightSteelBlue	LightYellow
Lime	LimeGreen	Linen	Magenta
Maroon	MediumAquama-rine	MediumBlue	MediumOrchid
MediumPurple	MediumSeaGreen	MediumSlateBlue	MediumSpring-Green
MediumTurquoise	MediumVioletRed	MidnightBlue	MintCream
MistyRose	Moccasin	NavajoWhite	Navy
OldLace	Olive	OliveDrab	Orange
OrangeRed	Orchid	PaleGoldenrod	PaleGreen
PaleTurquoise	PaleVioletRed	PapayaWhip	PeachPuff
Peru	Pink	Plum	PowderBlue

Download at WoweBook.Com

TABLE 11.7: Named Colors in the .NET Compact Framework (Continued)

Purple	Red	RosyBrown	RoyalBlue
SaddleBrown	Salmon	SandyBrown	SeaGreen
SeaShell	Sienna	Silver	SkyBlue
SlateBlue	SlateGray	Snow	SpringGreen
SteelBlue	Tan	Teal	Thistle
Tomato	Transparent	Turquoise	Violet
Wheat	White	WhiteSmoke	Yellow
YellowGreen			

The following code draws in the background of a window with the color `PapayaWhip`:

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.Clear(Color.PapayaWhip);
}
```

11.3.1.3 *Colors from RGB Values*

The third approach that the .NET Compact Framework supports for specifying colors is to specify the three components—red, green, and blue—that make up a color. These three components are packed together into a 32-bit integer with one byte for each. The range for each component is from `0` to `255` (`FF` in hexadecimal). Table 11.8 summarizes color triplet values for common colors.

TABLE 11.8: Color Triplets for Common Colors

Color Name	RGB Triplet (Decimal)	RGB Triplet (Hexadecimal)
Black	(0, 0, 0)	(0, 0, 0)
White	(255, 255, 255)	(0xFF, 0xFF, 0xFF)
Red	(255, 0, 0)	(0xFF, 0, 0)
Green	(0, 255, 0)	(0, 0xFF, 0)

Continues

Download at WoweBook.Com

TABLE 11.8: Color Triplets for Common Colors *(Continued)*

Color Name	RGB Triplet (Decimal)	RGB Triplet (Hexadecimal)
Blue	(0, 0, 255)	(0, 0, 0xFF)
Cyan	(0, 255, 255)	(0, 0xFF, 0xFF)
Magenta	(255, 0, 255)	(0xFF, 0, 0xFF)
Yellow	(255, 255, 0)	(0xFF, 0xFF, 0)
Dark gray	(68, 68, 68)	(0x44, 0x44, 0x44)
Medium gray	(128, 128, 128)	(0x80, 0x80, 0x80)
Light gray	(204, 204, 204)	(0xCC, 0xCC, 0xCC)

Knowing Black from White

We give programmers in our training classes the following tip to help them remember the correct RGB for black (0, 0, 0) and white (255, 255, 255). The RGB color encoding is a light-based scheme, which in a computer CRT is often used to correlate the power to apply to the electron guns in the monitor. Turn the power off, which causes the power to go to zero, and you see black. When the power to the red, green, and blue is turned up all the way, you get white.

In Table 11.8, notice the different shades of gray. By studying the color triplets, you can observe what makes the color gray: equal parts of red, green, and blue.

To create a color from an RGB triplet, use the `Color.FromArgb` method. There are two overloaded versions for this method. We find the following one easier to use:

```
public static Color FromArgb(
   int red,
   int green,
   int blue);
```

When you read the online documentation for this method, you see a reference to a fourth element in a color, the *alpha value.* The .NET Compact Framework does not support this, so you can safely ignore it. In the desktop .NET Framework, the alpha value defines the transparency of a color, where

Download at WoweBook.Com

a value of 0 is entirely transparent and 255 is entirely opaque. In a .NET Compact Framework program, all colors have an alpha value of 255, which means that all colors are 100% opaque.

The following code draws the window background using a light gray color:

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.Clear(Color.FromArgb(204,204,204));
}
```

11.3.2 Creating Brushes

A brush specifies the color and pattern to use for area-filling methods, such as FillRectangle. The .NET Compact Framework does not support patterns in brushes, however, so a brush just specifies the color when filling areas. Brushes also specify the color to use when drawing text. The second parameter to the DrawString method, for example, is a brush.

The desktop .NET Framework supports five different kinds of brushes, including solid brushes, bitmap brushes, and hatch brushes. Windows CE supports solid brushes and bitmap brushes but not hatch brushes. And in the .NET Compact Framework, things are even simpler: Only solid brushes are supported, by the SolidBrush[19] class. This class has a single constructor, which takes a single parameter—Color. The SolidBrush constructor is defined as follows:

```
public SolidBrush(
   Color color);
```

With one constructor, it is natural to assume that there is one way to create a solid brush. But because there are three ways to define a color, there are three ways to create a brush:

- Using the system colors
- Using a named color
- Using an RGB value

The following subsections discuss each of these briefly.

19. System.Drawing.SolidBrush.

Download at WoweBook.Com

11.3.2.1 *Creating Brushes with System Colors*

The following code creates a brush from a system color. This brush is suitable for drawing text within a program's main form or in a dialog box.

```
Brush brText = new SolidBrush(SystemColors.WindowText);
```

The resulting brush provides the same color the operating system uses to draw text. You are not required to select this color, but in doing so you help ensure that your application fits into the color scheme established by the user.

There might be reasons to design your own color scheme. For example, when dealing with financial figures you might display positive numbers in black and negative numbers in red. Or perhaps when displaying certain types of documents you could highlight keywords in different colors, in the same way that Visual Studio .NET highlights language keywords in blue. To handle these situations, you need to specify the brush color with one of the two other color-defining schemes: using either named colors or RGB colors.

11.3.2.2 *Creating Brushes with Named Colors*

Here are examples of creating brushes using named colors:

```
Brush brRed = new SolidBrush(Color.Red);
Brush brPaleGreen = new SolidBrush(Color.PaleGreen);
Brush brLightBlue = new SolidBrush(Color.LightBlue);
```

You might wonder where these color names come from. Some—such as `Red`—are, of course, names for common colors. But when you read through the list of names, you see colors such as `AliceBlue`, `GhostWhite`, and `WhiteSmoke`. The colors are sometimes called *HTML color names* because the more exotic names were first supported as color names in HTML by various browsers. Officially, however, HTML 4.0 includes only 16 color names, not the 140+ names defined in the `Color` structure.

11.3.2.3 *Creating Brushes with RGB Values*

To create a brush using an RGB value, call the `FromArgb` method in the `Color` class and pass the return value to the `SolidBrush` constructor. This method accepts three integer parameters, one each for red, green, and blue. Here is how to create three brushes from RGB triplets:

```
Brush brRed = new SolidBrush(Color.FromArgb(255, 0, 0));
Brush brGreen = new SolidBrush(Color.FromArgb(0, 255, 0));
Brush brBlue = new SolidBrush(Color.FromArgb(0, 0, 255));
```

Download at WoweBook.Com

11.3.3 Creating Bitmaps

A bitmap is a two-dimensional array of pixels with a fixed height and a fixed width. Bitmaps have many uses. One is to hold scanned images, such as a company logo. Photographs are stored as bitmaps, commonly in the highly compressed format of JPEG[20] files. Bitmaps can be used to create interesting effects on a display screen, such as smooth scrolling and seamless animation.

Bitmaps are often used to store complex images that a program can easily draw in one or more locations by making a single method call. As useful as this approach can be, it is important to always remember that bitmaps require a lot of room—both in memory and in the file system. If you plan to include any bitmaps with your program, give some thought to the format of those bitmaps. We address this issue later in this chapter.

Bitmaps are sometimes referred to as *off-screen bitmaps* because of the important role bitmaps have historically played in supporting display screen graphics. The Bitmaps on the Desktop sidebar discusses how bitmaps are used on desktop versions of Windows to support various user interface objects. That same support does not exist in Windows CE because of memory constraints. But bitmaps are still available to Windows CE programs for all of their other uses.

Bitmaps on the Desktop

On desktop versions of Windows, bitmaps support the quick appearance and disappearance of menus, dialog boxes, and various other user interface elements. For example, before a menu appears on the screen, a snapshot is taken of the area to be covered by the menu. When the menu disappears, the bitmap is used to redraw the affected part of the screen. This technique helps make the elements of the user interface appear and disappear very quickly. This technique is not employed in Windows CE because of the tight memory restrictions of mobile and embedded systems. But your program could use bitmaps in other ways to support the display of your program's user interface.

20. *JPEG* stands for Joint Photographic Experts Group, the name of the original committee that created the standard. For details on this compression standard, visit www.jpeg.org.

Download at WoweBook.Com

In our programming classes, we observe that programmers often get confused when first starting to work with bitmaps. The confusion seems to come from not grasping that bitmaps are inherently off-screen. Or it may arise from an understanding that display screens are supported by memory-mapped video devices and that the memory occupied by a bitmap must somehow be related to the memory used by the display adapter. After creating a bitmap and drawing into a bitmap, some programmers expect that bitmaps are going to appear somewhere on the screen. That does not happen, however, because bitmaps appear on a display screen only when your program explicitly causes them to appear.

11.3.3.1 *Bitmaps: Drawing Surface or Drawing Object?*

Bitmaps play two roles in every graphics library built for Microsoft Windows: (1) as drawing surfaces and (2) as drawing objects used to draw onto other surfaces. This is another reason why bitmaps can at first seem confusing for some programmers.

A bitmap is a drawing surface like other drawing surfaces. We say this because a program can obtain a `Graphics` object for a bitmap and then use the methods of that object to draw onto the surface of the bitmap. All of the drawing methods in the `Graphics` object are supported for bitmaps, including text, raster, and vector drawing methods.

The second role played by bitmaps is that of a drawing object. Like other drawing objects, such as pens, brushes, and fonts, a bitmap holds a pattern that can be applied to a drawing surface. Each drawing object has its particular uses, and each produces a different effect, as determined by the various drawing methods. The .NET Compact Framework supports four overloads for the bitmap drawing method, which is named `DrawImage`.

An example might clarify what we mean by each of these two roles. Using a `Graphics` object, a program can draw onto a bitmap by calling drawing methods. One such drawing method is `DrawImage`, which draws a bitmap onto a drawing surface. A program can call the `DrawImage` method to draw one bitmap (the drawing object) onto the surface of another bitmap (the drawing surface).

To push the example one step further, a bitmap can be both the drawing surface and the drawing object. You could do this by using the `DrawImage` method to draw onto a bitmap while using the bitmap itself as the image source. This may sound like a snake eating its own tail, a seemingly

Download at WoweBook.Com

impossible operation. It is possible, however, because it involves copying a rectangular array of pixels from one part of a bitmap to another part. The work required for this type of bitmap handling is well understood and has been part of Windows display drivers for more than a decade. The bitmap drawing code in Windows CE can easily—and correctly—handle cases where, for example, source and destination rectangles overlap. This describes what happens, for example, when a user picks up and moves a window.

11.3.3.2 *The* `Bitmap` *Class*

The .NET Compact Framework supports in-memory bitmaps with the `Bitmap`[21] class. This class is derived from the `Image` class, which is a common base class for the `Bitmap` class and the `Metafile` class. As we mentioned earlier in this chapter, metafiles are not supported in the .NET Compact Framework. But because the .NET Compact Framework maintains consistency with the desktop .NET Framework, our bitmap drawing method is called `DrawImage` (instead of, for example, `DrawBitmap`). On the desktop, where metafiles are supported, the `DrawImage` method draws both bitmaps and metafiles.

The dimensions of a bitmap are available through two properties of the `Bitmap` object: `Height` and `Width`. The dimensions are also available through the `Size` property, which provides a convenient package for height and width. These are read-only properties because an image cannot change size once it has been created.

On the desktop, the `Bitmap` class supports 12 constructors, while in the .NET Compact Framework there are only 4 constructors. You can create a bitmap in these ways:

- By opening an image file
- By reading an image from a stream
- By starting from an existing bitmap
- By specifying the width and height for an empty bitmap

Table 11.9 maps the constructors to six common sources you might use to create a bitmap.

21. `System.Drawing.Bitmap`.

Download at WoweBook.Com

TABLE 11.9: Sources for Bitmaps and Associated Bitmap Class Constructors

Source	Constructor Parameters	Comments
An external image file	(String)	Provides the path to the bitmap in the file system
A portion of a file	(Stream)	Uses the `FileStream` class to open the file and move the seek position to the first byte of the bitmap
Data in memory	(Stream)	Uses the `MemoryStream` class to assemble the bitmap bits as a byte array
A resource	(Stream)	Reads bitmap data from a managed resource created as an untyped manifest resource
An existing bitmap	(Image)	Copies an existing bitmap
An empty bitmap	(int,int)	Specifies the width and height of the empty bitmap

11.3.3.3 *Creating an Empty Bitmap*

One way to create a bitmap is to specify the desired dimensions of the bitmap to the following constructor in the `Bitmap` class:

```
public Bitmap(
   int width,
   int height);
```

This constructor creates a bitmap in program memory with the specified size. This is the quickest and easiest way to create a bitmap, but the empty (i.e., all-black) image means you must draw into the bitmap before displaying its contents. You might call this a **scratch space** or **double-buffer bitmap** because it provides an off-screen drawing surface for doodling, just like scratch paper. The term *double-buffer* refers to a technique of creating smooth graphics effects by doing complex drawing off-screen and sending the resulting output to the display screen with a single, fast drawing operation. Let's use a bitmap created with this constructor.

After creating the bitmap itself, a program typically obtains a `Graphics` object for the bitmap. As we mentioned earlier in this chapter, we need a `Graphics` object for any type of drawing. We obtain a `Graphics` object for the bitmap by calling the `FromImage` method of the `Bitmap` class. Before

Download at WoweBook.Com

drawing anything else in the bitmap, it makes sense to first erase the bitmap's background.

We need to think about cleanup. This is a subject that can often be ignored in managed code but not when working with resource-intensive objects such as bitmaps. So, when done working with a bitmap, your program must use the `Dispose` method to clean up two objects: the bitmap itself and the `Graphics` object. The code in Listing 11.1 shows the whole life cycle of our created bitmap: The code creates a bitmap, erases the bitmap's background, draws the bitmap to the display screen, and then cleans up the two objects that were created.

Listing 11.1: Dynamic Bitmap Creation

```
private void
CreateAndDraw(int x, int y)
{
   // Create a bitmap and a Graphics object for the bitmap.
   Bitmap bmpNew = new Bitmap(100,100);
   Graphics gbmp = Graphics.FromImage(bmpNew);

   // Clear the bitmap background.
   gbmp.Clear(Color.LightGray);

   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();

   // Copy the bitmap to the window at (x,y) location.
   g.DrawImage(bmpNew, x, y);

   // Clean up when we are done.
   g.Dispose();
   gbmp.Dispose();
   bmpNew.Dispose();
}
```

11.3.3.4 *Creating a Bitmap from an External File*

Another way to create a bitmap is by specifying the path to an image file. This is accomplished with a constructor that accepts a single parameter, a string with the path to the candidate file. This second `Bitmap` class constructor is defined as follows:

```
public Bitmap(
   string filename);
```

Download at WoweBook.Com

This method has two important requirements. One is that there must be enough memory to accommodate the bitmap. If there is not, the call fails. The second requirement is that the specified file must have an image in a format that the constructor understands. We have been able to create bitmaps from the following file types:

- Bitmap files (.bmp) with 1, 4, 8, or 24 bits per pixel
- JPEG (.jpg) files
- GIF (.gif) files
- PNG (.png) files

Among the unsupported graphics file formats are TIFF (.tif) files.

This constructor throws an exception if the filename provided is not a recognized format or if it encounters other problems when attempting to open the file or create the bitmap. For that reason, it makes sense to wrap this constructor in a `try…catch` block. Listing 11.2 provides an example of calling this constructor, with a filename provided by the user in a File Open dialog box.

Listing 11.2: Creating a Bitmap with a Filename

```
try
{
   bmpNew = new Bitmap(strFileName);
}
catch
{
   MessageBox.Show("Cannot create bitmap from " +
      "File: " + strFileName);
}
```

11.3.3.5 *Creating a Bitmap from a Resource*

When a program needs a bitmap for its normal operation, it makes sense to package the bitmap as a resource. Resources are read-only data objects that are bound into a program's executable file[22] at program build time. The benefit of binding a bitmap to an executable file is that it is always available and cannot be accidentally deleted by a user.

Resources have been a part of Windows programming from the very first version of Windows. In a native-mode program, resources are used

22. Resources can be bound into any executable module, meaning any program (.exe) or library (.dll) file.

Download at WoweBook.Com

for bitmaps and icons and also to hold the definitions of dialog boxes and menus. In managed-code programs, resources are still used for bitmaps and icons, although some program elements—including dialog boxes and menus—are not defined in a resource but instead are defined using code in the `InitializeComponent` method by the Designer.

Where Do Resources Live?

Because memory is scarce on a Windows CE–powered device, it helps to know when and how memory gets used. When a resource gets added to a module, the resource occupies space in the module's file but uses no program memory until the resource is explicitly opened and used. This is true for both native resources and managed resources.

While resources are used in both native code and managed code, native resources can be used only from native-mode code, and managed resources can be used only from managed code. The only exception is the program icon for a managed-code program, which is defined as a native icon. In managed code, there are two types of resources: **typed resources** and **untyped resources.**

Typed Resources We like to use typed resources to hold literal strings, which aid in the localization of programs. To access typed resources, a program creates an instance of a `ResourceManager`[23] class and then makes calls to methods such as `GetObject` and `GetString`.

Typed resources are defined using XML in files that have an extension of .resx. In a typed resource, an XML attribute provides type information, as shown in this example:

```
<data name="dlgFileOpen.Location" type="System.Drawing.Point,
 System.CF.Drawing, Version=7.0.5000.0, Culture=neutral,
 PublicKeyToken=b03f5f7f11d50a3a">
     <value>125, 17
     </value>
</data>
```

23. `System.Resources.ResourceManager`.

Download at WoweBook.Com

The Designer makes extensive use of typed resources. For each form created in the Designer, an associated file is used to store a variety of details about the form. The Visual Studio .NET Solution Explorer does not normally display resource files, but you can make them appear by clicking the Show All Files icon.

For example, bitmaps in the image collection of an `ImageList` control on a form are stored as typed resources in the typed resource file of the form that contains the control. The bitmaps themselves are serialized into the XML and are not stored in their original, binary form. While a programmer could convert bitmap files into XML resources, we prefer to avoid this extra step and use untyped resources when we add bitmap resources to a project.

Untyped Resources Untyped resources are also known as *manifest resources* because they are made available through an assembly's manifest (or table of contents). As the name implies, an untyped resource contains no type information and is made available as a raw stream of bytes. It does have a name, however, that is created by combining the default project namespace with the filename that contained the original resource data. You must know this name because you use the name to retrieve the resource. If you have trouble figuring out the resource name, the `ildasm.exe` utility can help. Open the program file and then click on the manifest. Listing 11.3 shows three bitmap resource names in a fragment from the manifest for the `ShowBitmap` sample program presented later in this chapter.

LISTING 11.3: Three Bitmap Resource Names from the `ShowBitmap` Manifest

```
.mresource public ShowBitmap.SPADE.BMP
{
}
.mresource public ShowBitmap.CUP.BMP
{
}
.mresource public ShowBitmap.HEART.BMP
{
}
```

Visual Studio .NET creates an untyped resource from an external file when you add the file to a project and assign a build action of Embedded Resource. The default build action for bitmap files, Content, allows the bitmap file to be downloaded with a program but as a separate file and not as

Download at WoweBook.Com

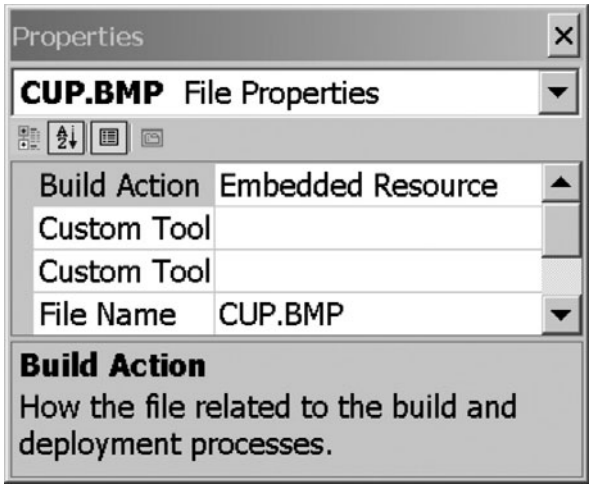


FIGURE 11.2: The Settings to Turn CUP.BMP into an Embedded Bitmap Resource

an embedded resource. Figure 11.2 shows the Visual Studio .NET settings to turn the file CUP.BMP into an embedded bitmap resource. The name of the resource is ShowBitmap.CUP.BMP, which we need to know to access the resource from our code.

You can access an embedded resource by calling a method in the `Assembly` class, named `GetManifestResourceStream`.[24] As suggested by the method name, the return value is a `Stream` object; more precisely, you are provided a `MemoryStream`[25] object. You can use all of the elements associated with a `Stream`-derived class (including the ability to query the resource length, which is the same as the resource input file) to seek a location in the stream and to read bytes (the `CanSeek` and `CanRead` properties are both set to `true`). In keeping with the read-only nature of Windows resources, you cannot write to a resource stream[26] (`CanWrite` returns `false`).

The code fragment that is shown in Listing 11.4 shows two methods from the `ShowBitmap` sample program. These methods are helper routines to handle the initialization and cleanup of resource-based bitmaps. The `LoadBitmapResource` method creates a bitmap from a resource; the `DisposeBitmap` method provides the cleanup.

24. `System.Reflection.Assembly.GetManifestResourceStream`.
25. `System.IO.MemoryStream`.
26. In contrast to an Apple Macintosh resource fork, which supports both read and write operations.

Download at WoweBook.Com

What Can Go into an Untyped Resource?

This chapter provides an example of putting a bitmap into an untyped resource. But this is not the only type of resource you can create. You can put any custom data into untyped resources, which can then be used to access the data at runtime. When you request an untyped resource, you are provided with a `Stream` object that you can use as you wish. You might, for example, read a resource into an array of bytes and then parse those bytes in whatever way your application needs. Such resources can be any read-only data that your program needs: tax tables, sports scores, or—as we show in the `ShowBitmap` sample program—a set of bitmaps.

A benefit of using custom resources is that we have access to data we need at runtime. But when we are not using that data, it does not occupy scarce program memory. This makes custom resources a useful tool in our toolkit for building memory-wise programs.

Listing 11.4: Creating Bitmaps from Untyped Manifest Resources

```
private Bitmap LoadBitmapResource(string strName)
{
   Assembly assembly = Assembly.GetExecutingAssembly();
   string strRes = "ShowBitmap." + strName;
   Stream stream = assembly.GetManifestResourceStream(strRes);
   Bitmap bmp = null;
   try
   {
      bmp = new Bitmap(stream);
   }
   catch { }
   stream.Close();

   return bmp;
}
private void DisposeBitmap(ref Bitmap bmp)
{
   if (bmp != null)
   {
      bmp.Dispose();
   }
   bmp = null;
}
```

Download at WoweBook.Com

The `LoadBitmapResource` method creates a bitmap by opening a resource stream and uses data read from that stream to create a bitmap. This method gets a reference to the program's assembly by calling a static method in the `Assembly` class named `GetExecutingAssembly`. After creating a bitmap, the stream can be closed. Once a bitmap has been created, it is self-contained and needs no external data. That is why we can close the stream once the `Bitmap` object has been created.

The `DisposeBitmap` method deletes the bitmap to free up its associated memory. It does this by calling the `Dispose` method for a `Bitmap` object. There are only a few situations in which it is mandatory to call the `Dispose` method.[27] Sometimes, however, it is still a good idea—even if it is not, strictly speaking, required. Bitmaps can be large, so we suggest you consider explicitly deleting bitmaps, as we have done in our sample. Among the factors to consider are the size of your bitmaps and the number of bitmaps. We suggest that you explicitly delete bitmaps when you have either many small bitmaps or a few large bitmaps.

We call our two methods using code such as the following:

```
private void
mitemResourceCup_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("CUP.BMP");
   Invalidate();
}
```

After cleaning up the old bitmap, we create a new bitmap and request a `Paint` event by calling the `Invalidate` method.

Next, we discuss image file size, and how to save memory by changing the format you use for your images.

11.3.3.6 *Image File Sizes*

Bitmaps can occupy a lot of memory, which can create problems in a memory-scarce environment such as Windows CE. When placing bitmaps in resources, we recommend that you test different formats and use the smallest one. To provide a starting point, we conducted some tests with three 100×100-pixel images stored in different formats. Table 11.10 summarizes our results, which provide the size in bytes for each image file.

27. The only situation requiring a call to the `Dispose` method is to release a `Graphics` object obtained in a control by calling the `CreateGraphics` method.

Download at WoweBook.Com

TABLE 11.10: Size Comparison for Three 100 × 100 Images in Various Image File Formats

Format	Bits per Pixel	Size of Single-Color Image (Bytes)	Size of Multicolor Image with Regular Data (Bytes)	Size of Multicolor Image with Irregular Data (Bytes)
Monochrome DIB	1	1,662	1,662	1,662
16-color DIB	4	5,318	5,318	5,318
256-color DIB	8	11,078	11,078	11,078
True-color DIB	24	30,054	30,054	30,054
GIF	8	964	3,102	7,493
PNG	8	999	616	5,973
JPEG	24	823	3,642	5,024

Four formats are uncompressed and three are compressed. The first four entries in the table are for DIB files. This well-known format is thoroughly documented in the MSDN Library and is the format that Visual Studio .NET provides for creating bitmap images. Notice that the size of these images is the same for a given number of bits per pixel. This reflects the fact that DIB files are uncompressed.

The last three formats are the compressed formats: GIF, PNG, and JPEG. To make sense of these formats, we must discuss the contents of the three images. The single-color image was a solid black rectangle. Each of the three compressed formats easily beat any of the uncompressed formats for the single-color image. The reason is that compressed formats look for a pattern and use that information to store details of the pattern. A single color is a pretty easy pattern to recognize and compress.

The second column, the multicolor image with regular data, shows the results for an image created with a solid background and vertical stripes. We used vertical stripes in an attempt to thwart the compression because run-length encoding of horizontal scan lines is an obvious type of compression. We were surprised (and pleased) to find that PNG compression was able to see through the fog we so carefully created—it created the smallest image in the table.

The third column, the multicolor image with irregular data, shows the sizes for images created with very random data. For this test, we copied

Download at WoweBook.Com

text (.NET Compact Framework source code) into an image file. (We never want our work to be called "random," but we wanted an irregular image to push the envelope for the three compression formats.) The result was more like a photograph than any of the other images, which is why JPEG—the compression scheme created for photographs—was able to provide the best compression. It provided the smallest file size with the least loss of information (the monochrome image was smaller, but the image was lost).

To summarize, the two compression schemes that created the smallest image files were PNG (for regular data) and JPEG (for irregular data). One problem is that Visual Studio .NET does not support either of these formats. But Microsoft Paint (`mspaint.exe`) supports both, so we recommend that you make sure your images have been compressed as much as possible prior to embedding your images as resources.

11.3.4 Drawing Bitmaps

The `Graphics` class supports four overloaded versions of the bitmap drawing method, `DrawImage`. These alternatives support the following types of bitmap drawing:

- Drawing the entire bitmap at the original image size
- Drawing part of a bitmap at the original image size
- Drawing part of a bitmap with a change to the image size
- Drawing part of a bitmap with a change to the image size and with transparency

We discuss these four methods in the sections that follow.

11.3.4.1 *Drawing the Entire Bitmap at the Original Image Size*

The simplest version of the `DrawImage` method copies an entire bitmap onto a device surface with no change in the image size, as shown here:

```
public void DrawImage(
   Image image,
   int x,
   int y);
```

Listing 11.5 shows an example of calling this method in a `Paint` event handler.

Download at WoweBook.Com

Listing 11.5: Drawing an Entire Bitmap at the Original Size

```
private void
FormMain_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   int x = 10;
   int y = 10;

   g.DrawImage(bmpDraw, x, y);
}
```

11.3.4.2 *Drawing Part of a Bitmap at the Original Image Size*

While we sometimes want to draw an entire bitmap, there are also times when we want to see only a portion of a bitmap. The second version of the `DrawImage` method provides the support we need to do just that, as shown here:

```
public void DrawImage(
   Image image,
   int x,
   int y,
   Rectangle srcRect,
   GraphicsUnit srcUnit);
```

This version of the `DrawImage` method has five parameters, while the earlier one has only three. One of the extra parameters is useful, and the second is not so useful. The fourth parameter, `srcRect`, is the useful one, which identifies the rectangular area in the source bitmap that we wish to copy to the destination surface.

The fifth parameter, `srcUnit`, can be set to only one valid value in the .NET Compact Framework: `GraphicsUnit.Pixel`. On the desktop, the presence of this parameter gives the caller the freedom to select a convenient unit of measure for the source rectangle (e.g., inches or millimeters). But the .NET Compact Framework supports only pixel drawing units, which is why this parameter is not so useful in the context of a .NET Compact Framework program. The `srcUnit` parameter is present because of the high level of compatibility between the desktop .NET Framework and the .NET Compact Framework. As such, it represents a small price to pay for the convenience of allowing smart-device code to have binary compatibility with the desktop runtime.

Download at WoweBook.Com

11.3.4.3 *Drawing Part of a Bitmap with a Change to the Image Size*

The third overloaded version of the `DrawImage` method allows a portion of a bitmap to be selected for drawing, and that portion can be stretched (or shrunk) to match a specified size on the destination surface. Of course, nothing requires the image to change size: If the width and height of the destination rectangle are the same as the width and height of the source rectangle, no size change occurs. This version of the `DrawImage` method is defined as shown here:

```
public void DrawImage(
   Image image,
   Rectangle destRect,
   Rectangle srcRect,
   GraphicsUnit srcUnit);
```

11.3.4.4 *Drawing Part of a Bitmap with a Change to the Image Size and with Transparency*

The final version of the `DrawImage` method adds a new feature to the drawing of bitmaps. It enables transparency while drawing a bitmap. In some ways, this feature breaks our definition of raster graphics. You might recall that we refer to raster graphics as those operations that operate on arrays of pixels. Implicit in this definition is that all operations are rectangular.

The ability to draw a raster operation and touch only a nonrectangular set of pixels on a drawing surface is, therefore, something of a heresy (like having nonrectangular windows on a display screen or a late-night coding session without ordering large quantities of unhealthy food). We hope readers can accept this change with little loss of sleep. We certainly are happy to break the shackles that have previously limited almost all raster graphics to the boring world of rectangular arrays of pixels. This amazing new feature is available through the following version of the `DrawImage` method:

```
public void DrawImage(
   Image image,
   Rectangle destRect,
   int srcX,
   int srcY,
   int srcWidth,
   int srcHeight,
   GraphicsUnit srcUnit,
   ImageAttributes imageAttr);
```

Download at WoweBook.Com

With its eight parameters, this version of the `DrawImage` method is the most complicated one that the .NET Compact Framework supports. Perhaps it is appropriate that this version matches the other versions in capabilities: It can draw an entire bitmap at its original size, draw a portion of a bitmap at its original size, and draw a portion of a bitmap at a different size.

What makes this version different is the final parameter, a reference to an `ImageAttributes` object. On the desktop, this class supports a variety of color adjustments that can be applied when drawing a bitmap onto a surface. The .NET Compact Framework version is much simpler, with what amounts to a single property: a color key. The color key defines the range of colors that represent transparent portions of an image. In other words, any color that matches the color key is a color that is *not* copied by the call to the `DrawImage` method. The color key settings are controlled through two methods: `SetColorKey` defines the transparency range, and `ClearColorKey` disables the transparency range.

Figure 11.3 shows an example of transparency at work. A 100 × 100 bitmap is first drawn without transparency at the window origin. That same bitmap is then drawn three times, using the version of the `DrawImage` method that supports transparency. The color key is set to light gray, which corresponds to the color outside the ellipse (the interior of the ellipse is set to yellow). Listing 11.6 shows the code, a handler for a `MouseDown` event, which we used to create the example.

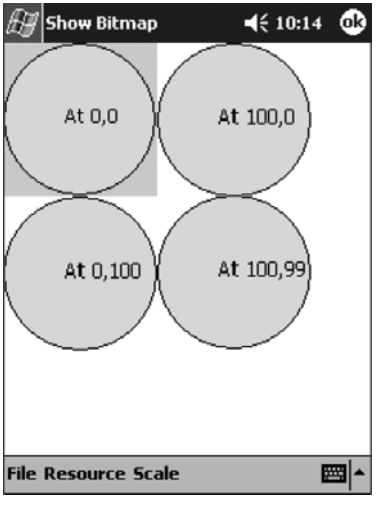


FIGURE 11.3: Four Calls to the `DrawImage` Method, Three with Transparency Enabled

Download at WoweBook.Com

Listing 11.6: Event Handler That Draws a Bitmap with Transparency

```
bool bFirstTime = true;

private void
FormMain_MouseDown(object sender, MouseEventArgs e)
{
   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();

   // Create a bitmap and a Graphics object for the bitmap.
   Bitmap bmpNew = new Bitmap(100,100);
   Graphics gbmp = Graphics.FromImage(bmpNew);

   // Clear the bitmap background.
   gbmp.Clear(Color.LightGray);

   // Create some drawing objects.
   Pen penBlack = new Pen(Color.Black);
   Brush brBlack = new SolidBrush(Color.Black);
   Brush brYellow = new SolidBrush(Color.Yellow);

   // Draw onto the bitmap.
   gbmp.FillEllipse(brYellow, 0, 0, 98, 98);
   gbmp.DrawEllipse(penBlack, 0, 0, 98, 98);
   gbmp.DrawString("At " + e.X.ToString() + "," + e.Y.ToString(),
      Font, brBlack, 40, 40);

   // Copy the bitmap to the window at the MouseDown location.
   if (bFirstTime)
   {
      // Copy without transparency.
      g.DrawImage(bmpNew, e.X, e.Y);
      bFirstTime = false;
   }
   else
   {
      // Copy the bitmap using transparency.
      Rectangle rectDest = new Rectangle(e.X, e.Y, 100, 100);
      ImageAttributes imgatt = new ImageAttributes();
      imgatt.SetColorKey(Color.LightGray, Color.LightGray);
      g.DrawImage(bmpNew, rectDest, 0, 0, 99, 99,
         GraphicsUnit.Pixel, imgatt);
   }

   // Clean up when we are done.
   g.Dispose();
   gbmp.Dispose();
   bmpNew.Dispose();
}
```

Download at WoweBook.Com

11.3.5 A Sample Program: ShowBitmap

Our bitmap drawing sample program shows several features of bitmaps that we have been discussing. This program can open files and create a bitmap. Several formats are supported, including the standard Windows DIB (.bmp) files and a few compressed image file formats such as GIF (.gif) files, JPEG (.jpg) files, and PNG (.png) files. Figure 11.4 shows the ShowBitmap program with a JPEG image of Chandler (the office beagle at The

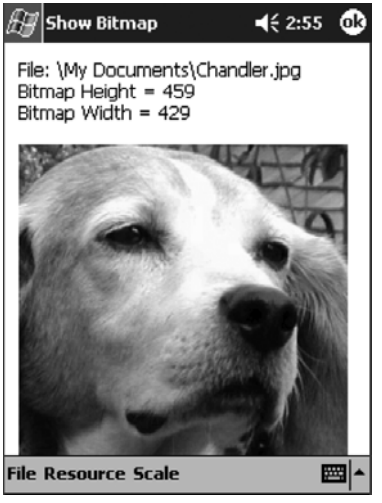


FIGURE 11.4: ShowBitmap Displaying a JPEG File Scaled to 50%

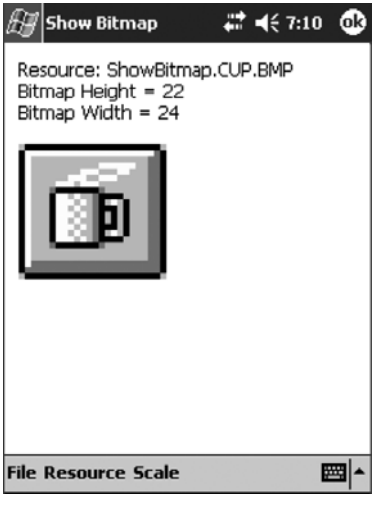


FIGURE 11.5: ShowBitmap Displaying a Bitmap from a Resource

Download at WoweBook.Com

Paul Yao Company). This image is drawn scaled to 50%, an effect made possible by selecting the appropriate version of the `DrawImage` method.

Our sample program contains a set of bitmap files that are bound to the program files as embedded resources (see Listing 11.7). As with all types of resources, the resource data does not get loaded into memory until we explicitly load the resource. In this program, we load the resource when the user selects an item on the program's resource menu. Figure 11.5 shows the bitmap resource that was read from a resource identified as ShowBitmap.CUP.BMP, drawn at 400% of its original size.

Listing 11.7: Source Code for ShowBitmap.cs

```
using System.Reflection; // Needed for Assembly
using System.IO;         // Needed for Stream
using System.Drawing.Imaging;  // Needed for ImageAttributes
// ...

private Bitmap bmpDraw;
bool bFirstTime = true;
bool bResource = false;
string strResName;

// Draw a bitmap using transparency where the MouseDown
// event is received.
private void
FormMain_MouseDown(object sender, MouseEventArgs e)
{
#if false
   CreateAndDraw(e.X, e.Y);
#endif
   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();

   // Create a bitmap and a Graphics object for the bitmap.
   Bitmap bmpNew = new Bitmap(100,100);
   Graphics gbmp = Graphics.FromImage(bmpNew);

   // Clear the bitmap background.
   gbmp.Clear(Color.LightGray);

   // Create some drawing objects.
   Pen penBlack = new Pen(Color.Black);
   Brush brBlack = new SolidBrush(Color.Black);
   Brush brYellow = new SolidBrush(Color.Yellow);

   // Draw onto the bitmap.
   gbmp.FillEllipse(brYellow, 0, 0, 98, 98);
   gbmp.DrawEllipse(penBlack, 0, 0, 98, 98);
```

Download at WoweBook.Com


```
      gbmp.DrawString("At " + e.X.ToString() + "," + e.Y.ToString(),
         Font, brBlack, 40, 40);

      // Copy the bitmap to the window at the MouseDown location.
      if (bFirstTime)
      {
         // Copy without transparency.
         g.DrawImage(bmpNew, e.X, e.Y);
         bFirstTime = false;
      }
      else
      {
         // Copy the bitmap using transparency.
         Rectangle rectDest = new Rectangle(e.X, e.Y, 100, 100);
         ImageAttributes imgatt = new ImageAttributes();
         imgatt.SetColorKey(Color.LightGray, Color.LightGray);
         g.DrawImage(bmpNew, rectDest, 0, 0, 99, 99,
            GraphicsUnit.Pixel, imgatt);
      }

      // Clean up when we are done.
      g.Dispose();
      gbmp.Dispose();
      bmpNew.Dispose();
   }

   private void FormMain_Paint(object sender, PaintEventArgs e)
   {
      Graphics g = e.Graphics;
      float sinX = 10.0F;
      float sinY = 10.0F;
      SizeF szfText = g.MeasureString("X", Font);
      float cyLine = szfText.Height;

      Brush brText = new SolidBrush(SystemColors.WindowText);
      if (bmpDraw != null)
      {
         if (bResource)
         {
            g.DrawString("Resource: " + strResName,
               Font, brText, sinX, sinY);
         }
         else
         {
            g.DrawString("File: " + dlgFileOpen.FileName,
               Font, brText, sinX, sinY);
         }
         sinY += cyLine;

         g.DrawString("Bitmap Height = " + bmpDraw.Height,
```

Download at WoweBook.Com


```
            Font, brText, sinX, sinY);
         sinY += cyLine;

         g.DrawString("Bitmap Width = " + bmpDraw.Width,
            Font, brText, sinX, sinY);
         sinY += cyLine;
         sinY += cyLine;

         if (mitemScale100.Checked)
         {
            g.DrawImage(bmpDraw, (int)sinX, (int)sinY);
         }
         else
         {
            Rectangle rectSrc = new Rectangle(0, 0,
               bmpDraw.Width, bmpDraw.Height);
            int xScaled = 0;
            int yScaled = 0;
            if (mitemScale50.Checked)
            {
               xScaled = bmpDraw.Width / 2;
               yScaled = bmpDraw.Height / 2;
            }
            else if (mitemScale200.Checked)
            {
               xScaled = bmpDraw.Width * 2;
               yScaled = bmpDraw.Height * 2;
            }
            else if (mitemScale400.Checked)
            {
               xScaled = bmpDraw.Width * 4;
               yScaled = bmpDraw.Height * 4;
            }

            Rectangle rectDest = new Rectangle((int)sinX,
               (int)sinY, xScaled, yScaled);
            g.DrawImage(bmpDraw, rectDest, rectSrc,
               GraphicsUnit.Pixel);
         }
      }
      else
      {
         g.DrawString("File: None", Font, brText, sinX, sinY);
      }
   }

   private void
   mitemFileOpen_Click(object sender, EventArgs e)
   {
      dlgFileOpen.Filter = "Bitmap (*.bmp)|*.bmp|" +
```

Download at WoweBook.Com


```
                        "Picture (*.jpg)|*.jpg|" +
                        "PNG Files (*.png)|*.png|" +
                        "TIF Files (*.tif)|*.tif|" +
                        "GIF Files (*.gif)|*.gif |" +
                        "All Files (*.*)|*.*";
   if (dlgFileOpen.ShowDialog() == DialogResult.OK)
   {
      Bitmap bmpNew = null;
      try
      {
         bmpNew = new Bitmap(dlgFileOpen.FileName);
         bResource = false;
      }
      catch
      {
         MessageBox.Show("Cannot create bitmap from " +
            "File: " + dlgFileOpen.FileName);
         return;
      }

      DisposeBitmap (ref bmpDraw);
      bmpDraw = bmpNew;
      Invalidate();
   }
}

private void
mitemScale_Click(object sender, EventArgs e)
{
   // Clear the checkmark on related items.
   mitemScale50.Checked = false;
   mitemScale100.Checked = false;
   mitemScale200.Checked = false;
   mitemScale400.Checked = false;

   // Set the checkmark on selected menu item.
   ((MenuItem)sender).Checked = true;

   // Request paint to redraw bitmap.
   Invalidate();
}


private void
mitemResourceCup_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("CUP.BMP");
   Invalidate();
}
```

Download at WoweBook.Com


```
private void
mitemResourceBell_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("BELL.BMP");
   Invalidate();
}

private void
mitemResourceSpade_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("SPADE.BMP");
   Invalidate();
}

private void
mitemResourceHeart_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("HEART.BMP");
   Invalidate();
}

private void
mitemResourceDiamond_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("DIAMOND.BMP");
   Invalidate();
}

private void
mitemResourceClub_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("CLUB.BMP");
   Invalidate();
}

private Bitmap LoadBitmapResource(string strName)
{
   Assembly assembly = Assembly.GetExecutingAssembly();
   string strRes = "ShowBitmap." + strName;
   Stream stream = assembly.GetManifestResourceStream(strRes);
   Bitmap bmp = null;
   try
   {
      bmp = new Bitmap(stream);
```

Download at WoweBook.Com


```
            strResName = strRes;
            bResource = true;
        }
        catch { }
        stream.Close();

        return bmp;
    }

    private void DisposeBitmap(ref Bitmap bmp)
    {
        if (bmp != null)
        {
            bmp.Dispose();
        }
        bmp = null;
    }

    // Simplest possible bitmap: Create a bitmap, clear the
    // bitmap background, draw the bitmap to the display screen.
    private void
    CreateAndDraw(int x, int y)
    {
        // Create a bitmap and a Graphics object for the bitmap.
        Bitmap bmpNew = new Bitmap(100,100);
        Graphics gbmp = Graphics.FromImage(bmpNew);

        // Clear the bitmap background.
        gbmp.Clear(Color.LightGray);

        // Get a Graphics object for the form.
        Graphics g = CreateGraphics();

        // Copy the bitmap to the window at (x,y) location.
        g.DrawImage(bmpNew, x, y);

        // Clean up when we are done.
        g.Dispose();
        gbmp.Dispose();
        bmpNew.Dispose();
    }
```

11.4 Vector Graphics

The available vector drawing methods in the .NET Compact Framework are summarized in Table 11.11 (which appeared earlier in this chapter as Table 11.4 and is repeated here for convenience). As indicated in the table, some shapes are drawn with a pen, a drawing object used for lines. The

Download at WoweBook.Com

TABLE 11.11: `System.Drawing.Graphics` Methods for Vector Drawing

Method	Comment
`DrawEllipse`	Draws the outline of an ellipse using a pen
`DrawLine`	Draws a straight line using a pen
`DrawPolygon`	Draws the outline of a polygon using a pen
`DrawRectangle`	Draws the outline of a rectangle using a pen
`FillEllipse`	Fills the interior of an ellipse using a brush
`FillPolygon`	Fills the interior of a polygon using a brush
`FillRectangle`	Fills the interior of a rectangle using a brush

.NET Compact Framework supports only pens that are 1 pixel wide (unless a programmer drills through to the native GDI drawing support). Other shapes in the table are drawn with a brush. We discussed the three methods for creating brushes earlier in this chapter. Here we'll cover the creation of pens in this discussion of vector graphics.

The vector methods with names that start with `Draw` are those that use a pen to draw a line or a set of connected lines. The call to the `DrawRectangle` method, for example, draws the outline of a rectangle without touching the area inside the line. If you pass a blue pen to the `DrawRectangle` method, the result is the outline of a rectangle drawn with a blue line. The .NET Compact Framework supports four line-drawing methods.

Vector methods whose names start with `Fill`, on the other hand, use a brush to fill in the area bounded by the lines. For example, if you pass a red brush to the `FillRectangle` method, the result is a solid red rectangle. There are three such methods in the .NET Compact Framework for drawing ellipses, polygons, and rectangles.

The `Draw` and `Fill` methods complement each other. You could, for example, pass a red brush to the `FillRectangle` method and pass a blue pen to the `DrawRectangle` method using the same coordinates that you used to draw the red, filled rectangle. The result would be a two-colored rectangle, with a blue border and a red interior. This type of two-colored figure is natively available in the Windows API. Yet it seems apparent that few programs need to draw two-colored vector figures. That is, no doubt, a factor that contributed to the design of vector drawing in the .NET Framework and the .NET Compact Framework.

Download at WoweBook.Com

If a programmer is willing to do a bit of work, almost all vector drawing can be accomplished by calling two of these methods: `DrawLine` and `FillPolygon`. Each of the supported method names is of the form <verb><shape>. In the `DrawLine` method, for example, the verb is `Draw` and the shape is `Line`.

11.4.1 Creating Pens

Pens draw lines. The desktop supports a very sophisticated model for pens, including support for scalable geometric pens and nonscalable cosmetic pens. Pens on the desktop support features that allow you to fine-tune how an end of a line appears (rounded or squared) and even how the "elbow" joints are drawn. Pens can be wide or narrow, and even nonsolid pen colors are supported.

In the .NET Compact Framework, pens are always 1 pixel wide. Pens provide a quick and simple way to define the color used to draw a line. From the 17 properties supported for pens on the desktop, one has survived to the .NET Compact Framework: `Color`. And so it should come as no surprise that the one constructor for the `Pen`[28] class has a single parameter, a color as shown here:

```
public Pen(
   Color color);
```

There are three ways to define a pen in a .NET Compact Framework program because there are three ways to specify a color:

1. With a system color
2. With a named color
3. With an RGB value

Earlier in this chapter, we described some of the details about the three ways to pick a color. We showed that each of the color-specifying approaches could be used to create a brush. Now the time has come to show the same thing for pens.

The following code fragment creates three pens. One pen is created using a system color; another pen is created using a named color; and finally, the third pen is created with an RGB value.

---

28. `System.Drawing.Pen`.

Download at WoweBook.Com


```
// Pen from a system color
Pen penCtrl = new Pen(SystemColors.ControlDark);

// Pen from a named color
Pen penRed  = new Pen(Color.Red);

// Pen from an RGB value
Pen penBlue = new Pen(Color.FromArgb(0, 0, 255));
```

11.4.2 A Game: JaspersDots

While writing this book, we watched Paul's son, Jasper, playing a paper-and-pencil game with one of his friends. They were having so much fun that we decided to write a .NET Compact Framework version. The game was Dots, which may be familiar to some readers. In this two-person game, players take turns connecting dots that have been drawn in a grid. A player is awarded a point for drawing the last line that creates a box. We named our version of the game `JaspersDots`, in honor of Paul's son. The playing board for this game is drawn entirely with the following vector graphics methods:

- `FillEllipse`
- `FillRectangle`
- `DrawLine`
- `DrawRectangle`

This program provides extensive use of various `Graphics` objects including colors, pens, and brushes.

Figure 11.6 shows the New Game dialog box. Each player enters a name and picks a color to use for claimed squares. The default board size is 8×8, which can be overridden in the New Game dialog box (the maximum board size is 11×9).

The New Game dialog box is a simple dialog box drawn with regular controls, with one small enhancement: This dialog handles a `Paint` event, which draws a selection rectangle around each player's currently selected color. The set of available colors is drawn with `Panel` controls, five for each player. Listing 11.8 shows the source code for the event handlers for responding to the `Click` event for each set of `Panel` controls and to the `Paint` event for the New Game dialog box.

Download at WoweBook.Com

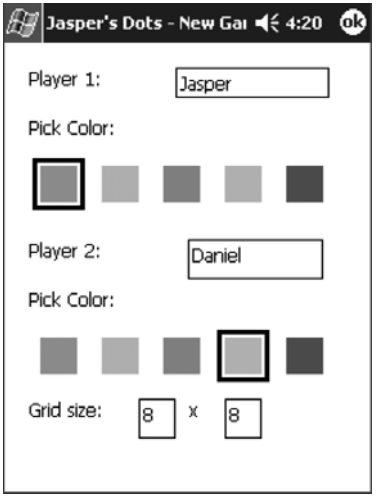


FIGURE 11.6: New Game Dialog Box for the `JaspersDots` Program

LISTING 11.8: `Paint` and `Click` Event Handlers for the New Game Dialog Box

```
private void
Panel1_Click(object sender, EventArgs e)
{
   if (sender == (object)panel1)
      iColor1 = 0;
   else if (sender == (object)panel2)
      iColor1 = 1;
   else if (sender == (object)panel3)
      iColor1 = 2;
   else if (sender == (object)panel4)
      iColor1 = 3;
   else if (sender == (object)panel5)
      iColor1 = 4;

   // Redraw the window.
   Invalidate();
}

private void
Panel2_Click(object sender, EventArgs e)
{
   if (sender == (object)panelA)
      iColor2 = 0;
   else if (sender == (object)panelB)
      iColor2 = 1;
   else if (sender == (object)panelC)
      iColor2 = 2;
```

Download at WoweBook.Com


```
   else if (sender == (object)panelD)
      iColor2 = 3;
   else if (sender == (object)panelE)
      iColor2 = 4;

   // Redraw the window.
   Invalidate();
}
private void
GameNewDialog_Paint(object sender, PaintEventArgs e)
{
   Panel panel = panel1;

   //
   // Player 1
   //
   // What is the current player 1 panel?
   switch(iColor1)
   {
      case 0:
         panel = panel1;
         break;
      case 1:
         panel = panel2;
         break;
      case 2:
         panel = panel3;
         break;
      case 3:
         panel = panel4;
         break;
      case 4:
         panel = panel5;
         break;
   }
   clr1 = panel.BackColor;

   // Draw a rectangle around the color selected by player 1.
   Pen penBlack = new Pen(Color.Black);
   Rectangle rc = new
      Rectangle(panel.Left - 3,
      panel.Top - 3,
      panel.Width + 5,
      panel.Height + 5);
   e.Graphics.DrawRectangle(penBlack, rc);
   rc.Inflate(1, 1);
   e.Graphics.DrawRectangle(penBlack, rc);
   rc.Inflate(1, 1);
   e.Graphics.DrawRectangle(penBlack, rc);
```

Download at WoweBook.Com


```
   //
   // Player 2
   //
   // What is the current player 2 panel?
   switch(iColor2)
   {
      case 0:
         panel = panelA;
         break;
      case 1:
         panel = panelB;
         break;
      case 2:
         panel = panelC;
         break;
      case 3:
         panel = panelD;
         break;
      case 4:
         panel = panelE;
         break;
   }
   clr2 = panel.BackColor;

   // Draw a rectangle around the color selected by player 2.
   rc = new Rectangle(panel.Left - 3,
      panel.Top - 3,
      panel.Width + 5,
      panel.Height + 5);
   e.Graphics.DrawRectangle(penBlack, rc);
   rc.Inflate(1, 1);
   e.Graphics.DrawRectangle(penBlack, rc);
   rc.Inflate(1, 1);
   e.Graphics.DrawRectangle(penBlack, rc);
}
```

There is a bug in Visual Studio .NET that affects C# programmers. The bug is that supported events for certain controls do not appear in the Designer. You can, however, add an event handler manually. Inside the Visual Studio code editor, you type the control name, the event name, and the += operator, and IntelliSense helps by providing the rest.

In our `JaspersDots` game, we found that the Designer did not support the `Click` event for `Panel` controls. To create `Click` event handlers for the `Panel` controls in the New Game dialog box, we manually typed in event handler names, which IntelliSense completed for us. The resulting code appears in Listing 11.9.

Download at WoweBook.Com

Listing 11.9: Adding Event Handlers Manually

```
// Set up the Click handler for player 1 panels.
// Note: The Designer does not support this
// so we have to do it manually.
panel1.Click += new EventHandler(this.Panel1_Click);
panel2.Click += new System.EventHandler(this.Panel1_Click);
panel3.Click += new System.EventHandler(this.Panel1_Click);
panel4.Click += new System.EventHandler(this.Panel1_Click);
panel5.Click += new System.EventHandler(this.Panel1_Click);

// Set up the Click handler for player 2 panels.
// Note: The Designer does not support this
// so we have to do it manually.
panelA.Click += new EventHandler(this.Panel2_Click);
panelB.Click += new System.EventHandler(this.Panel2_Click);
panelC.Click += new System.EventHandler(this.Panel2_Click);
panelD.Click += new System.EventHandler(this.Panel2_Click);
panelE.Click += new System.EventHandler(this.Panel2_Click);
```

Figure 11.7 shows an example of the `JaspersDots` game in play. Each dot is drawn with a call to the `FillEllipse` method that is drawn in a bounding rectangle that is 4 pixels by 4 pixels. Players draw lines by clicking in the area between dots, and when a hit is detected a line is drawn by calling the `DrawLine` method. A player's claimed boxes are drawn with calls to the `FillRectangle` method.

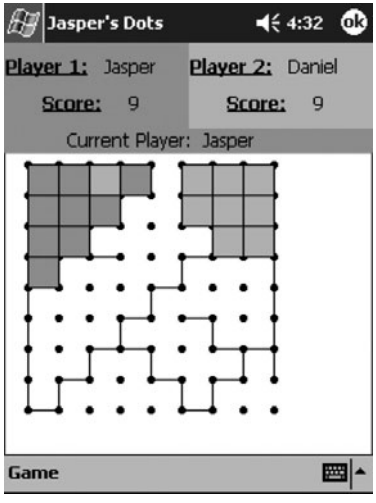


Figure 11.7: `JaspersDots` with a Game under Way

Download at WoweBook.Com

The JaspersDots program uses a custom control for the game window, our DotControl class. Listing 11.10 shows the source code to the DotControl class.

LISTING 11.10: The DotControl Class

```
public class DotControl : System.Windows.Forms.Control
{
   private FormMain formParent;
   private Brush m_brPlayer1;
   private Brush m_brPlayer2;
   private Squares sq;

   public DotControl(FormMain form)
   {
      formParent = form;

      formParent.Controls.Add(this);
      this.Paint += new
         PaintEventHandler(this.DotControl_Paint);
      this.MouseDown += new
         MouseEventHandler(this.DotControl_MouseDown);
      this.Left = 0;
      this.Top = 64;
      this.Width = 240;
      this.Height = 240;

      sq = new Squares(this);
   }

   public bool SetGridSize(int cxWidth, int cyHeight)
   {
      return sq.SetGridSize(cxWidth, cyHeight);
   }

   public bool SetPlayerColors(Color clr1, Color clr2)
   {
      m_brPlayer1 = new SolidBrush(clr1);
      m_brPlayer2 = new SolidBrush(clr2);

      return sq.SetPlayerBrushes(m_brPlayer1, m_brPlayer2);
   }


   private void
   DotControl_MouseDown(object sender, MouseEventArgs e)
   {
      // Check result.
```

Download at WoweBook.Com


```
      int iResult = sq.HitTest(e.X, e.Y,
         formParent.CurrentPlayer);

      // Click on the available line, no score.
      if(iResult == 1)
      {
         formParent.NextPlayer();
      }

      // Click on the available line, score.
      if (iResult == 2)
      {
         int iScore1 = sq.GetScore(1);
         formParent.DisplayScore(1, iScore1);
         int iScore2 = sq.GetScore(2);
         formParent.DisplayScore(2, iScore2);

         int count = sq.Height * sq.Width;
         if (iScore1 + iScore2 == count)
         {
            string strResult = null;

            if (iScore1 > iScore2)
               strResult = "Player 1 wins! ";
            else if (iScore1 < iScore2)
               strResult = "Player 2 wins! ";
            else
               strResult = "Tie Game! ";

            MessageBox.Show(strResult, "JaspersDots");
         }
      }
   }

   private void
   DotControl_Paint(object sender, PaintEventArgs e)
   {
      // Fill squares which players now own.
      sq.FillSquares(e.Graphics);

      // Draw lines which players have selected.
      sq.DrawLines(e.Graphics);

      // Draw dots in grid.
      sq.DrawDots(e.Graphics);
   }

} // class
```

Download at WoweBook.Com

The `DotControl` class handles two events: `MouseDown` and `Paint`. Most of the work for these events is done by a helper class named `Squares`. The source code for the `Squares` class appears in Listing 11.11.

LISTING 11.11: The `Squares` Class

```
public class Squares
{
   public int Width
   {
      get   { return cxWidth; }
   }
   public int Height
   {
      get   { return cyHeight; }
   }

   private int cxLeft = 15;
   private int cyTop  = 15;
   private int cxWidth;
   private int cyHeight;
   const int cxLine = 20;
   const int cyLine = 20;
   const int cxyDelta = 5;
   private Square [,] m_asq;

   private Control m_ctrlParent;
   private Brush m_brPlayer1;
   private Brush m_brPlayer2;
   private Brush m_brBackground = new _
      SolidBrush(SystemColors.Window);
   private Brush hbrBlack = new SolidBrush(Color.Black);
   private Point ptTest = new Point(0,0);
   Rectangle rc = new Rectangle(0, 0, 0, 0);
   private Size  szDot = new Size(4,4);

   Pen penLine = new Pen(Color.Black);

   public Squares(Control ctrlParent)
   {
      m_ctrlParent = ctrlParent;
   } // Squares()

   public bool SetGridSize(
      int cxNewWidth,     // Width of array.
      int cyNewHeight     // Height of array.
      )
   {
      // Temporary scratch space
```

Download at WoweBook.Com


```
Rectangle rcTemp = new Rectangle(0,0,0,0);
Point     ptTemp = new Point(0,0);
Size      szTemp = new Size(0,0);

// Set up an array to track squares.
cxWidth = cxNewWidth;
cyHeight = cyNewHeight;
m_asq = new Square[cxWidth, cyHeight];
if (m_asq == null)
   return false;

int x, y;
for (x = 0; x < cxWidth; x++)
{
   for (y = 0; y < cyHeight; y++)
   {
      m_asq[x,y].iOwner = 0; // No owner.
      int xLeft = cxLeft + x * cxLine;
      int yTop = cyTop + y * cyLine;
      int xRight = cxLeft + (x+1) * cxLine;
      int yBottom = cyTop + (y+1) * cyLine;
      int cxTopBottom = cxLine - (2 * cxyDelta);
      int cyTopBottom = cxyDelta * 2;
      int cxLeftRight = cxyDelta * 2;
      int cyLeftRight = cxLine - (2 * cxyDelta);

      // Main rectangle
      ptTemp.X = xLeft + 1;
      ptTemp.Y = yTop + 1;
      szTemp.Width = xRight - xLeft - 1;
      szTemp.Height = yBottom - yTop - 1;
      rcTemp.Location = ptTemp;
      rcTemp.Size = szTemp;
      m_asq[x,y].rcMain = rcTemp;

      // Top hit rectangle
      m_asq[x,y].rcTop =
         new Rectangle(xLeft + cxyDelta,
         yTop - cxyDelta,
         cxTopBottom,
         cyTopBottom);
      m_asq[x,y].bTop = false;

      // Right hit rectangle
      m_asq[x,y].rcRight =
         new Rectangle(xRight - cxyDelta,
         yTop + cxyDelta,
         cxLeftRight,
         cyLeftRight);
      m_asq[x,y].bRight = false;
```

Download at WoweBook.Com


```
            // Bottom hit rectangle
            m_asq[x,y].rcBottom =
               new Rectangle(xLeft + cxyDelta,
               yBottom - cxyDelta,
               cxTopBottom,
               cyTopBottom);
            m_asq[x,y].bBottom = false;

            // Left hit rectangle
            m_asq[x,y].rcLeft =
               new Rectangle(xLeft - cxyDelta,
               yTop + cxyDelta,
               cxLeftRight,
               cyLeftRight);
            m_asq[x,y].bLeft = false;

         } // for y
      } // for x

      return true;
   }

   public bool
   SetPlayerBrushes(
      Brush br1,      // Brush color for player 1
      Brush br2       // Brush color for player 2
      )
   {
      m_brPlayer1 = br1;
      m_brPlayer2 = br2;

      return true;
   }

   //------------------------------------------------------
   public void
   FillOneSquare(Graphics g, int x, int y)
   {
      Brush brCurrent = m_brBackground;
      if (m_asq[x,y].iOwner == 1)
         brCurrent = m_brPlayer1;
      else if (m_asq[x,y].iOwner == 2)
         brCurrent = m_brPlayer2;
      g.FillRectangle(brCurrent, m_asq[x,y].rcMain);
   }
continues

continued
   // FillSquares -- Fill owned squares with a player's color.
   //
```

Download at WoweBook.Com


```
public void
FillSquares(Graphics g)
{
   int x, y;
   for (x = 0; x < cxWidth; x++)
   {
      for (y = 0; y < cyHeight; y++)
      {
         if (m_asq[x,y].iOwner != 0)
         {
            FillOneSquare(g, x, y);
         }
      }
   }
} // method: FillSquares

//
// DrawOneLineSet
//
public void DrawOneLineSet(Graphics g, int x, int y)
{
   int xLeft = cxLeft + x * cxLine;
   int yTop = cyTop + y * cyLine;
   int xRight = cxLeft + (x+1) * cxLine;
   int yBottom = cyTop + (y+1) * cyLine;

   if (m_asq[x,y].bTop)
      g.DrawLine(penLine, xLeft, yTop, xRight, yTop);
   if (m_asq[x,y].bRight)
      g.DrawLine(penLine, xRight, yTop, xRight, yBottom);
   if (m_asq[x,y].bBottom)
      g.DrawLine(penLine, xRight, yBottom, xLeft, yBottom);
   if (m_asq[x,y].bLeft)
      g.DrawLine(penLine, xLeft, yBottom, xLeft, yTop);
} // DrawOneLineSet()

//
// DrawLines -- Draw lines which have been hit.
//
public void DrawLines(Graphics g)
{
   int x, y;
   for (x = 0; x < cxWidth; x++)
   {
      for (y = 0; y < cyHeight; y++)
      {
         DrawOneLineSet(g, x, y);
      }
   }
} // DrawLines()
```

Download at WoweBook.Com


```
public void DrawDots (Graphics g)
{
   // Draw array of dots.
   int x, y;
   for (x = 0; x <= cxWidth; x++)
   {
      for (y = 0; y <= cyHeight; y++)
      {
         ptTest.X = (cxLeft - 2) + x * cxLine;
         ptTest.Y = (cyTop - 2) + y * cyLine;
         rc.Location = ptTest;
         rc.Size = szDot;
         g.FillEllipse(hbrBlack, rc);
      }
   }
} // DrawDots

public enum Side
{
   None,
   Left,
   Top,
   Right,
   Bottom
}

//
// HitTest - Check whether a point hits a line.
//
// Return values:
// 0 = miss
// 1 = hit a line
// 2 = hit and completed a square.
public int HitTest(int xIn, int yIn, int iPlayer)
{
   int x, y;
   bool bHit1 = false;
   bool bHit2 = false;
   Side sideHit = Side.None;
   for (x = 0; x < cxWidth; x++)
   {
      {
      for (y = 0; y < cyHeight; y++)
         // If already owned, do not check.
         if (m_asq[x,y].iOwner != 0)
            continue;

         // Otherwise check for lines against point.
         if (m_asq[x,y].rcTop.Contains(xIn, yIn))
         {
```

Download at WoweBook.Com


```
   // Line already hit?
   if (m_asq[x,y].bTop) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Top;
   m_asq[x,y].bTop = true;
}
else if (m_asq[x,y].rcLeft.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bLeft) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Left;
   m_asq[x,y].bLeft = true;
}
else if (m_asq[x,y].rcRight.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bRight) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Right;
   m_asq[x,y].bRight = true;
}
else if (m_asq[x,y].rcBottom.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bBottom) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Bottom;
   m_asq[x,y].bBottom = true;
}

// No hit in current square -- keep looking.
if (sideHit == Side.None)
   continue;

// We hit a side.
bHit1 = true;

// Draw sides.
Graphics g = m_ctrlParent.CreateGraphics();
DrawOneLineSet(g, x, y);

// Check whether square is now complete.
// We hit a line - check for hitting a square.
if (m_asq[x,y].bLeft &&
      m_asq[x,y].bTop &&
```

Download at WoweBook.Com


```
                m_asq[x,y].bRight &&
                m_asq[x,y].bBottom)
            {
               // Side is complete.
               m_asq[x,y].iOwner = iPlayer;
               bHit2 = true;

               // Fill current square.
               FillOneSquare(g, x, y);
            }

            g.Dispose();

         } // for y
      } // for x

      if (bHit2) return 2;
      else if (bHit1) return 1;
      else return 0;

   } // HitTest

   //
   // GetScore - Get current score for player N.
   //
   public int GetScore (int iPlayer)
   {
      int iScore = 0;
      int x, y;
      for (x = 0; x < cxWidth; x++)
      {
         for (y = 0; y < cyHeight; y++)
         {
            if (m_asq[x,y].iOwner == iPlayer)
               iScore++;
         }
      }
      return iScore;
   } // GetScore

} // class Squares
```

Finally, we define two simple data structures—`Square` and `Players`—to hold details about individual game board squares and details about individual players, respectively. Listing 11.12 shows the code.

Download at WoweBook.Com

Listing 11.12: The `Square` and `Players` Structures

```
public struct Square
{
   // Coordinate of main rectangle
   public Rectangle rcMain;
   public int iOwner;

   // Hit-rectangles of four edges of main rectangle
   public Rectangle rcTop;
   public bool bTop;
   public Rectangle rcRight;
   public bool bRight;
   public Rectangle rcBottom;
   public bool bBottom;
   public Rectangle rcLeft;
   public bool bLeft;
} // struct Square

public class Players
{
   public string strName1;
   public string strName2;
   public bool bComputerPlaying;
   public System.Drawing.Color clr1;
   public System.Drawing.Color clr2;
}
```

11.5 Conclusion

It should be clear from this discussion that the .NET Compact Framework's `System.Drawing.dll` library provides rich capabilities for creating graphical output. This chapter looked in detail at the four types of drawing surfaces that Windows programmers are used to, two of which are supported in the .NET Compact Framework. This chapter also discussed the three families of drawing functions, which are reasonably well represented by the .NET Compact Framework's capable, though small, feature set.

This chapter explored two drawing families in depth: raster and vector drawing functions. The next chapter provides an in-depth look at the third family of output: text.

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

12
Text and Fonts

This chapter covers the process of drawing text using members of the `System.Drawing` *namespace.*

THE FOCUS OF THIS CHAPTER is on drawing text on the screen of a smart device using the .NET Compact Framework. As expected, the Compact Framework supports only a subset of text-drawing features of the full .NET Framework. Nonetheless, text-drawing support is surprisingly rich. All commonly required text operations are supported, such as selecting named fonts, along with choosing font sizes, styles, and text color. Some things are missing, but the most common operations are present.

12.1 Drawing Text

A graphical environment gives programmers a rich set of tools for creating a wide range of graphical effects. You can freely mix pictures with text, display text in different sizes and styles, and use various effects to convey subtle—and sometimes not-so-subtle—messages to your users. Text itself is a graphical object.

Drawing text in a graphical environment is paradoxically more complex than drawing text in a nongraphical environment. In a character-oriented world such as a console program, fixed-pitch fonts[1] place text in orderly rows and columns. In a graphical world, by contrast, fixed-pitch fonts and variable-pitch fonts float in a sea of pixels. Drawing text in a graphical environment takes a little more effort, but the results are well worth the time and energy you put into it.

1. In a fixed-pitch font, the width of every character in the font is the same.

Download at WoweBook.Com

12.1.1 Text-Drawing Support in the .NET Compact Framework

A .NET Compact Framework program can draw text using a built-in font, or you can add new fonts by dropping font (*.ttf) files into the `\windows\fonts` directory. The use of TrueType fonts allows text to be scaled to any size from 8 points up to 72 points (and beyond). That text can be drawn in any available color, although most programs use black. And you can even rotate that text. Table 12.1 summarizes Compact Framework text-drawing features, how to access these features, and this chapter's sample programs which illustrate each feature.

TABLE 12.1: Text-Drawing Features and Sample Programs

Feature	Comment	Sample Program
Simple text drawing	Call the `DrawString` method to draw text in a control or form	`SimpleDrawString`—shows simplest text drawing, which involves creation of a brush (for text color) and use of the form's default font
Simplest font creation	Create a font using the `FontFamily` enumeration to select between a fixed-pitch, serif, or sans serif font without regard to font face name	`GenericFonts`—creates and draws with each of the three generic font families, in a range of styles (regular, bold, italic, strikeout, and underline)
Font enumeration	Enumerated fonts using the `InstalledFontCollection` object	`FontPicker`—Compact Framework program that creates fonts of a specific face name
Rotate text	To rotate text, you draw using a rotated font; do this by filling a `LOGFONT` structure and calling the `FromLogFont` member of the `Font` class	`RotateText`—shows how much fun rotated text can really be
ClearType fonts	ClearType is a font technology that makes text more legible on LCD displays. Create ClearType fonts by filling in a `LOGFONT` structure and specifying `CLEARTYPE_QUALITY` in the `lfQuality` field	Does not explicitly create a ClearType font; the `RotateText` sample is useful because it shows how to fill in a `LOGFONT` structure

Download at WoweBook.Com

TABLE 12.1: Text-Drawing Features and Sample Programs *(Continued)*

Feature	Comment	Sample Program
Calculate size of graphical text	Optimal positioning of text requires calculating the size of the bounding box of drawn text. This is accomplished using the `MeasureString` method	`MeasureString`—shows how to use results from the `MeasureString` method in drawing
Setting text alignment	By default, text is aligned to the upper-left corner of the text box. Override this default by setting alignment properties in the `StringFormat` structure and calling the `DrawString` overrides which accept a `StringFormat` as a parameter	`TextAlign`—shows nine ways to align text by mixing and matching three vertical alignments and three horizontal alignments

12.1.2 The `DrawString` Method

All Compact Framework text drawing is done with the `DrawString` method, a member of the `Graphics`[2] class with four available overloaded implementations. (For details on the `Graphics` class, see Chapter 11, Creating Graphical Output.) We start our discussion with the simplest of the overloaded functions, which accepts a pair of (x,y) single-precision floating-point values for the text location.

The simplest version of the `DrawString` method, with five parameters, is defined as follows:

```
public void DrawString(
 string str,
 Font font,
 Brush brText,
 Single x,
 Single y);
```

The first parameter, `str`, identifies the string to draw. While automatic word wrap is not supported, a carriage return within a string draws multiple lines. (In C#, insert a new line with the `\n` character.)

2. `System.Drawing.Graphics`.

Download at WoweBook.Com

The second parameter, `font`, is the font used for drawing the characters. This could be the default font of a control (the `Font` property) or a font that you create.

The third parameter, `brText`, identifies the brush for drawing text foreground pixels; in .NET programming, background pixels are always untouched[3] in text drawing.

The fourth and fifth parameters, `x` and `y`, indicate the text-drawing location. This location is the upper-left corner of the rectangle that bounds the text. These coordinates are single-precision floating-point values, which is different from the integer coordinates used to draw raster and vector graphics.

12.1.3 A Sample Program: `SimpleDrawString`

Our first sample program shows the simplest way to draw in a form. Figure 12.1 shows the program's output. This program uses the form's default font to draw a character string using the system's default window text color.

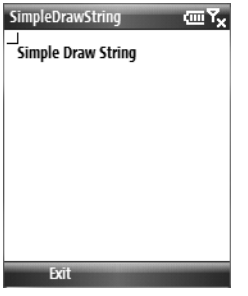


FIGURE 12.1: Output of the `SimpleDrawString` Program

As we discussed in Chapter 11, the `Paint` event plays a key role in any control with graphical output. Listing 12.1 shows the `Paint` event handler method for our simple text-drawing sample. This method is called whenever the window's client area needs drawing.

LISTING 12.1: Fragment from SimpleDrawString.cs Showing the `Paint` Event Handler

```
private void
formMain_Paint(object sender, PaintEventArgs e)
{
   Single xDraw = 10;
   Single yDraw = 10;
```

3. Win32 programmers may recall that the `BackgroundColor` attribute in a device context allows a text-drawing operation to also affect the background pixels.

Download at WoweBook.Com


```
    Brush brText = new SolidBrush(SystemColors.WindowText);
    e.Graphics.DrawString("Simple Draw String", Font, brText,
                          xDraw, yDraw);

    // Highlight origin.
    int x = (int)xDraw;
    int y = (int)yDraw;
    Pen penBlack = new Pen(Color.Black);
    e.Graphics.DrawLine(penBlack, x, y, x-8, y);
    e.Graphics.DrawLine(penBlack, x, y, x, y-8);
}
```

In this sample, the text is drawn at (10,10), the coordinates assigned to the `xDraw` and `yDraw` variables. The second parameter specifies the font to use; in this code, we use the `Font` property from the `formMain` form. In later examples, we show several ways to provide total control over the appearance of text.

Our program calls the `DrawLine` method to draw two lines—one horizontal and one vertical—that intersect at (10,10). The intersection of those two lines shows the default text alignment, at the upper-left corner of the drawn text.

The .NET Compact Framework supports other alignments, as we show later in this chapter. This capability lets you do things such as centering a string over a column of data or centering a text label within a graphical image. This chapter's `TextAlign` sample shows all nine possible alignments.

12.2 Font Selection

We now turn our attention to the subject of fonts. Font selection is the primary way to control text appearance. Windows CE supports two basic font methods: bitmap fonts and TrueType (also known as OpenType, for PostScript compatibility) fonts. A given platform can support one but not both of these font types. The decision about which to support is made by the platform creator and cannot be changed by an application program. Some platforms support bitmap fonts to minimize the size of the operating system image. The benefits of supporting TrueType fonts include the ability to draw scalable fonts and the ability to draw rotated text.

What font support is found on Windows mobile devices? Windows Mobile Professional (Pocket PC) supports TrueType fonts, and has these built in: Tahoma, Courier New and Bookdings. Windows Mobile Standard

Download at WoweBook.Com

(Smartphone) also supports TrueType fonts and has a single built-in font: Segoe Condensed.

12.2.1 The `Font` Property of Controls

The built-in controls use fonts. As shown in Figure 12.2, you select a control's font at design time in the Properties window of the Designer provided as part of Visual Studio. A word of caution is in order: The Designer cannot know what fonts are installed on your Windows Mobile device, so it provides a best guess. For example, when using an SDK for a Smartphone, you will see a choice of two fonts: Nina and Segoe Condensed. This is a pretty good guess, because Nina is the name of the font on older Smartphones, and the other name is on the newer ones.

However, a given device might have more fonts than these. You can, for example, add fonts to a Windows Mobile device by copying font files to the \Windows\Fonts folder. If you install new fonts when you install your application, you can list those for use in a control. Just remember that Visual Studio has no way to know which specific fonts you have installed.

Font support in Windows CE—as in desktop Windows—uses a process that is sometimes called font mapping. This process takes a request for a font and tries to map that request to a font that is actually present in the system. It works best when the font name or font family is present and when the size you request is available. It fails when either of these conditions is not met; in such situations this feature is sometimes called font mangling. The solution is to make sure you request font names and sizes that are available on the device.

As shown in the `SimpleDrawString` sample program, you can draw in a form or in a control by accessing the `Font` property of the form or control.

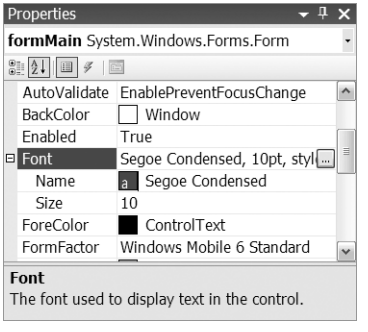


FIGURE 12.2: Selecting Control Fonts in the Visual Studio Forms Designer

Download at WoweBook.Com

Here is the line of code that shows this being done:

```
e.Graphics.DrawString("Simple Draw String", Font, brText,
xDraw, yDraw);
```

The `Font` property is a read/write property, so you can modify a control's font at runtime using any of the font creation techniques discussed in this chapter.

.NET Compact Framework Controls Do Not Support Ambient Properties

Programmers familiar with the desktop .NET Framework might notice that .NET Compact Framework controls do not support the concept of ambient properties. On the desktop, controls use the ambient properties from the parent control if that property is not specified for the control itself. The following properties of controls in the desktop .NET Framework are ambient properties: `BackColor`, `Cursor`, `Font`, and `ForeColor`.

The built-in .NET Compact Framework controls do not support ambient properties. This means that, for example, if you want to select a new font for a group of controls, you cannot simply change the selected font in the parent for the group of controls. Instead, you must change the font for each and every individual control.

12.2.2 Enumerating Fonts

You can create a font by name, just by specifying the name in a `Font` constructor. Get a list of available fonts by enumerating the members of the `InstalledFontCollection` object. We illustrate font enumeration in our `FontPicker` sample.

12.2.3 A Sample Program: `FontPicker`

Figure 12.3 shows the `FontPicker` sample program. This program lets you select a font by specifying a face name, a desired size, and (by means of a menu selection) a font style. A `TextBox` control displays text using the resulting font.

Listing 12.2, taken from the `FontPicker` sample, shows how to create a font and attach it to a text box control.

Download at WoweBook.Com

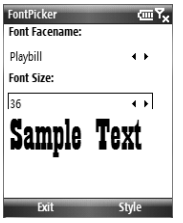


FIGURE 12.3: Output of `FontPicker` Program with a Sample of an Available Font

LISTING 12.2: Font Creation Code from `FontPicker` Sample

```
private void SetFont()
{
    int iFont = cboxFont.SelectedIndex;
    int iSize = cboxSize.SelectedIndex;
    if (iFont != -1 && iSize != -1)
    {
        string strFont = (string)cboxFont.Items[iFont];
        string strSize = (string)cboxSize.Items[iSize];
        iSize = int.Parse(strSize);

        FontStyle fs;
        if (m_FontStyle == "Regular")
            fs = FontStyle.Regular;
        else if (m_FontStyle == "Bold")
            fs = FontStyle.Bold;
        else if (m_FontStyle == "Italic")
            fs = FontStyle.Italic;
        else if (m_FontStyle == "Strikeout")
            fs = FontStyle.Strikeout;
        else
            fs = FontStyle.Underline;

        // Create a new font.
        Font font = new Font(strFont, (float)iSize, fs);

        // Connect font to textbox
        textSample.Font = font;
    }
}
```

12.2.4 A Sample Program: `RotateText`

The `RotateText` sample draws rotated text by creating rotated fonts. This program has several output styles, as illustrated in Figures 12.4 and 12.5. Figure 12.4 shows a single line of text rotated at 45 degrees. Figure 12.5 shows a fan blade—a series of rotated text drawings that spread out from the same drawing point.

Download at WoweBook.Com

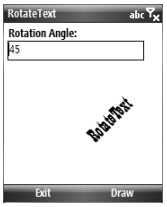


FIGURE 12.4: A Single Line of Rotated Text

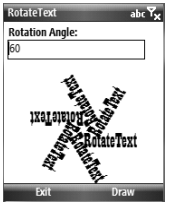


FIGURE 12.5: A Fan Blade of Rotated Text

Font rotation is made possible using TrueType fonts, which are currently the standard for all Windows Mobile devices including standard (Smartphone) and professional (Pocket PC) editions.

Listing 12.3 shows a fragment taken from the `RotateText` program, including the `Paint` event handler and other core pieces used to draw the rotated text.

LISTING 12.3: Fragment from `RotateText` with the Elements to Support Rotated Text

```
public partial class formMain : Form
{
    float m_xDraw;
    float m_yDraw;
    int m_degTextRotate;
    bool m_bDrawFanBlade;

    public formMain()
    {
        InitializeComponent();

        // Calculate center of client area.
        m_xDraw = (float)ClientRectangle.Right / 2;
        m_yDraw = (float)(2*ClientRectangle.Bottom / 3);

        // Initialize degree of rotation.
        m_degTextRotate = 0;

        // Initialize drawing flag.
        m_bDrawFanBlade = false;
    }
```

Download at WoweBook.Com


```
private void
mitemExit_Click(object sender, EventArgs e)
{
    Close();
}

private Font
CreateRotatedFont(string strFaceName, int PointSize,
                  int degRotation, float pixResolution)
{
    // Convert point size to pixel height.
    float flHeight = ((float)PointSize * pixResolution)
                     / 72.0F;
    int pixHeight = (int)(flHeight + 0.5);

    // Set height negative for "Em-Height" (versus
    // "character-cell height" for positive size)
    pixHeight = pixHeight * (-1);

    // Fill in LOGFONT structure.
    var lf = new LogFont();
    lf.CharSet = LogFontCharSet.Default;
    lf.ClipPrecision = LogFontClipPrecision.Default;
    lf.Escapement = 10 * degRotation;
    lf.FaceName = "";
    lf.Height = pixHeight;
    lf.Italic = 0;
    lf.Orientation = 0;
    lf.OutPrecision = LogFontPrecision.Default;
    lf.PitchAndFamily = LogFontPitchAndFamily.Default;
    lf.Quality = LogFontQuality.Default;
    lf.StrikeOut = 0;
    lf.Underline = 0;
    lf.Weight = LogFontWeight.Normal;
    lf.Width = 0;

    return Font.FromLogFont(lf);
}

private void
formMain_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    // Create black brush.
    Brush brBlack = new SolidBrush(Color.Black);

    // Create StringFormat for text drawing details.
    var sf = new StringFormat();
    sf.FormatFlags = StringFormatFlags.NoClip;
```

Download at WoweBook.Com


```
    if (!m_bDrawFanBlade)  // Normal drawing.
    {
        // Create font for drawing.
        Font font = CreateRotatedFont("", 18,
                    m_degTextRotate, g.DpiY);

        g.DrawString(this.Text, font, brBlack, m_xDraw,
                     m_yDraw, sf);
    }
    else
    {
        int cIncrement;
        cIncrement = (m_degTextRotate == 0) ?
                     20 : m_degTextRotate;
        for (int i = 0; i < 360; i += cIncrement)
        {
            // Create font for drawing.
            Font font = CreateRotatedFont("", 18, i, g.DpiY);

            g.DrawString(this.Text, font, brBlack, m_xDraw,
                         m_yDraw, sf);
        }
    }
}

private void
textAngle_TextChanged(object sender, EventArgs e)
{
    int    angle;
    try
    {
        angle = int.Parse(textAngle.Text);
    }
    catch
    {
        MessageBox.Show("Integers Only", "RotateText");
        textAngle.Text = m_degTextRotate.ToString();
        return;
    }
    m_degTextRotate = angle;
}

private void
mitemRedraw_Click(object sender, EventArgs e)
{
    Invalidate();
}

private void
mitemFanBlade_Click(object sender, EventArgs e)
```

Download at WoweBook.Com


```
    {

        // Set drawing flag.
        m_bDrawFanBlade = true;

        // Force redraw.
        Invalidate();
        Update();

        // Reset drawing flag.
        m_bDrawFanBlade = false;
    }

    private void
    mitemAnimate_Click(object sender, EventArgs e)
    {
        // Store current value.
        int degTextRotateTemp = m_degTextRotate;

        // Use degrees as rotational increment.
        int cIncrement;
        cIncrement = (m_degTextRotate == 0) ?
                      20 : m_degTextRotate;

        for (int i = 0; i < 360; i += cIncrement)
        {
            m_degTextRotate = i;
            Invalidate();
            Update();
        }

        // Reset current value.
        m_degTextRotate = degTextRotateTemp;
        Invalidate();
        Update();
    }

} // Class
```

12.3 Placing Text

In a graphical world, text is treated as a graphical object. This has some nice benefits. You can freely mix text and graphical images together. You can also use several fonts together to create a richer style of output. The drawback to this added power is that some effort is required to coordinate the placement of the various graphical objects in the available drawing area. By contrast, a console-style application can send its text to an output

Download at WoweBook.Com

window, where automatic word wrapping and automatic scrolling take the worry out of creating text output. In a graphical environment, you—the programmer—must measure the space that text occupies. The reward for this effort is total control in text placement.

Even if you are drawing only text, with no other kind of graphical object, you still have some work to do. For example, drawing multiple lines of text requires you to calculate the height of each line of text so that you can position each line the correct distance from the previous line. For this and other text-measuring tasks, the .NET Compact Framework provides the `MeasureString` method, which resides in the `Graphics` class. We discuss this next.

12.3.1 Text Size and the `MeasureString` Method

Most people have looked at text their entire lives. Yet, using text to read and write is not the same as managing the display of graphical text. If you look closely at individual characters—for example, the letter *G*—you see a very complicated piece of artwork that a font designer spent many hours perfecting. To efficiently deal with text without getting lost in each glyph, graphics programmers simplify a text string.[4]

We handle graphical text in terms of an imaginary bounding box that surrounds the text string. To place graphical text, we organize the placement of the bounding box within the drawing area, relative to other text and nontext graphical objects. This simplification is the basis of all efforts to measure the size of text in a graphics system.

The `MeasureString` method returns the width and height of a text string by providing a `SizeF` structure. The `F` at the end of the structure name indicates that the structure contains floating-point values (a Hungarian-like suffix). For our purposes, `SizeF` has two interesting properties:

```
public float Width {get; set;}
public float Height {get; set;}
```

The `MeasureString` method returns the height and width for a specified string drawn with a given font. To calculate the bounding box,

4. A *glyph* is the graphical image of a character. Display and printer driver writers, among others, use this term to make sure that the output—the mapping from the character in a font on a specific device—looks correct. Two images of the letter *A*—one drawn with 10-point Arial and the other with 12-point Arial—represent two glyphs but the same character code.

Download at WoweBook.Com

we specify both the character string and the font to be used in drawing the string. Here is the definition of the `MeasureString` method:

```
public SizeF MeasureString(
 string text,
 Font font);
```

12.3.2 A Sample Program: `MeasureString`

The `MeasureString` method is simple, so the accompanying sample program is also simple. The greatest challenge is that the bounding box size is always provided as floating-point values. You must create conversion logic if your text-drawing code interacts with other code that uses integer coordinates. Among other things, this includes mouse input coordinates and the coordinates for drawing vector or raster graphics.

To make our sample more interesting, we add font selection logic from the `FontPicker` sample that appeared earlier in this chapter. A text box allows the user to type in some text. As shown in Figure 12.6, the sample draws white text on a black background to highlight the size and placement of the bounding box. We also went a little crazy with some line drawing to illustrate the relationship between the reported height and width and the resulting text. Listing 12.4 provides a fragment of the code in the `Paint` event handler for measuring and drawing the string.

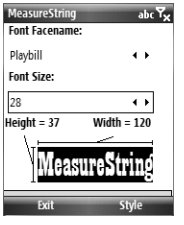


FIGURE 12.6: `MeasureString` **Illustrating Exactly What Is Being Measured**

LISTING 12.4: Fragment Showing the Call to the `MeasureString` Method

```
string str = this.Text;
Brush brFore = new SolidBrush(SystemColors.Window);
Brush brBack = new SolidBrush(SystemColors.WindowText);

// Upper-left corner of text box.
float sinX = (float)(labelWidth.Right/5);
float sinY = (float)(labelHeight.Bottom + 12);

SizeF szf = e.Graphics.MeasureString(str, Font);
```

Download at WoweBook.Com


```
// Draw rectangle in text background.
int xRect = (int)sinX;
int yRect = (int)sinY;
int cxRect = (int)szf.Width;
int cyRect = (int)szf.Height;
Rectangle rc;
rc = new Rectangle(xRect, yRect, cxRect, cyRect);

e.Graphics.FillRectangle(brBack, rc);

// Draw string.
e.Graphics.DrawString(str, Font, brFore, sinX, sinY);
```

The bounding box for the text is visible in Figure 12.6. When we specify the location for the text, the drawing coordinate in the `DrawString` method corresponds with the upper-left corner of the bounding box. Our next sample program, `TextAlign`, shows how to get eight other alignments.

You might notice that the top of the bounding box contains a blank horizontal area. In the world of digital typography, this is referred to as the internal leading. The term *leading* comes from typesetting as done in the predigital era when cubes holding character images had to be individually placed to form lines of text. Using this printing technology, a typesetter placed a long, thin piece of metal—known as the leading—between the rows of letters to provide the interline spacing. In the world of digital fonts, leading is split into two parts: the internal leading and the external leading. These two names indicate whether or not the leading is included in the reported height.

The interline spacing that the `MeasureString` method reports works reasonably well in most cases, especially given the small screen size of a typical device such as a Pocket PC that runs .NET Compact Framework programs. Programmers who work in depth with digital font technology may notice that absolute precision in text placement requires an additional element, the external leading value. For many TrueType fonts at smaller sizes, the external leading is zero and so can often be ignored.

12.3.3 Text Alignment

The .NET Compact Framework supports four overloads to the `DrawString` method. One overload positions the text with an (x,y) coordinate pair. We use this overload in all the samples presented up to this point. The second overload positions the text within a rectangle, and it performs word wrapping of the text and clipping to the rectangle. We consider this overload later in this

Download at WoweBook.Com

chapter. For the purposes of this discussion, we consider the simpler of the two functions—that is, the function that involves a coordinate pair.

When text is drawn with the coordinate-pair version of the `DrawString` method, the text is drawn below and to the right of the point. In other words, the default text alignment is to the top and left of the text box. For example, in Figure 12.7, the text coordinate is the intersection of the vertical and horizontal lines.

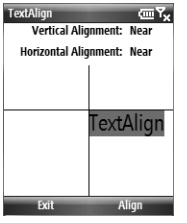


FIGURE 12.7: Default Text Alignment

This approach has been part of every version of Windows since Microsoft Windows 1.01 shipped in 1985. It is likely an outgrowth of the fact that the default origin of a drawing surface is the upper-left corner, which no doubt results from the convention in European languages of reading and writing from left to right and from top to bottom.[5]

Whatever the origin, the default alignment is not always the most convenient. This alignment does not help when you try to center some text within an object (e.g., a rectangle or a window). Nor does this default alignment help center text over a column of numbers.

The .NET Compact Framework does not provide automatic text alignment. For other alignments, you calculate an offset from the desired point to the upper-left corner of the text box.

12.3.4 A Sample Program: `TextAlign`

The `TextAlign` sample program lets you visualize the nine possible text alignments. There are nine possible alignments because there are three vertical alignment values (near, center, and far) and three horizontal alignment values (near, center, and far). The default for text alignment, shown in Figure 12.8, is `near` and `near`, which corresponds to the text drawn with the upper-left corner of the text box at the reference point indicated in the `DrawString` call.

5. Not every language follows this convention. Some languages are read from right to left.

Download at WoweBook.Com

There is an alternative to fussing with these alignment values: You can call one of the versions of the `DrawString` function which accepts the bounding rectangle as input. It is nice to have choices, which is a key benefit of having overloaded functions in an object-oriented programming language.

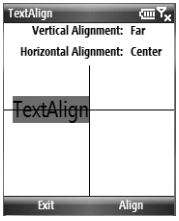


FIGURE 12.8: One of Nine Possible Text Alignments

Listing 12.5 shows the code for the `TextAlign` sample program.

LISTING 12.5: `TextAlign` Program Showing How to Support Nine Text Alignment Settings

```
public partial class formMain : Form
    {
    // Text alignment flags.
    StringAlignment valign;
    StringAlignment halign;

    // Adjustment for bounding box.
    private int m_cxAdjust = 0;
    private int m_cyAdjust = 0;

    public formMain()
    {
        InitializeComponent();

        // Set initial alignment values.
        valign = StringAlignment.Near;
        halign = StringAlignment.Near;
    }

    private void mitemExit_Click(object sender, EventArgs e)
    {
        Close();
    }

    private void
    mitemAlignHorizontal(object sender, EventArgs e)
    {
        var mi = (MenuItem)sender;
```

Download at WoweBook.Com


```
        // Set label to menu name.
        labHorz.Text = mi.Text;

        // Calculate size of string bounding box.
        Graphics g = CreateGraphics();
        SizeF size = g.MeasureString(this.Text, this.Font);
        g.Dispose();

        // Set flag and
        // Calculate bounding box offset.
        if (mi.Text == "Near")
        {
            halign = StringAlignment.Near;
            m_cyAdjust = 0;
        }
        else if (mi.Text == "Center")
        {
            halign = StringAlignment.Center;
            m_cyAdjust = (int)(size.Height / 2);
        }
        else if (mi.Text == "Far")
        {
            halign = StringAlignment.Far;
            m_cyAdjust = (int)size.Height;
        }

        // Force a redraw.
        Invalidate();
    }

    private void
    mitemAlignVertical(object sender, EventArgs e)
    {
        var mi = (MenuItem)sender;

        // Set label to menu name.
        labVert.Text = mi.Text;

        // Calculate size of string bounding box.
        Graphics g = CreateGraphics();
        SizeF size = g.MeasureString(this.Text, this.Font);
        g.Dispose();

        // Set flag and
        // Calculate bounding box offset.
        if (mi.Text == "Near")
        {
            valign = StringAlignment.Near;
            m_cxAdjust = 0;
        }
```

Download at WoweBook.Com


```
    else if (mi.Text == "Center")
    {
        valign = StringAlignment.Center;
        m_cxAdjust = (int)(size.Width / 2);
    }
    else if (mi.Text == "Far")
    {
        valign = StringAlignment.Far;
        m_cxAdjust = (int)size.Width;
    }

    // Force a redraw.
    Invalidate();
}

private void
formMain_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    // Calculate reference point for text drawing.
    float xDraw = (float)(ClientRectangle.Right / 2);
    float yDraw = (float)(ClientRectangle.Bottom / 2);

    // Create drawing objects.
    Brush brBlack = new SolidBrush(Color.Black);
    Pen penBlack = new Pen(Color.Black);

    // Create StringFormat for text drawing details.
    var sf = new StringFormat();
    sf.FormatFlags = StringFormatFlags.NoClip;
    sf.Alignment   = valign;
    sf.LineAlignment = halign;

    // Calculate size of string bounding box.
    SizeF size = g.MeasureString(this.Text, Font);
    int cxWidth = (int)size.Width;
    int cyHeight = (int)size.Height;

    // Adjust values to accommodate alignment request.
    float sinTextX = (float)(xDraw - m_cxAdjust);
    float sinTextY = (float)(yDraw - m_cyAdjust);

    // Draw text bounding box.
    Brush hbrFill = new SolidBrush(Color.Gray);
    Rectangle rc = new Rectangle((int)sinTextX,
                                 (int)sinTextY,
                                 cxWidth,
                                 cyHeight);
    g.FillRectangle(hbrFill, rc);
```

Download at WoweBook.Com


```
// Draw text string.
g.DrawString(this.Text, Font, brBlack, xDraw,
             yDraw, sf);

// Draw reference cross-hairs.
g.DrawLine(penBlack, (int)xDraw, 0, (int)xDraw,
           this.Height);
g.DrawLine(penBlack, 0, (int)yDraw, this.Width,
          (int)yDraw); }
```

12.4 Conclusion

As this chapter showed, a graphical programming environment gives you a lot of control over drawing text. Most of the control for text drawing comes from creating a font, an abstraction that refers to more than just the size and face name of the drawn text. It also refers to the style of the text, as well as the rotation. Once you select a font, you have a great degree of control in how and where the text is placed. You have the freedom—and the associated challenge—of placing text with pixel-level resolution. Such control means that your Compact Framework programs can create whatever visual effects you need from your text output.

Download at WoweBook.Com

A

Hungarian Notation for .NET Programs

Version 1.3
March 15, 2004
By Paul Yao and David Durant

TIME IS a software developer's most precious resource. For that reason, anything that can help you save time—at any phase of the project—while maintaining or increasing productivity is a good thing. When we started to work on this book,[1] we began by collaborating on a set of guidelines to minimize the differences between our coding styles. As the project moved forward, we refined and updated various elements of this file.

We offer this set to you as a starting point for your own coding guidelines. To get an editable copy of this file in Microsoft Word format, e-mail info@paulyao.com. Be sure to ask for the Hungarian Notation for .NET Programs file.

We expect to continue updating the various elements of this file and maintaining it on our Web site (www.paulyao.com/cfbook), and we welcome your comments and suggestions. We particularly welcome any additions for types not included in this document; if you develop extensions to

1. This appendix first appeared on the book review Web site for the following two titles by Paul Yao and David Durant: *.NET Compact Framework Programming with C#* and *.NET Compact Framework Programming with Visual Basic .NET* (Boston: Addison-Wesley, 2004).

Download at WoweBook.Com

the set described here, please send them to us! We will update the document and post it to our Web site (with, of course, credit given to contributors).

A.1 Goals and Objectives

Our primary goal in creating this set of Hungarian names is to increase efficiency and productivity, for both the developer writing the code and—most importantly—all future developers who need to read, update, maintain, and debug code. Toward that end, we created these guidelines that would meet the following goals.

- Make it easy to create variable names.
- Make it easy to write code that is consistent from one developer to another.
- Enhance code readability by attaching type-specific tags to names so that a variable makes sense on its own, without the need to search for its definition.
- Create names for program elements—classes, properties, methods, events, and fields—that help distinguish program-specific names from names that come from the .NET Compact Framework.
- Work equally well in both C# and Visual Basic .NET.
- Work equally well in .NET Compact Framework code as well as desktop .NET Framework code.

A.2 Guidelines

We created these guidelines for a coding project that exclusively uses Microsoft's .NET technology: the C# and Visual Basic .NET languages. Thus, these recommendations apply specifically to .NET projects. These guidelines were created with the following principles in mind.

- *Use Hungarian notation.* This style of naming is officially out of favor, but we have always liked it. In addition, Hungarian notation seems to sneak back through the back door of the official documentation, in sample code, and in some of the code automatically generated by the various wizards. For that reason, we decided to throw open the front door and embrace a style that has served us well in the past.

Download at WoweBook.Com

- *Create case-insensitive names.* C# is case-sensitive, but Visual Basic .NET is not. For that reason, we avoided any convention that relied on distinguishing uppercase from lowercase names. The exceptions to this rule are the names for constants and for members of enumerations, which we decided to make all uppercase.
- *Use default names when possible.* The Visual Studio .NET Forms Designer suggests names, some of which can be left alone but others which are likely to change. For example, the Designer gives label controls names such as `label1`, `label2`, `label3`, and so on. In many cases, programmers have no interest in changing these. As much as possible, the naming scheme described here works with the default names provided by the Designer.
- *Include elements in the .NET Compact Framework.* The focus of this book is the .NET Compact Framework, so we attempt to make this as complete as possible for the classes that are in the .NET Compact Framework, especially the more commonly used classes. Programmers targeting the .NET Framework may still find this list useful, but many .NET Framework types are not defined here. You are welcome, however, to use this as a starting point for your own guidelines.
- *Avoid terse abbreviations for prefixes and variable names.* Instead, in general we chose to make prefixes and variable names more meaningful. While terse names are easier to type, they are somewhat harder to remember. For example, the prefix for `TextBox` is `tbox` rather than the shorter but less understandable `txt`.
- *Use terse abbreviations for commonly used types.* There is a bit of a judgment call for this guideline because our commonly used types might differ from yours. For strings (`str`), integer loop indexes (`i`), and events (`e`), it seems that shorter is better; these abbreviations become easy to remember because they are used so often.
- *Use longer abbreviations (or no abbreviation) for less commonly used types.* Again, this involves a judgment call. Operating system threads (`thread`) are arguably common objects that everyone needs and uses, but the creation and management of those threads in code are done by a relatively small amount of code. The savings to be gained by some arbitrary abbreviation—such as `thrd`, `thd`, or `thred`—would not offset the clarity of just using the longer `thread`. (In other words, we avoid abbreviating for the sake of abbreviating for less common types.)

Download at WoweBook.Com

- *Do not try to make every single instance in all code exactly consistent with a standard.* The goal is to increase your productivity, which does not occur if you rework all your old code to match some change in your standard. For that reason, there are inconsistencies with this set of definitions and what you find in this printed book and in the online code. So be it. Consistency within a single program is far more important than consistency across different programs. (For example, with the `TextBox` control prefix, we waffled between using `txt` and `text`, finally settling on `tbox`—and this could even change again.)

A.3 .NET Naming Guidelines

The Microsoft .NET documentation provides some guidelines for .NET naming. Of particular interest is a section titled "Design Guidelines for Class Library Developers."[2] This strict set of rules helps ensure that new class libraries—custom controls, SQL data adapters, and so on—are easily understood by programmers who are already familiar with .NET conventions.

Rather than provide the copious detail of those design guidelines, we have summarized the key points in Table A.1. This book follows most of these guidelines for classes that we expect to be public; for our private code, we use more Hungarian notation than these guidelines allow. These guidelines can help you write public class library code that is generally more readable and maintainable by you and more usable and understandable by those programmers who use your class library.

TABLE A.1: Microsoft Guidelines for .NET Naming

Type	Convention	Examples
Event handlers	Use the `EventHandler` suffix for event handlers	Examples: `TreeViewEventHandler`, `KeyEventHandler`
Event handling methods	Name protected methods as `OnXXX`, where `XXX` is the event name	Examples: `OnParentChanged`, `OnRowUpdated`, `OnKeyDown`, `OnMouseMove`

2. See details at http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnamingguidelines.asp.

Download at WoweBook.Com

TABLE A.1: Microsoft Guidelines for .NET Naming *(Continued)*

Type	Convention	Examples
Event names	Use verbs for event names	Examples: `ParentChanged`, `EnabledChanged`, `GotFocus`, `LostFocus`
Exceptions	Use the `Exception` suffix for exception names	Examples: `ArithmeticException`, `DivideByZeroException`, `IndexOutOfRangeException`
Interfaces	Use the `I` prefix for interface names. Use nouns, noun phrases, or adjectives that describe behavior for the rest of the name	Examples: `ICollection`, `IComparer`, `IDictionary`, `IEnumerable`, `IEnumerator`, `IList`
Methods	Use verbs or verb phrases for method names	Examples: `Activate`, `Close`, `Invoke`, `ShowDialog`
Method parameters	Use descriptive parameter names. Do not use Hungarian notation. Use type-based parameter names sparingly	Examples: `bool disposing`, `object sender`
Contained Elements		
Properties	Use nouns or noun phrases for property names. Do not use Hungarian notation	Examples: `BackColor` `Font` `Text` `Width`
Static fields	Use nouns, noun phrases, or noun abbreviations for static field names. Use Hungarian notation	Examples: `Empty` `NotDefinedValue`

Hungarian notation has fallen out of favor at Microsoft, so the official word in the .NET community is to avoid Hungarian. Some .NET programmers—ourselves included—continue to use this style; others do not. Feelings run deep on this issue. In a Web log entry for November 14, 2003,[3] a reader of a Web-based copy of this book, Paul Gielens, asked for feedback

3. You can find the Web log entry, titled "What? Hungarian naming for .NET," at this URL: http://weblogs.asp.net/pgielens/archive/2003/11/14/37566.aspx.

Download at WoweBook.Com

from .NET programmers about whether they use .NET naming. The majority admitted to using Hungarian notation at least some of the time.

You will encounter programmers who advise against using Hungarian. Others advise using it for private names but sticking to the Microsoft guidelines for public symbols. Others use the notation only for user interface controls. You decide: Pick what works for you and your development team, always keeping in mind that the purpose of this notation is to save you time and energy. Where it accomplishes that end, use the notation; where it does not, don't use the notation.

A.4 Hungarian Notation

Hungarian notation—often referred to as *Hungarian naming*—refers to a style of creating meaningful variable names. An MSDN Technical Article gives the following explanation for how this approach got its name:

> It came to be known as "Hungarian notation" because the prefixes make the variable names look a bit as though they're written in some non-English language and because Simonyi is originally from Hungary.—Note from Dr. Gui, November 1999

Rather than repeat all the details that are so well described in that technical article, we refer you to a reprint of Charles Simonyi's original white paper on this naming convention if you wish to learn more.[4]

Hungarian notation is not a standard. In other words, you will not find an ISO standards committee (or, for that matter, an ANSI, IEEE, or ECMA committee) that dictates which prefixes you can and cannot use. While there are many lists of suggested prefixes, including this appendix, none claim to be the definitive one that supersedes all others.

Hungarian notation is, instead, a *style.* Lists (such as this one) that call themselves "standards" (or, in our case, "guidelines") are starting points for project-specific data types because, with few exceptions, every programming project has its own unique set of types. You'll get the greatest benefits from Hungarian notation on projects where everyone agrees to use the same set of types. It provides a common language so that any team member can look at any code—whether that individual wrote it or it was written by someone else—and read it, understand it, and modify it with a minimum of extra effort.

4. See http://msdn.microsoft.com/library/en-us/dnvsgen/html/hunganotat.asp.

Download at WoweBook.Com

We have a friend who used to work at Microsoft. He states that the disciplined use of Hungarian notation—and a project-specific set of types and associated Hungarian prefixes—allowed him to transfer to the Excel team during a crunch period and be productive very quickly.

A.4.1 Our Use of Hungarian Notation

We use Hungarian notation throughout this book for data types. We do so because we find it makes our code more readable and therefore more understandable.

The use of Hungarian notation also provides another important benefit: It allows you to quickly find names in an IntelliSense list. IntelliSense is a feature of the Visual Studio .NET text-editing windows. Just about every time you type a dot operator, IntelliSense pops up with possible choices for the next part of an operation.

For example, suppose you are writing some code and need to know the name of a string variable that is a data member of the class you are writing. The old-fashioned way to find the variable name would be to do a search or to scroll to the top of your source file (or wherever you normally put such things). When you get there, you might find a list of string names defined like this:

```
string strFirstName;
string strLastName;
string strAddress;
string strCity;
string strState;
string strPostalCode;
```

Another alternative is to ask IntelliSense to help you. By typing `this.str` in the text-editing window, you give IntelliSense enough details to help you remember the specific name you were looking for. Figure A.1 shows the IntelliSense window displaying the available data members that start with an `str` prefix.

A.4.2 The `m_` Prefix for Private Data

A variation of the Hungarian style that we adopt in this book is the use of an additional prefix, `m_`, for private class data members. This helps us instantly distinguish public data from private data. It is particularly helpful in allowing a public property and an associated private data member to have essentially the same name—the only difference being that the private data member has the `m_` prefix. Here is an example:

Download at WoweBook.Com

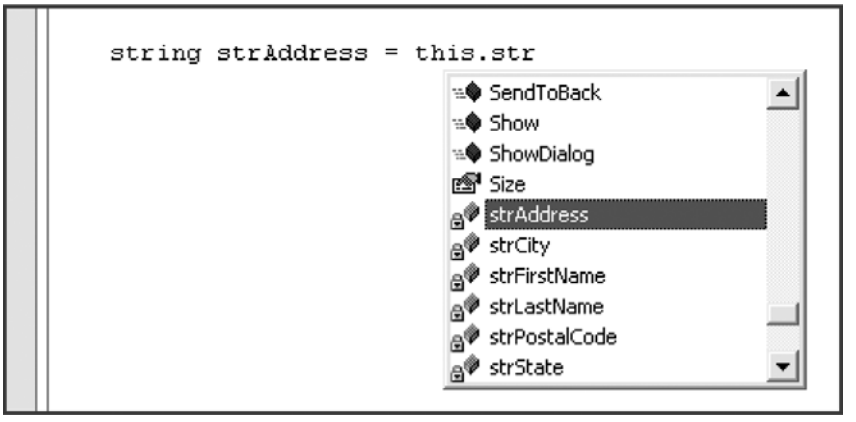


FIGURE A.1: Hungarian Notation Helping IntelliSense Remind You of Variable Names

```
private bool m_bThreadContinue; // Continue flag

public bool bThreadContinue // Continue property
{

 get { return m_bThreadContinue; }
 set { m_bThreadContinue = value; }
}
```

The programmer writing code that calls into this class always uses the public `bThreadContinue` property, which gets translated into the private `m_bThreadContinue` field that is visible only within our class.

The Designer creates some private data members for you—one for each control created in a form. We do not use the `m_` prefix for such private data members because the extra effort would not be worth the potential benefit.

While we were assembling this set of guidelines, the `m_` prefix prompted the most discussion. We both found it useful but wanted to make sure it would be reasonable for .NET Compact Framework code. We found it widely used in pre-.NET technologies, including C++ projects (for the MFC class library and the ATL template library), Visual Basic projects, and Active Server Pages.

What convinced us of its applicability to the .NET Compact Framework was its use in various .NET-based technologies, including Rotor, the shared source code implementation of the Common Language Infrastructure (CLI). We also found that the .NET Compact Framework itself uses `m_` for all of its private fields, none of which are documented but which become visible with a tool such as `ildasm.exe`.

Download at WoweBook.Com

A.4.3 Hungarian Prefixes for CTS Value Types

The common type system (CTS) defines a set of types that all languages must support to be compatible with .NET. The CTS, which we discussed in detail in Chapter 3, is part of a specification known as the Common Language Specification (CLS). C# supports types outside this specification, most of which are unsigned types. In the .NET documentation, these are marked as "not CLS-compliant." To enhance code portability, we use only CLS-compliant types, and so Table A.2 includes only CLS-compliant types.

TABLE A.2: Hungarian Prefixes for Standard Value Types

.NET Class	C# Alias	Visual Basic .NET Alias	Size (Bytes)	Prefix
`System.Boolean`	`bool`	`Boolean`	1	`b`
`System.Char`	`char`	`Char`	2	`ch`
`System.DateTime`	N/A	`Date`	8	`date`
`System.Decimal`	`decimal`	`Decimal`	16	`dec`
Integer Types				
`System.Byte`	`byte`	`Byte`	1	`byt`
`System.Int16`	`short`	`Short`	2	`sh`
`System.Int32`	`Int`	`Integer`	4	`i`
`System.Int64`	`long`	`Long`	8	`l`
Platform-Specific Pointer or Handle Type				
`System.IntPtr`	N/A		4	`iptr`[a]
`Floating-Point Types`				
`System.Single`	`float`	`Single`	4	`sin`
`System.Double`	`double`	`Double`	8	`dbl`
Other Type				
`System.Guid`	N/A		16	`guid`

a. In general, we do not use this prefix for `IntPtr`. Instead, we use the prefix for the type being represented by the `IntPtr` variable. For example, a Win32 window handle will have a prefix of `hwnd` instead of `iptr`. This accomplishes the goal of making our code more readable and reduces the coding errors that arise from misusing variables.

Download at WoweBook.Com

Tables A.3 through A.14 list our recommendations for Hungarian prefixes for various .NET Compact Framework types. Because .NET Compact Framework namespaces, classes, and other types have the same names as their .NET Framework counterparts, this set of Hungarian prefixes can also work for .NET Framework projects.

TABLE A.3: Hungarian Prefixes for `Usage` and `System` Classes

Class or Usage Type	Prefix	Examples
Integer used as a count of bytes	`cb`	`cbReturnBuffer, cbLength`
Integer used as a count of characters	`cch`	`cchMaxName, cchSearchBuffer`
Integer used as an index for a loop	`i`	`iItem, iFont, iFile`

Array of any type	`a` (used prior to type-specific type)	`int [] ai = new int[10]; // Array of integers`
		`ai[0] = 1;`
		`ai[1] = 2;`
		`string [] astrName; // Array of strings`
		`char [] achPassword; // Array of characters`
		`Button [] acmdWizard; // Array of buttons`
		`Control [] actrlSearch; // Array of controls`

`Delegate`	`dele`	`deleFileFound, deleAddItem`
`EventArgs`	`e`	`e` (the standard event argument for all event handlers)
`Exception`	`ex`	`ex` (the standard exception argument for all exception types)
`Object`	`obj`	`objDebugInput, objItem, objCollectionStart`
`String`	`str`	`strPath, strFileName, str`

Download at WoweBook.Com

TABLE A.4: Hungarian Prefixes for `System.Text` Classes

Class	Prefix	Examples
`Decoder`	`decode`	`decodeFileBuffer, decodeHTML`
`Encoder`	`encode`	`encodeDisplayBuffer, encodeConvert`
`StringBuilder`	`strbld`	`strbldInputBuffer, strbldErrorLog`

TABLE A.5: Hungarian Prefixes for `System.Collection` Classes

Class	Prefix	Examples
`ArrayList`	`al<type>`	`ArrayList ali = new ArrayList(); // Integer ArrayList ali.Add(1); ali.Add(2);`
`BitArray`	`bitarr`	`bitarrDisplayFlags, bitarrCollectionType`
`HashTable`	`hash`	`hashFileNames, hashLeftover`
`Queue`	`queue`	`queueSendRequest, queueWaitingList`
`Stack`	`stack`	`stackFunctionCall, stackUndoCommands`

TABLE A.6: Hungarian Prefix for `Microsoft.Win32` Class (Desktop .NET Framework Only)

Class	Prefix	Examples
`RegistryKey`	`regkey`	`regkeyRoot, regkeySoftware, regkeyVer10`

TABLE A.7: Hungarian Prefixes for `System.Windows.Forms` Classes

Class	Prefix	Examples
`Button` (used for command buttons)	`cmd`	`cmdOk` (OK button), `cmdReset` (button to reset a form)
`CheckBox`	`chk`	`chkWordWrap`, `chkToolbar`, `chkCloseWhenDone`
`ComboBox`	`combo`	`comboStates`, `comboTaxSettings`
`ContextMenu`	`cmenu`	`cmenuMain`, `cmenuOptions`
`Control`	`ctrl`	`ctrlParent`
`DataGrid`	`dgrid`	`//a "data-bound grid" dgridPhoneList`

Continues

Download at WoweBook.Com

TABLE A.7: Hungarian Prefixes for `System.Windows.Forms` Classes *(Continued)*

Class	Prefix	Examples
DomainUpDown	dupdown	dupdownState, dupdownCountry
Form	frm	frmMain (main form), frm (temporary variable)
Form used as a dialog box	dlg	dlgFileOpen, dlgFileSave
HScrollBar	hscroll	hscrollLines, hscrollElevation
ImageList	ilist	ilistFile, ilistEdit
Label	label	Label1 (default label name), labelStatus
ListBox	lbox	lboxFileNames
ListView	lview	lviewVisitors, lviewSymptoms
MainMenu	menu	menuMain, menuNew
MenuItem	mitem	mitemFileOpen, mitemEditCopy
NumericUpDown	nupdown	nupdownMaxItems, nupdownPassengers
Panel	panel	panelColorScroll, panelDiscountGroup
PictureBox	pbox	pboxPreview, pboxThumbnail
ProgressBar	pbar	pbarDownload, pbarRecalc
RadioButton	opt	optRed, optGreen, optBlue (opt stands for "option button")
StatusBar	status	statusMain, statusHelpWindow
TabControl	tab	tabFileDialog, tabOptionsDialog
TabPage	tpage	tpageOptions, tpageFormat
TextBox	tbox	tboxName, tboxFirst, tboxPassword
Timer	timer	timerPowerSave
ToolBar	tbar	tbarFile, tbarOptions
ToolBarButton	tbb	tbbFileOpen, tbbEditCopy
TrackBar	trbar	trbarItems
TreeView	tview	tvwDirectory, tvwOrgChart
VScrollBar	vscroll	vscrollRed, vscrollGreen, vscrollBlue

Download at WoweBook.Com

TABLE A.8: Hungarian Prefixes for `Microsoft.WindowsCE.Forms` Classes (.NET Compact Framework Only)

Class	Prefix	Examples
`InputPanel`	`sip`	`sipMain` (`sip` stands for "Software Input Panel")
`Message`	`msg`	`msgIn`
`MessageWindow`	`msgwnd`	`msgwndTextBox`, `msgwndSearchDone`

TABLE A.9: Hungarian Prefixes for `System.Drawing` Classes

Class	Prefix	Examples
`Bitmap`	`bmp`	`bmpThumb, bmpNext`
`Brush`	`br`	`brRed, brGreen, brYellow`
`Color`	`color`	`colorBlack, colorActiveWindow`
`Font`	`font`	`fontCurrent, fontArial`
`Graphics`	`g`	`G` (standard for `Graphics` objects)
`Icon`	`icon`	`iconHelp, iconQuestion, iconMain`
`ImageAttrib-utes`	`imgatt`	`imgattWhiteColorKey, imgattRedColorKey`
`Pen`	`pen`	`penRed, penBlack`
`Point`	`pt`	`ptStart, ptEnd`
`PointF`	`ptf`	`ptfName, ptfString`
`Rectangle`	`rect`	`rectStory`
`RectangleF`	`rectf`	`rectfPreview`
`Region`	`reg`	`regScroll`
`Size`	`sz`	`szBackground`
`SizeF`	`szf`	`szfStringSize`

TABLE A.10: Hungarian Prefixes for `System.Data` Classes

Class	Prefix	Examples
`ConstraintException`	`ex`	`ex`
`DataColumn`	`dcol`	`dcolName, dcolProduct`

Continues

Download at WoweBook.Com

TABLE A.10: Hungarian Prefixes for `System.Data` Classes *(Continued)*

Class	Prefix	Examples
DataColumnChangeEventArgs	e	e
DataException	ex	ex
DataRelation	drel	drelCustomerOrder, drelStudentSchedule
DataRow	drow	drowFirst, drowNext
DataRowChangeEventArgs	e	e
DataSet	dset	dsetCustomer, dsetBugReport
DataTable	dtab	dtabOrder, dtabAddress
DataView	dview	dviewCurrentOrder, dviewProduct
DBConcurrencyException	ex	ex
DeletedRowInaccessibleException	ex	ex
DuplicateNameException	ex	ex
EvaluateException	ex	ex
FillErrorEventArgs	e	e
ForeignKeyConstraint	fkey	fkeyCustomerOrder, fkeyCustomerProducts
InRowChangingEventException	ex	ex
InvalidConstraintException	ex	ex
InvalidExpressionException	ex	ex
MissingPrimaryKeyException	ex	ex
NoNullAllowedException	ex	ex
ReadOnlyException	ex	ex
RowNotInTableException	ex	ex
StateChangeEventArgs	e	e
SyntaxErrorException	ex	ex
VersionNotFoundException	ex	ex

Download at WoweBook.Com

TABLE A.11: Hungarian Prefixes for `System.Data.SqlServerCe` Classes

Class	Prefix	Examples
SqlCeCommand	cmnd	cmndAddOrderItems, cmndSelectNextItem
SqlCeCommandBuilder	cbld	cbldUpdate
SqlCeConnection	conn	connInternalServer, connMain
SqlCeDataAdapter	dapt	daptNewOrders, daptProductList
SqlCeDataReader	drdr	drdrResult
SqlCeEngine	db	db
SqlCeError	err	errLastCommand
SqlCeException	ex	ex

TABLE A.12: Hungarian Prefixes for `System.IO` Classes

Class	Prefix	Examples
BinaryReader	bread	breadPasswordInfo
BinaryWriter	bwrite	bwritePasswordInfo
EndOfStreamException	ex	ex
ErrorEventArgs	e	e
FileInfo	finfo	finfoNew, finfoNext
FileStream	fs	fsInput, fsOutput, fsDebugLog
BinaryWriter	bwrite	bwritePasswordInfo
EndOfStreamException	ex	ex
ErrorEventArgs	e	e
FileInfo	finfo	finfoNew, finfoNext
FileStream	fs	fsInput, fsOutput, fsDebugLog
IOException	ex	ex
MemoryStream	ms	msBitmap, msString
StreamReader	sread	sreadInput
StreamWriter	swrite	swriteOutput

Continues

Download at WoweBook.Com

TABLE A.12: Hungarian Prefixes for `System.IO` Classes *(Continued)*

Class	Prefix	Examples
StringReader	stringread	stringreadInput
StringWriter	stringwrite	stringwriteOutput

TABLE A.13: Hungarian Prefixes for `System.Threading` Classes

Class	Prefix	Examples
AutoResetEvent	autoevent	autoeventStartLogging
ManualResetEvent	manevent	maneventAutoSave
Mutex	mutex	mutexInputBuffer, mutexCheckRealtimeThread
Thread	thread	threadMain, threadNetworkListener, threadCommPortReadWrite, threadPrint
Timer	timer	timerStart

TABLE A.14: Hungarian Prefixes for `System.Xml` Classes

Class	Prefix
NameTable	ntable
XmlAttribute	xattr
XmlAttributeCollection	xattcoll
XmlCDataSection	xcdatasec
XmlCharacterData	xchardata
XmlComment	xcom
XmlDeclaration	xdecl
XmlDocument	xdoc
XmlDocumentFragment	xdfrg
XmlElement	xel
XmlEntityReference	xeref
XmlImplementation	ximp
XmlNamedNodeMap	xnnm

Download at WoweBook.Com

TABLE A.14: Hungarian Prefixes for `System.Xml` Classes *(Continued)*

Class	Prefix
XmlNamespaceManager	xnsmanager
XmlNodeChangedEventArgs	e
XmlNodeReader	xnreader
XmlParserContext	xcontext
XmlProcessingInstruction	xpi
XmlQualifiedName	xqname
XmlResolver	xresolver
XmlSignificantWhitespace	xsigwhite
XmlText	xtext
XmlTextReader	xtr
XmlTextWriter	xtw
XmlUrlResolver	xuresolver
XmlWhitespace	xwhite

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

B

Windows API Allocation and Cleanup Functions

When doing .NET Compact Framework programming, we prefer to write managed code. Sometimes, however, we want to do something—or must do something—that we cannot do with managed code alone. At such times, we call through to the underlying Windows API using the P/Invoke support built into the .NET Compact Framework (see Chapter 3, Platform Invoke).

The Windows API is an object-based programming interface, which is to say that objects are allocated and handles are issued for each object. In managed code, when you create managed objects, the cleanup of memory for those objects is handled for you automatically. This is often referred to as *garbage collection,* although we prefer calling this *automatic garbage collection* to highlight the fact that sometimes *manual garbage collection* is needed.

Manual garbage collection is needed for most Windows API objects that are allocated; but which objects need to be freed up, and how do you do it? We provide our answer in this appendix. In particular, Table B.1 shows the functions that create Windows API objects and the functions that clean up the objects, and it also indicates whether cleanup is required. If nothing else, this table should help programmers appreciate the complexities that the managed classes handle for them. There are more than 40 types of objects in this table and more than twice that number of functions for allocating these objects. Many objects have two or more allocation functions. There are, for example, eight different functions for creating bitmaps.

Download at WoweBook.Com

TABLE B.1: Windows API Objects and Object Support Functions

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?

Accelerators	Keyboard accelerators	`CreateAcceleratorTable`
		`LoadAccelerators`
		`DestroyAcceleratorTable`
		`DestroyAcceleratorTable`
			Yes	Yes

Atoms	String table	`GlobalAddAtom`	`GlobalFreeAtom`	Yes
Bitmap: new with no initialization	Device-dependent bitmap (DDB)	`CreateBitmap` `CreateCompatibleBitmap`	`DeleteObject` `DeleteObject`	Yes
Bitmap: in-memory DIB, with no initialization	Device-independent bitmap (DIB)	`CreateDIBSection`	`DeleteObject`	Yes
Bitmap: from resource	DDB	`LoadBitmap`	`DeleteObject`	Yes
Bitmap: from a file	DDB	`LoadImage` `SHLoadDIBitmap`	`DeleteObject` `DeleteObjec`	Yes Yes
Bitmap: from a GIF	DDB	`SHLoadImageFile` `SHLoadImageResource`	`DeleteObject` `DeleteObject`	Yes Yes

Brush	Used for filling areas	`CreateDIBPatternBrushPt`
		`CreatePatternBrush`
		`CreateSolidBrush`
			`DeleteObject`
			`DeleteObject`
			`DeleteObject`
				Yes
				Yes
				Yes

Caret	Keyboard pointer	`CreateCaret`	`DestroyCaret`	Yes

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: Windows API Objects and Object Support Functions *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?
Clipboard	Used for user-driven data sharing	`OpenClipboard`	`CloseClipboard`	Yes
Critical section	Synchronization object for use in a single process	`InitializeCriticalSection`	`DeleteCriticalSection`	Yes
Cursor	Mouse pointer	`CreateCursor` `LoadCursor`	`DestroyCursor` `DestroyCursor`	Yes Yes

Database	Windows CE–property database	`CeCreateDatabaseEx`
		`CeCreateDatabaseEx` (RAPI version)
		`CeOpenDatabaseEx`
		`CeOpenDatabaseEx` (RAPI version)
			`CloseHandle`
			`CeCloseHandle` (RAPI version)
			`CloseHandle`
			`CeCloseHandle` (RAPI version)
				Yes
				Yes
				Yes
				Yes

Database enumerator	Enumerates Windows CE property databases	`CeFindFirstDatabase` `CeFindFirstDatabase` (RAPI version) `CeFindAllDatabases` (RAPI version)	`CloseHandle` `CeCloseHandle` (RAPI version) `CeRapiFreeBuffer` (RAPI version)	Yes Yes Yes

Device context	Connects graphics devices, including the display screen, printers, bitmaps, and metafiles	`CreateDC`
		`CreateCompatibleDC`
		`GetDC`
		`BeginPaint`
			`DeleteDC`
			`DeleteDC`
			`ReleaseDC`
			`EndPaint`
				Yes
				Yes
				Yes
				Yes

Continues

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: **Windows API Objects and Object Support Functions** *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?

Dialog box	Modeless and modal dialog boxes	`CreateDialog`
		`CreateDialogIndirect`
		`CreateDialogIndirectParam`
		`CreateDialogParam`
		`DialogBox`
		`DialogBoxIndirect`
		`DialogBoxIndirectParam`
		`DialogBoxParam`
			`DestroyWindow`
			`DestroyWindow`
			`DestroyWindow`
			`DestroyWindow`
			`EndDialog`
			`EndDialog`
			`EndDialog`
			`EndDialog`
				Yes
				Yes
				Yes
				Yes
				Yes
				Yes
				Yes
				Yes

Event	Win32 synchronization object	`CreateEvent`	`CloseHandle`	Yes
File (and device)	Connection to file in file system, to installable stream-interface drivers, and to devices (`COM1:` port)	`CreateFile` `CeCreateFile` (RAPI version)	`CloseHandle` `CeCloseHandle` (RAPI version)	Yes Yes
File enumerator	Enumerates files and directories	`FindFirstFile` `CeFindFirstFile` (RAPI version) `CeFindAllFiles` (RAPI version)	`FindClose` `CeFindClose` (RAPI version) `CeRapiFreeBuffer` (RAPI version)	Yes Yes Yes

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: Windows API Objects and Object Support Functions *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?
File-mapping object	Maps the contents of a file to memory; also supports sharing memory between processes	`CreateFileForMapping` `CreateFileMapping`	`CloseHandle` `CloseHandle`	Yes Yes
Font	Used to draw text	`CreateFontIndirect` `AddFontResource`	`DeleteObject` `RemoveFontResource`	Yes Yes
FTP file handle	Connection to a remote file for reading or writing	`FtpOpenFile`	`InternetCloseHandle`	Yes
GDI stock object	Built-in set of fonts, pens, and brushes.	`GetStockObject`	N/A	No
Hook	A mechanism for intercepting message traffic (present but not documented in Windows CE)	`SetWindowsHookEx`	`UnhookWindowsHookEx`	Yes
HTTP connection	Creates a connection to a server using the HTTP protocol	`HttpOpenRequest`	`InternetCloseHandle`	Yes

Continues

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: **Windows API Objects and Object Support Functions** *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?
Icon	Created from two bitmap masks, an AND mask and an XOR mask	`CreateIcon` `LoadIcon`	`DestroyIcon` `DestroyIcon`	Yes Yes
Library	Also called dynamic link library (DLL). DLLs provide the foundation for both native and managed programming interfaces	`LoadLibrary` `LoadLibraryEx`	`FreeLibrary` `FreeLibrary`	Yes Yes
Memory: component	Functions support allocations shared between COM/ ActiveX components	`CoTaskMemAlloc` `CoTaskMemRealloc`	`CoTaskMemFree` `CoTaskMemFree`	Yes Yes
Memory: heap	A Windows CE process can contain one or more heaps	`LocalAlloc` `malloc`	`LocalFree` `free`	Yes Yes

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: Windows API Objects and Object Support Functions *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?
Memory: page	Functions allocate pages; are the foundation for all other allocators	`VirtualAlloc` `MapViewOfFile`	`VirtualFree` `UnmapViewOfFile`	Yes Yes

Menu	Two types of menus: program and pop-up menus	`CreateMenu`
		`CreatePopupMenu`
		`LoadMenu`
			`DestroyMenu`
			`DestroyMenu`
			`DestroyMenu`
				No
				No
				No

Metafile	A mechanism to record and play back GDI drawing calls (present but not documented in Windows CE)	`CreateEnhMetaFile`	`DeleteEnhMetaFile`	Yes
Mutex	A type of Win32 synchronization object	`CreateMutex`	`CloseHandle`	Yes
Palette	Color table for use with bitmaps and for updating the set of colors displayed by a device	`CreatePalette`	`DeleteObject`	Yes
Pen	Used for drawing lines	`CreatePen` `CreatePenIndirect`	`DeleteObject` `DeleteObject`	Yes Yes

Continues

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: **Windows API Objects and Object Support Functions** *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?
Process	Operating system process—maximum of 32 in Windows CE. Closing a process handle does not terminate the process	`CreateProcess` `CeCreateProcess` (RAPI version)	`CloseHandle` `CeCloseHandle` (RAPI version)	Yes
RAPI connection	Links a desktop program to a Windows CE smart device	`CeRapiInit` `CeRapiIniEx`	`CeRapiUninit` `CeRapiUninit`	Yes Yes
Region	Defines the clipping areas for graphical output	`CreateRectRgn` `CreateRectRgnIndirect`	`DeleteObject` `DeleteObject`	Yes Yes
Registry key	Needed to access the registry (see Chapter 11)	`RegOpenKeyEx` `RegCreateKeyEx`	`RegCloseKey`	Yes
Resource	A read-only data object mapped to a program's address space; these functions access native resources, not managed resources	`FindResource` `LoadResource`	N/A N/A	No No

From the Library of Lee Bogdanoff

Download at WoweBook.Com

TABLE B.1: Windows API Objects and Object Support Functions *(Continued)*

Object	Comment	Win32 Allocation Function	Win32 Cleanup Function	Cleanup Required?
Semaphore	A type of Win32 synchronization object	`CreateSemaphore`	`CloseHandle`	Yes
Socket	Part of a network API	`socket`	`closesocket`	Yes
String resource	Native string tables support localization and delayed loading of string data	`LoadString`	N/A	No
Thread	The unit of scheduling in the operating system; after starting a thread, must close handle to allow object cleanup	`CreateThread`	`CloseHandle`	Yes
Timer	Low-resolution timer oriented to the user interface	`SetTimer`	`KillTimer`	No
Window	Visible user interface object, dialog box, dialog box control, and so on	`CreateWindow` `CreateWindowEx`	`DestroyWindow` `DestroyWindow`	Yes Yes

From the Library of Lee Bogdanoff

Download at WoweBook.Com

This page intentionally left blank

Download at WoweBook.Com

Index

A

ActiveSync
  connection detection by, 499–501
  P/Invoke and, 61
  Remote API (RAPI) and, 181, 451–458
  SQL Server CE and, 389
  for synchronization, 226
  TCP/IP connection for, 386
  wrapper library of, 462, 512
ActiveX controls, 18
Add Hardware Wizard, 528
AddProgramsInDirectory function, 484–486
address of service, in WCF, 518
AddSubscription method, 419
Adobe Flash Lite, 5, 18
ADO.NET, 223–335
  change tracking in, 357–358
  classes of, 229–233
  data sets of, 236–254
    creating and accessing, 237–240
    XML for reading and writing, 252–254
  data sets of, data binding in, 240–252
    bound data table updating in, 251–252
    to multi-item controls, 244–245
    overview, 240–244
    rows controls assigned to, 250
    rows displayed in, 246–250
    to single-item controls, 245–246
  error handling in, 234–235
  layered approach of, 227–229
  in LINQ sample application, 345, 355
  overview, 223–226
  for relational data handling, 16
  Remote Data Access (RDA) and, 388
  SQL Server CE in, 254–287
    CE database created for, 261–263
    CE database populated for, 263–266
    CE files in, 255–256
    CE query analyzer in, 259–261
    CE syntax in, 256–259
    schema queries in, 282–287
    SqlCEDataAdapter class for, 276–282
    SqlCEDataReader class for, 266–275
    updating, 275–276
  SQL Server in, 287–318
    command objects for, 297
    connecting to, 289–297
    overview, 287–289
    stored procedures of
      concurrency and, 307
      with DataSet objects, 303–307
      using, 297–303
    typed data sets of, 310–318

Download at WoweBook.Com

ADO.NET *(Continued)*
SqlCeRemoteDataAccessclassin,398
typed data sets of, 431–432
Web Services in, 318–335
client application for, 331–335
sample application for, 321–331
XML, XSD, and SOAP for, 319–321
Advise Sink, for connection detection, 501, 505
Agents
SQL Server CE Server, 390–393, 395–396
Synchronize method of, 440
alignment, of text, 623, 635–636
allocated object map, 25
allocation and cleanup functions, 659–667
Amazon.com, 6
ambient properties, 626
ampere units, in battery ratings, 45
Anonymous authentication, 392
anonymous function, in C#, 343
anonymous objects, 353–354, 373
anonymous pipes, 111
anonymous subscriptions, in Merge Replication, 411
ANSI (American National Standards Institute), 112, 257, 283, 288
APIs (application programming interfaces)
allocation and cleanup functions for, 659–667
mobile phone programming, 4–19
for client/server applications, 5–8
Internet Explorer Mobile Web browser, 17–18
.NET Compact Framework, 14–17
RIAs (rich Internet applications), 18–19
for thick clients, 4–5
Win32, 8–14
application data, 347, 349
application domains, 19
application setup, CAB (deployment) files and, 12
ApplyChangeFailed event, 445
ApplyChanges method, 425
ARM instruction set, 9, 65
arrays
character, 106
of pixels, 554
in Platform Invoke (P/Invoke) feature, 90–92
registry support of, 213
articles of publication, in Merge Replication, 410
.asmx pages, on Web sites, 320
ASP.NET Web Services, 517
assemblies, WCF client, 538–539
Assembly class, 587
assembly references, in ADO.NET classes, 232–233
AT&T, 51
attributes, in managed code, 67
authentication, 389, 392
automated in-place editing capability, 167–177
automatic garbage collection, 659
automatic memory cleanup, 63
automatic word wrap, 623, 635–636
AutoResetEvent, 111
Azure operating system (cloud computing), 5, 530

B

background thread
for RAPI file search, 475–479
for RAPI startup, 462, 465
backlight power usage, 54–57
Basic authentication, 392
battery life, 39–58
device power study of, 51–58
backlight usage in, 54–57
communications usage in, 57
multimedia usage in, 57–58
standby usage in, 53–54
measuring, 43–51
hardware approach to, 48–51
software-only approach to, 45–48

Download at WoweBook.Com

talk time versus standby time in, 44–45
memory leaks and, 14
problems in, 40–43
Bezos, Jeff (Amazon.com), 6
binary file I/O for storage, 196–202
BinaryFormatter serialization class, 184, 197, 211
BinaryWriter class, 188–190
binding. *See* data binding
BindingCollection property, 118, 126–129
bitmaps, 579–602
area filling brushes for, 577
class of, 581–582
device-independent, 552, 554
drawing, 591–595
as drawing surface or object, 580–581
empty, 582–583
external files to create, 583–584
Graphics object for, 562
Image class and, 563
image files sizes of, 589–591
as pixel arrays, 554
resources to create, 584–589
sample program on, 596–602
block mode, in device-side DLL loading, 509, 511–512
blocking functions, 469
Bluetooth, discoverable, 54
Boolean values, 77–79
boot time, system, 491–496
Borland, Inc., 98
bound data table updating, 248, 251–252
bound DataGrid information, 147–148
boundaries, drawing, 552, 559. *See also* graphics
bounding box, 635, 637
brushes
for fill colors, 570
for raster graphics, 577–578
for text foreground pixels, 624
for vector graphics, 555, 603
business objects
for central location of business logic, 355
data binding provided by, 375
in sample application, 364–374
read-only properties of, 369–374
read-write properties of, 366–369
state table in, 364–365
BusinessSolutionAccelerator2008, for Windows Mobile, 359
byte-level file I/O for storage, 185–186

C

C programming language, parameter passing supported by, 80
CAB (deployment) files, 12
caching
CustCache.Client.Designer.cs file for, 432–433
CustCache.SyncContract.cs file for, 433–434
of JITted code, 22
Local Data Cache for, 428–429, 432
call and block, in native and managed code communication, 111
CallWin32 sample program, 93–96
cancel editing event handlers, 173
CancelCurrentEdit method, 251
caption text parameter, 68–69
Category Modification Form class, 279–281
CeCreateDatabaseEx function, 180–181
CeFindAllFiles function, 479
CEFUN tool, 63
CeRapiInit startup function, 458–459
CeRapiInitEx startup function, 458–459
CeRapiInvoke function, 512–514
CeRapiUninit shutdown function, 459–460
CeRegCreatKeyEx function, 488
change tracking, 357, 426, 429
Char [ ] array, 90
character arrays, 106

Download at WoweBook.Com

CIL code, 64
circular references, objects with, 212
classes
  ADO.NET, 229–233
  Assembly, 587
  bitmap, 581–582
  Category Modification Form, 279–281
  Control, 563, 567
  data provider, 224–225, 227
  DataRelation, 252–253
  DataSupplier Web Services, 327
  Directory, 184–185, 255
  Engine, 262–263
  Environment, 472
  File, 184–185, 255
  FormMain, 417
  Graphics, 558–559, 591
  Hungarian notation for, 650–657
  Image, 563
  I/O, 191–193
  MainForm, 467
  managed code encounter of, 20
  Marshal
    to access data, 483
    for function parameters and return values, 76–77
    for manual parameter passing, 101–102
    static methods in, 100
    for strings, 113
    for structure size, 86
    structures tuned by, 105–108
  Matrix, 558
  memory-resident database, 223, 227, 236
  .NET, 15–16
  Proxy, 433
  Queue, 341
  Rapi, 462
  ResourceWrapper, 33–35
  serialization, 184
  set in .NET, 340–341
  SoapHttpClientProtocol, 437
  SqlCeCommand, 263–266
  SqlCeConnection, 227, 263–265
  SqlCeDataAdapter, 276–282
  SqlCeDataReader, 266–275
  SqlCeRemoteDataAccess, 398
  SqlCeReplication, 416, 416–418, 419
  Stack, 341
  StartupThread, 465–469
  SynchAgent, 432
  SynchProvider, 432
  X, 379
cleanup functions, 659–667
Clear method, 560
ClearColorKey method, 593
ClearType fonts, 622
Click event handler, 314, 606–609, 608
CLI-compatible languages, 32
client-oriented computing, 517
client/server applications, 5–8
client/server topology, 523
clipboard, 74
clipping, for drawing boundaries, 552
Close method, 30, 541
CloseHandle function, 83
cloud computing, 5, 530
CodeSnitch (Entrek Software), 23
collections, LINQ, 342
color specifications, 570–578
  named, 573–575, 578
  pens defined by, 604
  RGB values for, 575–578
  system, 571–573, 578
  transparency in, 593–595
columns
  accessing by name, 239
  in DataGrid control, 139–142
  GridColumnStyle objects for, 147
  Identity property, 401
  in Merge Replication publication design, 411–414
  public properties of, 119
  Ranged Identity, 409
  runtime creation of styles for, 142–144
  sequence of, 147
COM (Component Object Model)
  device connection detection and, 500–501

Download at WoweBook.Com

distributed, 18
P/Invoke support of, 65–66
SendMessage function in, 466
WCF (Windows Communication Foundation) and, 517
COM Interop, 62, 65
ComboBox control, 241–242
binding tasks to, 130–131
for complex data binding, 125–126
for data binding, 116–117
in LINQ sample application, 350
Command object
generating, 301
SQL Server, 297
stored procedures in, 306
commit editing event handlers, 173
Common Language Specification, 459
communications power usage, 57
Compact Flash memory cards, 470–471
compact the heap phase, in object cleanup, 26
companion helper, in WCF Service Host utility, 524
compatibility
code verification for, 20
MSIL (Microsoft Intermediate Language) for, 14
PostScript, 625
complex data binding
with ComboBox controls, 125–126
controls supporting, 124–125
with DataGrid control, 136–137
to display tables, 118
composable queries, 344
compressed formats, 579, 590
concurrency
in Data Synchronization Services, 444–447
DataSet objects and, 307
optimistic, 358, 397
in SQL Server, 289
Configure Data Synchronization Wizard, 426, 428–429, 433, 447
conflict detection, 282, 388, 407
connected approach, of ADO.NET, 227–228
connection parameters, 418
connection status flag, 468
connectivity, for remote data synchronization, 392–396
contract, in WCF, 518, 520, 535
Control class, 567
Control Panel, system colors from, 571
Control.Invoke method, 466
controls
assigned to rows, 250
ComboBox, 241–242, 350
for complex data binding, 124–125
data binding, 117–119. *See also* DataGrid control for data binding
display screen drawing in, 564–566
font selection for, 626–627
Label, 245
ListBox, 241
multi-item, 244–245
Panel, 608
read-only, 144
single-item, 245–246
TextBox, 245
coordinate system, for drawing, 557–558
C++ programming
applications written in, 6
functions of, 97–100
long type of, 79
RAPI startup and, 467
unchecked keyword of, 81
C# compiler, 32
C# programming
anonymous function in, 343
P/Invoke wrappers from, 458–460, 462
RDA code in, 397
CPUs, of Windows Mobile phone, 2
CREATE INDEX statements, 265
Create Publication Wizard, 416
CreateGraphics method, 30, 563, 565
CreateProcess function, 83
credentials, proxy, 392

Download at WoweBook.Com

C-style function declarations, 72–75
CurrencyManager object, 246, 248, 251
current row concept, 129
CurrentCell object, 161–162, 167
CurrentCellChanged event
  in in-place editing, 168, 172
  row filter set from, 244
  sample application with, 148–149, 153–155
cursoring capability, 128
CustCache.Client.Designer.cs file, 432–433
CustCache.SyncContract.cs file, 433–434

D

data binding, 115–177
  in ADO.NET data sets, 241–252
    bound data table updating in, 251–252
    to multi-item controls, 244–245
    overview, 240–244
    rows controls assigned to, 250
    rows displayed in, 246–250
    to single-item controls, 245–246
  business objects providing, 375
  complex, 124–126
    with ComboBox controls, 125–126
    controls supporting, 124–125
    with DataGrid control, 136–137
    to display tables, 118
  controls for, 117–119
  DataGrid control for, 132–177
    accessing information from, 149–155
    automated in-place editing capability for, 167–177
    complex data binding with, 136–137
    display styles in, 137–139
    drill-down capability for, 155–161
    events inherited from, 135
    in-place editing capability for, 161–167
    methods inherited from, 133–134
    PMEs specific to, 132–136
    properties inherited from, 132–133
    runtime creation of styles in, 142–144
    table and column styles in, 139–142
    user input to, 144–149
  DataViews and, 241
  in LINQ sample application, 354, 356
  objects for, 119–124
  overview, 115–117
  simple, 126–131
  in WCF, 518–521, 525
data contract, in WCF, 520–522
data forms, 15
data handling, 15
Data Manipulation Language (DML), 256
data modification routine, 439
data provider classes, 224–225, 227
data sets
  from databases, 284–286
  LINQ to, 348–355
  multitable, 309–310, 350
  as provider-independent memory-resident database classes, 227
  stored procedures with, 303–307
  typed, 310–318, 431–432
data sets, ADO.NET
  creating and accessing, 237–240
  data binding in
    bound data table updating in, 251–252
    to multi-item controls, 244–245
    overview, 240–244
    rows controls assigned to, 250
    rows displayed in, 246–250
    to single-item controls, 245–246
  XML for reading and writing, 252–254
Data Synchronization Services, 424–448
  beginning development of, 428–433
  benefits and weaknesses of, 389
  client in, 435–444
  completing, 433–435
  concurrency errors in, 444–447
  functionality added to, 447–448

Download at WoweBook.Com

requirements of, 425–426
data view, rows assigned controls by, 246
DataAdapter object, 348, 358
database layer, in data objects, 116
databases, property, 180–181
data-bindable control ADO.NET classes, 229, 231
DataGrid control for data binding, 116, 132–177
accessing information from, 149–155
automated in-place editing capability for, 167–177
complex data binding with, 136–137
display styles in, 137–139
drill-down capability for, 155–161
events inherited from, 135
in-place editing capability for, 161–167
methods inherited from, 133–134
PMEs specific to, 136
properties inherited from, 132–133
runtime creation of styles in, 142–144
table and column styles in, 139–142
user input to, 144–149
DataReader
application source code for, 268–273
as SQL Server object, 290–295
DataRelation classes, 252–253
DataRelation object, 240–241
DataSet objects, 307
DataSet.WriteXML class, 330
DataSource property, 247
DataSupplier Web Services class, 327
DataTable memory-resident database classes, 227, 236–240
DataTable objects, 123
DataView memory-resident database classes, 227, 236–241
DB_2 and DB Everyplace (IBM), 225
debugging, 10, 64–65
declarations in P/Invoke, 66–75
C-style function, 72–75
function return values in, 72
MessageBox function in, 66–68
native function details in, 68–71
defensive coding, RAPI startup, 469–470
deferred execution, in LINQ (Language Integrated Query), 343–344
Delegate parameter, 466
DELETE statements, 365
DeleteCommand property, 278
Designer, in Visual Studio
bug in, 608
for table styles, 140
for tables versus table adapters, 318
typed resources and, 586
.xsd files in, 311
desktop computers, drawing in, 555–558
destructors, Finalize method versus, 32
Details property, 370
device connection state, RAPI detection of
auto-start approach to, 500–501
callback approach to, 501–507
device context, handle to (hdc), 565–566
device driver code, Win32 for, 11
Device Emulator Manager, 528
device power study, 51–58
backlight usage in, 54–57
communications usage in, 57
multimedia usage in, 57–58
standby usage in, 53–54
device-independent bitmaps (DIBs), 552, 554, 590, 596
Dictionary class, 213
Directory class, 184–185, 255
disconnected approach, of ADO.NET, 229
disconnected connections, 470
disconnected operations, support for, 357–359
discoverable Bluetooth, 54
display screen drawing
in controls, 564–566
graphics objects for, 562–564
non-Paint event drawing in, 568–569
Paint event for, 566–568

Download at WoweBook.Com

display styles, in DataGrid control, 137–139
DisplayMember property, 247
Dispose method, 29–30, 33, 563, 583
DisposeBitmap method, 589
distributed computing, 516
DLLs (dynamic link libraries)
  C-callable functions access by, 62
  C++ functions declared in, 97–100
  DllImport attribute for, 67, 100
  ISAPI extension, 391–392
  P/Invoke to call Win32, 59, 96–100
  Remote API (RAPI) and, 509–514
  Win32 code and, 12
  Windows CE installation of native, 96
DMA (direct memory access) high-speed channel, 528
DML (Data Manipulation Language), 256
DNS servers, 527
documentation, 73, 562
double-buffer bitmap, 582
DrawEllipse method, 561
DrawIcon method, 560
DrawImage method, 560
drawing. *See* graphics
DrawLine method, 561
DrawPolygon method, 561
DrawRectangle method, 561, 603
DrawRectangle sample program, 568–569
DrawString method, 559, 623–624
drill-down capabilities, 155–161
drill-up capabilities, 159
drivers
  stream interface, 63, 491
  Win32 for, 11
Dynamic Data Exchange (DDE), 466
dynamic horizontal partitions, 409

E

echoing back, in synchronization, 442
editing
  automated in-place, 167–177
  in-place, 161–167
"elbow" joints, drawing, 604
electricity usage application example, 530–538
Embedded Resource build action, 586–587
eMbedded Visual Basic applications, 6
empty bitmaps, 582–583
encapsulation, 13, 202
EncodingClass, 190–191, 193
EndCurrentEdit method, 251
endpoint, in WCF, 518, 524
end-user experience, 6
Engine class, 262–263
Enhanced Metafiles (EMFs), 553
Entrek Software, 23
enumeration
  cell name in, 145
  for file filter and retrieval flags, 481
  for flag parameters, 70
  of fonts, 622, 627
  indexing versus, 341
  of installed file systems, 181
  World Wide Web Consortium (W3C) definitions of, 379
Environment class, 472
ErrorCategories table, 406–407
errors
  ADO.NET handling of, 234–235
  FetchAndDisplayError function for, 475
  synchronization, 446
  WCF Service Host utility detection of, 525
event handlers
  bitmap transparency, 595
  cancel editing, 173
  Click, 314, 606–609, 608
  commit editing, 173
  CurrentCellChanged, 168, 172
  in Dispose method for memory management, 33–35
  FormLoad, 157–158
  Load, 151, 163–164, 171
  manually adding, 608–609
  MouseDown, 172

Download at WoweBook.Com

MouseUp, 173
Paint, 563, 565, 606–609, 629
for system timers, 47
Validating, 169, 174–175
events
ApplyChangeFailed, 445
Click, 608
CurrentCellChanged, 148–149, 153–155, 244
from DataGrid control, 135
Format, 129–130
Mouse, 146
MouseDown, 612
MouseMove, 148–149, 151–153, 160
Paint, 566–568, 612
of threads, 111
Exchange Server, 520
eXecute In Place (XIP) region, 182
Execute method, 263–264
extensibility, 12, 518
external files, for bitmaps, 583–584

F

FAT file system, 179
fault contract, in WCF, 520, 523
FetchAndDisplayError function, 475
field versions, of rows, 239
File class, 184–185, 255
File Explorer, 255
file I/O for storage, 183–213
binary, 196–202
byte-level, 185–186
encoding and decoding data in, 190–191
File and Directory classes for, 184–185
higher-level, 187–190
I/O classes for, 191–193
text, 193–196
XML, 203–210
XML serialization for, 210–213
file systems, 229. *See also* storage
FileStream object, 189–190, 195, 210
fill colors, 570
Fill method, 276–277, 308, 349, 603
FillEllipse method, 561
FillPolygon method, 561
FillRectangle method, 561, 577
FillRegion method, 560
filters, in Merge Replication, 410
Finalize method, for memory management, 29–36
FindMemoryCard sample program, 105–108
fingerprint readers, 19
firewalls, 386, 389
fixed-pitch fonts, 621
flag parameter, 68–70, 468, 481
Flash memory cards, 105
floating-point values, 72, 557
flushing JITted code pool, in object cleanup, 27
FontPicker sample program, 627–628
fonts, 554, 625–632
ClearType, 622
for controls, 626–627
enumerating, 627
fixed- and variable-pitch, 621
FontPicker sample program for, 627–628
RotateText sample program for, 628–632
setting structure for, 200–202, 205
TrueType, 622
foreign keys, 259, 409
formatting, 129–131. *See also* graphics
FormLoad event handler, 157–158
FormMain class, 417
Framework Class Library, 427
FromImage method, 563–564
function parameters, 77
function return values in P/Invoke declarations, 72

G

game development, 6
garbage collection, for memory management, 22–27
automatic, 25–27
calling, 27–29

Download at WoweBook.Com

garbage collection, for memory management *(Continued)*
data and, 23–25
garbage collector pools for, 22–23
weak references and, 36
for Windows API, 659
GC_Collect method, 28
GetChanges method, 425
GetChildRows method, 240
GetDC function, 565
GetMessage function, 505
GetParentRows method, 240
GetSchema method, 425
GetServiceInfo method, 425
GetSystemPowerStatusEx2, 47
GIF graphics format, 552, 590, 596
GlobalAlloc function, for memory, 513
Global.asax Web Services page, 328–330
GlobalMemoryStatus function, 84, 86
goes to variable, 343
Google.com, 517
granularity
of managed heaps, 20–21
in power usage, 41–42, 48
in SmartMeter application sample, 535
in SQL server security, 289
graphics, 549–619. *See also* text
display screen drawing, 561–569
in controls, 564–566
graphics objects for, 562–564
non-Paint event drawing in, 568–569
Paint event for, 566–568
drawing functions for, 553–555
drawing surfaces for, 551, 551–553
.NET Compact Framework, 555–561
overview, 549–551
raster, 570–602
bitmaps for, 579–602
class of, 581–582
drawing, 591–595
as drawing surface or object, 580–581
empty, 582–583
external files to create, 583–584
image files sizes of, 589–591
resources to create, 584–589
sample program on, 596–602
brushes for, 577–578
color specifications in, 570–577
vector, 602–619
game sample of, 605–619
DotControl class for, 610–612
paint and click event handlers for, 606–609
Square and Players structures for, 619
Squares class for, 612–618
overview, 602–604
pens for, 604–605
Graphics class, 558–559, 562–564, 591
Graphics Device Interface (GDI), 549
GridColumnStyle objects, 147

H

Hall, Mike, 74
handle to a device context (hdc), 565–566
hardware, measuring battery life with, 48–51
hatch brushes, for area filling, 577
"heap," the, 19. *See also* garbage collection, for memory management
Height property, of Bitmap objects, 581
help pages, for Web services, 324
hidden files, 471
hidden windows, 567
higher-level file I/O for storage, 187–190
HitTest application, 144–146, 150–153, 160
HRESULT values, Win32, 459
HTC (High Tech Computing) Kaiser phone, 50–51, 54, 57
HTTP and HTTPS transport protocol, 518

Download at WoweBook.Com

HTTP binding, 434–435
HTTPS (Secure Sockets Layer), 331
Hungarian notation for .NET programs, 85, 641–657
  for data types, 647
  guidelines for, 642–644
  m_ prefix for private data, 647–648
  .NET naming guidelines for, 644–646
  objectives of, 642
  standard value type prefixes, 649
  usage and system classes, 650–657
hypermedia, 518

I

IBM, Inc., 225, 517
IComparable.Compare routine, 367
Identity property, RDA, 401–402
IEnumerable interface, 341–342
IIS (Internet Information Systems)
  in Merge Replication, 415
  synchronization and, 388–392
ildasm.exe utility, 20
Image class, 563
image file sizes for bitmaps, 589–591
ImageAttribute object, 594
immutable strings, 82
In Search of Excellence (Peters), 7
In The Hand, Inc., 225, 233
Inch coordinate system, for drawing, 558
include files set, 73
indexing
  enumeration versus, 341
  rows displayed by, 245, 248–250
  Seek method and, 267
  TrackingOnWithIndexes option for, 402–403
inheritance, Win32 versus, 13
Init Settings selection, 221
in-memory bitmaps, 552
in-place editing capability, 161–167
INSERT statements, 365
InsertCommand property, 278
installable file systems, 181, 470–472
instance methods, 20
instantiating WCF clients, 541–542
Int values, 459
integers
  passing array of, 92
  registry support of, 213
Integrated authentication, 392
IntelliSense, in Visual Studio
  LINQ and, 351, 374
  Rapi class and, 462
  typed data sets and, 310
  WCF services and, 542
  Web services client application and, 331
interline spacing, 635
internal leading, 635
Internet Explorer Mobile Web browser, 5, 17–18
Internet host loopback address, in WCF, 527
invalid windows, 566–567
Invalidate method, 567–568
Invoke function, 465–466, 506, 511–512
IP address, hard-coded, 527
ISAPI (Internet Server Application Programming Interface), 391–392

J

Jaspers Dots game sample, 605–619
  DotControl class for, 610–612
  paint and click event handlers for, 606–609
  Square and Players structures for, 619
  Squares class for, 612–618
JITted code pool
  flushing, 27
  for memory management, 20–22
  MSIL (Microsoft Intermediate Language) converted to, 9
JPEG graphics format, 552, 579, 590

K

Kerberos Authentication, 392
key constraints for databases, 259

Download at WoweBook.Com

keys
  foreign, 259, 409
  registry, 213, 215–217
  search, 246, 249–250
keywords
  for by-reference parameters, 77, 80–81
  PASCAL, 98
  ref, 77, 80, 482
  unchecked, 459

L

Label control, 116, 245, 568
lambda expression operator, 343
LANs (local area networks), 527
layered approach, in ADO.NET, 225, 227–229
leading, internal, 635
line colors, 570
LINQ (Language Integrated Query), 337–383
  in Compact Framework, 341–343
  data binding and, 242
  deferred execution of, 343–344
  overview of, 337–340
  sample application in, 344–377
    business object properties in, 366–374
    hybrid version of, 374–377
    LINQ to datasets in, 348–355
    LINQ to objects in, 355–365
    overview, 344–348
  set classes in .NET and, 340–341
  to XML, 378–382
List class, 213
ListBox controls, 241
ListView control, 116
Live Mesh (Mesh Operating Environment), 5
Load event handlers, 151, 163–164, 171
Load method, 315
load ordering issues, 11
LoadBitmapResource method, 589
LoadProductIDs routine, 273–274
Local Data Cache item, 428–429, 432
LocalAlloc function, for memory, 513
LocalProvider property, 445
location, for text drawing, 624
loopback address, Internet host, 527
low-level code, Win32 for, 10–12
LP prefix, in Hungarian naming, 85

M

m_ prefix for private data, in Hungarian notation, 647–648
Mail Transport, 518, 520
MainForm class, 467
managed code. *See also* .NET Compact Framework
  attributes in, 67
  CeRapiInvoke function from, 513–514
  native code communication with, 109–112
  native code ported to, 64–65
  portability of, 60, 63
Management Studio, 443
manifest resources, 586–589
manual cleanup, for memory management, 29–30
manual garbage collection, 659
manual memory management, 24, 36–37
manual parameter marshaling, 101
manual parameter passing, 100–108
  copying to native memory, 102–104
  Marshal attribute to tune structures in, 105–108
  Marshal class for, 101–102
ManualResetEvent, 111
mapping, font, 626
mark and sweep phase, in object cleanup, 25–26
Marshal class
  to access data, 483
  for function parameters and return values, 76–77
  for manual parameter passing, 101–102
  static methods in, 100

Download at WoweBook.Com

for strings, 113
for structure size, 86
structures tuned by, 105–108
Marshal.AllocHGlobal function, 104
Marshal.SizeOf method, 484
matching binding, 245–248
Matrix class, 558
MeasureString method, 559, 633–634
MeasureString sample program, 634–635
memory
allocation of, 104
automatic cleanup of, 63
bitmaps in, 552, 589
DMA high-speed channel for, 528
NAND flash, 179
native, copying to, 102–104
Win32 API allocation functions for, 513
memory leaks, 13–14, 22, 22–23
memory management, for mobile phones, 19–37
desktop versus, 3
garbage collection for, 22–27
JITted code pool for, 20–22
manual, 36–37
metadata maps for, 19–20
special handling of managed data for, 27–36
memory-resident database classes, 223, 227, 236
MemoryStatus, P/Invoke declaration for, 84–90
menu handler, synchronization, 441
Merge Replication, 225–226, 409–424
benefits and weaknesses of, 389
configuring, 415–416
Data Synchronization Services versus, 448
design for, 410–415
modifying data at device in, 421–424
process of, 420–421
Remote Data Access (RDA) versus, 424
SqlCeReplication class for, 416–418
subscribing to publication in, 419–420
for synchronization, 255
"message pump" function, 505
message queues, 62, 111
message text parameter, 68–69
MessageBox function, 15
calling native, 103–104
as P/Invoke declaration, 66–68
MessageWindow class, 108
metadata maps, 19–20
metafiles, to store pictures, 553, 563
method tables, 20–21
methods. *See also* PMEs (properties, methods, events)
AddSubscription, 419
ApplyChanges, 425
CancelCurrentEdit, 251
Clear, 560
ClearColorKey, 593
Close, 30, 541
Control.Invoke, 466
CreateGraphics, 30, 563, 565
Dispose, 29–30, 33, 563, 583
DisposeBitmap, 589
DrawEllipse, 561
DrawIcon, 560
DrawImage, 560
DrawPolygon, 561
DrawRectangle, 561, 603
DrawString, 559, 623–624
EndCurrentEdit, 251
Execute, 263–264
Fill, 276–277, 308, 349, 603
FillEllipse, 561
Fillpolygon, 561
FillRectangle, 561, 577
FillRegion, 560
Finalize, 29–36
FromImage, 563–564
GC_Collect, 28
GetChanges, 425
GetChildRows, 240
GetParentRows, 240
GetSchema, 425
GetServiceInfo, 425

Download at WoweBook.Com

methods *(Continued)*
inherited from DataGrid control, 133–134
instance, 20
Invalidate, 567–568
Load, 315
LoadBitmapResource, 589
in Marshal class, 100
Marshal.SizeOf, 484
MeasureString, 559, 633–634
MoveToContent, 207
Pull, in RDA, 399–401, 403–404
Push, in RDA, 405–407
Read, 268
ReRegisterForFinalize, 31
Seek, 267–268
SetColorKey, 593
static, 20, 185
SubmitSQL, in RDA, 407–408
SuppressFinalize, 31
Synchronize, 440
ThreadMainStartup, 465
Union, 353–354
WriteXML, 252
MEX (metadata exchange) endpoint, 524
MFC applications, 6
Microsoft Loopback Adapter, 528
Microsoft Synchronization Services SP1 (Devices), 428
Microsoft Zune media player, 6
middle tier objects, 357
Millimeter coordinate system, for drawing, 558
mobile phone programming
APIs for, 4–19
for client/server applications, 5–8
Internet Explorer Mobile Web browser, 17–18
.NET Compact Framework, 14–17
RIAs (rich Internet applications), 18–19
for thick clients, 4–5
Win32, 8–14
memory management for, 19–37
garbage collection for, 22–27
JITted code pool for, 20–22
manual, 36–37
metadata maps for, 19–20
special handling of managed data for, 27–36
overview, 1–4
MOE (Mesh Operating Environment), 5
Monsoon Solutions, Inc. (Power Monitor), 49–50, 53
mouse events, 146
MouseDown event handler, 172, 612
MouseMove events, 148–149, 151–153, 160
MouseUp event handler, 173
MoveToContent method, 207
MSDN library, 562
MSIL (Microsoft Intermediate Language) instruction set, 9, 20
MSMQ (Microsoft Message Queue), 62, 108, 519
multi SELECT stored procedures, 317–318
multi-item controls, 244–245
multimedia power usage, 57–58
multithreaded startup, of Remote API (RAPI), 462–465
mutex operating system objects, 111

N

name decorating, 97
name mangling, 97
named colors, 573–575, 578, 604
Named Pipes, Win32, 517, 519
namespaces
in ADO.NET classes, 232–233
drawing, 556
.NET, 15
WCF client, 538–539
NAND flash memory, 179
native code, 8. *See also* Win32 API
managed code communication with, 109–112
ported to managed code, 64–65

Download at WoweBook.Com

SQL Server CE databases accessed from, 255
as unsafe code, 61, 64
native function details in P/Invoke declarations, 68–71
native image cache, 22
native memory, copying to, 102–104
nested relationships, DataRelation class for, 253
.NET Compact Framework
APIs (application programming interfaces) of, 5, 14–17
file I/O for, 209–210. *See also* storage
graphics in, 555–561
Hungarian naming convention for, 85
LINQ (Language Integrated Query) in, 341–343
MSIL (Microsoft Intermediate Language) instruction set for, 9
WCF (Windows Communication Foundation) in, 522–523
.NET Micro Framework, 5
.NET Remoting, 517
network stack, 519
NetworkStream object, 190, 210
no-network hosts, 527–528
normal files, 471
NotepadCE element, 191, 196, 206
NTFS file system, 179
N-tier topology, 523
NTLM, 392

O

Object Browser, in Visual Studio, 20
object-oriented programming (OOP)
encapsulation in, 202
quotation mark (".") notation in, 338
SOAP formatting for, 319
Win32 versus, 13
objects
anonymous, 353
for API allocation and cleanup, 659–667
bitmap, 580–581
business, 366–374
for central location of business logic, 355
data binding provided by, 375
in sample application, 364–374
read-only properties of, 369–374
read-write properties of, 366–369
state table in, 364–365
with circular references, 212
cleanup of, 25
Command, 297, 301, 306
CurrencyManager, 246, 248, 251
CurrentCell, 161–162, 167
data, 115–117, 119
data binding, 119–124
DataAdapter, 348, 358
DataGridTableStyle, 137–139
DataReader, 290–295
DataRelation, 240–241
DataSet, 236–240, 303–307
DataTable, 123, 236–240
DataView, 236–240
FileStream, 189–190, 195
Graphics, 562–564, 580–581
GridColumnStyle, 147
HitTest, 144–146, 150–153
ImageAttribute, 594
I/O, 183
LINQ to, 355–365
middle tier, 357
NetworkStream, 190
proxy, 320
Remote API (RAPI)
device files access from
finding files in, 474–479
overview, 471–473
speed in, 479–486
device property database access from, 496–499
registry entries access from, 486–496
SqlCeCommandBuilder, 282
SqlCeDataAdapter, 276–278
Stream, 183, 186, 194, 210, 588
SynchAgent, 445
TableStyles, 147

Download at WoweBook.Com

objects *(Continued)*
UpdateCommand, 306
weak references to, 36
XmlTextWriter, 192
ObjectState value, 365
Occasionally Connected Applications (OCAs), 386
Office Suite, code in, 7
off-screen bitmaps, 579
OLE (Object Linking and Embedding), 466
OpenType fonts, 625
operating system extensions, Win32 for, 10, 12
optimistic concurrency, 358, 397
overloads
in device-side DLL loading, 509
drawing, 557
of function names in C++ programs, 97
in Graphics class, 591
for Invoke method, 466
of SqlCeDataAdapter object, 277
word wrap, 635–636
owner-draw support feature, 564

P

Paint event handler
for game sample application, 606–609, 612
generating, 565–568
for Graphics object access, 563
in RotateText sample application, 629
Panel controls, 608
parameters
connection, 418
Delegate, 466
floating-point, 557
function, 104
manual passing of, 100–108
copying to native memory, 102–104
Marshal attribute to tune structures in, 105–108
Marshal class for, 101–102
MessageBox, 68
Platform Invoke (P/Invoke) function, 75–92
arrays passed as, 90–92
passing by value versus reference, 79–82
passing string, 82
simple data types as, 76–79
structures as, 82–88
type of types versus, 88–90
reserved, 510
Win32 versus .NET types of, 481
parent-child relationships, in datasets, 236
parentheses, placement of, 352
parsing, 16, 129–131
PASCAL keyword, 98
Paul Yao Company, 74
PDA applications, 470
PeekMessage function, 505
peer-to-peer topology, 523
pens
for line colors, 570
for vector graphics, 603–605
permissions, 288, 396
Peters, Tom (*In Search of Excellence*), 7
Petzold, Charles (author), 13
photographs, 578. *See also* graphics
PIE (Pocket Internet Explorer), 17
P/Invoke Wizard, 73–75, 80, 85. *See also* Platform Invoke feature
P/Invoke wrapper, 59. *See also* Platform Invoke feature
pipes, anonymous, 111
Pixel coordinate system, for drawing, 558
pixels, arrays of, 554
placing text, 632–640
alignment in, 635–636
MeasureString sample program for, 634–635
text size in, 633–634
TextAlign sample program for, 636–640
Platform Builder packages, 182
platform independence, 318

Download at WoweBook.Com

Platform Invoke (P/Invoke) feature, 59–113
for ActiveSync, 458–460
to call Win32 functions, 8
declarations created in, 66–75
C-style function, 72–75
function return values in, 72
MessageBox function in, 66–68
native function details in, 68–71
manual parameter passing in, 100–108
copying to native memory, 102–104
Marshal attribute to tune structures in, 105–108
Marshal class for, 101–102
native and managed code communication by, 109–112
overview, 60–66
parameters supported in, 75–92
arrays in, 90–92
parameter passing by value vs. reference in, 79–82
simple data types in, 76–79
string parameter passing by value in, 82
structures passed as parameters in, 82–88
type of types in, 88–90
PowerMeter program and, 47
property databases and, 181
QueryPerformanceCounterfunction and, 472
RAPI registry query function wrappers of, 488–489
reserved parameters and, 510
sample program in, 93–96
support of, 112–113
Win 32 and NETCF benefits in, 4
Win32 DLLs called by, 96–100
PMEs (properties, methods, events). *See also* event handlers; events; methods
of Binding objects, 126–127
of DataGrid control for data binding, 132–136
for DataGridTableStyle object, 137–139
of DataReader, 233
of HitTest object, 145–146
of SQL Server, 289
PNG graphics format, 552, 590
Pocket Access databases, 496
Pocket Internet Explorer (PIE), 17
PocketAccess data provider (In The Hand, Inc.), 225, 233
"pointer to a pointer" function, 480
point-to-message queues, 111
point-to-point message queues, 62, 111
polymorphism, 13
portability
of managed code, 60, 63
.NET Compact Framework for, 15–16, 60, 63
of XML, 203
PostMessage function, 109
PostScript compatibility, 625
Power Monitor (Monsoon Solutions, Inc.), 49–50, 53
Power Toys for .NET Compact Framework 3.5, 539
power usage. *See* battery life
PowerMeter program, 47
presentation layer, in data objects, 116
primary key constraint for databases, 259
printing, 552
process handles, 83
PROCESS_INFORMATIONstructure, 83
processors, 3, 9
profiling functions, 472
program startup at system boot time, 491–496
Projects DataTable objects, 123
properties, ambient, 626
property databases
C-callable functions in, 180–181
RAPI objects for access to, 496–499
provider-independent ADO.NET classes, 229–230, 232

Download at WoweBook.Com

provider-independent memory-resident database classes, 227
provider-specific ADO.NET classes, 229
proxy, WCF client, 539–540
Proxy class, 433
proxy credentials, 392
proxy objects, 320
PtrToStructure function, 483
public properties, of columns, 119
publisher, SQL Server as, 410
Pull method, RDA, 399–401, 403–404
Push method, RDA, 405–407

Q

query analyzer, CE, 259–261
QueryPerformanceCounter function, 472
QueryPerformanceFrequency function, 472
Queue class, 341

R

Ranged Identity columns, 409
RAPI. *See* Remote API (RAPI)
RapiConnectDetect.cs source file, 501–505
raster graphics, 570–602
  bitmaps for, 579–602
    class of, 581–582
    drawing, 591–595
    as drawing surface or object, 580–581
    empty, 582–583
    external files to create, 583–584
    image files sizes of, 589–591
    resources to create, 584–589
    sample program on, 596–602
  brushes for, 577–578
  color specifications in, 570–577
  methods for, 560
  output from, 554
Read method, 268
read-only controls, 144
read-only files, 471
read-only properties, 369–374, 571
ReadSmartMeter sample WCF client, 542–546
read-write properties, 366–369
real-time threads, 62
ref keyword, 77, 80, 482
reference, parameters passed by, 79–82
reference types, 89
Reflection feature, in .NETCF, 20
regedit.exe, 486
registry access
  P/Invoke wrappers for RAPI query of, 488–490
  for program startup at system boot, 491–496
  RAPI objects for, 470
  Remote Registry Editor for, 486–488
  for storage
    keys for, 214–215
    Storage sample application for, 217–221
    values for, 215–217
regular expressions, 16
ReleaseDC function, 565
Remote API (RAPI), 451–514
  device connection state detection by, 499–507
    auto-start approach to, 500–501
    callback approach to, 501–507
  device-side DLLs loaded to, 509–514
  device-side programs loaded to, 507–509
  functions of, 181, 452–458
  .NET ActiveSync applications in, 452–458
  object store of, 470–499
    device files access from, 471–486
      finding files in, 474–479
      overview, 471–473
      speed in, 479–486
    device property database access from, 496–499

Download at WoweBook.Com

registry entries access from, 486–496
P/Invoke wrappers for RAPI query of, 488–490
for program startup at system boot, 491–496
Remote Registry Editor for, 486–488
shutdown of, 459–460
startup of, 458–470
defensive coding for, 469–470
multithreaded, 462–465
simple, single-threaded, 460–462
StartupThread class for, 465–469
for synchronization, 226
Remote Data Access (RDA)
benefits and weaknesses of, 388–389
Data Synchronization Services versus, 448
Identity property of, 401–402
Merge Replication versus, 424
methods and properties overview, 398–399
overview, 225–226
Pull method of, 399–401, 403–404
pulled schema of, 402–403
Push method of, 405–407
SubmitSQL method of, 407–408
for synchronization, 255
remote procedure calls (RPCs), 18, 451. *See also* Remote API (RAPI)
Remote Registry Editor, 214, 486–488
ReRegisterForFinalize method, 31
reserved parameters, 510
resources, for bitmaps, 584–589
ResourceWrapper class, 33–35
REST (Representational State Transfer), 517–518
restoring setting, 219–221
rewriting older code, decision to, 6–8
RGB color values, 575–578, 604
RIAs (rich Internet applications), 5, 18–19
ROM-based files, 182–183
RotateText sample program, 622, 628–632
routines
data modification, 439
IComparable.Compare, 367
to set property values, 366–369
structure-specific, 310
unbinding, 131
utility, 235, 346, 417
rows
in ADO.NET data sets, 246–250
business objects from, 359–364
controls assigned to, 250
data view to assign controls to, 246
DataGrid control headers for, 158
echoed back, 442
indexing to display, 245, 248–250
in Merge Replication publication design, 411–414
status value of, 239–240
update selected, 302–303
UpdateSelectRow method for, 275–276
RPCs (remote procedure calls), 18, 451. *See also* Remote API (RAPI)
runtime
DataGrid control styles at, 142–144
message generated by, 109
object instantiation as, 20
of single-threaded embedded operating system, 5
Windows Mobile-compatible, 18
runtime callable wrappers (RCWs), 66

S

safe code, managed code as, 60
SalesForce.com, 517
Save handler, 194
SaveAs handler, 195
SaveSettingToFile routing, 198
SByte [ ] array, 90
scheduling units, threads as, 109
schema
GetSchema method for, 425
RDA Pull method for, 401–403
SQL Server CE queries to, 282–287
scratch space, 582

Download at WoweBook.Com

screen size, mobile phone versus desktop, 3
scroll bars, 245
scrolled windows, 567
search keys, 246, 249–250
Secure Digital (SD) cards, 105, 181, 470
security
  HTTPS (Secure Sockets Layer) for, 331
  IIS exposure of, 392
  of managed code, 63
  SQL server, 289
seed value, Identity property, 401–402
Seek method, 267–268
SELECT statements
  data tables populated by, 238
  for DataTable objects, 123
  Fill method and, 276
  in LINQ sample application, 349
  for LoadProductIDs routine, 273
  in Pull RDA method, 399
  stored procedures as, 297, 307–308
SelectCommand property, 278
SelectedItemIndex property, 128
SendMessage function, 109, 466
serialization
  classes for, 184
  Smart Device client use of, 433
  XML
    for data formatted in, 16
    easier, 210–213
    LINQ to Objects for, 356
    to move data to device object store, 345
    to save data to device storage, 360–363, 382
server-side code, Win32 for, 10, 12–13
service contract, in WCF, 520–522
Service Model Metadata Utility, 539
service-oriented computing (SOC), 516–517
set classes in .NET, 340–341
SetColorKey method, 593
share permissions, 396
shell extension DLLs, 12
shell functions, 62
ShowBitmap sample program, 596–602
ShowDatabases program, 497–499
ShowParam.cpp code, 98–100
shutdown, Remote API (RAPI), 459–460
signed integers, in P/Invoke declarations, 79, 81–82
Silverlight, for RIA building, 5, 18
simple, single-threaded startup, of Remote API (RAPI), 460–462
simple data binding, 118, 126–131
simple data types, in P/Invoke, 76–79
SimpleDrawString sample program, 624–626
single-item controls, 245–246
single-threaded embedded operating system, 5
SIP (Software Input Panel), 12
sleep state, 491
smart devices. *See also* mobile phone programming
  data storage for, 179–183
  drawing in, 555–558. *See also* graphics
  XML serialization and, 213
SmartMeter sample application, 530–538
SMTP e-mail standard, 518
SOAP (Simple Object Access Protocol)
  in ADO.NET Web Services, 319–321
  messages of, 319–320, 325–326, 330, 334
  SoapFormatter serialization class for, 184, 211
  SoapHttpClientProtocol class for, 437
software-only, measuring battery life by, 45–48
solid brushes, for area filling, 577
Solution Explorer, in Visual Studio, 96, 313, 332, 586
sorting
  anonymous objects, 373
  in LINQ, 339, 352–353
spacing, in drawing, 559, 635

Download at WoweBook.Com

special handling for memory management, 27–36
SPOT (Smart Personal Objects Technology), 5
SQL DML (Data Manipulation Language), 256
SQL Server
  in ADO.NET
    command objects for, 297
    connecting to, 289–297
    overview, 287–289
    as provider of, 225
    stored procedures of
      concurrency and, 307
      with DataSet objects, 303–307
      using, 297–303
    typed data sets of, 310–318
  for Data Synchronization Services, 389, 428
  Enterprise Manager of, 416, 423
  Merge Replication and, 389
  RDA code in, 397
  Remote Data Access (RDA) and, 388
  stored procedures of, 346–347
SQL Server Authentication, 392
SQL Server CE
  in ADO.NET
    CE database created for, 261–263
    CE database populated for, 263–266
    CE files in, 255–256
    CE query analyzer in, 259–261
    CE syntax in, 256–259
    as provider of, 225
    schema queries in, 282–287
    SqlCEDataAdapter class for, 276–282
    SqlCEDataReader class for, 266–275
    updating, 275–276
  for Data Synchronization Services, 428
  in synchronization, 387–392
SQLServerConnectivityManagement program, 393–394, 396
SqlCeCommand class, 263–266
SqlCeCommandBuilder object, 282
SqlCeConnection class, 227, 263–266
SqlCeDataAdapter class, 276–282
SqlCeDataReader class, 266–275
SqlCeRemoteDataAccess class, 398, 404
SqlCeReplication class, 416, 416–418, 419
Stack class, 341
standard value types, Hungarian notation for ., 649
standards-driven approach, in ADO.NET, 233
standby power usage, 44–45, 53–54
startup, Remote API (RAPI), 458–470
  defensive coding for, 469–470
  multithreaded, 462–465
  simple, single-threaded, 460–462
  StartupThread class for, 465–469
startup programs
  at system boot time, 491–496
  Win32 code for, 11
StartupCallback function, 467–468
StartupThread class, 465–469
static methods, 20, 185
storage, 179–222. *See also* Remote API (RAPI)
  file I/O for, 183–213
    binary, 196–202
    byte-level, 185–186
    encoding and decoding data in, 190–191
    File and Directory classes for, 184–185
    higher-level, 187–190
    I/O classes for, 191–193
    text, 193–196
    XML, 203–210
    XML serialization for, 210–213
  mobile phone versus desktop, 3
  registry access for, 213–221
    keys for, 214–215
    Storage sample application for, 217–221
    values for, 215–217
  for smart-device data, 179–183
  for Windows CE-powered devices, 224

Download at WoweBook.Com

stored procedures
to access records, 346–347
concurrency and, 307
with DataSet objects, 303–307
multi SELECT, 318
for typed data sets, 317–318
for user access rights, 288
using, 297–303
stream interface drivers, 63, 491
stream mode, in device-side DLL loading, 509, 511
Stream object, 183, 186, 194, 210, 588
StringBuilder class, 82
strings
character arrays in, 90–92
marshaling, 104
.NET Compact Framework for, 16
parameters of, 82
passed as parameters, 88–89
passed by value, 82
registry keys defined as, 213
structures
passed as parameters, 82–88
structure-specific routines for, 310
in typed data sets, 310–311
SubmitSQL method, RDA, 407–408
subnet mask, 527
subscribing to publication, in Merge Replication, 411, 419–420
SuppressFinalize method, 31
Sync Services Designer Wizard, 427
SynchAgent class, 432
SynchAgent object, 440, 445
SynchProvider class, 432
synchronization of mobile data, 385–449
Data Synchronization Services for, 424–448
beginning development of, 428–433
client in, 435–444
completing, 433–435
concurrency errors in, 444–447
functionality added to, 447–448
requirements of, 425–426
Live Mesh for, 5
Merge Replication for, 409–424
configuring, 415–416
design for, 410–415
modifying data at device in, 421–424
process of, 420–421
Remote Data Access (RDA) versus, 424
SqlCeReplication class for, 416–418
subscribing to publication in, 419–420
options for, 225–226
overview, 385–387
Remote Data Access (RDA) for, 397–408
capabilities and overhead of, 397–398
Identity property of, 401–402
Merge Replication versus, 424
methods and properties overview, 398–399
Pull method of, 399–401, 403–404
pulled schema of, 402–403
Push method of, 405–407
SubmitSQL method of, 407–408
remote data connectivity for, 392–396
SQL Server CE, 387–392
of versions, 239
synchronous functions, 469
syntax, CE, 256–259
syntax checker, in compiler, 80
Sysbase SQL (Anywhere), 225
system boot time, 11, 491–496
system colors, 571–573, 578, 604
system files, 471
system timers, for power usage, 42, 47–48
System.Runtime.InteropServices namespace, 100
System.Threading.Mutex class, 111

T

TableMappings collection property, 309
tables
data, 123, 238–239

Download at WoweBook.Com

in DataGrid control, 139–142
DataGridTableStyle object for, 137–139
datasets for multi, 350
ErrorCategories, 406–407
memory-resident data based on, 223
in Merge Replication publication design, 411–414
RDA Pull method for, 401
runtime creation of styles for, 142–144
SELECT statements to populate, 238
SqlCeConnection and SqlCeCommand, 263–266
table adapters versus, 318
TableStyles object, 147
tombstone version of, 426
updating bound data, 248, 251–252
talk time power usage, 44–45
Task structure, 119–123
Tasks DataTable objects, 123
TCP/IP connections, 386, 500–501, 519. *See also* synchronization of mobile data
text, 621–640
colors for, 570
drawing, 621–625
font selection for, 625–632
for controls, 626–627
enumerating, 627
FontPicker sample program for, 627–628
RotateText sample program for, 628–632
as graphics, 554, 559–560
placing, 632–640
alignment in, 635–636
MeasureString sample program for, 634–635
text size in, 633–634
TextAlign sample program for, 636–640
text file I/O for storage, 193–196
TextBox control
binding tasks to, 130–131
for data binding, 116, 126
editing initiation by, 165–166
overview, 245
for update requests, 161–163
for XML file I/O, 203–204, 206
TFAT (Transaction-Save File System), 179
thick clients
APIs (application programming interfaces) for, 4–5
computing with, 517
RIAs (rich Internet applications) and, 18
thread handles, 83
thread startup code, 465
ThreadMainStartup method, 465
thread-safe communication code, 465–466
thread-safe message delivery, 109
three-layer approach, of ADO.NET, 229
"thunk," in method tables, 21
Tick-Count property, 472
Tilt phone (AT&T), 51
time card application, on mobile phone, 3
time tracker application example, 115–116
timer function, for battery status power usage, 42, 47–48
"today screen" customization, 12
token-passing topology, 523
ToList operator, 362
tombstone version of tables, 426
TrackingOnWithIndexes option, 402–403
Transaction-Save File System (TFAT), 179
transparency, of colors, 576–577, 593–595
travel expense application, on mobile phone, 3
TreeView control, in data binding, 119
triggers, in SQL Server syntax, 288, 409
TrueType fonts, 622, 625
type of types, in P/Invoke, 88–90

Download at WoweBook.Com

typed data sets, 310–318
benefits of, 311–314
code to create and load, 314–316
Data Synchronization Services and, 431–432
multi SELECT stored procedure for, 317–318
partial class code for, 316–317
structure specified in, 310–311
typed resources, 585–586

U

UI (user interface) code, 371–372, 375
unbinding routines, 131
unchecked keywords, 459
Unicode, 8, 66, 112–113
Union method, 353–354
Universal Flash Storage (UFS), 181
unsafe code, native code as, 61, 64
unsafe keyword, 80–81
unsigned integers, in P/Invoke declarations, 79
untyped resources, 586–589
updating
in data binding, 131, 161, 166
data locally, 422
databases with SqlCeDataAdapter object, 278–282
selected rows, 302–303
SQL Server CE in ADO.NET, 275–276
Storage sample application, 217–221
UPDATE statements for, 365
UpdateCommand object for, 306
UpdateCommand property for, 278, 282
URLs
Server Agent, 391, 395
as WCP endpoint addresses, 518–519
user access permissions, 288
user interface skin, 12
user interface (UI) code, 371–372
UTF-16 strings, 66
utility routines, 235, 346, 417
UtilRegistry class, 215–217

V

Validating event handler, 169, 174–175
validation, 357, 369
ValueMember property, 128
values
Boolean, 77–79
floating-point, 72, 557
Identity property seed, 401–402
ObjectState, 365
as parameter type, 89
parameters passed by, 79–82
property, 366–369
registry, 215–217
RGB color, 575–578, 604
row status, 239–240
string parameters passed by, 82
Win32 HRESULT, 459
in XML file, 207–209
ValueType helper routine, 366–367
variable-pitch fonts, 621
vector graphics, 602–619
game sample of, 605–619
DotControl class for, 610–612
paint and click event handlers for, 606–609
Square and Players structures for, 619
Squares class for, 612–618
methods for, 561
output from, 554–555
overview, 602–604
pens for, 604–605
virtual consoles, windows as, 551
virtual directories, for synchronization, 392–396
Virtual Directory Creation Wizard, 393–394
virtual private networks (VPNs), 386
Visual Basic, 79, 81, 458, 462
Visual Studio
ARM4 instruction set for, 9
constraints detected by, 313
data set class code from, 315
for Data Synchronization Services, 427–428

Download at WoweBook.Com

debugging features of, 10
Designer in
bug in, 608
for table styles, 140
for tables versus table adapters, 318
typed resources and, 586
.xsd files in, 311
enumeration and, 70
graphics formats and, 552
IntelliSense function in
LINQ and, 351, 374
Rapi class and, 462
typed data sets and, 310
WCF services and, 542
Web services client application and, 331
.NET ActiveSync applications built in, 453
Object Browser in, 20
P/Invoke code debugging and, 64
proxy objects generated by, 320
Remote Registry Editor in, 214, 486
Solution Explorer in, 96, 313, 332, 586
SQL Server connection from, 295, 299
for table styles, 140
typed resources and, 586
untyped resources and, 586
WCF and
client for, 538–539
service for, 433–434, 523, 525–526
Service Library project for, 427
Web services and, 319
for Windows Mobile, 2
XNA plug-ins for, 6

W

watt units, in battery ratings, 45
WCF (Windows Communication Foundation), 515–547
cloud computing via, 5
description of, 515–518
in .NET Compact Framework, 16, 522–523
service creation for, 523–538
code generation for, 523–525
host address for, 526–530
SmartMeter sample of, 530–538
Windows Mobile-compatible, 525–526
terminology of, 518–522
Visual Studio and
client created in, 538–539
service created in, 433–434, 523, 525–526
Service Library project created in, 427
Windows Mobile client of, 538–546
WCF Service Configuration Editor, 526
WCF Service Host utility, 524–525
WCF Test Client program, 524
weak references, objects marked with, 36
Web browsers, 16, 16–18
Web Forms applications, 226
Web services
in ADO.NET, 226, 318–335
client application for, 331–335
sample application for, 321–331
XML, XSD, and SOAP for, 319–321
clients for, 16
for thick client applications, 18
for Windows Mobile, 13
Web Services Description Language (WSDL), 324–325
Web-centric companies, 517
WHERE clause, 342, 358
Width property, of Bitmap objects, 581
Win32
ActiveSync libraries in, 458
DLLs of, 507
FindFirstFlashCard function of, 181
HRESULT values of, 459
memory allocation in, 513
Named Pipes of, 517
owner-draw support feature in, 564
QueryPerformanceCounterfunction of, 472
Win32 API
Hungarian naming convention for, 85

Download at WoweBook.Com

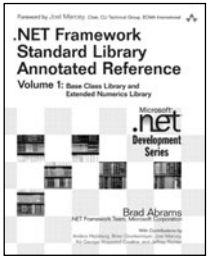
Win32 API *(Continued)*
overview, 8–14
Platform Invoke (P/Invoke) feature and, 96–100
for thick clients, 5
WINAPI declaration, 98
window handle parameter, 68–69
Windows Authentication, 392
Windows Azure cloud computing system, 5, 530
Windows CE. *See also* ADO.NET; mobile phone programming; SQL Server CE; Windows Mobile
file system of, 182
native DLLs installed in, 96
property databases for, 180–181
as Unicode operating system, 112–113
Windows Communication Foundation. *See* WCF (Windows Communication Foundation)
Windows Device Center, 451
Windows Forms applications, 226
Windows Mobile
Business Solution Accelerator 2008 for, 359
TFAT (Transaction-Save File System) for, 179
WCF client for, 538–546
WCF service compatible with, 525–526
Windows Desktop PC versus, 2–3
Windows sandwich, 569
wireless cards, 386
WM_CLOSE message, for garbage collection, 27–28
WM_HIBERNATE message, for garbage collection, 27–28
word wrap, automatic, 623, 635–636
working set of managed code, 21
World Wide Web, 517
World Wide Web Consortium (W3C), 203, 319, 378–379
WriteXML method, 252
WSDL (Web Services Description Language), 324–325
www.RegExpLib.com, 368

X

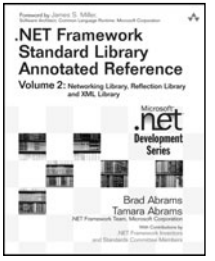
X classes, 379
XElement, 380
XIP (eXecute In Place) region, 182
XML
ADO.NET data sets read by, 252–254
in ADO.NET Web Services, 319–321
Close method and, 30
file I/O for, 203–210
LINQ (Language Integrated Query) to, 378–382
.NET 2.0 support for, 378
in .NET Compact Framework, 16
typed resources and, 585
XML persistence format, 330
XML serialization
easier, 210–213
in LINQ sample application, 345, 356, 360–363, 382
LINQ to Objects for, 356
to move data to device object store, 345
to save data to device storage, 360–363, 382
Smart Device client use of, 433
for XML-formatted data, 16
XmlIgnore attribute, 370
XmlReader class, 188
XmlSerializer serialization class, 184
XmlTextWriter object, 192
XmlWriter class, 188
XNA Game Studio, 6
.xsd files, 311
XSD.EXE, 319–321, 382

Download at WoweBook.Com

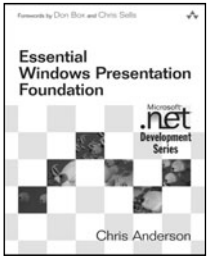
Microsoft .NET Development Series



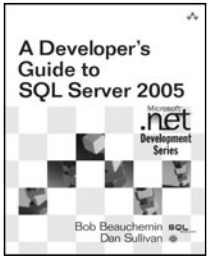
978-0-321-15489-7



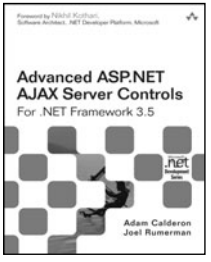
978-0-321-19445-9



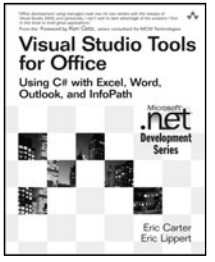
978-0-321-37447-9



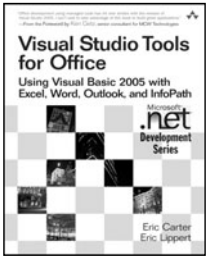
978-0-321-38218-4



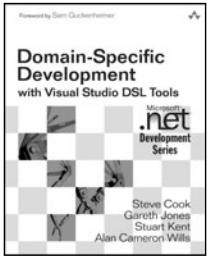
978-0-321-51444-8



978-0-321-33488-6

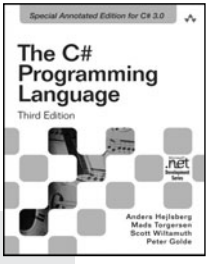


978-0-321-41175-4

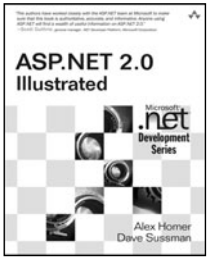


978-0-321-39820-8

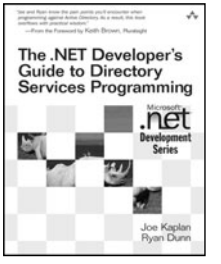
978-0-321-27872-2



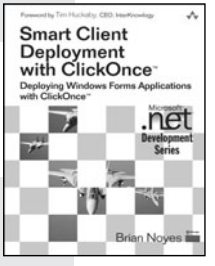
978-0-321-56299-9



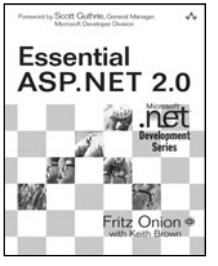
978-0-321-41834-0



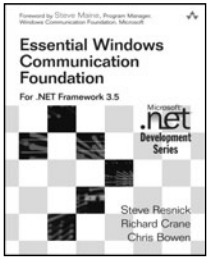
978-0-321-35017-6



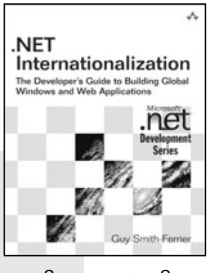
978-0-321-19769-6



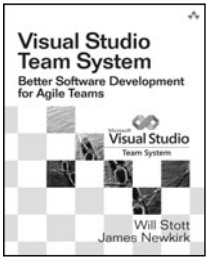
978-0-321-23770-5



978-0-321-44006-8



978-0-321-34138-9

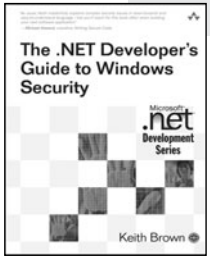


978-0-321-41850-0

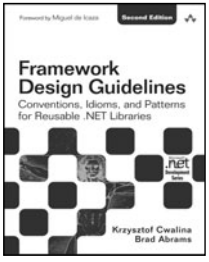
Download at WoweBook.Com

For more information go to informit.com/msdotnetseries/

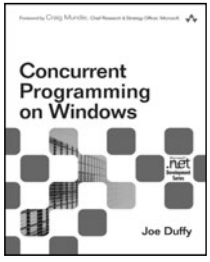
978-0-201-73411-9



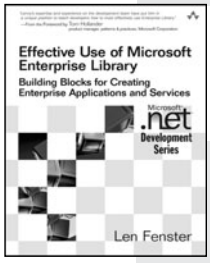
978-0-321-22835-2



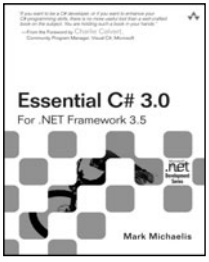
978-0-321-54561-9



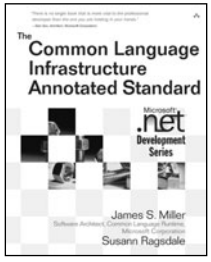
978-0-321-43482-1



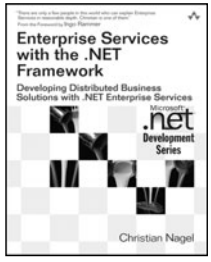
978-0-321-33421-3



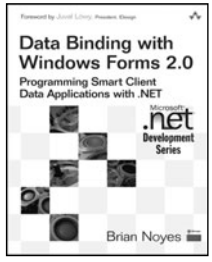
978-0-321-53392-0



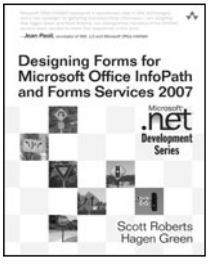
978-0-321-15493-4



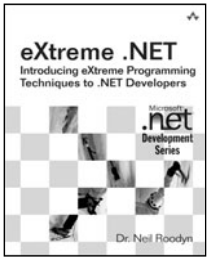
978-0-321-24673-8



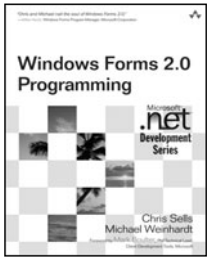
978-0-321-26892-1



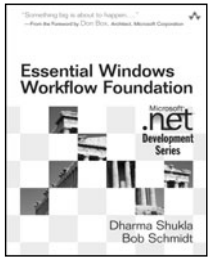
978-0-321-41059-7



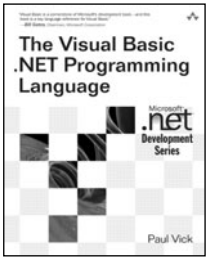
978-0-321-30363-9



978-0-321-26796-2

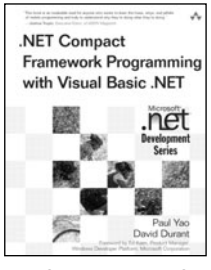


978-0-321-39983-0



978-0-321-16951-8

978-0-321-17403-1



978-0-321-17404-8

Download at WoweBook.Com

BE A LEAN MEAN CODING MACHINE

get the book source code at **www.paulyao.com**

Featuring the Power Programming Technique:

- On-site Workshops
- eCoaching for Programmers
- Open Enrollment Workshops
- Design Reviews
- Consulting
- Programmer Productivity Tools

Win the battle against time. Learn the secrets of Windows® CE® and Compact Framework Programming from the industry's most experienced Grand Masters. We can help you get ahead of the learning curve so you'll be more productive, more competitive, and more efficient.

The Paul Yao Company delivers intensive training aimed at providing programmers with insights into the internals of Microsoft® Windows®.

~Since 1987~

THE PAUL YAO COMPANY

The Paul Yao Company
1700 – 7th Avenue, Suite 116 PMB 320
Seattle, WA 98101 USA
training_info@paulyao.com
www.paulyao.com

Download at WoweBook.Com

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams
SAFARI BOOKS ONLINE

Download at WoweBook.Com

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari

Download at WoweBook.Com

Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos

Safari Books Online

FREE TRIAL—GET STARTED TODAY!
www.informit.com/safaritrial

Find trusted answers, fast

Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.

Master the latest tools and techniques

In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

WAIT, THERE'S MORE!

Keep your competitive edge

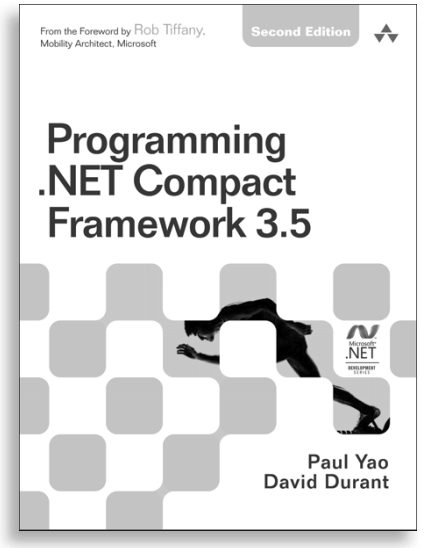
With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.

Stay current with emerging technologies

Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.

Addison Wesley · Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · lynda.com · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice Hall · Que · Redbooks · Sams · SAS Publishing · Sun Microsystems · Wharton School Publishing · Wiley

Download at WoweBook.Com



FREE Online Edition

Your purchase of ***Programming .NET Compact Framework 3.5, Second Edition,*** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, Que, and Sams.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

Activate your FREE Online Edition at www.informit.com/safarifree

- **STEP 1:** Enter the coupon code: WAODHBI.
- **STEP 2:** New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Safari Books Online

Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · lynda.com · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice Hall · Que · Redbooks · Sams · SAS Publishing · Sun microsystems · Wharton School Publishing · Wiley