RibbonX FOR DUMMIES®

by John Paul Mueller

Wiley Publishing, Inc.

RibbonX For Dummies®

Published by
Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030-5774

www.wiley.com

Copyright © 2007 by Wiley Publishing, Inc., Indianapolis, Indiana

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, the Wiley Publishing logo, For Dummies, the Dummies Man logo, A Reference for the Rest of Us!, The Dummies Way, Dummies Daily, The Fun and Easy Way, Dummies.com, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 800-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

For technical support, please visit www.wiley.com/techsupport.

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

Library of Congress Control Number: 2007932463

ISBN: 978-0-470-16994-0

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

About the Author

John Mueller is a freelance author and technical editor. He has writing in his blood, having produced 74 books and over 300 articles to date. The topics range from networking to artificial intelligence and from database management to heads-down programming. Some of his current books include a Windows power optimization book, a book on .NET security, and books on Amazon Web Services, Google Web Services, and eBay Web Services. His technical editing skills have helped over 52 authors refine the content of their manuscripts. John has provided technical editing services to both *DataBased Advisor Magazine* and *Coast Compute* magazines. He's also contributed articles to magazines like *CIO.com*, *DevSource*, *InformIT*, *Informant*, *DevX*, *SQL Server Professional*, *Visual C++ Developer*, *Hard Core Visual Basic*, *asp.netPRO*, *Software Test and Performance*, and *Visual Basic Developer*.

When John isn't working at the computer, you can find him in his workshop. He's an avid woodworker and candlemaker. On any given afternoon, you can find him working at a lathe or putting the finishing touches on a bookcase. He also likes making glycerin soap and candles, which comes in handy for gift baskets. You can reach John on the Internet at `JMueller@mwt.net`. John is also setting up a Web site at `http://www.mwt.net/~jmueller/`; feel free to look and make suggestions on how he can improve it. Check out his weekly blog at `http://www.amazon.com/gp/blog/id/AQOA2QP4X1YWP`.

Author's Acknowledgments

Thanks to my wife, Rebecca, for working with me to get this book completed. I really don't know what I would have done without her help in researching and compiling some of the information that appears in this book. She also did a fine job of proofreading my rough draft.

Russ Mullen deserves thanks for his technical edit of this book. He greatly added to the accuracy and depth of the material that you see here. I really appreciated the time that he devoted to checking my code for accuracy. I also spent a good deal of time bouncing ideas off of Russ as I wrote this book, which is a valuable aid to any author.

Matt Wagner, my agent, deserves credit for helping me get the contract in the first place and taking care of all the details that most authors don't really consider. I always appreciate his assistance. It's good to know that someone wants to help.

A number of people read all or part of this book to help me refine the approach, test the examples, and generally provide input that every reader wishes they could have. These unpaid volunteers helped in ways too numerous to mention here. I especially appreciate the efforts of Eva Beattie who read the entire book and selflessly devoted herself to this project. A number of other people, including Ant Burnham helped me in many and significant ways. I'd love to thank each person by name who wrote me with an idea, but there are simply too many.

Finally, I would like to thank Kyle Looper, Nicole Sholly, Barry Childs-Helton, and the rest of the editorial and production staff for their assistance in bringing this book to print. It's always nice to work with such a great group of professionals.

Publisher's Acknowledgments

We're proud of this book; please send us your comments through our online registration form located at www.dummies.com/register/.

Some of the people who helped bring this book to market include the following:

Acquisitions, Editorial, and Media Development

Project Editor: Nicole Sholly

Acquisitions Editor: Kyle Looper

Senior Copy Editor: Barry Childs-Helton

Technical Editor: Russ Mullen

Editorial Manager: Kevin Kirschner

Media Development and Quality Assurance: Angela Denny, Kate Jenkins, Steven Kudirka, Kit Malone

Media Development Coordinator: Jenny Swisher

Media Project Supervisor: Laura Moss-Hollister

Media Development Associate Producer: Richard Graves

Editorial Assistant: Amanda Foxworth

Senior Editorial Assistant: Cherie Case

Cartoons: Rich Tennant (www.the5thwave.com)

Composition Services

Project Coordinator: Patrick Redmond

Layout and Graphics: Carl Byers, Joyce Haughey, Stephanie D. Jumper, Alicia B. South, Ronald Terry

Proofreaders: Aptara, David Faust, Todd Lothery

Indexer: Aptara

Anniversary Logo Design: Richard Pacifico

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Acquisitions Director

Mary C. Corder, Editorial Director

Publishing for Consumer Dummies

Diane Graves Steele, Vice President and Publisher

Joyce Pepple, Acquisitions Director

Composition Services

Gerry Fahey, Vice President of Production Services

Debbie Stailey, Director of Composition Services

Contents at a Glance

Introduction 1

Part I: An Overview of RibbonX 9

Chapter 1: Getting to Know the Ribbon 11
Chapter 2: Creating an Effective RibbonX Design 31

Part II: Interacting with the Ribbon 47

Chapter 3: Designing New RibbonX Elements 49
Chapter 4: Writing RibbonX Scripts 71
Chapter 5: RibbonX and Visual Studio 91

Part III: Creating New RibbonX Applications 119

Chapter 6: Developing Business Applications for Word 121
Chapter 7: Developing Business Applications for Excel 163
Chapter 8: Developing Business Applications for Access 201
Chapter 9: Developing Business Applications for Outlook 235
Chapter 10: Developing Business Applications for PowerPoint 261
Chapter 11: Working with Web Services 285

Part IV: Converting Existing Toolbars and Macros 309

Chapter 12: Simple Fixes for Older Menus and Toolbars 311
Chapter 13: Conversion Techniques for VBA Users 327
Chapter 14: Conversion Techniques for Visual Studio Users 345

Part V: The Part of Tens 363

Chapter 15: Ten New Tasks You Can Perform with RibbonX 365
Chapter 16: Ten RibbonX Resources 375

Index 385

Table of Contents

Introduction 1

About This Book 1
Conventions Used in This Book 2
What You Should Read 3
What You Don't Have to Read 3
Foolish Assumptions 4
How This Book Is Organized 4
Part I: An Overview of RibbonX 5
Part II: Interacting with the Ribbon 5
Part III: Creating New RibbonX Applications 5
Part IV: Converting Existing Toolbars and Macros 6
Part V: The Part of Tens 6
The accompanying Web site 6
Icons Used in This Book 7
Where to Go from Here 7

Part I: An Overview of RibbonX 9

Chapter 1: Getting to Know the Ribbon 11

Understanding the Office Ribbon 12
Considering Office support for the Ribbon 13
Understanding support for old toolbars and menus 16
Defining the RibbonX Elements 23
Understanding tabs 24
Understanding groups 26
Understanding controls 26
Considering the Ribbon in Office 2007 27
Understanding the common Ribbon elements 27
Looking at the Ribbon in Word 28
Looking at the Ribbon in Excel 28
Looking at the Ribbon in Access 29
Looking at the Ribbon in Outlook 29
Looking at the Ribbon in PowerPoint 30

Chapter 2: Creating an Effective RibbonX Design 31

Developing RibbonX Element Goals 32
Considering RibbonX Element Accessibility and Visibility 34
Using tooltips 34
Using existing Office features 36

Using the Office Menu....36
Using Contextual Tabsets....37
Repurposing the MiniToolbar....38
Defining an Effective RibbonX Design....39
Using names effectively....40
Considering the number of items on a tab....40
Looking at groups from the user's perspective....41
Using the right control....41
Providing user hints....43
Using feature hiding effectively....43
Understanding the XML Connection....44

Part II: Interacting with the Ribbon....47

Chapter 3: Designing New RibbonX Elements....49

Creating a RibbonX Tab....50
Using Groups to Your Advantage....52
Defining the RibbonX Controls....55
An overview of the RibbonX controls....56
Common RibbonX control attributes....59
Common RibbonX control callbacks....61
Developing with the Office 2007 Custom UI Editor....64
Creating Custom Control Graphics....66
Obtaining a list of Office icons....67
Tools for creating control graphics....68
Choosing between bitmaps and icons....69
Understanding how transparency works....69
Relying on 32-bit images....70

Chapter 4: Writing RibbonX Scripts....71

Understanding RibbonX Basics for VBA Developers....72
Considering the RibbonX Limitations in VBA....73
Creating a Basic Tab....75
Writing the Scripts....76
Automatically generating the callback subs....76
Coding a new tab with groups and controls....77
Obtaining an identifier for an existing tab, group, or control....78
Modifying or repurposing existing tabs, groups, and controls....79
Modifying or repurposing the Office menu....83
Performing tasks when the Ribbon loads....86
Creating a Ribbon Using startFromScratch Mode....87
Adding Forms Instead of RibbonX Controls....88

Chapter 5: RibbonX and Visual Studio 91

Defining the Advantages of Using Visual Studio 92
Understanding the levels of RibbonX support 92
Working with dynamic document content 93
Creating a secure environment 94
Considering the advantages of managed code 94
Creating the RibbonX Environment in Visual Studio 95
Understanding RibbonX Basics for VB.NET and C# Developers 98
Choosing Between VBA and Visual Studio 99
Creating a Basic Tab 100
Defining the project 100
Adding the RibbonX files 101
Adding some code 102
Creating a package for end users 103
Removing the add-in 104
Writing Code Behind for RibbonX 106
Handling graphics in Visual Studio 107
Performing tasks when the Ribbon loads 109
Creating new tabs, groups, and controls 110
Modifying or repurposing existing tabs, groups, and controls 111
Modifying or repurposing the Office menu 114
Creating a Ribbon Using startFromScratch Mode 116

Part III: Creating New RibbonX Applications.....119

Chapter 6: Developing Business Applications for Word 121

Getting Started with Word Applications 122
Understanding Word and VBA 122
Understanding Word and Visual Studio 123
Creating a Letter/Memo Tab 123
Setting the style 124
Adding a recipient 129
Working with dates 133
Adding the sender 136
Greeting the recipient and adding a signature 138
Considering the CC, routing, and approval requirements 139
Automating Envelopes 142
Creating Labels 145
Filling Out Forms 146
Creating the forms 147
Selecting a form 150
Adding the user information 156
Including a date 159

Chapter 7: Developing Business Applications for Excel 163

Getting Started with Excel Applications 164
Understanding Excel and VBA 164
Understanding Excel and Visual Studio 164
Combining VBA and Visual Studio in Excel applications 165
Creating a Nonstandard Equations Tab 165
Creating a starting element 166
Choosing the correct equation 167
Defining the multiple Ribbon elements 168
Obtaining the data entered in the Ribbon 172
Performing the calculation 174
Performing Redundant Calculations 176
Defining the problem solution 177
Designing the dialog boxes 178
Creating the calculation code 179
Defining linkages to existing data 180
Performing the redundant calculations 182
Considering the data identification requirements 186
Automating Data Entry with Forms 187
Creating the form 187
Adding the Ribbon code 192
Performing content sleight-of-hand 192
Creating the worksheet linkage 194
Defining the employee selections 195
Calculating the cost 198

Chapter 8: Developing Business Applications for Access 201

Getting Started with Access Applications 202
Creating the XML File 202
Loading the Ribbon Changes 208
Defining the Ribbon macros 208
Using the system table USysRibbons 209
Using a standard user table 212
Using an XML file directly 214
Obtaining the Sample Database 218
Generating Temporary Tables or Filtered Results 220
Hiding the Add-Ins tab 221
Placing the groups in the correct order 221
Creating a temporary table 223
Defining a filtered result 226

Chapter 9: Developing Business Applications for Outlook 235

Getting Started with Outlook Applications 235
Creating a Mail-Management Tab 237
Trying the default project 237
Detecting the caller's class 239

Designing the filing interface 241
Obtaining the folder list....243
Creating the copy 245
Saving as a draft 246
Processing Incoming Mail Based on User Selections 247
Considering multiple Outlook class issues 247
Designing the task-creation interface 251
Defining the task....253
Adding supplemental information 258
Closing the task 259

Chapter 10: Developing Business Applications for PowerPoint . . .261

Getting Started with PowerPoint Applications 262
Defining the Custom Presentation Tab Interface 263
Creating the Initial Slide 267
Starting the process 268
Saving custom properties for later use 271
Getting built-in property values 272
Getting custom property values....273
Interacting with Outlook to obtain user data 274
Adding the Optional Slide Elements 276
Supporting Constant Data....278
Providing a Presentation Ending....280
Saving and Using the Template 283

Chapter 11: Working with Web Services 285

Getting Started with Web Service Applications 286
Understanding Public and Private Web Service Differences 287
Creating an Amazon.com Custom Application....288
Getting an AWS developer tag....289
Seeing how queries work in a browser....290
Understanding AWS tasks....292
Defining the AWS Ribbon interface 293
Adding the dialog-box launcher 298
Making a query 300

Part IV: Converting Existing Toolbars and Macros 309

Chapter 12: Simple Fixes for Older Menus and Toolbars 311

Training Users for the New Paradigm 311
Substituting Forms for Menus and Toolbars 313
Relying on the Menu Control....318
Using Existing Office Features 320
Relying on context menus 322
Relying on task panes 323

Performing Simple Interface Changes and Storing Them 323
Customizing the Quick Access Toolbar 324
Modifying the Quick Style Set in Word and storing it in the template 325

Chapter 13: Conversion Techniques for VBA Users 327

Defining the Issues behind VBA Conversion 328
Creating a Conversion Strategy 329
Using forms 330
Using existing menus and toolbars 330
Designing toolbars and menus, rather than creating them 331
Using the Quick Access Toolbar (QAT) 331
Developing a List of RibbonX Changes 333
Creating a Conversion Solution for Word, Excel, and PowerPoint 334
Creating a Conversion Solution for Access 335
Creating a Conversion Solution for Outlook 337
Designing Parallel Version Solutions 339
Considering the Office XP/2003 user 339
Considering the Office 2007 user 341
Defining the common code 342

Chapter 14: Conversion Techniques for Visual Studio Users 345

Using Existing Add-Ins 346
Existing add-ins from the RibbonX perspective 346
Considering project conversion 348
Re-creating a project 350
Defining a Conversion Strategy 352
Converting VBA Solutions 355
Performing a VBA walkthrough 357
Working with menus and toolbars 358
Developing workflows and task-based solutions 358
Considering Application-specific Conversion Requirements 359
Creating Custom Conversions When Necessary 360

Part V: The Part of Tens 363

Chapter 15: Ten New Tasks You Can Perform with RibbonX 365

Creating a Workflow Solution 366
Targeting Specific User Needs 367
Defining Alternatives for Common Tasks 368
Developing Organizational Aids 368
Performing User-Assisted Application Integration 369
Working with Web Services 370

Working with Hybrid Applications....370
Considering the User Task Criteria....371
Using Code More Than Once....372
Reining In Support and Training Costs....373

Chapter 16: Ten RibbonX Resources....375

Starting with the Microsoft Developer Network....376
Getting Tips from the Microsoft Blogs....377
Finding Other News Sources for RibbonX....377
Interacting with Others Through the Microsoft Forums....378
Obtaining Answers from Other Sources....379
Getting Tools, Examples, and Products from PSchmid.net....379
Working with the RibbonCustomizer....380
Using Blogs to Your Advantage with Technorati....382
Using OpenXMLDeveloper.org....382
Using MSDN and Other Print Magazines....383

Index....385

Introduction

The new Ribbon interface is here, and it's here to stay. According to Microsoft, this is the interface that everyone in the world voted to have through surveys and direct conversations. (Of course, if you truly believe that, I have a really cool-looking bridge to sell you.) No matter where the idea for the Ribbon comes from, though, the fact of the matter is that you're stuck (or is that blessed?) with it. Fortunately, this book can ease your pain. *RibbonX For Dummies* provides a comprehensive view of the Ribbon and makes your new application development or application conversions significantly easier.

Many people see the Ribbon as an entirely different way to work, and it is. For many people, different isn't better or worse, it's just plain scary. The chapters in this book help you discover techniques for making the Ribbon considerably less scary and reduce the learning curve for people who've used the menu-and-toolbar interface for years. As you work through these issues, you may come to conclude that the Ribbon truly is a good solution for some tasks. Although power users may find it a hindrance to working quickly, most novice users are going to find that the Ribbon does reduce what they need to know about the application; the need to remember one less thing often translates into improved productivity.

The Ribbon can benefit you as well. Instead of spending your weekends providing support for poorly designed applications, you can create a new workflow application that does much of the thinking for the user. That's right! The Ribbon is part of your support staff. It tends to enforce certain actions on the user's behalf, which means the user makes fewer mistakes and works faster. Fewer mistakes translates into fewer support calls and a longer weekend for you. Yes, as much as you (or your users) might like to dislike the Ribbon for being newfangled and unfamiliar, it has something to offer you.

About This Book

RibbonX For Dummies contains everything you need to work with the Ribbon. You get complete details on the RibbonX interfaces, discover how to use controls effectively, and learn about all of the ways you can use the Ribbon to make your applications better. You'll find a comprehensive listing of the

controls, their attributes, and the callbacks associated with them. Each control appears in use at least once in the book, and most significantly more often than that. Consequently, you won't have to worry about seeing an interesting control and then not knowing how to use it.

You'll also find examples for both VBA and Visual Studio users. The Visual Studio examples are available on both VB.NET and C#. All of the things you should consider for RibbonX development appear in your favorite language so you won't have to worry about translating the code. In addition, this book doesn't treat either group as a second-class citizen — all of the material appears in a manner that addresses the needs of both VBA and Visual Studio developers.

This book contains an entire chapter of coding examples for each major Office application, so you don't have to worry about finding examples for your product. The examples include Word, Excel, Access, Outlook, and PowerPoint. Not only will you find you find a great selection of Office applications, you'll also find examples of using Web services from the new Ribbon interface as well.

Converting existing applications is a major concern for all companies. After all, you have a huge investment in all that code you created for older versions of Office. *RibbonX For Dummies* contains a solution for every RibbonX need. Not only will you find multiple solutions in Part IV of the book, you'll also find the reasons that each solution works for particular needs so you don't have to guess about which solution to use in a particular situation.

Conventions Used in This Book

I always try to show you the fastest way to accomplish any task. In many cases, this means using a menu command such as Tools➪Macro➪Visual Basic Editor. When working with the Ribbon, I tell you which tab to access first, and then which feature to use on that tab.

Whenever possible, I use shortcut keys to help you access a command faster. For example, you can also start the VBA Integrated Development Environment (IDE) by pressing Alt+F11.

This book also uses special type to emphasize some information. For example, entries that you need to type appear in **bold.** All code, Web site URLs, and on-screen messages appear in `monofont type`. Whenever I define a new word, you'll see that word in *italics.*

Because you use multiple applications when you're working with the Ribbon, I always point out when to move from one application to the next. When a chapter begins, I introduce the main application for that chapter. All the commands in that chapter are for the main application until I specifically tell you to move to another application. I also specifically tell you when it's time to move back to the main application.

What You Should Read

What you read depends on your level of experience. If you've already worked with the Ribbon for a while, you can probably skip Chapter 1. In addition, if you've already created a simple Ribbon design, you can probably skip Chapter 2 as well.

VBA developers should read Chapters 3 and 4 to get started. If you're converting an application, then you should also read Chapters 12 and 13.

Visual Studio developers should read Chapters 3 and 5 to get started. If you're converting an application, then you should also read Chapters 12 and 14.

The Part III chapters you read depend on the Office products you work with. These chapters discuss Word (Chapter 6), Excel (Chapter 7), Access (Chapter 8), Outlook (Chapter 9), and PowerPoint (Chapter 10). The book also includes special information for working with Web services in Chapter 11.

No matter which programming language you use (or Office product you work with), you'll want to read Chapters 15 and 16 at some point. Chapter 15 tells you about ten new tasks you can perform with the Ribbon. Chapter 16 tells you about resources that will make your Ribbon programming experience significantly easier.

What You Don't Have to Read

Most of the chapters contain some advanced material that will interest only some readers. When you see one of these specialized topics (such as obtaining the correct visual effects in a Visual Studio add-in when your add-in has only a minimal interface), feel free to skip it.

You can also skip any material marked with a Technical Stuff icon. This material is helpful, but you don't have to know it to work with the Ribbon. I include this material because I find it helpful in my programming efforts and hope that you will, too.

Foolish Assumptions

You might find it difficult to believe that I've assumed anything about you — after all, I haven't even met you yet! But I have made a few assumptions. Although most assumptions are indeed foolish, I made these assumptions to provide a starting point for the book.

I assume you've worked with Windows long enough to know how the keyboard and mouse work. You should also know how to use menus and other basic Windows features. It's also essential to know how to use at least one Office application. I assume that you've already spent time discovering how to use the Ribbon in a new Office 2007 application. Some portions of the book work with Web pages, and others use eXtensible Markup Language (XML); you need to know at least a little about these technologies to use those sections. You don't have to be an expert in any of these areas, but more knowledge is better. You must also have a very good knowledge of the programming language you use to work with the Ribbon.

This is a book for someone who does have development experience. I assume that you have a very good knowledge of either VBA or a .NET programming language such as VB.NET or C#. Knowledge of both VBA and a .NET programming language is helpful for this book, but not a requirement. It's also important to know how to work with both the Office 2003 product and the Office 2007 product you want to use for creating your Ribbon application.

How This Book Is Organized

This book contains several parts. Each part demonstrates a particular Ribbon concept. In each chapter, I discuss a particular topic and include example programs that you can use to discover more about the Ribbon on your own. You can find the source code for this book on the Dummies.com Web site at `http://www.dummies.com/go/ribbonxfd`.

Part I: An Overview of RibbonX

You may not know what RibbonX is, except that it promises to become the next major annoyance for your overworked IT department. Chapters 1 and 2 of this book export the Ribbon and the RibbonX interface used to interact with it. In Chapter 1, you'll find an introduction to basic concepts, but already in this chapter you'll begin working with the XML that defines RibbonX. Chapter 2 begins the process of creating your first Ribbon. The example in this chapter actually has useful code so you can see the Ribbon in action. More importantly, Chapter 2 begins providing you with the tips and techniques you need to create Ribbon applications fast.

Part II: Interacting with the Ribbon

This part of the book gets into the details of working with the Ribbon. Chapter 3 introduces all of the controls, their attributes, and the callbacks you can use to interact with them. You'll also begin working with the specialized tools used to create Ribbon applications in Chapter 3. Chapter 4 provides specialized coverage of how to create a full-fledged application for VBA developers. Chapter 5 provides the same coverage for Visual Studio developers. Since each development group has special needs, you'll find sections that address concerns that only a VBA or a Visual Studio developer will have.

Part III: Creating New RibbonX Applications

At this point, it's time to begin working with real-world applications. You'll create applications in this part of the book that you can augment and use within your company. In fact, many of the examples in this section are simplified versions of applications already in use in companies just like yours. For example, the letter-writing example in Chapter 6 is already in use at a company. The users of this application produce letters in about a third of the time they did when working with the menu-and-toolbar interface. In addition, user error is almost down to zero, which is amazing; the rate using the menu-and-toolbar interface was significantly higher.

This part provides examples for Word (Chapter 6), Excel (Chapter 7), Access (Chapter 8), Outlook (Chapter 9), and PowerPoint (Chapter 10) users. By the time you finish this part, you'll be able to write a Ribbon application for any Office application. The detailed information tells you about all of the tips and tricks you can use with each Office product to make that product work better. You'll also know where Microsoft has deviated from the basic Ribbon strategy in a particular application, and what you need to look out for when you create your Ribbon application.

Part IV: Converting Existing Toolbars and Macros

Once you see just how well the Ribbon works, you'll want to begin converting some of your applications. Either that or someone at the top will decide that it's time for you to convert the applications as part of an overall strategy for adopting Office 2007. Whatever your reason for converting existing applications, this part of the book will provide you with the best techniques possible. Chapter 12 provides you with some general techniques that work for both VBA and Visual Studio developers. Chapter 13 provides specific conversion techniques for VBA developers, while Chapter 14 provides conversion techniques for Visual Studio developer.

Part V: The Part of Tens

This final part of the book provides you with some helpful tips and resources you can use to make your Ribbon development experience even better. Chapter 15 provides a list of ten new tasks you can perform now that you're more expert with the Ribbon. Chapter 16 provides ten truly useful resources that will help reduce your development time.

The accompanying Web site

This book contains a lot of code, and you might not want to type it. Fortunately, you can find the source code for this book on the Dummies.com Web site at `http://www.dummies.com/go/ribbonxfd`. The source code is organized by chapter, and I'll always tell you about the example files in the text. The best way to work with a chapter is to download all the source code for it at one time.

Icons Used in This Book

As you read this book, you'll see icons in the margins that indicate material of interest (or not, as the case may be). This section briefly describes each icon used in this book.

Tips are nice because they help you save time or perform some task without a lot of extra work. The tips in this book are timesaving techniques or pointers to resources that you should try to get the maximum benefit from VBA.

I don't want to sound like an angry parent or some kind of maniac, but you should avoid doing anything marked with a Warning icon. Otherwise you could find that your program melts down and takes your data with it.

Whenever you see this icon, think *advanced* tip or technique. You might find these tidbits of useful information just too boring for words, or they could contain the solution that you need to get a program running. Skip these bits of information whenever you like.

If you don't get anything else out of a particular chapter or section, remember the material marked by this icon. This material usually contains an essential process or bit of material that you must know to write VBA programs successfully.

Where to Go from Here

It's time to start your Ribbon adventure! I recommend that anyone who has only a passing knowledge of Ribbon go right to Chapter 1. This chapter contains essential, get-started information that you need for writing your first Ribbon program.

Those who already know Ribbon might want to skip to Part III to sink their teeth into some complex examples. If you've seen the Ribbon, but haven't developed a Ribbon application, start with Part II first. You might want to check out the resources in Part V if you find your current Ribbon development experience lacking.

Anyone who needs to convert an existing application should avoid the temptation to go directly to Part IV. Make sure you build a good foundation for your programming efforts first: I recommend going to Part II, and then seeing how a Ribbon application should work for the target Office product in Part III. Only then should you skip ahead to Part IV and discover the conversion techniques there.

Part I

An Overview of RibbonX

"We've got a machine over there that monitors our quality control. If it's not working, just give it a couple of kicks."

In this part...

You take your first look at Office 2007, notice that all your menus and toolbars are missing, and instantly know that the upgrade is going to be hard — or perhaps worry that it might be impossible. Don't fret! The Ribbon isn't nearly as difficult to configure as you might imagine. In fact, it offers some advantages that aren't immediately apparent. Chapter 1 describes the issues the Ribbon presents and describes why it might actually provide some benefits to your organization. Chapter 2 starts you down the path to creating your first RibbonX application.

Chapter 1

Getting to Know the Ribbon

In This Chapter

- Getting the overview of the Office Ribbon
- Understanding the RibbonX elements
- Using the Ribbon in Office 2007

Microsoft has made significant changes to Office 2007. The most noticeable change is the new Ribbon, which appears across the top of applications in place of the menu-and-toolbar interface of old. Unfortunately, in creating this new interface, Microsoft also decided against backward compatibility. All of those well-ordered toolbars and menus you created in the past now appear on a single Ribbon tab, Add-Ins. Yes, your code will still work — but users will find it significantly more difficult to use your applications.

If you have a relatively simple application, using the Add-Ins tab might not result in a loss of productivity. Most applications, however, don't translate well to the Add-Ins tab. That's because they rely on the context of the old menu-and-toolbar system — which means you probably end up with a mess instead of the nice interface you used in the past. Microsoft hasn't provided a clear and easy method to overcome the mess they created, which is (presumably) why you're reading this book.

This chapter introduces you to the Ribbon. The new interface really does have an appeal from an ease-of-use perspective. The chapter also examines the few aids that Microsoft has provided to help you overcome the problems with the new interface and tells you how to use them.

You'll discover that the Ribbon does have a few redeeming features for the developer as you examine the Ribbon elements. Theoretically, once you transition your application to the Ribbon, users can become more productive because the Ribbon hides complexity and makes application features more visible. By using the new screen elements carefully, you can create amazing new interfaces. Not only do you have access to new controls, but you also use new grouping features to use those controls with greater success.

Finally, this chapter shows you how the Ribbon appears in the target applications for this book. Knowing how Microsoft has put the Ribbon together in the various applications can help you create better application elements of your own.

This book uses the term Ribbon to refer to the physical presentation of application elements such as tabs, groups, and controls. The Ribbon is the part of the application interface that you can see. A similar term, RibbonX, refers to the programming interface you use to create the Ribbon elements for your application. You define how the Ribbon appears in the application by using the RibbonX programming interface.

Understanding the Office Ribbon

The Ribbon is the new interface for Office. Microsoft doggedly pursues most of its innovations — so it's quite likely that you'll see the new Ribbon in most, or all, Microsoft applications of the future. Because of the change in interface, the options available to Office users who have a substantial investment in custom templates consist of the following:

- Use the Add-Ins tab to access custom features in existing templates, which isn't a viable option for custom templates of any complexity.
- Continue to use an older version of Office, which means that you won't have updates at some point to protect against real-world dangers such as viruses.
- Convert existing applications to use the Ribbon, which means writing the interface code from scratch.
- Scrap existing applications as they become outdated — which means you'll eventually incur substantial development costs, but have a completely updated application.

None of these options would be optimal in every situation. Yes, you can use the Add-Ins tab for less complex applications — but the moment you have more than one or two menu or toolbar entries, the Add-Ins tab quickly becomes unviable. Even if you do use the Add-Ins tab, users will require some level of retraining because the options won't appear in the same locations they occupied in the past.

It's tempting to think that you can simply ignore the Ribbon completely by holding on to your current version of Office. Certainly, some companies are still using Office 97 without a pressing need to update to obtain new features. However, the risks of this approach are many. The biggest risk is that a new virus will appear, and because Microsoft won't provide updates for your old

copy of Office, you can easily get it and lose all of your data. Keeping the old version of Office is really only viable if your office is completely closed to the outside world — especially the Internet — and few offices are in that position any longer.

The paragraphs that follow consider the last two options in a bit more detail. It's important to become familiar with the Ribbon, decide whether it meets specific needs in your organization, look at the tools that Microsoft provides to update your templates, and then perform all the required manual conversion. Of course, you may simply decide that the features the Ribbon provides are too significant to ignore, and create a completely new version of your application. The "new application" option is probably going to be a little challenging for many organizations, though, considering the amount of custom code they already have in place.

The techniques you discover in this book may help you in more than one way in the future. You may eventually find the Ribbon in other applications. Microsoft has decided to license the Ribbon interface to third parties free of charge. Obviously, Microsoft wants to entrench the new Ribbon interface into a broad range of applications as quickly as possible. Making RibbonX available to third parties is an efficient way to perform the task. You can discover more about third-party licensing at

`http://msdn2.microsoft.com/en-us/office/aa973809.aspx`

Considering Office support for the Ribbon

The first question anyone with a perfectly running Office application will ask is why Microsoft decided to change the interface completely. The old system of menus and toolbars had simply become too cumbersome for most users; some organizations have found that users have a hard time locating a particular command or feature. Even though the menu-and-toolbar system is straightforward, commands often appear several layers deep, or on a toolbar that the user doesn't have displayed. Microsoft designed the Ribbon to overcome these problems by making the interface simpler and options easier to find. For example, you can choose programmatically to display tabs when the user needs them, and then hide them when the user has finished performing a task. (Whether the new approach actually works remains to be seen.)

Figure 1-1 shows the old menu-and-toolbar approach used with Excel. Figure 1-2 shows the same view, using the new Ribbon. As you can see, the new Ribbon does tend to make features easier to find by grouping them together and reducing the view to just the current task. By organizing the application features and focusing user attention, Microsoft contends that it's possible to obtain a significant improvement in user performance of common tasks.

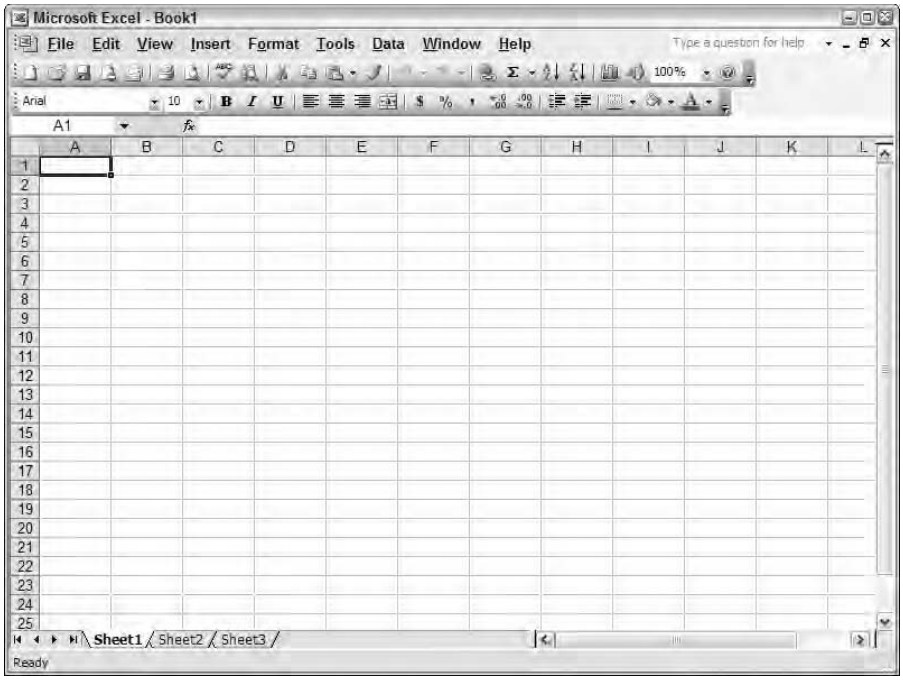


Figure 1-1: The old menu and toolbar interface looks cluttered.

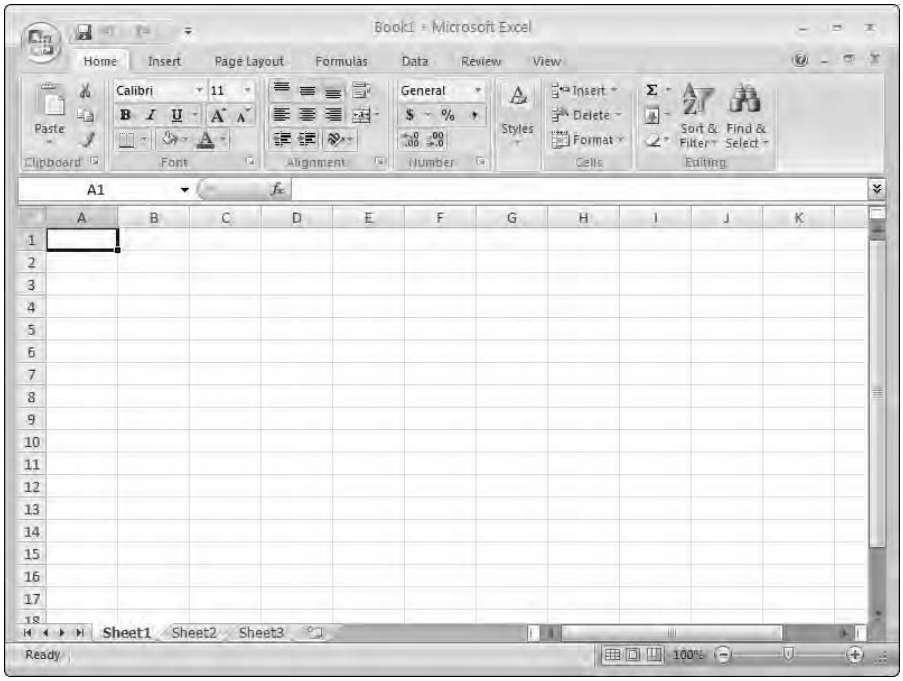


Figure 1-2: The new Ribbon interface groups like items together.

When you decide to update your application, you'll want to consider how you can also use these new features to your advantage. Chapter 2 discusses this topic in detail. However, for now, you'll want to think about the advantages that grouping and the new controls provide. Instead of placing certain features on a toolbar, you can place them on a tab. The tab need not be available at all times; you can make it visible only when the user's task context requires. In short, the developer also receives a number of benefits by using the Ribbon.

You might notice something else in Figure 1-2: The Ribbon seems to take up a lot of space. In fact, during beta testing, many people complained that the new interface requires too much space — so Microsoft makes it possible to hide the Ribbon when you're not using it. Simply right-click the menu bar and choose Minimize the Ribbon from the context menu. Consequently, any concerns that the Ribbon consumes more space than the old toolbar and menu interface are unfounded. In fact, the Ribbon can actually provide you with more screen real estate for editing documents.

However, the biggest plus of the new Ribbon is the ability to hide things in such a way that the user knows that they exist, but can easily ignore them. For example, many groups contain a special button in the lower-right corner that lets you display a dialog box with detailed settings, as shown in Figure 1-3. Simply click the button to display the associated dialog box.

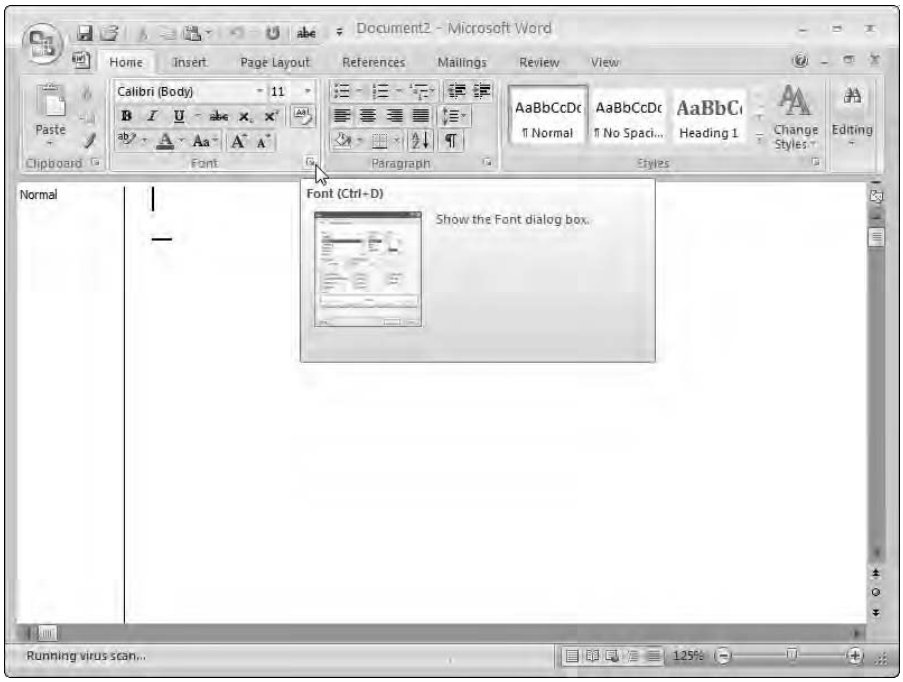


Figure 1-3: Use the Ribbon to hide special features in plain sight.

The user knows that the Font dialog box exists, but can choose to ignore it when the controls provided on the Ribbon suffice. Using this approach means you can monitor users of your application to determine which features they use most often, and then you can place those features on the Ribbon. This approach addresses the needs of normal users. Power users can display a dialog box that includes the full set of application features, so they won't have to give up flexibility to make the interface easier for novice users to use.

Understanding support for old toolbars and menus

You might be under the impression that the new Ribbon interface is an all-or-nothing proposition. In fact, there's a migration path of sorts; you can exploit that path in a number of ways. (For example, the "Performing Simple Interface Changes and Storing Them" section of Chapter 12 tells how you can make the transition easier by highlighting custom styles and adding some features to the user's Quick Access Toolbar.) The following sections describe the tools that Microsoft offers to make the transition easier. The Office Compatibility Pack lets you continue using Office 2003 even as you move to Office 2007 documents, while the Office Migration Planning Manager reduces the work required to plan an upgrade path.

Working with the Office Compatibility Pack

One capability that will undoubtedly make your life interesting is that you can provide backward compatibility for users of previous versions of Office as you move to Office 2007. The Office Compatibility Pack installs support for the new Office 2007 file formats for Office XP and Office 2003 users. Although users of these older versions of Office still can't use the new RibbonX applications, they can at least interact with the data found in the documents. In fact, you can create document templates in a way that provides support for *both* older and newer users, using parallel code. (The "Designing Parallel Version Solutions" section of Chapter 13 addresses this technique.)

Make certain you install all current updates for Office XP or Office 2003 before you install the Office Compatibility Pack. Otherwise, the installation will fail and you might find it difficult at best to obtain the desired results. Use the Office Update link at

```
http://office.microsoft.com/en-us/default.aspx
```

to locate and install the required updates.

After you download and install all the required Office updates, you can download the Office Compatibility Pack from

```
http://www.microsoft.com/downloads/details.aspx?familyid=
941b3470-3ae9-4aee-8f43-c6bb74cd1466
```

Install the Office Compatibility Pack as you would any other program. Figure 1-4 shows the changes to Word. Notice that you can now open Office 2007 documents.

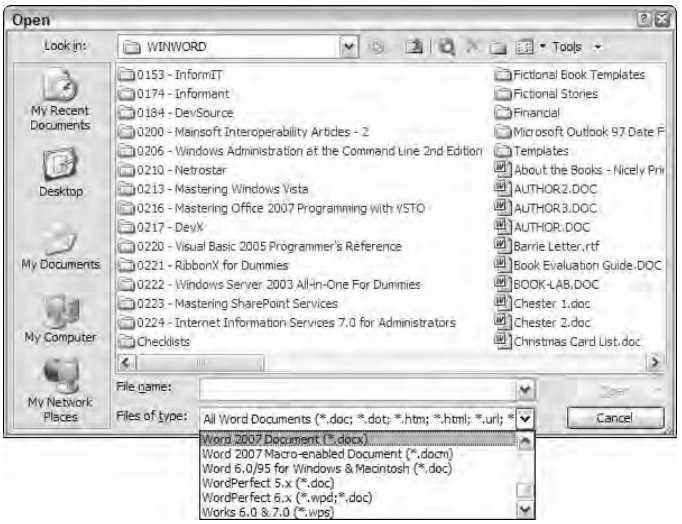


Figure 1-4: Open Office 2007 documents using the familiar Office XP/2003 interfaces.

It's important to remember that Office 2007 makes a distinction between standard documents and those that contain macros. For example, when working with Word 2003, you'd normally use the DOC file extension. However, when working with Word 2007, you'll always save documents with macros using the DOCM file extension. Likewise, if the file contains no macros, save it using the DOCX file extension.

Working with the Office Migration Planning Manager

It's important to plan the migration to Office 2007. No one is going to move a complex set of applications to Office 2007 without performing some type of review process and understanding precisely what the move requires. The

Office Migration Planning Manager (OMPM) performs an audit of your system and helps you plan the migration to Office 2007 with greater ease. You can download this tool from

```
http://www.microsoft.com/downloads/details.aspx?FamilyID=
13580cd7-a8bc-40ef-8281-dd2c325a5a81
```

The product actually consists of two downloads. The first download contains release information that tells you about the changes in the OMPM, which is in version 2 as of this writing. For example, if you don't read the release notes, you won't know that the OMPM doesn't work on 64-bit systems. The second download is the actual Office Migration Planning Manager utility.

You should also review the documentation at

```
http://technet2.microsoft.com/Office/en-us/library/
d8f318d4-84ea-4d3e-8918-ea8dacd14f7e1033.mspx
```

This documentation explains the inner workings of the Office Migration Planning Manager and tells how you can best use it to meet specific migration needs. You begin by extracting the files to a specific location on your hard drive, such as `\OMPM`. The rest of this discussion assumes you extracted the files to the `C:\OMPM` folder of your hard drive, but you can use any other folder you choose. (Choosing the `C:\OMPM` folder has the advantage of reducing the number of configuration choices you have to make.)

If you don't really want to work with OMPM now, you can probably skip the rest of this section. The OMPM tool uses a command line interface. You'll begin by configuring a special file to tell the program how to run. The listing of `Offscan.ini` entries appears at

```
http://technet2.microsoft.com/Office/en-us/library/
1850987f-87bb-47e9-b370-f4b8af3c39d71033.mspx
```

You'll find this file in the `C:\OMPM\scan` folder. Of all of the setting changes you can make, the most important configuration change (the one you must make) is the `[FoldersToScan]` entry. Beneath this heading, you include the `Folder=` entries that define the locations to scan. You must also provide a unique `RunID=` entry for each scan you perform. After you complete the `Offscan.ini` changes, you can run the Offscan utility. When working in Vista, you must open the command prompt by right-clicking the Command Prompt entry in the Start menu and choosing Run as Administrator from the context menu. Figure 1-5 shows typical output from the command (notice that the title bar begins with the word Administrator to show that this command prompt is in administrator mode).

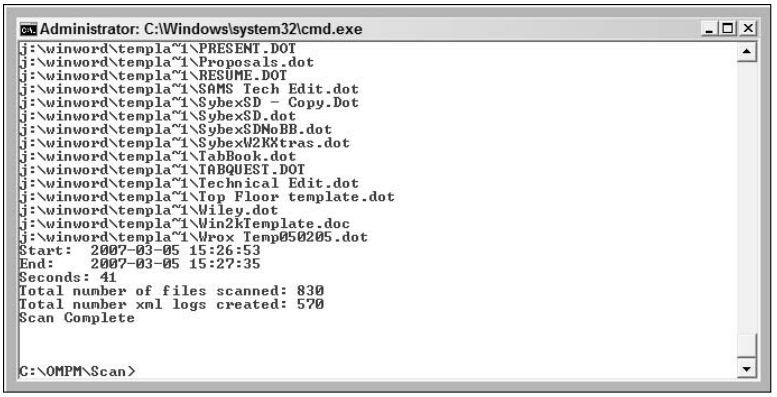


Figure 1-5: Scan your hard drive for potential problem documents and conversion issues.

Microsoft suggests that you distribute the scanning tools to everyone on the network to ensure you get all of the client drives, as well as the data, on the server. Whether you need to perform this step depends on how centralized you keep your data. Some organizations need only an administrator scan of the server's hard drive to locate the required updates.

Eventually you'll end up with a host of `CAB` files that are useless by themselves; you need to import the data into a database to make it useful. This tool requires a copy of SQL Server to run, and, because of compatibility problems, you need SQL Server 2005 SP2 or better when working in Vista. Fortunately, you can obtain a free copy of SQL Server 2005 Express SP2 at

`http://www.microsoft.com/sql/editions/express/default.mspx`

that performs well for this task. You can download the basic version of SQL Server Express (a 36.5MB download), but I recommend getting SQL Server 2005 Express Edition with Advanced Services (a 234MB download) to ensure you have everything you need. If you haven't worked with SQL Server before, check out *SQL Server 2005 For Dummies,* by Andrew Watt, or *SQL Server 2005 Express Edition For Dummies,* by Robert Schneider (both from Wiley Publishing, Inc.).

SQL Server requires a second installation to work with XML data. Be sure you download and install the SqlXml 3.0 Service Pack 3 (SP3) add-in found at

`http://www.microsoft.com/downloads/details.aspx?FamilyID=51d4a154-8e23-47d2-a033-764259cfb53b`

After you install SQL Server or SQL Server Express, use the CreateDB batch file found in the `C:\OMPM\Database` folder to create a database to hold the data you've collected. When you run CreateDB, you must supply the name of the server, the SQL Server instance, and the name of the database you use. For example, if you're using SQL Server Express and the name of your server is `MyServer`, you'll likely type something like

```
CreateDB MyServer\SQLExpress OMPM001
```

at the command line and then press Enter. You can always delete the database if you make a mistake by using the DeleteDB batch file.

You may get an error message from the `CreateDB` batch file, complaining that it can't create the required database. Don't worry; you can normally fix this problem with a simple change. The following steps tell you how to perform this task.

1. **Select Start➪Programs➪Microsoft SQL Server 2005➪Configuration Tools➪SQL Server Configuration Manager.**

   You'll see the SQL Server Configuration Manager window.

2. **Open the SQL Server Configuration Manager (Local)\SQL Server 2005 Network Configuration\Protocols for SQLEXPRESS (or other server instance) folder.**

   You'll see a list of protocols for the selected SQL Server instance, as shown in Figure 1-6.

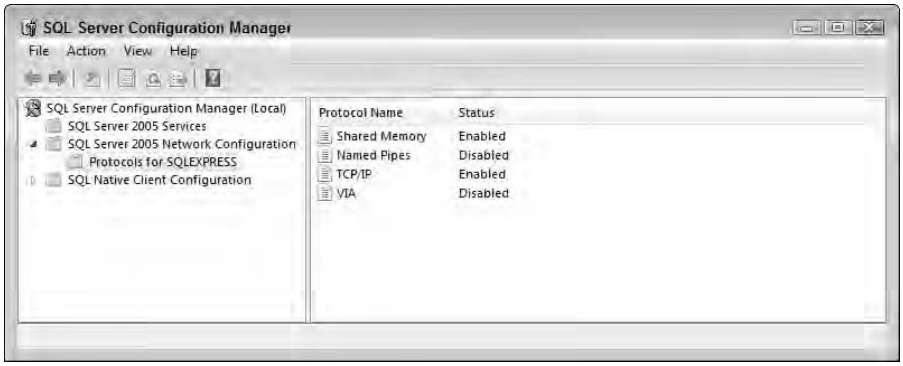


Figure 1-6: Enable the TCP/IP protocol so the system can communicate with SQL Server.

3. **Right-click TCP/IP and choose Enable from the context menu.**

   You'll see a warning message stating the change won't take effect until you restart the service.

4. **Click OK.**
5. **Open the Server Configuration Manager (Local)\SQL Server 2005 Services folder.**

   You'll see a list of SQL Server-related services installed on your system, as shown in Figure 1-7.

Figure 1-7: Restart the SQL Server service to make the changes permanent.

6. **Right-click the SQL Server (SQLEXPRESS) (or other instance) entry and choose Restart from the context menu.**

   The `CreateDB` batch file should execute properly when you try it again.

Now that you have a handy database to use for storage, you can import the data into it. The `ImportScans` batch file in the `C:\OMPM\Database` folder performs this task. As with the CreateDB batch file, you supply the name of the server, the SQL Server instance, and the database. In addition, you must supply the location of the scan files, which is `C:\OMPM\SCANDATA` if you use the default settings. Given the same setup as before, you might type

```
ImportScans MyServer\SQLExpress OMPM001 C:\OMPM\SCANDATA
```

at the command line and press Enter to complete this part of the task.

If you find that the ImportScans batch file fails because it can't find the OSQL utility, you can add the utility to the command prompt path. Normally you'll find this utility in the following folder on your system:

```
C:\Program Files\Microsoft SQL Server\90\Tools\Binn
```

To get there, type

```
Path = C:\Program Files\Microsoft SQL Server\90\Tools\
Binn;%PATH%
```

With this new path added to your command prompt, try the `ImportScans` batch file again.

After all of this work, you're probably wondering about the payoff. Importing the data into the database lets you begin analysis. The OMPM helps you create reports that you'll use to update your Office environment later. To begin using reports, double-click the `OMPM.ACCDR` icon in the `C:\OMPM\Report` folder. If you're using Vista, you'll very likely see a number of security messages that you'll use to elevate your privileges to the required level. After you supply the name of the database used to hold your data (OMPM001 in the example), you can begin working with reports. Figure 1-8 shows a typical example. In this case, the report tells you about the Word compatibility concerns for a set of existing documents.

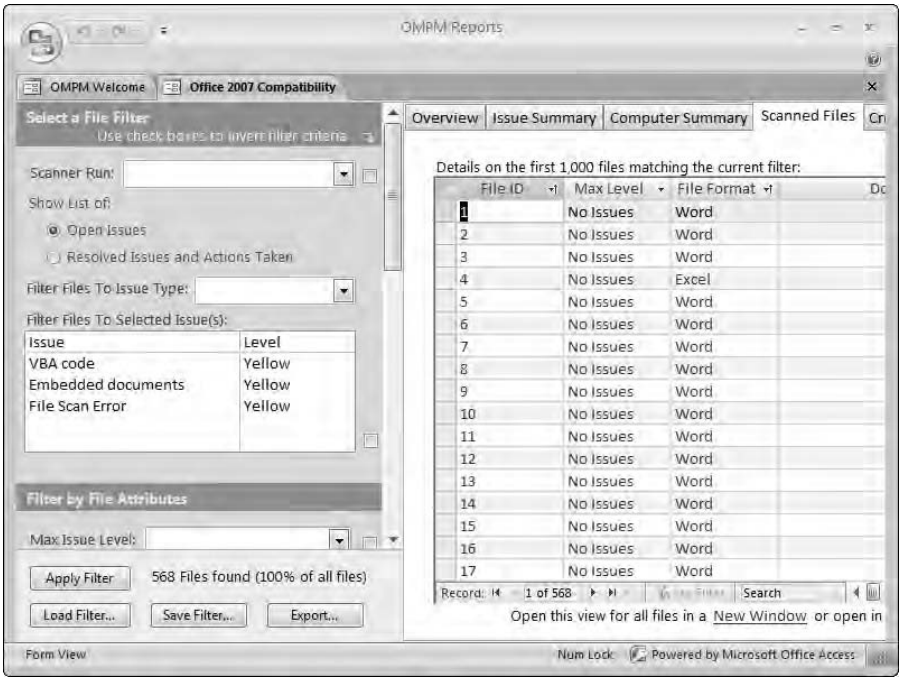


Figure 1-8: Create a strategy for updating to Office 2007 using these reports as a basis.

Getting quick information about XML

Many of the new features of Windows and Office rely on XML for configuration. The reasons that Microsoft uses XML are that it's easy to understand, very flexible, standardized in format, and text based. Of course, you may not have used XML and might not understand how it works. Fortunately, the XML used with Office for RibbonX is straightforward; you don't need to delve into the depths of XML to understand it. Even so, if you haven't used XML before, you might want to visit the XML tutorial at `http://www.w3schools.com/xml/`.

RibbonX also relies on at least one namespace to get the job done. In this case, the URL defines a location that specifies how the RibbonX entries work. Generally, you don't need to know too much about the namespace — except for how to include it (as described in the "Understanding tabs" section of the chapter). If you do want to know more about namespaces, check out the tutorial at `http://www.zvon.org/index.php?nav_id=172&ns=34`.

At some point, you might find that you want to go deeper into XML. You'll find great references online — including one at `http://www.xml.com/axml/axml.html`. Don't forget to review Microsoft's offerings at `http://msdn.microsoft.com/xml/`.

It's important to consider all the compatibility concerns for a set of documents before you begin updating your application. Only when the documents provide a reasonable level of compatibility should you consider updating your applications. If you have hundreds of documents that require considerable conversion, it may be better to stick with the old version of Office for those documents and start fresh with a new application for new documents. Obviously, the decision to update depends on a number of factors, including the value of the data to your organization. Sometimes you have to update the documents, no matter how much it costs, because of the data's value. You can find a complete listing of migration considerations at

`http://technet2.microsoft.com/Office/en-us/library/1db55715-df10-428d-ad42-4ce3c58a8edf1033.mspx`

Defining the RibbonX Elements

Every physical element in the Ribbon has a corresponding element in the RibbonX programming interface. The user sees the results of changes you make with code. Unlike previous versions of Office, however, RibbonX doesn't

rely on a hierarchical set of objects to control the interface. Instead, the interface relies on an XML file that describes the various elements. This file follows the following hierarchy of XML elements:

- Tabs
- Groups
- Controls

Unlike many other Microsoft offerings, the hierarchy for RibbonX is relatively absolute. A tab can't contain other tabs, and it only holds groups. Likewise, you place controls in groups; you don't place them directly in tabs. Chapters 2 and 3 provide detailed descriptions of how to build a good Ribbon interface. The following sections provide an overview of the various elements.

Understanding tabs

Tabs are the uppermost element in the Ribbon hierarchy. You can see an example of a custom tab in Figure 1-9. In this case, you're looking at a custom tab called My Tab that contains a single group. The group, My Group, contains a single button named My Button. Obviously, you wouldn't create a real-world tab like this, but it's good for explanation purposes.

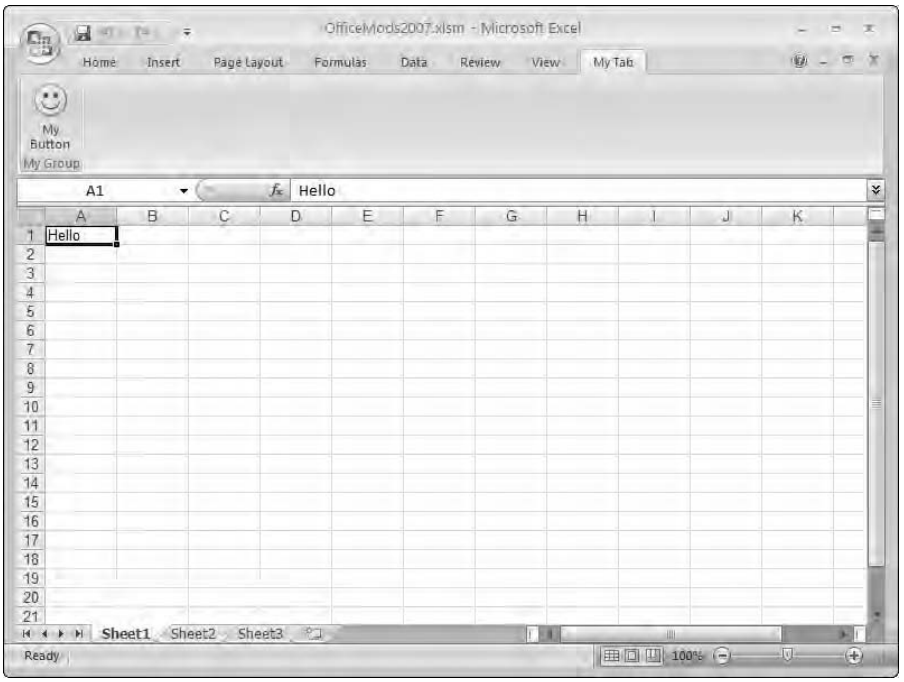


Figure 1-9: Each physical element of the Ribbon has a corresponding RibbonX element.

Generally, you'll use tabs to focus user attention on a particular task or requirement. For example, when you look at the Word Mailings tab, you see everything required to mail a document to someone. The tab focuses the user's attention on mailing the document; not on another task such as formatting it. You can also create custom tabs that focus user attention on specific tasks for your company.

To create My Tab, I wrote the XML file shown in Listing 1-1. As you can see, this is a standard XML file with all of the usual features including a processing instruction, root element (`<customUI>`), child elements such as `<ribbon>`, and properties. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Listing 1-1: Creating New Ribbon Elements Using XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customUI
xmlns=
"http://schemas.microsoft.com/office/2006/01/customui">

  <ribbon>
    <tabs>
      <tab id="myTab" label="My Tab">
        <group id="myGroup" label="My Group">
          <button id="myButton"
                  label="My Button"
                  imageMso="HappyFace"
                  size="large"
                  onAction="myButton_ClickHandler"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

You must define a namespace for the custom interface. In this case, the namespace resides at `http://schemas.microsoft.com/office/2006/01/customui`, which is the location you'll use for every custom interface element you create.

Office does a lot of the work for you when it comes to creating a new tab. Although this implementation is minimal, it gives you an idea of what you can do with very little XML. All you need to create a new tab is the `<tab>` element with `id` and `label` properties. You use the `id` attribute to access the tab from your application. The user sees the text you provide as part of the `label` attribute. As you can see, the physical presentation on the Ribbon always corresponds to the XML you provide as part of the RibbonX interface.

Understanding groups

Groups gather like controls together so the user doesn't spend as much time looking for the right control. Using groups reduces user confusion and makes it easier to show users how to perform a particular task. For example, when working with the Word Mailings tab, you'll see a Create group. The Create group contains controls for creating both envelopes and labels. The user doesn't have to look around for either item; they both appear in the same place.

In the example shown in Figure 1-9, you see My Group, which contains a single control. You create a group in XML by using the `<group>` element. As with a tab, you must include an `id` attribute, which identifies the group in your code, and a `label` attribute, which provides text that the user can use to identify the group on-screen.

Understanding controls

A control performs a specific task. You don't want to pile multiple tasks onto one control because, in many cases, doing so confuses the user. Each control should perform a specific task; you should always choose a control that performs the task well. For example, you can use a pushbutton control to help the user execute a task. On the other hand, you might use a check box to let the user make a choice. A drop-down list box lets the user choose between multiple choices instead of a simple yes/no choice.

Some controls come in multiple sizes. Look at the Word Review tab and you'll notice that the Proofing group contains multiple pushbutton sizes. The large pushbuttons draw the user's attention to major tasks, such as checking spelling and grammar. The small pushbuttons help the user perform less common tasks, such as setting the document language or performing a word count.

Controls always require more code than any other Ribbon element because they aren't static; they perform some task. Look again at Listing 1-1 — you'll notice that the simple `<button>` element requires multiple arguments. As with the other elements discussed so far, you must provide `id` and `label` attributes. When your button includes an image, you must provide the name of the image as part of the `imageMso` attribute. The image must also appear within the file that references it; you'll see how the referencing works in Chapters 3, 4, and 5. The `size` attribute defines the size of the pushbutton. Most pushbuttons also include an `onAction` attribute that connects the pushbutton to code you create.

The `onAction` attribute is special because it reflects an event instead of a property. When the user clicks the pushbutton, the `onAction` event occurs. Most controls provide access to more than one event (as you'll see in the "Defining the RibbonX Controls " section of Chapter 3).

Considering the Ribbon in Office 2007

Before you embark on the journey of creating your own Ribbon elements, it helps to understand what Microsoft provides within Office. You may find that you can make small changes to the existing setup and still obtain a usable interface. For example, you may decide to place your custom styles in the Styles group of the Home tab, rather than create a custom tab for the purpose. The following sections provide you with an overview of the existing Ribbon elements found in Office 2007 so you can choose which application features to update and which to make part of existing Ribbon elements.

Understanding the common Ribbon elements

Every Office application except Outlook has a Home tab included with it. The Home tab contains the elements that the user needs most often. For example, the Home tab for Word contains common formatting groups such as Font, Paragraph, and Styles. It also has common editing tools and provides access to the Clipboard. The Home tab for PowerPoint includes many of these elements, along with a Drawing group that contains common drawing tools. Access doesn't really feature much in the way of data formatting; its Home tab focuses more on data sorting and manipulation. Excel includes some formatting tools on its Home tab, but you'll also find Number, Alignment, and Cells groups that contain tools for working with worksheets.

The Office applications also include common task-based tabs. The two most common tabs are Insert and Review. The Insert tab contains groups that help the user insert data. When working with Word, you'll find groups for headers, footers, text, symbols, links, tables, illustrations, and pages. Excel has special features on its Insert tab to add charts to a worksheet; PowerPoint provides numerous graphics features on its Insert tab. The Insert tab is an example of a task-based tab that focuses on a particular task as it occurs in that application.

Contrast the Insert tab with the Review tab. The Review tab varies little between applications. For example, all of the applications that support it include Proofing and Comments groups since these are common review task requirements. The Review tab also includes document protection features, even though these features don't always appear as part of a Protect group. The point is that this task is common among applications and you should strive to maintain that commonality as much as possible when creating a custom application.

Because Word and Excel deal with larger documents, they both include a Page Layout tab as well. In both cases, you find Themes and Page Setup groups that contain controls for managing the pages as a whole. Word includes a Paragraph group because it works specifically with text in paragraphs. Excel, on the other hand, includes a Sheet Options group with controls that help you manage the appearance of a single large worksheet. The Page Layout tab is an example of a tab that includes both common and application-specific features.

Looking at the Ribbon in Word

Word is one of the most complex implementations of the Ribbon for good reason: Manipulating text so it looks perfect when printed is a difficult task, and even with the right tools you can make mistakes. The References tab in Word is an example of a tab that performs a specialized task. You use the groups on the References tab to add citations, footnotes, a table of contents, an index, and other document-specific features to a file. Not everyone will use this particular tab, so this is a tab you may choose to remove from view when creating a custom application.

The specialized toolbars and menus that developers add to Word also indicate the complexity of working with the written word. Unfortunately, Word has the dubious distinction of providing the applications least likely to move from previous versions of Office to Office 2007. The reason that people have created so many applications for Word is also the reason you'll probably end up moving the application to Office 2007 or starting over from scratch.

Looking at the Ribbon in Excel

Excel users require access to formulas to compute entries in a worksheet. The Formulas tab contains features that help the user work with formulas, including the Function Library group, which contains common formulas. You'll also

find groups for managing named ranges, auditing existing formulas, and performing calculations. Generally, if you have to add a new formula to Excel, this is the tab to hold it. Rather than create a custom tab for the task, you can remove the features you aren't using and add the custom features your application requires.

Worksheets often require external access to data stored in another location. The Data tab provides basic data-access functionality, including the Get External Data button. Unfortunately, these features are generic; and they may not meet your specific needs. If you have complex data requirements for your application, you'll very likely add those features to this tab. Unlike many other tabs, this one isn't loaded with a lot of extra features you won't use, so you should be able to maintain the default setup and simply add the new controls you need.

Looking at the Ribbon in Access

The first thing that will surprise you when you look at Access is the small number of tabs (as compared to other Office products). The two main tabs are Create and External Data. You use the Create tab to define new database objects, and the External Data tab to access data outside of Access. The Database Tools folder contains esoteric items for creating macros, defining relationships, analyzing and moving data, manipulating database content, and administering the database. Generally speaking, you probably won't want to modify any of these tabs, but you may want to hide them from users. Placing any custom features you want to provide on a special tab makes sense with Access, especially considering that you aren't fighting with the application for space.

Looking at the Ribbon in Outlook

When you initially start Outlook, you might wonder whether it even uses the Ribbon. The Ribbon doesn't appear as part of the main application, but it does appear as part of special features, such as creating a message, adding an appointment, and defining a new task. Consequently, any work you perform with the Outlook Ribbon will be task-specific.

Outlook developers are also going to have to maintain their current skills, because Outlook uses the older toolbar and menu interface for performing general tasks. For example, if you want to add a special feature for signing up

for Internet access, you'll need to add it to a toolbar or menu, not to the Ribbon. Only when you want to add a task-specific feature — such as special headings for an e-mail message — do you need to worry about the Ribbon in Access.

Looking at the Ribbon in PowerPoint

The special Ribbon features in PowerPoint all deal with the slides you create. These tabs include Design, Animations, and Slideshow.

All three tabs control the slides in some way. The Design tab includes features for changing the slide appearance. For example, you can use the features in the Themes group to control the overall theme for your presentation. The Animations tab controls how you move from one slide to another. For example, you can add a special transition or sound between slides. The Slideshow tab contains controls for changing the slide presentation. For example, the Set Up group contains controls that let you choose how the slideshow begins and ends. You can also use these features to rehearse the slideshow timing. When you want to add custom features to PowerPoint that affect the slides in any of these ways, you'll want to augment one of the existing tabs.

Of course, slideshows don't deal just with slides. You might want to add a special tab for accessing external data. In some cases, you won't find a standard tab to hold the special features you want to add to PowerPoint, in which case, you'll want to add a new tab, rather than change the focus of an existing tab.

Chapter 2

Creating an Effective RibbonX Design

In This Chapter

- Deciding on realistic and effective goals
- Making your RibbonX design work for everyone
- Considering the best way to arrange RibbonX elements
- Defining how XML plays a part in RibbonX

Developers spend a great deal of time on design issues. A productivity application isn't much good if the user doesn't become more productive because of using it. When an application places undue burden on the users and forces them to jump through hoops to accomplish even basic tasks, then the application has failed. As mentioned in Chapter 1, Microsoft's main goal in moving to the Ribbon interface is to make Office easier to use. Whether this design change proves effective depends a great deal on how users perceive the change. If the Ribbon doesn't truly make users more productive, Microsoft may very well end up going back to the (virtual) drawing board.

Your applications have to meet productivity, usability, visibility, and other goals as well. The Ribbon presents some special challenges to developers, as well as opportunities to excel. Overall, you'll need to change your development techniques to match the new Ribbon interface. While you may have placed a new command on a menu in the past, now you're going to need to find a tab to use instead. In many cases, you can't simply stick the new feature just anywhere; it's important to follow the task-oriented focus of the Ribbon interface.

This chapter helps you design effective RibbonX interfaces for your applications. An effective interface — at minimum — meets all of the user's productivity, usability, and visibility goals. In addition, the interface has to meet all the requirements for your application. Here's where those two requirements come together: It's important that the application behave as anticipated when the user clicks the button. Part of your ability to meet that goal resides in how the user interacts with the Ribbon, which means that the interface could make or break your application. The last part of this chapter delves a

little deeper into the XML part of the interface. Once you have an idea of how to design an effective interface, you'll have to put those design decisions to work by creating the required XML for your application.

Developing RibbonX Element Goals

This chapter considers the interface that the application you create relies on to perform a given task. The Ribbon forces you to consider how a user performs a task. When you work with menu-and-toolbar applications, it's easy to consider convenience as one of the major goals in an application design. An application designer can place a new feature on a toolbar when the user requires quick access; otherwise a menu may be more appropriate. The choice of toolbar or menu depends as much on the other items in that toolbar or menu as it does on how the user works with them. However, look at the Ribbon display shown in Figure 2-1: The focus is now on the task.

Figure 2-1: The Ribbon focuses attention on a task, rather than on convenience.

In this case, you're looking at the Formulas tab of the Excel Ribbon. The focus is on manipulating data using some type of math. The user isn't concerned about creating new data or importing data from another source. Everything on this tab concerns working with formulas in some way. If your application doesn't work with formulas, you shouldn't place any part of your application here because the focus is on formulas. In fact, this intense focus on tasks is going to cause many developers problems because (in many cases) they don't actually *know* how users work; questioning the users probably won't provide much good information, either.

The overall goal is to create a task-oriented application when working with RibbonX. However, designing such an application requires that you also achieve a number of sub-goals. The following list provides an overview of the sub-goals you should achieve to ensure that the RibbonX additions you make actually work with the rest of the Ribbon elements in the target application (try to achieve the goals in the order shown):

1. **Separate tasks into steps that a user is likely to perform at one time.** For example, writing a letter necessarily implies that the user will perform an addressing task at some point in the process.
2. **Observe a user performing the task.** Some organizations record keystrokes; others use cameras. The point is to capture the process the user actually relies on to perform the task.
3. **Break down the process into individual steps that each require interaction with (at most) one control.**
4. **Determine how often users are likely to perform a particular step.** For example, when addressing an envelope, not all users will look up the addressee; some users will have the address information memorized or obtain it from alternative sources.
5. **Create a list of controls that the user needs to perform the task.** Remember that the Ribbon provides a wealth of alternative controls that you didn't have access to when working with the menu and toolbar setup.
6. **Determine whether the controls fit within an existing tab.** Consider using Contextual Tabsets whenever possible so the tabs appear only when the user needs them. (Don't worry about how Contextual Tabsets work for now; you get a closer look at them later in the chapter.)

   If you don't find an existing tab to use for the controls, define a new tab to hold them. Never use multiple tabs to hold the controls for a single task.
7. **Determine whether any of the existing groups can hold your controls.** If not, define a series of task-oriented groups to hold the controls. You might break down an addressing task into locating an addressee, formatting labels, and adding postage groups.
8. **Order groups by usage.** The user's attention should move from left to right on-screen (or right to left if the user's language normally moves in that direction) while using the groups.
9. **Place the controls that you previously defined into their groups.**
10. **Assign prominence to each of the controls.** If a user uses a particular control every time, then the control should receive greater prominence than a control the user seldom requires. Use larger controls and better placement for controls with greater prominence.
11. **Determine whether you need to include dialog boxes to show advanced or alternative options.** If so, you'll want to add the appropriate button in the lower-right corner of the group area (some developers are calling this a *dialog-box launcher,* which is the term I use throughout the book).

You may find you need to perform additional steps — and create additional goals — for a particular application. For example, you might find that power users require one set of steps, and standard users another set of steps. In addition, power users may require access to features that you don't want standard users to use. All these goals should help you think about the Ribbon in a manner that you might not have thought about in the menu-and-toolbar interface.

Never include the Quick Access Toolbar (QAT) as part of your application design goals. Microsoft sets the QAT apart for user customization. The only way you can override the QAT is to create a completely new Ribbon, using the `StartFromScratch` mode. Using the `StartFromScratch` mode means the application removes all standard tabs and Office Menu entries. You start (of course) from scratch — and have to implement everything from the ground up. That's why it's usually easier to avoid using the QAT.

Considering RibbonX Element Accessibility and Visibility

When you're creating a RibbonX application, one of the most important enabling goals is to improve the accessibility and visibility of features. The focus on tasks, rather than features, attains part of this goal for you. Unfortunately, a pretty new interface and reorganization of existing features probably won't accomplish all of your application goals. Making it easier for the user to work with your application is at least as important. In some cases, achieving this goal means hiding features, but generally it means putting them in the right location and providing visible usage aids. The following sections describe some techniques you can use to achieve the goals of enabling better accessibility and visibility.

Using tooltips

The menu-and-toolbar interface provides limited opportunities to enhance the user experience. For example, you can add tooltips to a menu-and-toolbar application, but the tooltips provide a single line of limited information (even when the tooltip appears on multiple lines, it actually consumes just one line within the application code). The elements in a menu-and-toolbar interface are the same size and you don't have good access to icons. Yes, you can create an icon, but the icon size reduces the amount of information available to the user of the application.

The Ribbon overcomes some of the problems that users experienced in the past in figuring out what a particular control does. Hover the mouse pointer over the Paste icon in any of the applications that support that operation (you'll see it on the Home tab), and you'll see a standard tooltip like the one shown in Figure 2-2. Because most people understand what the Paste control does, you can use a standard tooltip. However, notice that even a standard tooltip provides more information than the tooltips of the past. In this case, the user knows that pressing Ctrl+V will perform a standard paste operation in the current application.

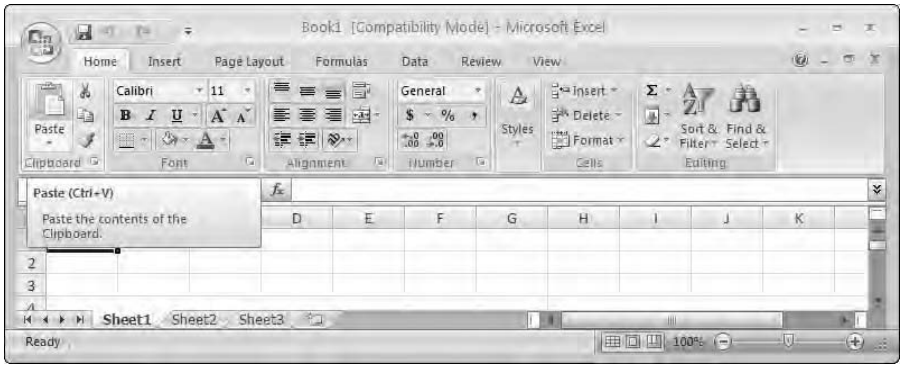


Figure 2-2: Standard tooltips in the Ribbon convey more information than standard tooltips in the past.

In addition to the standard tooltip, the Ribbon can also support a super-tooltip. Figure 2-3 shows an example of a super-tooltip. Notice that the help text appears on multiple lines. Using this approach lets you create clearer direction to the user. Notice that this tooltip also includes a help indicator. If the user presses F1 while the super-tooltip is displayed, the system displays context-sensitive help about the topic.

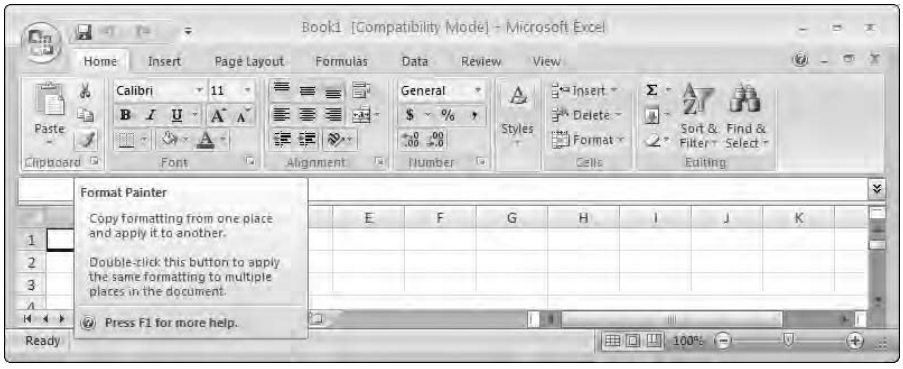


Figure 2-3: Use super-tooltips when a control performs a complex task and you want to provide a better explanation.

A third kind of tooltip displays little dialog boxes that you can display when the user hovers the mouse pointer over a dialog-box launcher. In addition, you'll see this kind of tooltip when you hover the mouse pointer over a special area, such as the Office button. (Figure 2-4 shows an example of a dialog-box tooltip.) Notice that you can see the entire dialog box as a thumbnail. The user obtains a complete description of what the dialog box contains. In addition, you can add a help indicator to the information the tooltip provides.

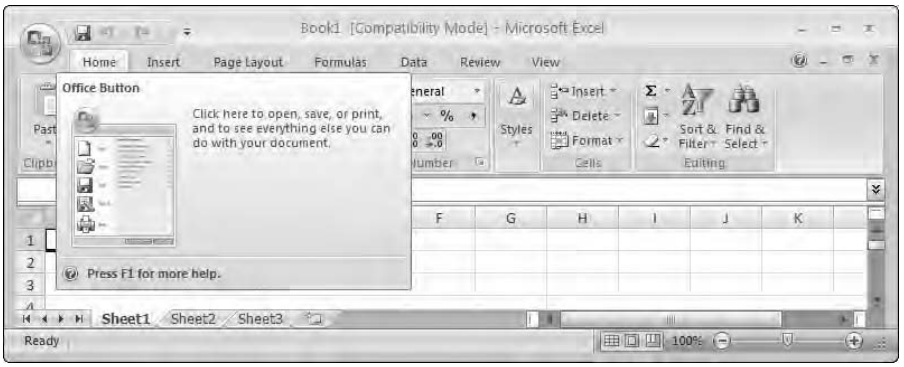


Figure 2-4: Dialog-box tooltips help a user see a dialog box before opening it.

Using existing Office features

In some cases, you might want to use some of the remaining Office features from previous versions of Office. For example, the task pane is the same as before from a programming perspective. It's possible to add some of your existing application features to a custom task pane so that the same feature appears in both Office 2007 and older versions of Office. Unfortunately, you can't use VBA to implement this solution.

Another alternative to consider is context menus, which rely on the existing CommandBars object model (so this strategy also works for all versions of Office). Right-clicking an object displays a context menu whose content is limited to tasks you can perform with the selected object. Even so, you may find that this is a good solution to overcome some version differences. You can even use VBA to implement this solution. The "Using Existing Office Features" section of Chapter 12 describes both the custom task pane and context menu strategies in detail.

Using the Office Menu

Remember that you can also change the Office Menu (shown in Figure 2-5) as needed to meet application requirements. Look at the Office Menu as a kind of advanced File menu from days past. You shouldn't use it to hold task-based features of your application (more about that in a minute). However, if your

application performs a special kind of save or other file-related task, you can place that feature on the Office Menu, rather than on the Ribbon.

You can add new options to an existing entry, such as an alternative printing method to the Print entry, or you can create entirely new entries. The Office menu can help you fill the gaps between standard and power users as needed. In addition, it does make the best place to put nontask-related items.

Make sure you avoid the trap of adding tasks to the Office menu, however. Adding a task to the Office menu makes it less visible to the end user (after all, one of the main purposes of using the Ribbon is to make application features more visible). Such an addition also runs counter to the workflow emphasis of the new Office products. Developers who are used to working with the menu-and-toolbar system will have to pay special attention to this trap.

Using Contextual Tabsets

Contextual Tabsets appear when you perform certain tasks in Office. For example, when you create a table in Word, you see the Table Tools Contextual Tabset shown in Figure 2-6. Notice that the Contextual Tabset appears above the normal tags on the Ribbon; this is your first clue that this is a Contextual Tabset, rather than a standard tab.

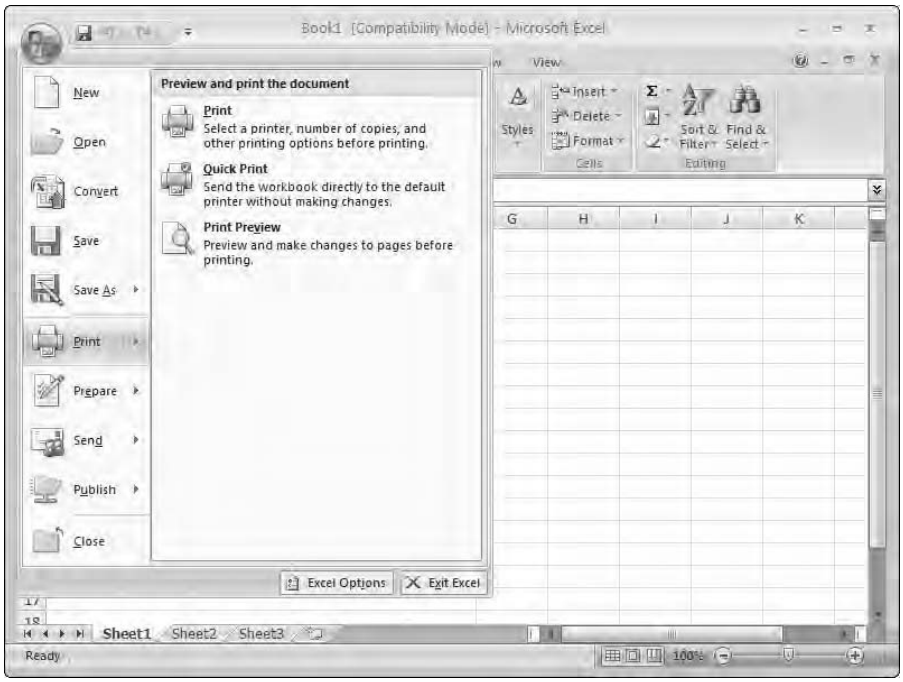


Figure 2-5: The Office menu is an excellent place to put nontask application features.

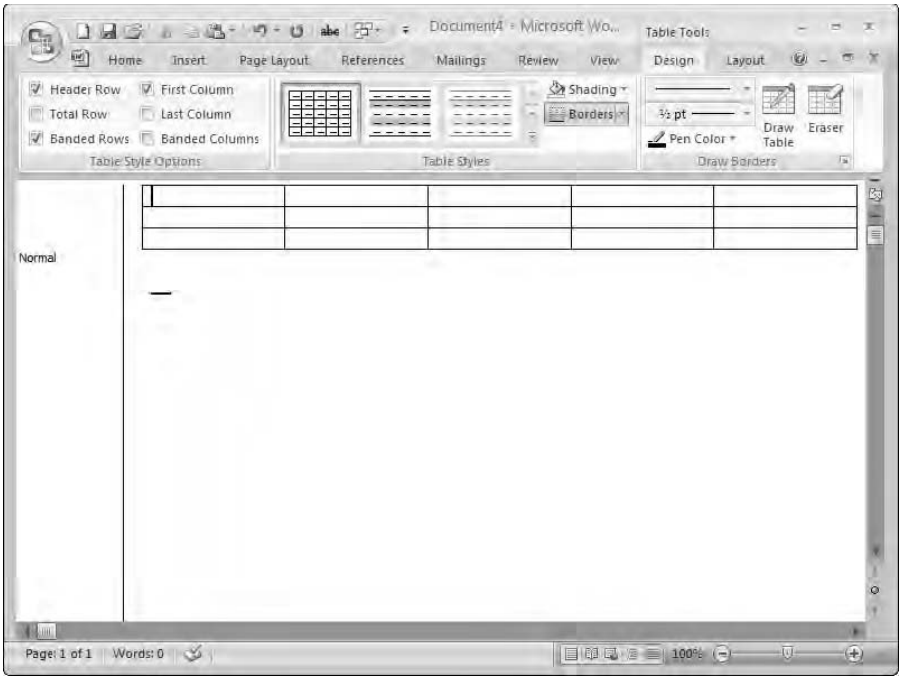


Figure 2-6: Contextual Tabsets offer an opportunity to add custom context-based features.

Below the Contextual Tabset are tabs related to the Contextual Tabset. A line extends from the Contextual Tabset to highlight the two tabs: Design and Layout. These tabs appear only when Word displays the Table Tools Contextual Tabset.

The bad news is that you can't create your own Contextual Tabsets in Office 2007. Microsoft purportedly plans to add this feature to a future version of Office (but they aren't saying when). Fortunately, you can modify the existing tabs for a Contextual Tabset, hide the existing tabs when that's required, and add custom tabs of your own.

Repurposing the MiniToolbar

The MiniToolbar provides a listing of formatting commands that the user can use with highlighted text or other objects in a document, as shown in Figure 2-7. It appears wherever you right-click on text within the document. You can't add any new content to the MiniToolbar, which means that you should probably consider the MiniToolbar as the last alternative for adding new features. You can, however, hide existing features or repurpose elements to perform another task.

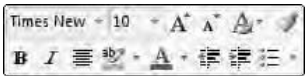


Figure 2-7: The MiniToolbar appears automatically when the user selects text or an object.

Repurposing is the act of hijacking a control for your own use. For example, you might decide to hijack the Format Painter control to add custom formatting required by your company to a document. You can actually repurpose controls in the Ribbon and in the Office Menu as well.

It's usually a bad idea to repurpose controls. When the user expects a control to perform a certain task and it no longer performs that task, you can expect a high level of confusion, not to mention added support costs for the application. The only time you should consider repurposing a control is when the control does something you don't want the user to do (because of company policy or simply because you consider the action dangerous). Make certain that you document any repurposing you do — completely — because other people might not have the same understanding about it that you do.

Whenever you repurpose a control, you should consider making the original functionality available to the user. For example, you might repurpose the Print control on the Office Menu to point the user to a specific printer on your network. You could also let the user choose the original printing feature when the designated printer is in use by someone else for a long print job.

Defining an Effective RibbonX Design

The previous sections of this chapter describe how to define Ribbon design goals and where to place various application elements that you define. After you make these decisions, consider the precise design of your Ribbon elements. For example, you can use text, icons, or a combination of both to relate the purpose of most controls. The method you use to convey the control purpose depends on the application user and the purpose of the control within your application. The following sections describe naming and other concerns for RibbonX applications.

Using names effectively

Look again at Figure 2-1. You'll notice that every tab and every group has a name. Whenever you create new tabs or groups, you must also provide a name. The name should meet the following criteria:

- **Short:** The name should contain only one or two words or you won't be able to see it if the user resizes the application.
- **Task-focused:** Make sure the name reflects the task you want the user to perform.
- **Unique:** Even though you could possibly create (say) two Page Layout tabs in Word, doing so would confuse the user.
- **Simple:** Use the simplest possible term to define the task.

Controls don't always include a name. Less-used controls often rely on smaller pushbuttons and only an icon. Using a larger control and text tends to draw the user's attention to that control. As with the text for tabs and groups, you should make the control label short, unique, and simple. In this case, however, you should ensure the label reflects the purpose of the control, rather than the task the user is performing.

Considering the number of items on a tab

The Ribbon automatically adjusts itself to display the amount of information that can fit within the space the user allows. In most cases, the user can see all the tabs and groups you provide, as long as the display area doesn't become too small. For example, I was able to size Excel down to 650 pixels in width before the group names began to disappear, and I could still access all groups — even when I reduced the application width to 420 pixels (as shown in Figure 2-8). However, many of the Home tab features begin to disappear at around 700 pixels. Consequently, the effective productive size for Excel is 700 pixels — which works for most systems today.

What happens, though, if you begin placing more items on a tab than Excel places on the Home tab? The effective productive size begins to increase. At some point, the user can't make the application large enough to display all of the controls you place on the tab. When the application reaches this point, the changes you make to the Ribbon are extraneous because the user won't even see them. In general, you should use the tab with the most default controls in an application as the basis for your custom changes. For most Office applications, the Home tab contains the most controls.

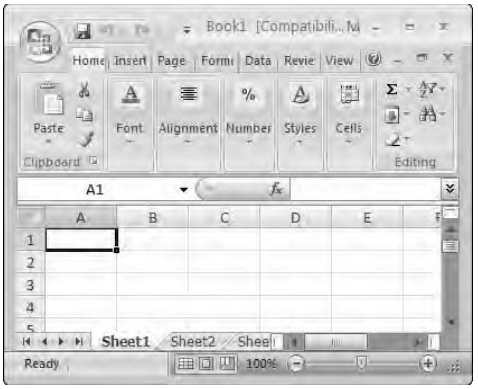


Figure 2-8: Excel can display all of its tabs and groups on the Home tab even at 420 pixels.

Looking at groups from the user's perspective

One of the disconnects between developers and users is that they look at applications differently. The Ribbon certainly helps to redirect developer attention toward user needs, but it still probably isn't enough to provide the ultimate solution. The problem is one that probably won't go away — simply because a user's perspective is always going to differ from that of the developer.

The user's goal in working with an application is to accomplish work that has (in most cases) absolutely nothing to do with computers or applications. The user really doesn't want to think too much about the computer; only the task matters. That's why the Ribbon is a good update to Office, even if it completely ruins years of interface work by developers (your business logic works just as well with the Ribbon as it did in the past).

Of course, a developer can still choose not to view tasks from the user's perspective or simply create the application incorrectly because the older application had a certain appearance. When creating groups for your application, always think about contiguous tasks — one step after another. Use action words that convey the task orientation of the Ribbon or nouns that describe the task activity that the group accomplishes. Always order the groups from left to right, just as you'd read a line of text. (Of course, if you normally read text from right to left, you'll want to place the groups from right to left as well.)

Using the right control

An important issue for the Ribbon is choosing the correct control to interact with the user. Unlike the menu-and-toolbar setup of the past, the Ribbon provides you with a wealth of very usable controls that make it easy for the

developer to provide just the right presentation to the user. For example, consider the simple pushbutton. The Ribbon supports a number of button types, including these:

- **Button:** Performs an action or displays a dialog box.
- **Toggle Button:** Turns a feature on or off.
- **Split Button:** Performs one of several actions based on user selection. The split includes a drop-down list of acceptable actions.

However, you actually have more than three button types at your disposal because you can modify the basic buttons. Here are some examples of potential modifications:

- **Size:** You can make the button large or small to change its emphasis.
- **Order:** Placing the buttons in a particular order changes their significance as well. Buttons placed in a particular order may make a user think of related functions or a particular process.
- **Custom images:** You don't have to use icons or text to describe the button. Using a custom image on a large button can significantly change how the user sees it.

The issue of images is an important one to consider because Ribbon controls let you do far more than you could in the past. You can use any of the graphics formats listed in Table 2-1 with the Ribbon. Table 2-1 also tells you about the advantages of using each type.

Table 2-1 Graphics Formats Supported by the Ribbon

File Type	*Advantages*
BMP	This graphic type is easy to create and there are a lot of tools for manipulating it.
JPG	Easily used on Web pages and useful for pictures or other complex graphics.
GIF	Easily used on Web pages. This format doesn't suffer from lossy data storage. It also provides a single level of transparency on the Ribbon for special effects.
PNG	Easily used on Web pages and used to store complex graphics. This format also provides full support for all Ribbon transparency effects.

Providing user hints

The Ribbon provides features you can use to improve the user experience through hints. Some of these hints are built in; you don't need to worry about them. For example, the Ribbon makes the control selection very clear when a user hovers the mouse pointer over the control.

Some of the hints are things you have done in the past. For example, you can use tooltips to improve user communication. The "Using Tooltips" section of the chapter describes this feature in detail. Adding super-tooltips to your application will significantly improve user communication.

Another kind of hint isn't so obvious. When you look at the Styles group in Word, you see an actual presentation of the various styles. This kind of hint isn't readily available in older versions of Office, yet it makes it easy for a user to choose the correct style for a particular need. Below the style representation, the user sees the actual style name — which makes it easy to choose the style by name as well.

Using feature hiding effectively

Feature hiding is a new Microsoft watchword, and they employ it effectively in Office 2007. One example of feature hiding is the use of Contextual Tabsets (described in the "Using Contextual Tabsets" section of this chapter). A Contextual Tabset is a set of tabs that appears on-screen only when you perform a specific task — such as editing a table. When you create an application feature that works with a Contextual Tabset, you can let Office hide the feature for you.

Unfortunately, the Contextual Tabsets won't answer every feature-hiding need. For example, you might need to hide features based on a user's role in an organization or a task for which Microsoft hasn't provided a Contextual Tabset. You can still hide tabs, groups, or individual controls programmatically. It's also possible to hide functionality according to a template or a particular document. Global features can appear as an add-in to ensure that the user can always see them.

Some developers hide features a bit too enthusiastically. What the user ends up with is what I call the Las Vegas Effect: the glittering array of appearing and disappearing controls that dazzles the eye, but doesn't do anything for productivity. If you plan to hide features in your application based on certain criteria, you should place the controls in a single group or on a single tab and hide everything at once if possible. Don't keep displaying and then hiding a feature if the user will need it occasionally while using the application.

Understanding the XML Connection

RibbonX relies on a hierarchical description of tabs, groups, and controls to modify the physical appearance of the Ribbon (the "Defining the RibbonX Elements" section of Chapter 1 describes how this works). The XML you create also affects design decisions. Microsoft hasn't provided a graphical design tool for Office 2007 yet, so you must write the actual XML by hand. In some cases, the use of hand-coding techniques will result in potential errors in the resulting Ribbon interface — so complete testing and verification is important. You should actually make a drawing of how you expect the resulting Ribbon additions to appear to reduce the potential for errors.

Keep in mind that the XML can become quite complex. As you add new features to a control, the number of attributes within the XML increases. For example, here's the XML for a button that includes an image:

```
<button id="MyButton"
        imageMSO="ButtonImage"
        size="Large"
        label="My Button"
        onAction="ClickMe" />
```

The `imageMSO` attribute defines the image to use with the button. You can store this image as part of the add-in, template, or document that you create because RibbonX won't look on the hard drive for the image. If you do decide to load the image, then you have to create special code to do it. Don't worry about the particulars now; you'll discover more about graphics in Chapter 3. When working with graphics — especially custom images — you'll need to use one of the image formats listed in Table 2-1.

It's also possible to use built-in images for your control. For example, if you wanted to use the icon from the `Underline` control, you would simply include `imageMSO="Underline"` as part of your control description. You can download a complete list of the control identifiers for Office 2007 at

`http://www.microsoft.com/downloads/details.aspx?familyid=4329d9e9-4d11-46a5-898d-23e4f331e9ae`

The order in which you place entries in the XML file is important. The application displays the tabs, groups, and controls in the order you specify. Consequently, if you want `Button1` to appear first in the physical presentation on-screen, you must also place it first in the XML file. Unfortunately, it's all too easy to obtain awkward, odd-looking layouts if you aren't careful. Listing 2-1 shows one such example (the listing shows only the tab code).

Listing 2-1: An Example of a Problem Layout

```
<tab id="myTab" label="My Tab">
  <group id="Group1" label="First Group">
    <button id="Button1"
            label="Big Button 1"
            imageMso="HappyFace" size="large"
            onAction="myButton_ClickHandler"/>
    <button id="Button2"
            label="Button 1"
            imageMso="Bold"
            onAction="myButton_ClickHandler"/>
    <button id="Button3"
            label="Big Button 2"
            imageMso="HappyFace"
            size="large"
            onAction="myButton_ClickHandler"/>
    <button id="Button4"
            label="Button 2"
            imageMso="Italic"
            onAction="myButton_ClickHandler"/>
    <button id="Button5"
            label="Button 3"
            imageMso="Underline"
            onAction="myButton_ClickHandler"/>
  </group>
  <group id="Group2" label="Second Group">
    <button id="Button6"
            label="Button 1"
            imageMso="Bold"
            onAction="myButton_ClickHandler"/>
    <button id="Button7"
            label="Button 2"
            imageMso="Italic"
            onAction="myButton_ClickHandler"/>
    <button id="Button8"
            label="Button 3"
            imageMso="Underline"
            onAction="myButton_ClickHandler"/>
  </group>
</tab>
```

You should notice some special issues in this example. All images listed in the code appear as part of Office, so you don't have to supply any of them. It also contains big buttons and regular buttons. However, because of the order in which the big buttons and regular buttons appear, the layout is less than optimal in `Group1`. Figure 2-9 shows what happens when you place the buttons in the wrong order.

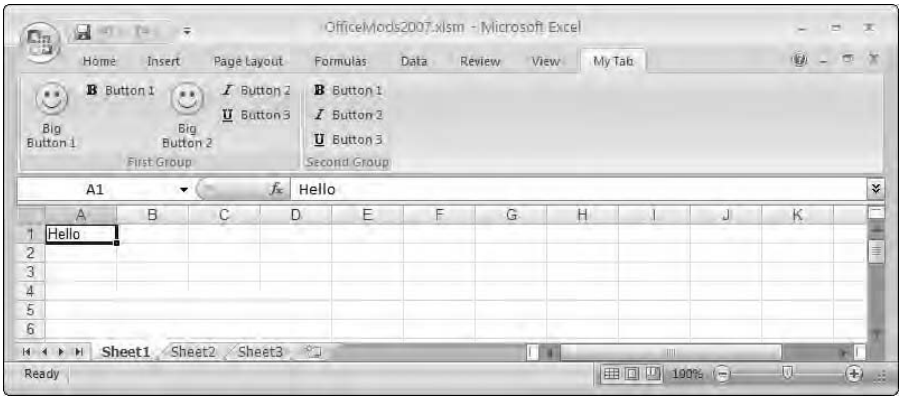
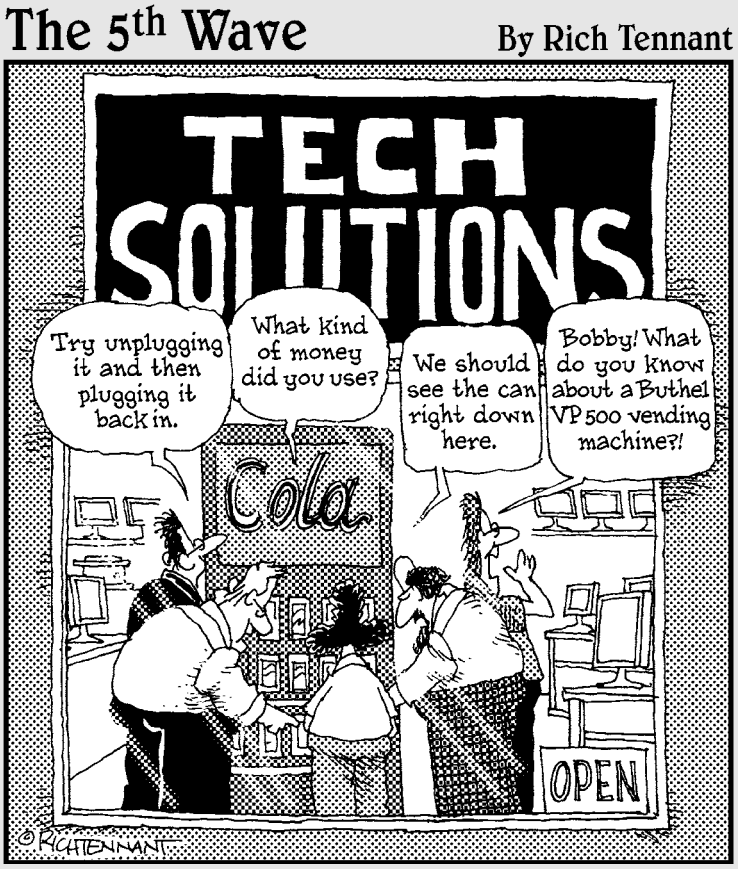


Figure 2-9: Ineffective layouts come from placing the buttons in the wrong order.

Notice the big gaps in this layout. You can actually get three buttons in the space of a single large button, as shown in `Group2`. A better layout for `Group1` would be to place the three regular buttons together so that the two large buttons would attract the proper user attention and the group would use Ribbon space efficiently. The Ribbon does provide a great deal of flexibility, so saying that one layout is the best is impossible. You'll see a considerable number of functional layouts as you review the examples in this book. The accompanying text describes why I used a particular layout so you can more easily decide on a layout for your own application.

Part II

Interacting with the Ribbon



In this part...

The Ribbon is amazingly flexible. In fact, the flexibility could overwhelm some developers who are used to handling just a few controls when they're working with the menu-and-toolbar setup. Chapter 3 starts the adventure by presenting the various strategies for creating a Ribbon. You discover some of the new tools for working with the Ribbon and begin to get a handle on the new techniques you'll use to improve the user experience.

Chapters 4 and 5 complement each other. Chapter 4 shows how to work with the Ribbon using scripts. You discover that VBA users have considerable flexibility in working with the Ribbon — they can even start the Ribbon from scratch — so you can have it your way. Chapter 5 looks at many of the same issues as Chapter 4, but from the perspective of the Visual Studio developer. Your add-ons never looked so good as they can with the Ribbon.

Chapter 3

Designing New RibbonX Elements

In This Chapter

- Developing a RibbonX tab
- Working with groups
- Adding controls
- Creating a simple RibbonX document
- Understanding how to work with graphics

It's time to create your first RibbonX tab. This chapter looks at the process of adding the graphical interface from a number of perspectives and using a variety of techniques. The manual method of adding a Ribbon isn't hard, but it can be cumbersome and error-prone. However, it's important to know how to use this technique (very carefully) if you ever have to repair a document you've created.

Fortunately, you also have an alternative method for working with the Ribbon by using the Office Custom UI Editor created by a third-party developer. This tool doesn't provide you with a graphical interface for working with the XML that RibbonX requires, but it does make the process of adding the XML to the document significantly easier *and* less error-prone.

Adding new RibbonX elements to a template is the same as adding them to a document. However, when you work with a template, you have to consider the effect of that template on every document that uses it. The problem that some developers might encounter is creating RibbonX elements that don't apply to an entire class of documents.

This chapter doesn't consider adding RibbonX elements to add-ins (DLLs). Creating add-ins requires that you use Visual Studio instead of VBA to perform tasks. Consequently, you'll find add-ins discussed in Chapter 5. However, the techniques for creating the RibbonX XML in this chapter apply to add-ins as well. In addition, even though this chapter works with Excel, you'll find that the RibbonX techniques apply equally well to other Office 2007 products. Part III of this book discusses each of the Office applications in individual chapters.

Creating a RibbonX Tab

The first step in creating a new RibbonX application is to define the goals for the application as described in Chapter 2. When you decide that you need a new tab to display the groups and controls for your application, it's a good idea to add the tab first and ensure that it appears as you think it will. In addition, adding the tab first lets you create and test any code required to make the tab functional. The following steps get you started working with an example tab in Excel. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

1. **Open Excel, define any data required for the example (none in this case), and save the file using the `XLSM` extension.**

   Make sure you use the `XLSM` extension or Excel won't recognize any macros you provide in the application. The example uses a filename of `NewElements.xlsm`.

2. **Close Excel and open the folder where you stored the Excel file in Windows Explorer.**

   You see the Excel file you created.

3. **Create a new folder named** customUI**.**

4. **Add a new text file to the folder named `customUI.xml`. Don't retain the `TXT` extension.**

   Windows asks whether you're sure you want to change the file extension. Make sure you answer yes.

5. **Change the extension of the Excel file from `XLSM` to `ZIP`.**

   Windows asks whether you're sure you want to change the file extension. Make sure you answer yes.

6. **Extract the `_rels` folder from the `ZIP` file.**

   You see the `_rels` folder and the `customUI` folder in the same folder as the Excel file. You're now ready to add the tab.

Adding the tab requires two steps. The first step is to define the tab in the `customUI.xml` file. Here's the XML you'll need to add to the file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customUI
xmlns="http://schemas.microsoft.com/office/2006/01/customu
          i">
```

```
  <ribbon>
    <tabs>
      <tab id="myTab" label="My Tab" keytip="MT">
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

This is the first time you've seen the `keytip` attribute. Whenever you add this attribute, you control the appearance of the keytips that appear when the user presses Alt. If you don't provide a keytip, the Office application assigns one randomly that doesn't affect the other keytips in use. The keytips are an accessibility feature that makes it easy for anyone to access any part of the Ribbon without using the mouse. After selecting that tab, a user can choose other elements on that tab. In this case, the user would select the tab by pressing Alt+MT. A keytip can have as many characters as needed to avoid collisions with other elements at the same level. For example, every tab must have a unique keytip, but a control can have the same keytip as a tab (as long as it doesn't have the same keytip as any other control).

After you add this code, save the file. The second step is to tell Excel that the `customUI.xml` file exists. Otherwise Excel won't display the custom tab. You perform this task by modifying the `.rels` file located in the `_rels` folder (that's right, the file doesn't have a filename). Add a new `<relationship>` element like the one shown here:

```
<Relationship
  Id="rID4"
Type="http://schemas.microsoft.com/office/2006/relationshi
          ps/ui/extensibility"
  Target="customUI/customUI.xml"/>
```

You must provide a unique `Id` attribute value. The example uses `rID4` (short for "relationship identify four"). The `Type` attribute has to point to the schema for the custom user interface extension, so you'll use the URL shown in the code. Finally, the `Target` attribute points to the location of the file. Your `.rels` file should look like the one shown in Figure 3-1 at this point.

Place the `_rels` and `customUI` folders inside the `NewElements.zip` file. You shouldn't compress the `customUI` folder. For some reason, Excel doesn't display the result properly when you do. Windows will ask if you want to overwrite the `_rels` folder if you use the default ZIP file handler. If you use an alternative ZIP file handler, such as WinZIP, make sure you retain the directory structure. Rename the `NewElements.zip` file to `NewElements.xlsm`. Open the file in Excel and you'll see the new tab shown in Figure 3-2. The new tab is blank because you haven't added any groups to it yet. Notice how this tab shows the MT keytip.

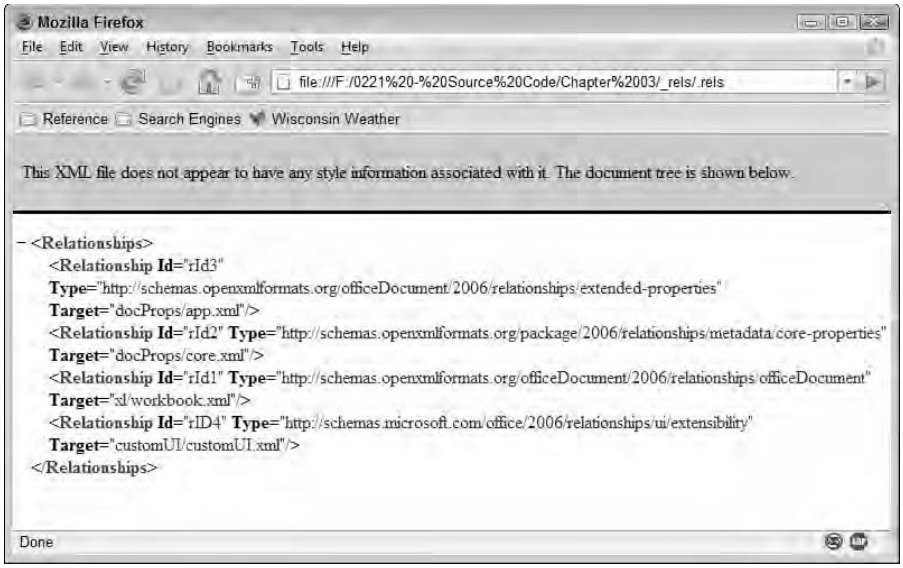


Figure 3-1: The .rels file contains one entry for each Excel relationship.

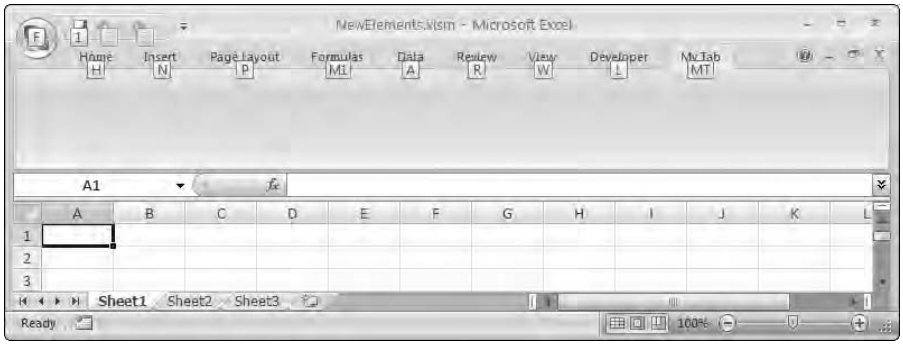


Figure 3-2: A blank tab may not be exciting, but it demonstrates that elements work individually.

Using Groups to Your Advantage

Adding groups to your new tab is relatively easy now that you've created the initial setup. All you need to do is modify the `customUI.xml` file with the new entries you want to make. It's important to start with just the basic layout. Don't try to connect anything to code for right now. After you get the basic layout finished, you can start adding code to each of the elements. Here's the updated `myTab` with three new groups added to it.

```
<tab id="myTab" label="My Tab" keytip="MT">

  <group id="Default" label="Default Group">
  </group>

  <group id="SeparatorGroup"
         label="Group with Separator">
    <separator id="SeparatorItem" visible="true"/>
  </group>

  <group id="DialogBoxLaunch"
         label="Group with Dialog Box Launcher">
    <dialogBoxLauncher>
      <button id="LaunchDialog"
              screentip="Launch Dialog Box"
              supertip="Clicking this button would
normally display a dialog box."
              keytip="LD"/>
    </dialogBoxLauncher>
  </group>

</tab>
```

The code begins with a standard group that doesn't contain any special features. Normally, you'll place controls between the beginning and ending `<group>` tags. Groups have a considerable number of additional attributes you can use, including

- `image` (adds an image to the group)
- `imageMso` (adds a built-in image to the group)
- `keytip` (provides an accelerator key for the group)
- `screentip` (displays a short tip for using the group)
- `supertip` (displays an extended tip for using the group)

Normally, you won't use these additional attributes even though they appear in the Microsoft documentation. In fact, experimentation shows that Office doesn't even react to these special attributes. For example, because you can't select a group anyway, there's little reason to assign an accelerator to it (using the `keytip` attribute). The document, template, or add-in will load as it normally does — it won't generate an error, but you won't see any difference because Office ignores the attribute. However, Office does pay attention to other special attributes (which appear in other areas of this book), such as `insertAfterMso`, `insertAfterQ`, `insertBeforeMso`, and `insertBeforeQ`, which control the positioning of the group in relation to other groups on the tab.

Groups only require occasional use of special attributes. However, controls normally require significant use of attributes. (The "Defining the RibbonX Controls" section of this chapter describes control attributes in detail.) A simple attribute change can affect the functionality of the various control elements considerably. Although this book does explore a considerable number of elements and their attributes, it doesn't provide you with a complete list. To obtain a complete list, download the 2007 Office System: XML Schema Reference from

`http://www.microsoft.com/downloads/details.aspx?familyid=15805380-F2C0-4B80-9AD1-2CB0C300AEF9`

Besides controls, the `<group>` element can have two additional child elements. The `<separator>` element places a line in the middle of the group. You can place the `<separator>` element anywhere within a list of controls to provide separation between the controls. The Window group of the View tab provides a good example of how to use a `<separator>` to further control the appearance of a group. You must provide the `visible="true"` attribute and value to see the `<separator>` in any group you create. Modifying this attribute in your code lets you control when the `<separator>` appears on-screen. Figure 3-3 shows how the `<separator>` appears in this example.

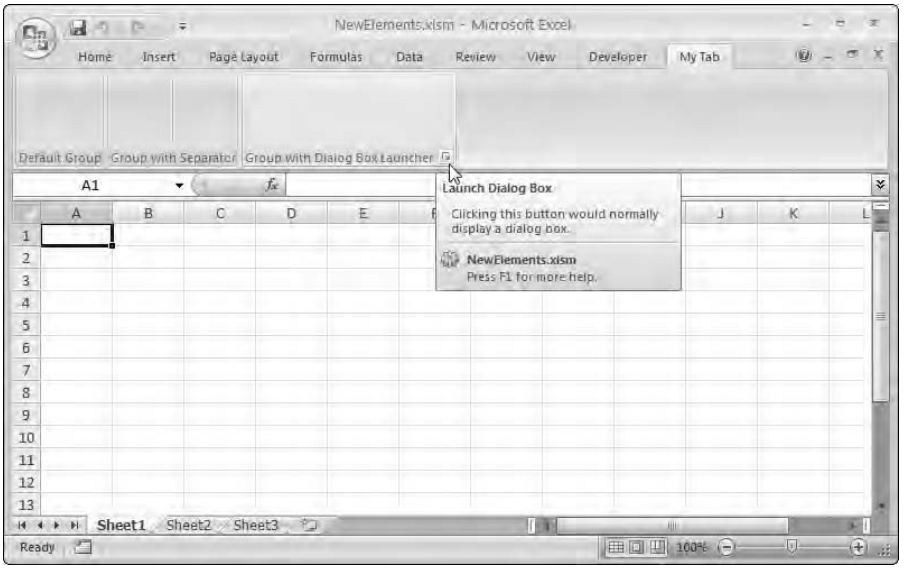


Figure 3-3: Three different kinds of group setups that you can use to manage your controls.

Many of the groups in Office include the dialog-box launcher, a feature that a power user can click to see advanced features. You use the `<dialogBoxLauncher>` element to add the visual element to a group. However, the `<dialogBoxLauncher>` element requires that you include a `<button>` control, or Office won't display the visual element. This is the only time that you must include a control within a group. In fact, Office is quite fussy about this particular element, and it could cause problems with your development efforts. The `<dialogBoxLauncher>` element must contain one and only one `<button>` control, and you can't use any other control. Look again at Figure 3-3, and you'll see the results of adding the dialog-box launcher to a group.

As shown in Figure 3-3, the button also includes both a `screentip` (the bold text) and a `supertip` (the regular text). You can add control characters and special characters to the text by prefacing the character value with `&#` and ending it with a semicolon (`;`). For example, if you want to add a new line to a `supertip`, you would use `&#13;` as the character value.

The current version of Office doesn't provide a hook for the help information that appears below the tip information shown in Figure 3-3. Consequently, you'll always see the name of the module that contains the group or control, plus the text for pressing F1 to obtain help about the element (even when help isn't available). Microsoft is supposedly fixing this problem in a future version of Office.

Now that you have a better idea of how the code works, you'll want to add it to the example. The following steps tell you how to perform this task. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

1. **Type the code into the `customUI.xml` file.**
2. **Rename `NewElements.xlsm` to `NewElements.zip`.**
3. **Add the customUI folder to the `NewElements.zip` file.**
4. **Rename `NewElements.zip` to `NewElements.xlsm`.**

Defining the RibbonX Controls

The third element of any RibbonX implementation is controls. Not only can you use more controls on the Ribbon than you have in the past, but, in many cases, you can also change the control size. This diversity of control options means you can create a better interface for the user by choosing the right control from the RibbonX toolbox and giving it the correct emphasis. The following sections describe controls in greater detail.

An overview of the RibbonX controls

The term *control* takes on a new meaning with RibbonX. A control not only has a purpose, but it also has a size and context. For example, a button is normally just that in most programming environments. However, in RibbonX, you can create either a large or a normal-size button. In addition, the use (and sometimes the appearance) of the button varies by context. A button has no less than eight contexts in the Ribbon:

- standard button in the Ribbon
- part of a dialog-box launcher
- part of a drop-down list
- within the Quick Access Toolbar (QAT)
- part of a split button
- part of a split button with a title
- within a controls group
- part of a menu

The context can make a significant difference in how you use the control. For example, you must include just one button within a dialog box launcher. Office limits the number of buttons in a drop-down list to 16, but you don't have to include any at all. A button that you include within a split button is always visible, and Office will raise an error if you try to use the visible attribute. Consequently, if you run into a problem, but you feel you haven't made any errors in creating the XML for a particular control, you'll want to check the 2007 Office System: XML Schema Reference at

```
http://www.microsoft.com/downloads/details.aspx?familyid=
15805380-F2C0-4B80-9AD1-2CB0C300AEF9
```

to ensure you've used the control correctly in context.

The actual list of RibbonX controls is impressive when compared to what Office provided in the past. Table 3-1 provides a list of these controls and provides a short description of each of them. You'll see all these controls used somewhere in the book.

Table 3-1 RibbonX Controls

Control Name	*Description*
`box`	Groups controls together within a group. You can place any control within a `box`, and flow the set of controls either horizontally or vertically. You must supply a `boxStyle` attribute with a value of `vertical` or `horizontal` to use this control. Unlike the `buttonGroup` control, a `box` control doesn't provide a visual presentation other than flowing the controls.
`button`	Provides a basic execution function. Click the button and something happens within Office. As previously mentioned, Office provides eight different contexts in which you can use the `button` control.
`buttonGroup`	Groups various types of buttons together. The buttons appear within a physical box, and Office places them closer together to show that they're associated in some way. You can use this grouping control with the `button`, `toggleButton`, `gallery`, `menu`, `dynamicMenu`, and `splitButton` controls.
`checkBox`	Provides a basic selection function. The user enables or disables an option by clicking the control. Office provides two contexts for the `checkBox` control, including as a standalone control or as part of a menu.
`comboBox`	Displays a list of options for the user. You create the list of options using the `item` control. Every `comboBox` control must include at least one `item` control as a child. When working with a `comboBox` control, the user can also type a value that doesn't appear in the list (a `dropDown` control requires the user to choose one of the options in the list).
`dropDown`	Displays a list of options for the user. You create the list of options using the `item` or `button` control. The list must contain at least one of the two acceptable controls. The user must choose one of the options in the list that you provide. When the user chooses a `button`, rather than an `item`, control, Office executes the requested action, rather than choose the desired option.
`dynamicMenu`	Defines a menu that you create at runtime, rather than during design time. The menu contents can change to meet specific needs. You must include the `getContent` callback to use this control. A `dynamicMenu` control can appear as part of a `buttonGroup`, `menu`, or `splitButton` control.

(continued)

Table 3-1 *(continued)*

Control Name	*Description*
`editBox`	Lets the user enter plain text into the Ribbon. You might use this feature to perform a task such as searching. Use this control for any input that you can't define through using some other control.
`gallery`	Displays a group of controls in a drop-down structure to save space on the Ribbon. Word uses such a grouping in the Styles group of the Home tab. A `gallery` control differs from other grouping controls in that it provides a drop-down list that you can control in various ways. You can change the presentation of the controls using the `rows` and `columns` attributes. The `itemWidth` and `itemHeight` attributes help you control the size of each item in the group. You use the `Gallery` control within a `buttonGroup` or `menu` control, or as a standalone control. To display items in a gallery, you add code to the `getItemCount`, `getItemImage`, and `getItemLabel` callbacks.
`labelControl`	Creates a label on-screen. You can use this control to label control groups or other elements that don't easily lend themselves to other forms of identification. The user can't interact with the labels you provide.
`menu`	Defines a menu that you create at design time. The menu can contain controls such as the `button` and `checkbox` controls. You can use a menu in standalone mode, or as part of a `splitButton` control. Use the `menuSeparator` control to place separations between menu elements. Unlike a `gallery` control, the `menu` control presents all of the options in a single column (much like the menu system in older versions of Office).
`menuSeparator`	Provides a means of separating elements within any control group.
`splitButton`	Creates a button that has a default action and a list of alternative options. One of the best examples of the split button is the Paste button in the Clipboard group on the Home tab. You must include a `button` or `toggleButton` control for the default control. The optional actions appear within a `menu` control, where you can add a `button` or `toggleButton` control.
`toggleButton`	Provides a combination of a `checkBox` and a `button` control. The user selects a state and performs an action by clicking the `toggleButton`.

Relating RibbonX to object terminology

Many developers are used to working with objects. The elements described in the "An overview of the RibbonX controls" section of the chapter correspond to objects. Calling these elements something else isn't improper because they really aren't objects; they simply take the *place* of objects from the developer's perspective. As far as you (the user) are concerned, the elements that RibbonX defines as controls work every bit as well as the objects you normally use. However, they work very differently under the hood — in a way that users really don't need to worry about.

As with objects, RibbonX elements have the concepts of properties and events. When you're working with an object, a property defines object functionality, such as the color used to present text on-screen. In RibbonX terminology, a property becomes an *attribute.* This word has the same meaning as it does for XML, but in reality, it affects how the RibbonX control functions in some way.

User actions signal events when you use them with objects. When a user clicks a button, the system receives an event message that tells your application to perform some action, such as displaying a dialog box. RibbonX calls events *callbacks*. Again, the name change is appropriate because your code sits outside the RibbonX environment — RibbonX literally makes a callback to your code to signal an event. However, from your perspective, the callback still works the same as an event.

Common RibbonX control attributes

RibbonX provides a number of common attributes that you can use to control the appearance of your application. Table 3-2 describes the most common attributes that you'll use when creating your application. You'll see many of these attributes in use throughout the book.

Table 3-2 Common RibbonX Attributes

Attribute	*Description*
`description`*	Specifies the description text that Office displays when the `itemSize` attribute for a menu is set too large.
`enabled`*	Determines whether Office enables or disables a control. You can choose between `true` and `false` as values.
`id`*	Specifies the identifier for a custom control. You can't use this attribute with either the `idMso` or `idQ` attribute.
`idMso`*	Specifies the identifier for a built-in control. You can't use this attribute with either the `id` or `idQ` attribute.

(continued)

Table 3-2 *(continued)*

Attribute	*Description*
`idQ`*	Specifies the identifier for a qualified control. (A *qualified control* relies on a custom namespace for definition.) You can't use this attribute with either the `id` or `idMso` attribute.
`image`*	Defines the name of a custom image to use with the control.
`imageMso`*	Defines the name of a built-in image to use with the control.
`insertAfterMso`*	Identifies the location of a new control, based on the position of a built-in control.
`insertAfterQ`*	Identifies the location of a new control, based on the position of a qualified control.
`insertBeforeMso`*	Identifies the location of a new control, based on the position of a built-in control.
`insertBeforeQ`*	Identifies the location of a new control, based on the position of a qualified control.
`itemSize`*	Determines the size of a menu or other control item. You can choose between `large` and `normal` as values.
`keyTip`*	Adds a specific accelerator-key combination to the control. The keytip appears when the user presses Alt. You can specify any key combination, using from one to three letters.
`label`*	Specifies the text that appears as part of the control when it's displayed on-screen.
`screenTip`*	Provides a short tip to help the user understand the purpose of a control.
`showImage`*	Determines whether Office displays the image associated with a control. You can choose between `true` and `false` as values.
`showItemImage`	Determines whether Office displays the image associated with a menu or other control item. You can choose between `true` and `false` as values.
`showItemLabel`	Determines whether Office displays the label associated with a menu or other control item. You can choose between `true` and `false` as values.

Attribute	*Description*
`showLabel`*	Determines whether Office displays the label associated with a control. You can choose between true and false as values.
`size`*	Determines the size of a control. You can choose between large and normal as values.
`sizeString`	Sets the size of the control to hold a string of the specified width.
`superTip`*	Provides a detailed tip (super-tooltip) to help the user understand the purpose of a control.
`tag`*	Contains user-defined data that you can use within your application to interact with the control or other element.
`title`	Specifies the text that Office displays in place of a horizontal line for a `menuSeparator` control.
`visible`*	Determines whether Office displays a control or other feature. You can choose between `true` and `false` as values.

** Used by four or more controls*

Common RibbonX control callbacks

RibbonX provides a number of common callbacks that you can use to monitor your application. Table 3-3 describes the most common callbacks that you'll use when creating your application. You'll see many of these callbacks in use throughout the book.

Table 3-3 Common RibbonX Callbacks

Callback	*Associated Controls*	*Description*
`getContent`	`dynamicMenu`	Defines the content of the control.
`getDescription`	Various*	Obtains a description of the control that you provide as part of your application, and displays it on-screen.
`getEnabled`	Various*	Lets your code enable or disable controls.
`getImage`	Various*	Retrieves a custom image you provide for the control.

(continued)

Table 3-3 *(continued)*

Callback	*Associated Controls*	*Description*
`getImageMso`	Various*	Retrieves a standard image that you define for the control.
`getItemCount`	`comboBox`, `dropDown`, `gallery`	Obtains the number of items in the control's item list.
`getItemHeight`	`gallery`	Determines the item height in pixels when displayed on-screen.
`getItemID`	`comboBox`, `dropDown`, `gallery`	Determines the ID of the current item.
`getItemImage`	`comboBox`, `dropDown`, `gallery`	Obtains the image associated with the current item.
`getItemLabel`	`comboBox`, `dropDown`, `gallery`	Obtains the label associated with the current item.
`getItemScreenTip`	`comboBox`, `dropDown`, `gallery`	Obtains the screentip associated with the current item.
`getItemSuperTip`	`comboBox`, `dropDown`, `gallery`	Objects the supertip associated with the current item.
`getItemWidth`	`gallery`	Determines the item width in pixels when displayed on-screen.
`getKeytip`	Various*	Obtains a keytip (accelerator) for the control that you provide as part of your application, and displays it on-screen.
`getLabel`	Various*	Obtains a label for the control that you provide as part of your application, and displays it on-screen.
`getPressed`	`checkBox`, `toggleButton`	Determines whether the user has clicked the control in a manner that activates it.

Callback	*Associated Controls*	*Description*
`getScreentip`	Various*	Obtains a screentip for the control that you provide as part of your application.
`getSelectedItemIndex`	`dropDown`, `gallery`	Determines which item the user has selected from the list.
`getSelectItemID`	`gallery`	Obtains the ID of the item that the user has selected from a list.
`getShowImage`	`button`	Determines whether the control displays an image (which allows you to suppress the image even when you've defined one for the control).
`getShowLabel`	`button`	Determines whether the control displays a label (which allows you to suppress the label, even when you've defined one for the control).
`getSize`	Various*	Defines the size for the control, based on your application's output.
`getSupertip`	Various*	Obtains a super-tooltip for the control that you provide as part of your application.
`getText`	`comboBox`, `editBox`	Obtains the text associated with the currently selected item in a list.
`getTitle`	`menuSeparator`	Provides a title for the specified control. Office displays the text in place of the horizontal line that it normally displays.
`getVisible`	`button`	Determines whether the control is visible.

(continued)

Table 3-3 *(continued)*

Callback	*Associated Controls*	*Description*
`loadImage`	`customUI`	Loads an image associated with the user interface as a whole.
`onAction`	Various*	Executes the action for the control as defined in your application code.
`onChange`	`comboBox`, `editBox`	Detects a change in the user selection or control content.
`onLoad`	`customUI`	Performs a specific act during the loading process.

** Used by four or more controls*

Developing with the Office 2007 Custom UI Editor

The official method for modifying the Ribbon has you changing the file extension, extracting the required files, making modifications in an editor, archiving the files again, and (finally) changing the file extension back every time you want to make any change at all. The beginning of this chapter uses that manual process — which is error-prone at best.

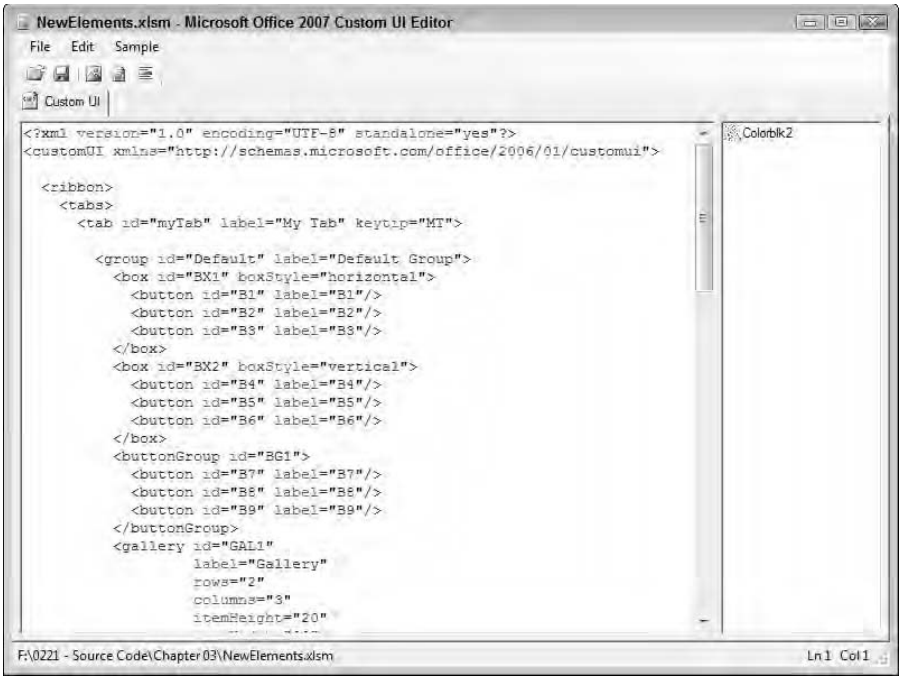
A better way to make changes is to rely on a utility called Office 2007 Custom UI Editor, or Custom UI Editor, for short. You can get it at

`http://openxmldeveloper.org/articles/CustomUIeditor.aspx`

The direct download link is at

`http://openxmldeveloper.org/attachment/808.ashx`

Using the Custom UI Editor is easier than the difficult process that Microsoft suggests. All you do is open your document, template, or add-in, make the required Ribbon additions, and save the file. The next time you open the file in the Office application, it contains the updated Ribbon. Figure 3-4 shows a typical example of the Custom UI Editor in action. The left pane contains the XML for the example for this chapter, while the right pane shows a graphic embedded in the file and used as a custom image.




```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">

  <ribbon>
    <tabs>
      <tab id="myTab" label="My Tab" keytip="MT">

        <group id="Default" label="Default Group">
          <box id="BX1" boxStyle="horizontal">
            <button id="B1" label="B1"/>
            <button id="B2" label="B2"/>
            <button id="B3" label="B3"/>
          </box>
          <box id="BX2" boxStyle="vertical">
            <button id="B4" label="B4"/>
            <button id="B5" label="B5"/>
            <button id="B6" label="B6"/>
          </box>
          <buttonGroup id="BG1">
            <button id="B7" label="B7"/>
            <button id="B8" label="B8"/>
            <button id="B9" label="B9"/>
          </buttonGroup>
          <gallery id="GAL1"
                   label="Gallery"
                   rows="2"
                   columns="3"
                   itemHeight="20"
```

Figure 3-4: The Custom UI Editor makes short work of Ribbon changes.

One of the tasks that you commonly perform is creating your own tabs on the Ribbon. However, just creating a tab isn't enough; you must also create a group and a control of some sort. Figure 3-4 shows a typical setup in Custom UI Editor. The following steps describe the setup for this example:

1. **Download and install Custom UI Editor on your system.**

   To do so, use the information earlier in this section.

2. **Create a file to hold the new Ribbon tab, group, and button.**

   The example uses Excel, but you can also create the example using Word or any other Office 2007 application that relies on the Ribbon. The Ribbon has certain advantages over the older menu-and-toolbar setup because it relies on a generic XML file to define the setup.

3. **Close the application you used to create the data file.**

   Always close the host application before you use the Custom UI Editor to make any changes to the target file. After you make the changes, save them and reopen the file in the host application. Never make any changes in the Custom UI Editor while you have the host application open. Although your data will remain safe, making changes with both the Custom UI Editor and the host application on the same file can have unpredictable results, including lost VBA code.

4. **Start the Custom UI Editor.**
5. **Open the data file you created.**

   You may need to set the Files of Type field to All Files (*.*) to see templates and documents.

   The Custom UI tab in the editor is blank because you haven't added any custom UI features to this file yet.
6. **Choose Sample➪Custom Tab in the Custom UI Editor.**

   The Custom UI Editor automatically creates the entries for a custom tab for you.
7. **Modify the entries as needed to create the custom interface.**

   You can open the sample code for this chapter to see the XML required to use most of the controls described in Table 3-1. Opening the Excel file will show you how these controls appear and let you work with them.
8. **Click Save.**
9. **Open the file in the application you used to create it.**

   You see a new tab like the one shown in Figure 3-5. The tab isn't functional yet, but you can see it.

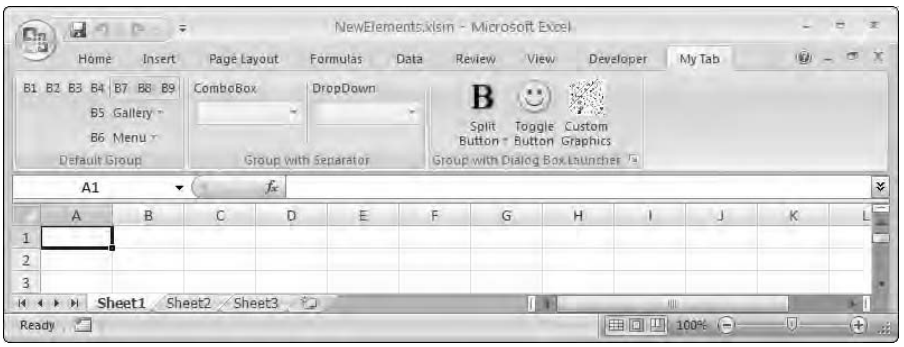


Figure 3-5: Adding a new tab, group, or control is the first step to programming the Ribbon.

Creating Custom Control Graphics

The Ribbon brings a number of new features for graphics as well as the developer goodies described in the rest of the chapter. For example, you'll find that Office supports standardized graphics better, and you won't spend nearly as much time trying to get the right look for your application. The following sections describe a number of interesting graphics considerations for your applications.

Obtaining a list of Office icons

Microsoft does provide a list of all of the Office icons. However, the list isn't a straightforward download; you have to do a little work to obtain the information. The following steps describe the process you use to download and view a list of Office icons.

1. **Download the `Office2007IconsGallery.EXE` file from the following location:**

   ```
   http://www.microsoft.com/downloads/details.aspx?family
           id=12b99325-93e8-4ed4-8385-74d0f7661318
   ```

   You'll need to validate your copy of Office to obtain this download, which means you can't use alternative browsers, such as Firefox, to perform the download. In addition, even though the `.EXE` file is an archive, you can't use WinZIP or other utilities to open it.

2. **Double-click the `.EXE` file.**

   You see a message box asking whether you want to install the icon gallery.

3. **Click Yes.**

   You see a license-agreement dialog box.

4. **Click Yes.**

   You'll see a WinZIP Self-Extractor dialog box.

5. **Choose a location for the icon gallery and click Unzip.**

   The default location of the icon gallery is `C:\2007 Office System Developer Resources\2007OfficeIconsGallery`. Microsoft uses the `C:\2007 Office System Developer Resources` folder for other Office 2007 downloads, so you'll normally want to use the default location.

6. **Double-click the `Office2007IconsGallery.xlsm` file found in the location you used to unzip the file.**

   You see an Excel window.

7. **Click the Office button and click Excel Options.**

   You'll see the Excel Options dialog box with the `Popular` folder selected.

8. **Check Show Developer Tab in the Ribbon and click OK.**

   The Developer tab contains a number of entries, including a special Office Icons group. You can now determine the names of all of the built-in icons.

Click any of the buttons in this group to display a list of the icons that it contains, such as Gallery 1, shown in Figure 3-6. You can determine the name used for a particular icon by hovering the mouse over it. For example, if you wanted to use the selected icon in your application, you'd access it by typing BevelShapeGallery as the `imageMSO` attribute value.

Tools for creating control graphics

Because Office provides a more standardized approach for working with graphics, you can use all of the standard industry tools to create images now. No longer do you need to worry about creating a transparency mask, because Office supports the *alpha bits* (the bits in addition to blue, green, and red used to code a particular bit within an image) that determine transparency.

In the past, many developers have used Paint — or the graphics features of Visual Studio — to create icons. This approach works fine when you're working with older versions of Office. Unfortunately, because Office 2007 now supports a number of new graphics features, these older tools are inadequate to the task. If you use an older tool to create your graphics, you can run into all kinds of problems. An icon may look just fine on your system, but it may not display very well at all on a user's machine.

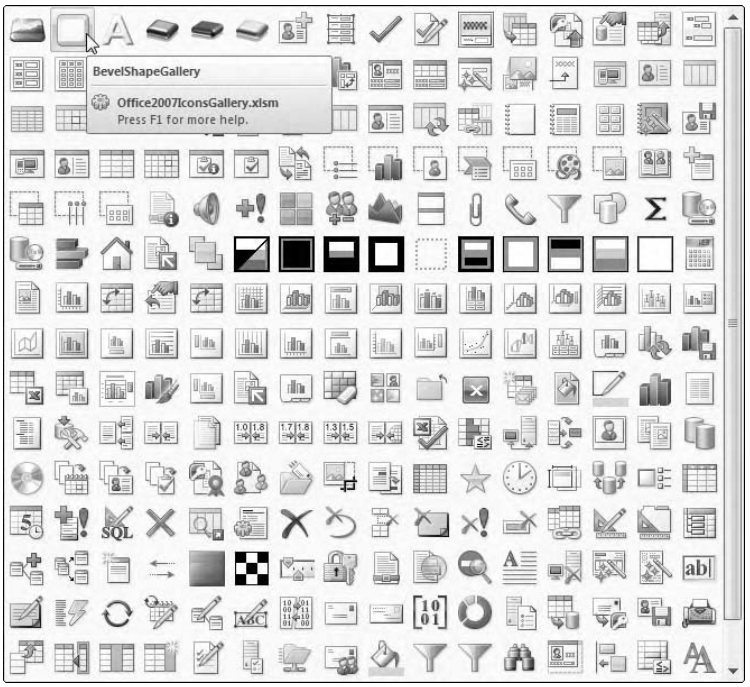


Figure 3-6: The Office 2007 Icons Gallery tells you how to access the built-in icons.

If you're going to update your graphics tools, make sure you obtain a product that supports all of the formats listed in Table 2-1. However, of all of the image formats that Office supports, the PNG format is the most important because it provides the best transparency support. The reason that transparency is so important is that images that don't support it tend to cause problems when working in Vista in the Aero Glass mode.

If you really do need a free alternative for creating your images, Paint.NET provides a great graphics editor that you can simply download and use. This editor supports JPEG, PNG, and BMP images, so you have great options for creating Office graphics. Even though it looks similar to Paint, you'll be amazed at the number of tasks that Paint.NET can perform — including the creation of transparent effects. You can obtain this product at

```
http://www.getpaint.net/index2.html
```

Choosing between bitmaps and icons

Many developers think that their only choice in graphics is icons. In the past, developers did experience a problem using bitmaps where an icon was the most appropriate choice. However, when working with Office, you'll find that bitmaps actually work very well when you need an icon. In addition, because they size better, bitmaps are actually a better choice for Office 2007. You can create an image at the size that feels most comfortable to you, and Office will automatically resize it to meet whatever need it has at the time.

The only problem with bitmaps is that sometimes Office does resize them in such a way that you see unusual effects, such as moiré patterns or a significant loss of detail. In addition, bitmaps can consume considerable space in your document, template, or add-in, so you need to consider the tradeoff between flexibility and file size when working with a large number of graphics.

Understanding how transparency works

Transparency is an important consideration for your graphics for a number of reasons. When a graphic doesn't support transparency correctly, you see the icon or bitmap background, rather than the Office background when you display it on-screen. If you just happen to have a rectangular icon or bitmap where every space contains information, you probably won't experience a problem. However, you'll always experience problems with images that contain irregular borders. The background will show through, and the image will prove quite distracting to your users (making them think that you've done a less-than-professional job).

Modern transparency relies on a fourth color. A single pixel relies on a combination of red, green, and blue to determine its color. A fourth color — called the *alpha color* — determines the transparency of the pixel. You can set a pixel to provide complete transparency, complete opacity, or something in-between. Graphics cards include alpha-blending features to ensure that the user can see through multiple levels of graphics as needed to obtain the desired visual effect. In fact, you can see this effect quite readily when viewing Vista's Aero Glass interface, and many game applications also use transparency to good effect.

Relying on 32-bit images

Most graphics programs today save your images in 32-bit format. Assuming that you also write your application to use 32-bit graphics, you won't ever experience a problem trying to display graphics on a range of machines. However, if you're using an older graphics program that stores images in 16-bit format, or you use the wrong programming technique to load the image, you can inadvertently end up with images that work great on one machine and not-so-great on another.

Windows supports a concept of Device-Dependent Bitmaps (DDBs) and Device-Independent Bitmaps (DIBs). A DIB always requires a 32-bit graphic — which allows 8 bits for each of the four colors (including transparency) for each pixel in the image. On the other hand, a DDB normally relies on a 16-bit graphic, which provides 5 bits for each of the colors you can see and only 1 bit for the transparency. Clearly, one measly bit of transparency support isn't enough (in most cases) to display graphics in Office correctly.

The best way to avoid this problem is to use a newer graphics editor and always save your images in 32-bit format. If you're using VBA to create your application, this is the only step you need to consider, because the Office application automatically uses the correct technique to load the image for you. When you're working with Visual Studio, always make sure you load the image as a DIB to ensure it looks just as good on the user's machine as it does on yours.

Chapter 4

Writing RibbonX Scripts

In This Chapter

- Understanding RibbonX scripting when using VBA
- Defining the limits of RibbonX for VBA developers
- Developing a basic tab
- Adding tab script
- Developing a Ribbon from scratch
- Working with forms in the RibbonX environment

Some VBA developers are under the impression that Microsoft is leaving them out in the cold when it comes to Office 2007. It's true that you can't perform as many tasks using VBA with Office 2007 as you can with Visual Studio, but that's always been true. VBA provides a level of access to Office that lets people do amazing things, but it never has provided complete access.

This chapter helps you create several RibbonX applications using VBA. The first application helps you understand the basics of working with VBA and RibbonX. It doesn't spend a lot of time with the visual elements of the Ribbon (those features appear as part of Chapters 2 and 3), but it does provide enough information so you can use the examples.

The second application shows how to design a Ribbon interface from scratch. Depending on the complexity of the templates you want to move from previous versions of Office or your security requirements, you may find that starting from scratch is the only way to obtain usable results. In addition, the starting from scratch approach does allow you to limit the users' access to Office features that you don't want them to use.

The third application moves from basic interaction with the Ribbon into using forms. Adding forms to your application could help you adapt older applications significantly faster than you could by creating an entirely new Ribbon interface for them. Of course, how well this technique works depends on how you put your application together. In some cases, a full conversion is the only reliable method.

Understanding RibbonX Basics for VBA Developers

VBA developers might initially feel as if Microsoft has left them out of the loop, especially after loading a carefully crafted application into an Office 2007 application and, when it appears on-screen, seeing that nothing's where they left it. In most cases, you'll see everything special about your application crammed onto the Add-Ins tab — where it's very hard to use and may not work at all. Many people feel that Microsoft's reasoning behind this particular developmental decision is flawed.

According to the blog at `http://pschmid.net/blog/2006/10/18/68`, Microsoft's statistics show that 99.7% of all Office sessions don't customize. Given the cost of supporting a version of Office that provides maximum customization, Microsoft simply decided not to support much customization at all. You can still modify the following features without resorting to using RibbonX:

- Quick Access Toolbar (QAT)
- Status bar
- Galleries
- Add-ins tab

Writing new applications may present a bit of a problem, too. If you're used to working with the older event-driven `CommandBar` objects, moving to the new setup could prove troublesome. The ideas are basically the same as before. When a user clicks a button, the system calls your code and your code reacts to the action. Now, however, you're working with callbacks and you have to describe the buttons as part of an XML file, rather than directly changing a toolbar. Even so, the issue is one of learning to work with new objects, not with an entirely new programming culture.

Because of the way you're working with Office 2007 documents, you have to remember to save them in a format that allows macros. For example, when working with Word, you'll save the file using a `DOCM` extension, rather than a `DOCX` extension. Interestingly enough, the file content doesn't appear to change when you use one extension over the other, so you can simply change the extension if you make a mistake and use the wrong one. The Office application simply opens the file differently when it sees one extension or the other. In fact, as you saw in Chapter 3, the Office documents aren't really extended `DOC` files at all; they're a kind of `ZIP` archive files in disguise.

The business-logic code you've used in the past continues to work in Office 2007. Working with a file on disk is the same as before, as is creating XML or interacting with a Web service. Accessing a database hasn't changed either. All of this business-logic code from the past can remain the same.

The part of your application that does change is the user interface. The examples in this chapter show that working with the user interface is perhaps easier than it was in the past because RibbonX passes more information to your application. However, the fact that the user interface code is different is going to cause problems for many developers, especially when the application interacts a lot with the user. In many cases, it's best to separate this code out of your application and provide functions for the business logic as a first step toward making the move to Office 2007. (Chapter 13 provides complete information on how best to move various kinds of VBA applications from older versions of Office to Office 2007 and even making the two versions work together.)

Considering the RibbonX Limitations in VBA

Even though developers have created amazing VBA applications, VBA simply doesn't provide all the programming features of a high-level language, such as VB.NET or C#. That doesn't mean you can't create significant applications with VBA; many developers do.

VBA is no longer a complete tool as it was in earlier versions of Office. You have to add some tools to the mix in order to achieve good results using Office 2007. Fortunately, the two tools you need are free and you'll find them discussed in Chapter 3. The first tool, Office 2007 Custom UI Editor (the book simply calls this tool the Custom UI Editor from this point on), makes it possible to create the XML needed to interact with the Ribbon in a reasonable amount of time (see the discussion in the "Developing with the Office 2007 Custom UI Editor" section of Chapter 3). The second tool is a good graphics editor. You'll find a discussion of Paint.NET in the "Tools for creating control graphics" section of Chapter 3.

You'll also hear a lot of discussion about add-ins for Office 2007 because that seems to be the Microsoft emphasis at the moment. A VBA developer can't create add-ins because writing VBA code doesn't result in an executable file. However, even though this is a VBA limitation, it isn't a new limitation; VBA developers have never created add-ins.

Some online sites also discuss exotic techniques that VBA developers are likely going to want to avoid. For example, you can inject graphics into an Office 2007 application so the images on-screen match some dynamic criteria. (The text styles presented in the Styles group of the Home tab in Word are an example of this approach.) When working with VBA, it's generally a better idea to place the graphics you need in the template or document (using the Custom UI Editor) than to try to inject the graphic from disk.

One type of injection that VBA developers can't use is XML injection. Although a Visual Studio developer can use this technique to create tabs, groups, and controls at runtime, the option simply isn't available to VBA developers.

More than a few early implementers have complained about a particular problem with VBA. If you're not careful, it's easy to erase all your hard work from a document or template. The `_rels` and `CustomUI` files apparently disappear or revert to an earlier state. An important issue for VBA developers is to realize that you're not using a single tool any longer. Modifying the document or template while you also have it open in the Office application will almost certainly result in disaster. The order in which you create a RibbonX application is important; here's the correct sequence:

1. Create the document or template.
2. Define any required styles or other document features.
3. After saving and closing the document or template and the associated application, add any required graphics and XML using the Custom UI Editor.
4. After saving and closing the document or template and the Custom UI Editor, open the document or template again in the application, and check the user interface.
5. Add the VBA code required to make the application work.

Custom Task Panes (CTPs) can greatly ease some movement tasks between earlier versions of Office and Office 2007. Placing tasks within a CTP makes it easy to access by anyone no matter which version of Office they use. Unfortunately, VBA programmers can't use CTPs because they require executable code — in particular, a COM (Component Object Model) component. Some developers have overcome this problem by creating a COM add-in that reads VBA user forms. For example, check out the CTP post at

`http://blogs.msdn.com/excel/archive/2006/07/11/662623.aspx`

One of the more interesting deficiencies for VBA developers is that you can use — but not create — menus and toolbars. Consequently, the Add-Ins tab will show you all the menus and toolbars you actually designed into the

template — but nothing that you create programmatically. If you're looking for missing application features, check to see if they're created programmatically as part of your update process.

Creating a Basic Tab

Chapters 2 and 3 describe the mechanics of creating a tab. The best way for a VBA developer to work with tabs is to use the Custom UI Editor. An important consideration before you begin working with VBA is how you plan to present the options to the user. When working with VBA, you can perform the following actions:

- Create a new tab with groups and controls
- Add new controls to an existing tab
- Add new controls to the Office menu
- Repurpose controls on an existing tab
- Repurpose controls on the Office menu

The first example in this chapter shows how to perform all five of these actions. This section begins with the tabs. The example shows how to create a new tab with the requisite group and controls, add new controls to both an existing tab and the Office menu, and repurpose controls on both an existing tab and the Office menu.

Repurposing is the act of changing the functionality of an existing control. You can still perform the old action after you perform the custom action, but the point is that you're changing the default behavior of a control. By and large, don't repurpose a control unless it's absolutely necessary.

This example also uses custom graphics for two of the controls. The first image is a simple bitmap, while the second is an icon. The following steps tell how to insert your own graphics into a file using the Custom UI Editor.

1. **Open the file you want to modify in the Custom UI Editor.**
2. **Click Insert Icons on the toolbar.**

   You'll see an Insert Custom Icons dialog box. This dialog box works like any typical file open dialog box.
3. **Locate the bitmap or icon that you want to include in the template or document and click Open.**

   You see the icon or bitmap added to the file. Figure 4-1 shows a typical example.

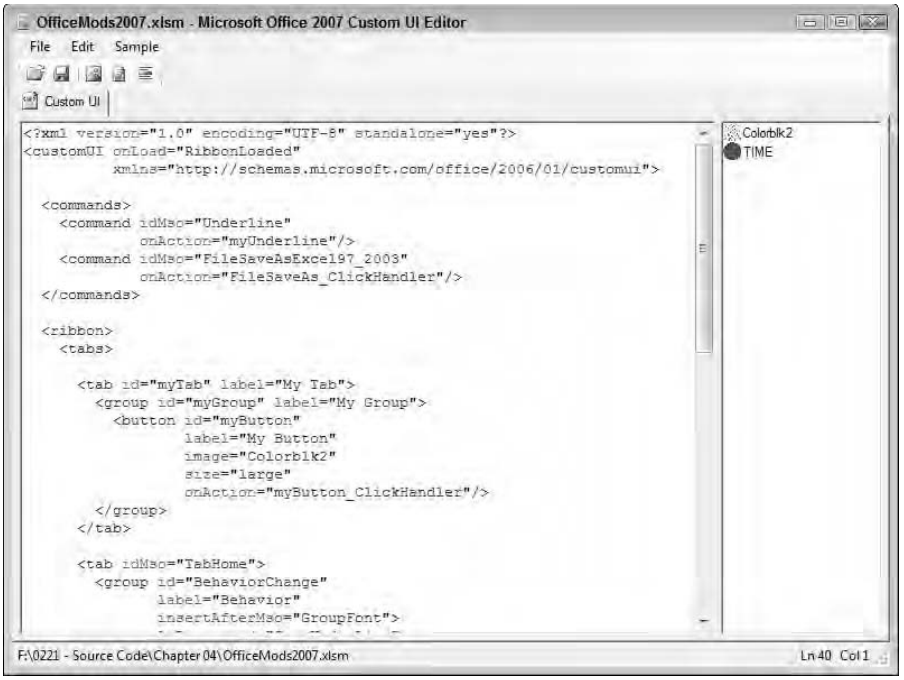


Figure 4-1: Add the XML and graphics required to create your custom interface.

Don't worry about the actual XML for the Excel worksheet changes right now. You'll see each of the XML elements in the "Writing the Scripts" section of the chapter. For now, all you really need to consider is the act of creating this content and adding graphics to dress up the interface.

Writing the Scripts

After you have the pieces of the interface in place, you can begin adding VBA code to your application. The techniques you use vary by addition type. The following sections describe how to implement each of the five additions described in the "Creating a Basic Tab" section of the chapter.

Automatically generating the callback subs

Writing code for the Ribbon differs significantly from writing code for the menu-and-toolbar setup. Fortunately, the Custom UI Editor can help reduce the difficulty of the coding experience for you, as shown in the following steps:

1. **Select the Custom UI Editor and click Generate Callbacks on the toolbar.**

   You see a Callbacks tab appear with VBA code in it. The code provides precisely what you need to access the button you created. Using this approach takes the guesswork out of creating a `Sub` for your button. Notice that the callback includes a variable you can use to access the button. You can't access the button directly from VBA.

2. **Highlight all the code and press Ctrl+C.**
3. **Open the VBA Editor and add a new module called RibbonX.**
4. **Select the VBA Editor. Place the cursor on a new line at the end of the file and press Ctrl+V.**

   You see the VBA code added to the module.

Always perform this task at the outset of the development effort, *before* you've written any other code. Using this approach ensures that you get your application off to a good and fast start.

Coding a new tab with groups and controls

One of the most common tasks you'll perform is creating a new tab with groups and controls. This section shows how to work with My Button in My Group on My Tab — which you create using the following XML:

```
<tab id="myTab" label="My Tab">
  <group id="myGroup" label="My Group">
    <button id="myButton"
            label="My Button"
            image="Colorblk2"
            size="large"
            onAction="myButton_ClickHandler"/>
  </group>
</tab>
```

Notice that this example uses a custom image, so it relies on the `image` attribute, rather than the `imageMso` attribute. You can see this image added to the example document in Figure 4-1. This button also includes the `onAction` attribute to provide a callback to the VBA code. You can see this tab in Figure 4-2.

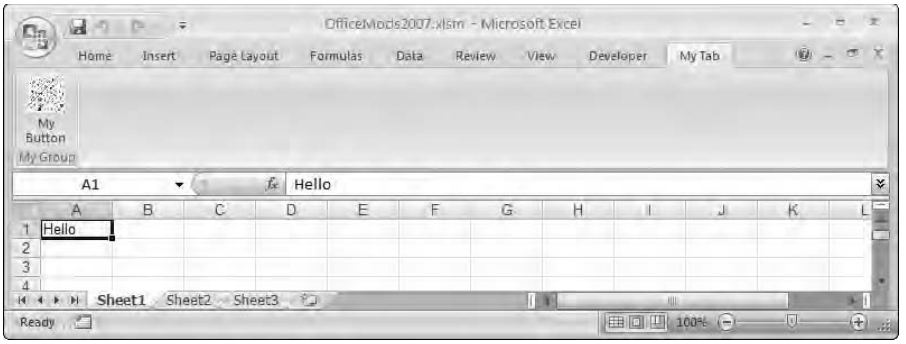


Figure 4-2: The simple tab example is now complete with VBA code to react to the button.

In this case, the example performs a simple interaction with the control passed by RibbonX. The `control` variable provides only a few, but essential, values you can use. The following code shows how you can display the control's identifier:

```
'Callback for myButton onAction
Sub myButton_ClickHandler(control As IRibbonControl)
    MsgBox control.ID + " Clicked"
End Sub
```

Add the `MsgBox` call, as shown, to display a dialog box. Save the file and click My Button on the My Tab tab. You see a dialog box that contains the My Button identifier.

The same techniques described for a Ribbon addition also work for the application menu. However, rather than use the `<tabs>` element, you use the menu element that you want to change, such as the `<fileMenu>` element. You can obtain a complete list of all the Ribbon schema elements (the special elements you use to write Ribbon additions) at

`http://www.microsoft.com/downloads/details.aspx?familyid=15805380-F2C0-4B80-9AD1-2CB0C300AEF9`

Obtaining an identifier for an existing tab, group, or control

Often, you'll find that a new feature you want to add fits better on an existing tab than creating a new tab of your own. You can modify any existing element as long as you know the required identifier. Microsoft provides these identifiers as a download at

http://www.microsoft.com/downloads/details.aspx?familyid=4329d9e9-4d11-46a5-898d-23e4f331e9ae

You can also view the identifiers by using a special feature of the Office products with the Ribbon interface. The following steps describe how:

1. **Click the Office menu to display its menu.**
2. **Click Options (such as Word Options in Word).**

   You see the Options dialog box for that application.
3. **Select the `Customize` folder in the Options dialog box.**
4. **Select an entry in the Choose Command From field.**
5. **Hover the mouse pointer over the control you want to work with.**

   You see a tooltip that displays the identifier for that control in parentheses after the control name.

Unfortunately, this technique works with controls only. If you want to find the identifiers for tabs or groups, you need to download the Microsoft-supplied documentation.

Modifying or repurposing existing tabs, groups, and controls

The example in this section shows (first) how to add a new control to an existing tab, and then how to work with an existing control. It demonstrates two techniques:

- The example adds a new group and control to an existing tab.
- You see how to change the behavior of an existing control by using repurposing.

Creating the custom user interface

Repurposing usually involves two steps. First, you must determine some mechanism to signal the change in behavior. The example uses a simple toggle button for the task, but you could use anything. For example, you might want to disable printing when the user is on the road and doesn't have a printer connected to the system. Second, you must provide the required linkage to repurpose the control. Listing 4-1 shows the code for the simple toggle button used in this example. You would insert this code into the `<ribbon>` element. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Using the Control UI Editor with care

I'm showing you this example because the Control UI Editor doesn't always provide the correct arguments for some events, such as `onAction`. When you run this application, you see an error message stating that something has the wrong number of arguments. Unfortunately, VBA doesn't tell you what piece of code has the wrong number of arguments, and troubleshooting doesn't help you find the problem. In this case, the signature (the arguments) for the `Underline` toggle-button callback is incorrect. You can see these signatures at `http://msdn2.microsoft.com/en-us/library/ms406047.aspx`. The correct callback signature looks like this:

```
sub OnAction(control as
   IRibbonControl, _
             pressed as
   Boolean, _
             ByRef
   fCancelDefault)
```

Listing 4-1: Creating a Toggle Button

```
<tab idMso="TabHome">
  <group id="BehaviorChange"
         label="Behavior"
         insertAfterMso="GroupFont">
    <toggleButton
      id="StopUnderline"
      label="Stop Underlining"
      onAction="StopUnderline_ClickHandler"
      getPressed="StopUnderline_GetPressed"
      size="large"
      imageMso="ColorPickerXLFill"/>
  </group>
</tab>
```

This new group actually appears on the Home tab. Notice that the `<tab>` element uses `idMso` instead of `id` as the identifier. The `idMso` attribute defines an existing identifier — `TabHome` in this case. The `<group>` element defines a new group on the existing tab. The example inserts the new group after the `GroupFont` group, as shown in Figure 4-3.

The toggle button includes a number of new features. Notice that you must provide two callbacks: `onAction` and `getPressed`. The `onAction` callback performs the same task as a standard button by letting you know when the user clicks the toggle button. The `getPressed` callback records the state of the toggle button. Notice that this example uses `ColorPickerXLFill` as the `imageMso` value. You can use the icon from any existing button for controls you create.

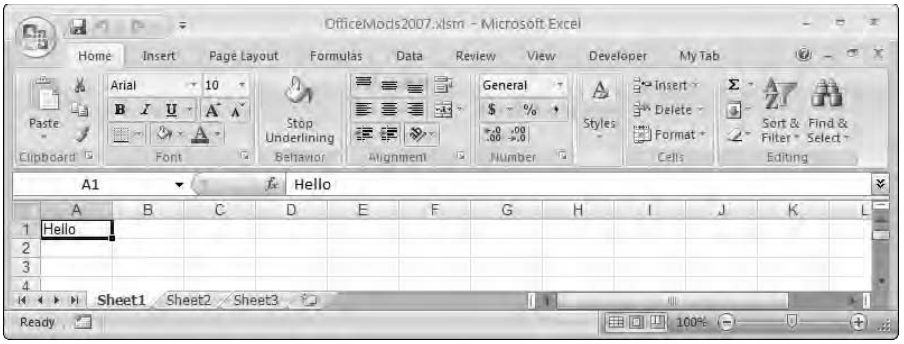


Figure 4-3: You can place new groups anywhere within an existing tab.

The linkage for the repurposing appears as a new child of the `<customUI>` element rather than in the `<ribbon>` element. The `<commands>` provides a list of commands you want to repurpose and the VBA scripts that handle them. Here's the repurposing linkage for this example:

```
<commands>
  <command idMso="Underline" onAction="myUnderline"/>
</commands>
```

Because you're creating linkage to an existing control, you must use the `idMso` attribute with a value that tells which control to repurpose. In this case, the code repurposes the `Underline` control. The `Sub` used as a callback is `myUnderline`.

Reacting to user input

Now that you have the custom Ribbon changes made, you can create code required to interact with the Ribbon in VBA. Listing 4-2 shows the code required to make this part of the example work. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Listing 4-2: VBA Interaction with the Ribbon

```
'Determines the behavior button state.
Dim lBehavior As Boolean

'Callback for StopUnderline onAction
Sub StopUnderline_ClickHandler( _
   control As IRibbonControl, _
   pressed As Boolean)

    ' Change the behavior state.
    lBehavior = pressed
```

(continued)

Listing 4-2 ***(continued)***

```
    ' Update the control.
    Rib.InvalidateControl (control.ID)
End Sub

'Callback for StopUnderline getPressed
Sub StopUnderline_GetPressed( _
   control As IRibbonControl, _
   ByRef returnedVal)

    ' Return the current behavior state.
    returnedVal = lBehavior
End Sub

'Callback for myUnderline onAction
Sub myUnderline(control As IRibbonControl, _
                pressed As Boolean, _
                ByRef fCancelDefault)
    If (lBehavior) Then
        MsgBox "No Underlined Allowed!"
        pressed = False
        fCancelDefault = True
    Else
        fCancelDefault = False
    End If
End Sub
```

The code begins by defining a variable to track the behavioral state of the application. You need this variable to ensure that the `StopUnderline` control you added reflects the correct state.

The `StopUnderline_ClickHandler() Sub` receives the current control and its pressed state. Theoretically, you can use the same `Sub` for all your controls by checking the control's identifier. However, most developers use a separate `Sub` for each control. The code stores the state of the control in `lBehavior` and then uses `InvalidateControl()` to redraw everything.

The `StopUnderline_GetPressed() Sub` completes the process of showing the current toggle button state by returning `lBehavior` to the Ribbon. Because the Ribbon calls this `Sub` when you first load the document, you must also provide a default value for `lBehavior` in `RibbonLoaded()`.

You can repurpose a control to perform any task you want, or you can turn it off completely. In this case, `myUnderline()` performs the default action when `lBehavior` is `False`. When the user sets Stop Underlining and `lBehavior` is `True`, however, the code displays a message telling the user that no underlining is allowed. In addition, the code tells the Ribbon not to depress the Underline toggle button and not to perform the default action.

Modifying or repurposing the Office menu

As with the Office tabs, you can modify or repurpose items on the Office menu. You can also add new controls and even new major menu items. However, there's only one Office menu, so you can't perform something similar to adding a new tab. The following sections describe how to add new items to the Office menu and repurpose existing items.

Adding items to the Office menu

Working with the Office menu requires use of a few new techniques. The Office menu items don't appear within the `<tab>` element, they appear within the `<officeMenu>` element instead, as shown in Listing 4-3. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Listing 4-3: Changing Items on the Office Menu

```
<officeMenu>

  <menu idMso="FilePrepareMenu">
    <button id="NewPrepButton"
            label="My Prepare Button"
            description="Prepare Time"
            image="TIME"
            insertBeforeMso="FileProperties"
            onAction="NewPrepButton_ClickHandler"/>
  </menu>

  <splitButton idMso="FileSaveAsMenu">
    <menu idMso="FileSaveAsMenu">
      <button id="SayHello"
              label="Say Hello"
              description="This button says hello!"
              imageMso="Colorblk2"
              onAction="SayHello_ClickHandler"/>
    </menu>
  </splitButton>

</officeMenu>
```

In the first case, the code simply adds a new control to the Prepare menu, as shown in Figure 4-4. This control uses a custom icon, as compared to the custom bitmap for the control shown in Figure 4-2. Notice how the example uses the `insertBeforeMso` attribute to place the control at the top of the list. It's also important to see how the example uses the `label` and `description` attributes because they work differently from the tabs. The `label` attribute provides a name for the control, while the `description` attribute tells how to use it.

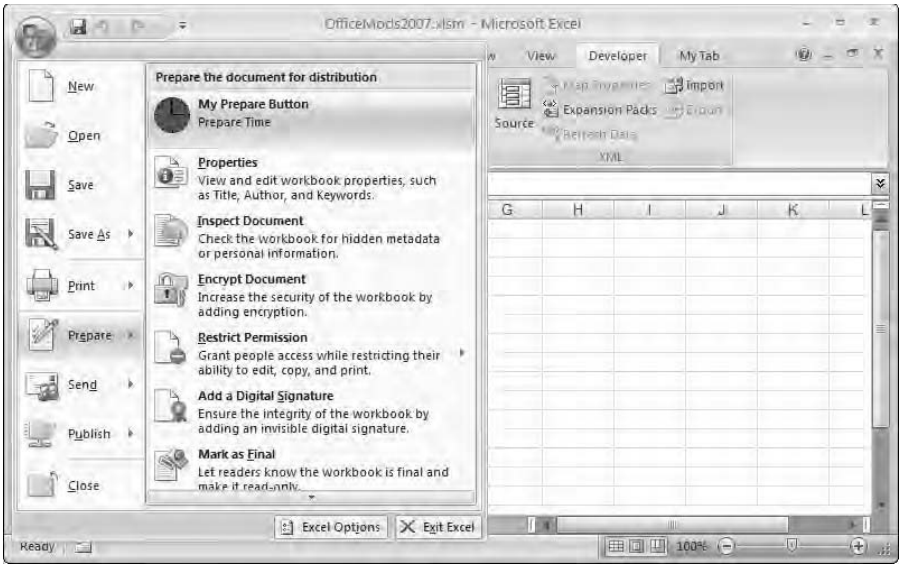


Figure 4-4: This Prepare menu option shows how to add controls to the Office menu.

The second case may take you by surprise. Notice that the `FileSaveAsMenu` entry starts with a `<splitButton>` element. If you attempt to add a new control without this entry, Office will raise an error despite the fact that the `FileSaveAsMenu` entry looks physically the same as the `FilePrepare Menu` entry. The only clue you have about a potential problem is the `WordRibbonControls.xlsx` file found in the `2007OfficeControlIDs Excel2007.exe` file you can download from the 2007 Office System Document: Lists of Control IDs site at

`http://www.microsoft.com/downloads/details.aspx?FamilyId=4329D9E9-4D11-46A5-898D-23E4F331E9AE`

This example demonstrates the reason you should always check the control specifics before you attempt to work with them. The listing of control IDs tells you that the `FileSaveAsMenu` entry is actually a split button and not a menu. Notice how the code adds a `<menu>` element to the `<splitButton>` element before adding the `<button>` control.

Interacting with Office menu controls is pretty much the same as interacting with those on the tabs, as shown in Listing 4-4. Each of the callback handlers displays a message box; the Custom UI Editor generated all these handlers correctly.

Listing 4-4: Reacting to Controls on the Office Menu

```
' Callback for My Prepare Button on the
' Prepare menu of the Office menu.
Sub NewPrepButton_ClickHandler(control As IRibbonControl)

    ' Display a message.
    MsgBox "Are you prepared?", vbYesNo
End Sub

' Provides support for the Say Hello Button on the
' Save As menu of the Office Menu.
Sub SayHello_ClickHandler(control As IRibbonControl)

    ' Display a message.
    MsgBox "Hello!"
End Sub
```

Repurposing Office menu controls

Repurposing an Office menu control works about the same as it does for the tab. You begin by adding an entry to the `<commands>` element, as shown here. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

```
<commands>
  <command idMso="FileSaveAsExcel97_2003"
           onAction="FileSaveAs_ClickHandler"/>
</commands>
```

In this case, the example provides an additional warning when someone tries to save an Office 2007 document in an older format. Listing 4-5 shows the VBA code required to handle this scenario.

Listing 4-5: Verifying a Conversion to an Older Office Format

```
' Repurposes callback for the File Save As button.
Sub FileSaveAs_ClickHandler( _
   control As IRibbonControl, _
   ByRef fCancelDefault)

    ' Holds the user response.
    Dim Answer As VbMsgBoxResult

    ' Display a message.
    Answer = _
    MsgBox("Saving as an old version. Are you sure?", _
    vbYesNo)
```

(continued)

Listing 4-5 *(continued)*

```
    ' Act on the response.
    If Answer = vbYes Then
        fCancelDefault = False
    Else
        fCancelDefault = True
    End If
End Sub
```

This example shows one use of the `fCancelDefault` argument. The code begins by displaying a message box. If the user clicks Yes, the code sets `fCancelDefault` to `False`, which means that Office displays the Save As dialog box. Otherwise the code sets `fCancelDefault` to `True`, which means the user doesn't see anything.

Performing tasks when the Ribbon loads

Not every task you perform with the Ribbon involves a control. You have access to a wealth of information about the Ribbon, and one of the most important callbacks concerns Ribbon loading. You use the `onLoad` attribute for the `<customUI>` element, as shown here:

```
<customUI
   xmlns="http://schemas.microsoft.com/office/2006/01/customui"
   onLoad="RibbonLoaded">
```

This callback looks for a `RibbonLoaded Sub` in your VBA code. As with button and other control events, you can ask the Custom UI Editor to generate the required `Sub` code for you automatically. All you need to do then is fill in the `Sub` with the actions you want to perform. Listing 4-6 shows an example of a common task you could perform in the Ribbon `onLoad` callback. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Listing 4-6: Defining a Callback for Ribbon Loading

```
'Define a global variable to hold the Ribbon reference.
Dim Rib As IRibbonUI

'Callback for customUI.onLoad
Sub RibbonLoaded(ribbon As IRibbonUI)

    'Save the ribbon reference.
    Set Rib = ribbon

    ' Tell the user the Ribbon is loaded.
    MsgBox "Ribbon Loaded"
End Sub
```

This example may not look like much, but you often need the Ribbon reference (`Rib` in this case) when you make changes to the Ribbon. You use the `Invalidate` method to tell the Ribbon to redraw itself and display any features you remove, change, or add.

Creating a Ribbon Using startFromScratch Mode

In most cases, you won't want to modify the Ribbon so much that it actually pays to start from scratch, without any controls on the display. The reasons are simple. If you modify the user interface completely, it's possible that you'll make the application unusable because it lacks a necessary standard control. In fact, this very problem happened with previous versions of Office, costing Microsoft (and other companies) a lot of money in support.

Even though you won't want to start from scratch normally, you might find occasion to remove unneeded controls from view. For example, you might create a special-purpose application that has no use beyond the single task it performs. (An application that keeps track of your bids on eBay might fall into this category.)

The essential change to the XML that enables you to begin from scratch appears in the `<ribbon>` element. You simply add the `startFromScratch` attribute, as shown here:

```
<ribbon startFromScratch="true">
```

To see `startFromScratch` in action, create a new Excel document. Open the document in the Custom UI Editor and choose Sample➪Excel – A Custom Tab. The Custom UI Editor creates a new interface for you. Change the `<ribbon>` element so that it includes `startFromScratch="true"`. You'll also want to remove the `insertAfterMso="TabHome"` attribute from the `<tab>` element. Close the file and you'll see something like the interface shown in Figure 4-5.

Except for the major change in interface, working in `startFromScratch` mode isn't any different from working in RibbonX normally. You'll create the interface, and then add VBA to support the interface. Any default controls that you add to the interface will react normally unless you repurpose them.

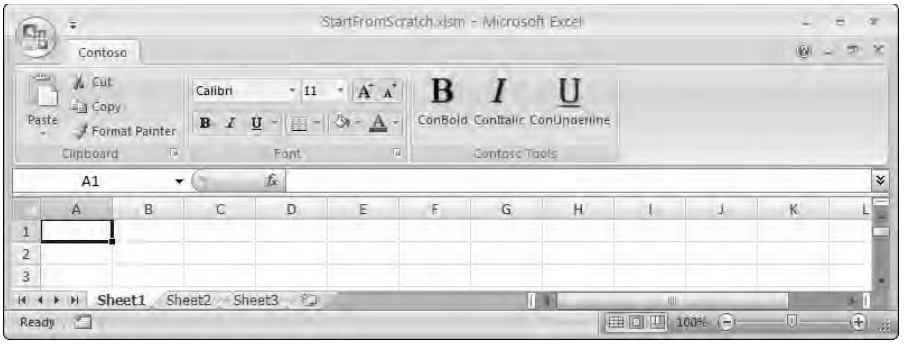


Figure 4-5: Starting from scratch means you won't see anything but the items you create.

Adding Forms Instead of RibbonX Controls

Don't get the idea that the Ribbon suddenly negates the techniques you've used in the past. Yes, you'll always experience some level of complexity — but you can significantly reduce that complexity by using approaches you've always used. For example, you can add a form to your application to display in place of a series of RibbonX controls. All you really need is a single button that displays the form.

Working with forms has several advantages for the VBA developer, not the least of which is keeping the majority of your code in the VBA editor. Using forms can also ease the learning curve for users who are more accustomed to forms than they are to the Ribbon interface. With this in mind, the example in this section shows how to display and work with a simple form. Listing 4-7 shows the XML you'll need to add to an Excel file to begin the example. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Listing 4-7: Adding a Button to Access a Form

```
<customUI
          xmlns="http://schemas.microsoft.com/office/2006
          /01/customui">
  <ribbon>
    <tabs>
      <tab idMso="TabHome">
        <group id="FormList" label="Forms">
          <button id="ShowForm"
                  label="Show Form"
                  imageMso="HappyFace"
                  size="large"
                  onAction="ShowMyForm" />
```

```
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

The example begins with a standard button displayed on the Home tab. When you use forms, you'll very likely want to place the buttons for them on existing tabs whenever possible. Creating a custom tab could reduce part of the benefit of using forms — after all, users still have to discover where you put the button before they can access it.

After you create the button to access the form, you work with the form just as you always have. You create the module to hold the callback, use the callback to create the form, let the user interact with it, and then process the form data. Nothing changes from what you've done in the past, and this technique even works across Office versions (with careful programming). Listing 4-8 shows the form-handling code for this example.

Listing 4-8: Handling a Form with RibbonX

```
' Callback for ShowMyForm onAction
Sub ShowMyForm(control As IRibbonControl)
    ' Call the common processing routine.
    ProcessForm
End Sub

' Provides common processing for all Office versions.
Sub ProcessForm()
    ' Contains the custom form.
    Dim ThisForm As SimpleForm

    ' Holds the message the user chooses.
    Dim Msg As String

    ' Create the form.
    Set ThisForm = New SimpleForm

    ' Display the form.
    ThisForm.Show

    ' Check which option the user selected.
    If ThisForm.optGoodbye = True Then
        Msg = ThisForm.optGoodbye.Caption
    ElseIf ThisForm.optHello Then
        Msg = ThisForm.optHello.Caption
    ElseIf ThisForm.optToday Then
        Msg = ThisForm.optToday.Caption
```

(continued)

Listing 4-8 *(continued)*

```
    ElseIf ThisForm.optYesterday Then
        Msg = ThisForm.optYesterday.Caption
    End If

    ' Display the result.
    MsgBox "You Chose: " + Msg
End Sub
```

In this case, the form contains four message output options, as shown in Figure 4-6. When the user selects an option and clicks OK, the code processes the option and displays a message.

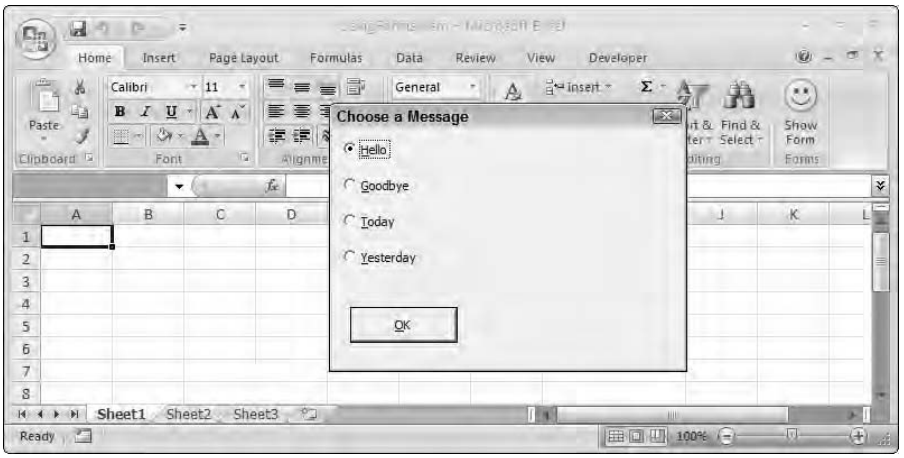


Figure 4-6: Forms let you provide extra options that don't appear on the Ribbon.

Chapter 5

RibbonX and Visual Studio

In This Chapter

- Understanding why you should use Visual Studio
- Configuring your system for RibbonX
- Defining how RibbonX works for C# and Visual Basic.NET developers
- Making a decision between VBA and Visual Studio
- Designing a tab
- Adding code behind to interact with the RibbonX controls
- Creating a Ribbon interface from scratch

RibbonX and Visual Studio are made for each other. You can use these two products together to create some amazing applications with less effort than you might think. Microsoft has really put a lot of effort into making Visual Studio and RibbonX work together. Not only are there special templates you can use to create every kind of RibbonX project, you'll find that Visual Studio does a lot more to help you along. However, don't get the idea that Visual Studio somehow magically makes everyone into a developer. You still need good programming skills to use Visual Studio — but once you have the basic skills, you'll find that working with Office is no longer the painful act of slow torture that it used to be.

This chapter helps you perform a number of tasks to use Visual Studio with RibbonX. The first piece of software you'll need is a copy of Visual Studio 2005. Theoretically, you could also create these applications using Visual Studio 2003, but Visual Studio 2005 provides so many advantages that you really owe it to yourself to use the newer product. (Microsoft doesn't support Visual Studio 2003 in Vista, so you must use Visual Studio 2005 when working in the Vista environment.) You'll have to perform a number of configuration tasks to make Visual Studio work with RibbonX (described in the first few sections of the chapter). You'll also find a discussion of when to use VBA versus when to use Visual Studio. The differences are actually easy to define, so you'll find that you have clear-cut choices among application-programming solutions.

The remainder of this chapter is all about examples. You'll see three kinds of examples in this chapter: document, template, and add-in. Although you can create documents and templates using Visual Basic for Applications (VBA), the add-in is unique to Visual Studio and one of the reasons you might want to use Visual Studio in place of VBA. The examples also highlight some special features that Visual Studio provides, such as the ability to provide dynamic content changes and inject XML into a document as needed to address specific document or user requirements.

Defining the Advantages of Using Visual Studio

Microsoft seems to be leaning toward creating Office applications with Visual Studio, rather than holding on to the legacy of developing with VBA. This is terrible news for VBA developers who have thousands of lines of code to update to Office 2007. The initial reason for this change might appear to be yet another way for Microsoft to grab more of your money. (Actually, as shown in Chapter 4, you can perform a considerable number of tasks using VBA, and VBA developers may not want to update to Visual Studio unless they have a pressing need to do so.) However, there are many good reasons to use Visual Studio for development. The following sections describe the benefits of using Visual Studio to develop your Office applications.

Understanding the levels of RibbonX support

Visual Studio provides three levels of support for RibbonX. These levels of support are more extensive than you might initially think because Visual Studio brings considerable programming functionality to the table. The following sections describe the three levels of support and how you might use them for an application.

Add-ins

Visual Studio creates add-ins as a DLL that you install into the Office environment. Because of the way that you install add-ins, you can actually make them global to the application as a whole or install them only as needed. You may need to use a particular add-in with one document and not with another, even if both documents use the same template or perform essentially the same task, such as a form (think about an invoice that you fill in locally versus one you fill in on the road — the invoice you create on the road may require an add-in to transmit it for processing). Add-ins are the only form of

support for Office 2007. If you want to work with the Ribbon using Visual Studio, you must do it using an add-in.

Templates

Templates affect a class of documents as a whole. For example, you might create an application that someone uses for all of the letters produced by a company. You use templates when you always want a special feature available for use with a particular kind of document, but not with any other document. Templates aren't global in nature; they always affect only one document. In some cases, you might find that you need to combine a template with an add-in to obtain the right mix of application functionality. Templates are a compatibility option for Visual Studio at this time, but Microsoft could add Office 2007 support for them later.

Documents

Creating code for a specific document is more specialized than you might initially think. Any code you create with Visual Studio for a document affects only that document. However, even document features have their use. For example, you might use a single document to interact with a Web service or check on the status of customer orders. Because the document doesn't fulfill a single, one-time purpose, but rather, an ongoing purpose, you can successfully create code without considering the effort a loss. Documents are a compatibility option for Visual Studio at this time, but Microsoft could add Office 2007 support for them later.

Many developers find it helpful to use a single document for experimentation as well. For example, you might have an idea for a particular kind of application that could reduce the time spent looking up resources online — but you need to test your theory before you present it to anyone. Working with a single document is significantly faster than trying to make the idea work with a template. Since Visual Studio 2005 provides add-in templates for Office 2007 only, you must create the add-in on your local machine and attach it to the local document, rather than using network resources to perform the task (which could possibly corrupt production documents, templates, or add-ins). Using a single document for experimentation makes sense from a development-time perspective and makes it less likely that you'll contaminate other documents accidentally (as you might with a template or add-in).

Working with dynamic document content

One of the best features of working with Visual Studio is the capability to inject both XML code and resources into Office. If you decide to create an additional tab, groups, and associated controls to meet a particular user need, you can do so. It's also possible to use this technique to customize the display to meet specific role requirements; a manager may see more controls

than other employees to meet role requirements. Using XML injection also means that you could possibly store the Ribbon changes in a standard XML file on disk and simply add the features as needed during runtime. Using standalone XML would make it considerably easier to create a graphical design tool for your development team.

Resources — such as the XML used to define the interface — are easy to inject into the Office environment. For example, you can easily add the icons or bitmaps required for buttons at runtime, rather than having to include them as part of the file. This added flexibility means that someone could update the icons or bitmaps without ever touching the add-ins, templates, or documents you create. The only criterion for successful implementation is that the new resource have the same filename as the one that the application originally used.

Creating a secure environment

Depending on your organization and how you use the Office applications you create, the secure environment provided by Visual Studio can become an overriding reason to work with it, rather than with VBA. When working with VBA, you can't hide your code even when you protect the document. In addition, VBA has all the security holes that Office and the host operating system have — without any additional functionality to provide a secure computing environment. When working with Visual Studio, you can secure both the user and the code in a way that greatly reduces the chances of a security breach.

Some developers are under the impression that they can create a completely secure environment through devious coding practices and other tactics. If someone wants to break your security, you'll have a hard time keeping them from doing so. As most major vendors have discovered, the largest obstacle in the world can't keep crackers from trying to discover some way inside your organization. Consequently, the best course of action is to couple good application security with vigilance. Regularly check your network for problems as part of a good security plan.

Considering the advantages of managed code

Anyone working with VBA code knows there's always the chance that a COM component is going to ruin your day by causing application failures. The COM component could cause memory leaks — which means the host system could run out of memory at some point. Security leaks occur in COM components as well. You may also find that the component simply fails to work, and the older

code doesn't provide the best error messages in many cases. Using managed code can reduce all these problems and eliminate (at least) the memory leak.

Some tasks are also simply easier to perform using the managed environment provided by Visual Studio. Yes, you can access and use Web services using VBA (I've done so with many public Web services including those offered by Amazon, eBay, and Google). However, the code for accessing a Web service can become quite convoluted in VBA; in Visual Studio, working with a Web service requires that you add a simple Web reference and access the Web-service features just as you would any other class.

Creating the RibbonX Environment in Visual Studio

VBA developers have a definite edge when it comes to setup for Office 2007; a couple of quick tool downloads and you're ready to go. Visual Studio developers have a relatively long journey by comparison. Setting up the RibbonX environment for Visual Studio begins with the Office 2007 installation. Make sure you install Office 2007 first so Visual Studio can find the correct Interop (IOP) modules when it installs. The .NET Programmability Support option, shown in Figure 5-1, lets Visual Studio interact with Office 2007. You must install this feature for every application you want to work with.

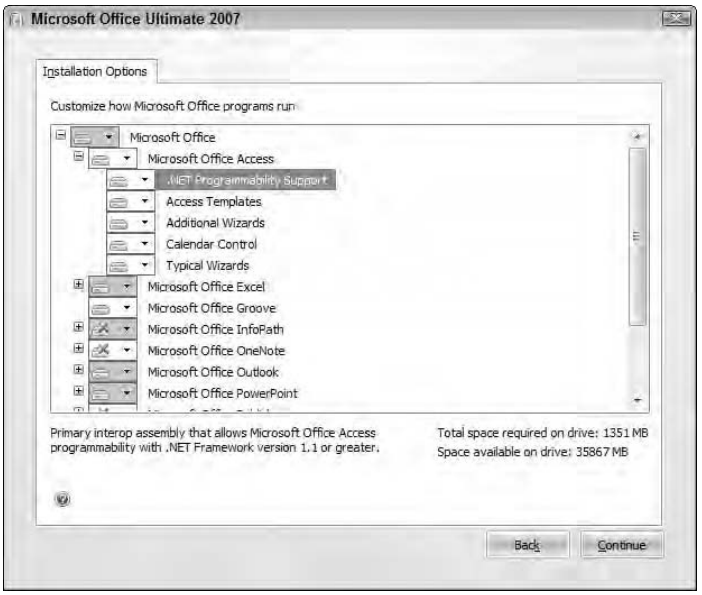


Figure 5-1: Install the .NET Programmability Support feature when you install Office 2007.

After you install Office 2007, install Visual Studio 2005. When working with Visual Studio 2005 Team Edition, you'll find a special Office-interoperability support feature for the language you want to use, as shown in Figure 5-2. (Users of other versions of Visual Studio 2005 need to download and install the required Office interoperability support separately from `http://www.microsoft.com/downloads/details.aspx?familyid=F5539A90-DC41-4792-8EF8-F4DE62FF1E81`.) Be sure to install this feature, even though the installation program will later tell you that you need to install Office to get the desired support.

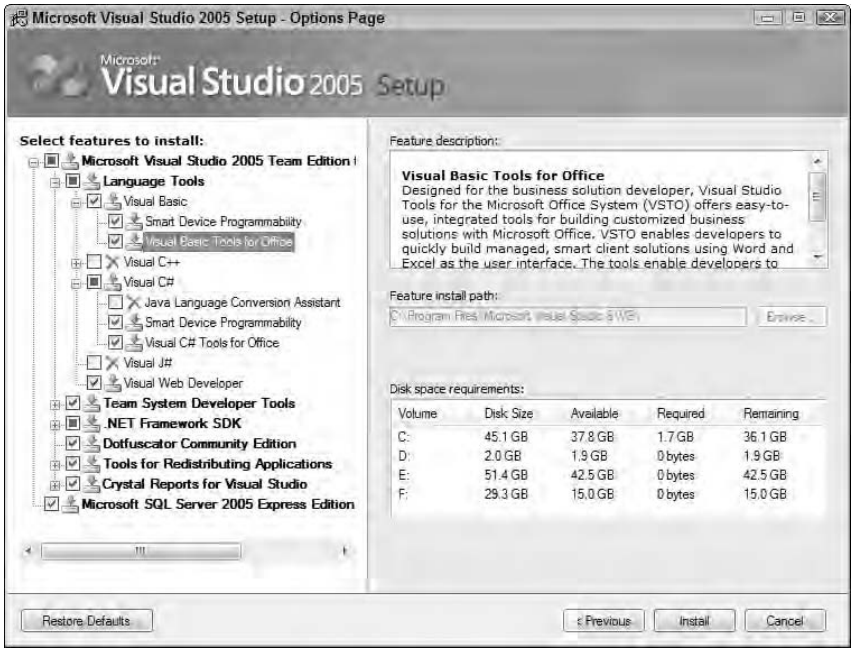


Figure 5-2: Install Office support as part of the Visual Studio installation.

Anyone installing Visual Studio on Vista is going to encounter an error message during installation. The error message tells you that there's a compatibility problem with Vista. The error message is correct, so you shouldn't try to run Visual Studio until you apply the patches described later in this section. For now, simply install Visual Studio on your system.

At this point, you need to install Visual Studio 2005 SP1 (a 431.7 MB download). When working with Windows XP or earlier versions of Windows, click the Check for Service Releases link in the Visual Studio 2005 Setup dialog box. Vista requires that you open the Windows Update applet in the Control Panel and click the Check for Updates link (perform this task even if Windows Update tells you that Vista is up to date). Make sure you install any required patches for Office 2007 during this time as well.

Running Visual Studio under Vista

Even after you apply all of the required patches, running Visual Studio under Vista isn't any picnic because of new Vista features such as the User Access Control (UAC). Many Visual Studio features require administrator privileges, and no one has administrator privileges in Vista by default (not even the administrator account).

The first change you'll want to make to Visual Studio is to run it in Administrator mode:

1. Right-click the Start➪Programs➪Microsoft Visual Studio 2005➪Microsoft Visual Studio 2005 entry.
2. Choose Properties from the context menu.

   You'll see a Microsoft Visual Studio 2005 Properties dialog box.
3. Choose the Compatibility tab and check the Run this Program as an Administrator option. Click OK.
4. Now repeat these steps for all the other Visual Studio applications.

   If more than one user uses Visual Studio on your machine, you can click Show Settings for All Users and change the settings on the new dialog box to let everyone run the application as an administrator.

Whenever you run Visual Studio after you make the change, you'll see a User Access Control dialog box that asks your permission to continue. Vista is elevating the application's privileges to let it run in Administrator mode, and you must approve this change. After you see the UAC message, you'll see a Visual Studio dialog box that leads you to believe it doesn't have the proper rights to run. Unfortunately, this dialog box shows up whether Visual Studio has the correct rights or not. Click Continue to clear it. At this point, Visual Studio will start, and you'll use it as you would normally. If you do experience other Visual Studio problems, you can find more information at the Visual Studio on Windows Vista site:

`http://msdn2.microsoft.com/en-us/vstudio/aa948853.aspx?lcid=1033`

If you're using Vista, you'll want to install a special Vista patch for Visual Studio (29 MB). You can download this patch from the Visual Studio 2005 Service Pack 1 Update for Windows Vista site at

`http://www.microsoft.com/downloads/details.aspx?familyid=90E2942D-3AD1-4873-A2EE-4ACC0AACE5B6`

This patch won't install until you have Visual Studio 2005 SP1 installed. The patch does fix the incompatibility problem that Vista warned you about earlier. Be patient when installing this update; even with the right setup, it seems to require an inordinate amount of time to complete its task.

Now it's time to install the actual Office 2007 support. You can download Visual Studio Tools for Office 2005 Second Edition (VSTO 2005 SE) from the site at

`http://www.microsoft.com/downloads/details.aspx?familyid=5e86cab3-6fd6-4955-b979-e1676db6b3cb`

This is only a 6.1 MB download. After you download the file, perform the installation and you're finally ready to work with Office 2007 in Visual Studio.

Understanding RibbonX Basics for VB.NET and C# Developers

You can create a number of Office application types in Visual Basic. The documents and templates appear in the Visual Basic\Office folder, as shown in Figure 5-3. Notice that these offerings work for Excel and Word. Don't use the Outlook Add-in template shown in the Office folder because it isn't Office 2007–specific.

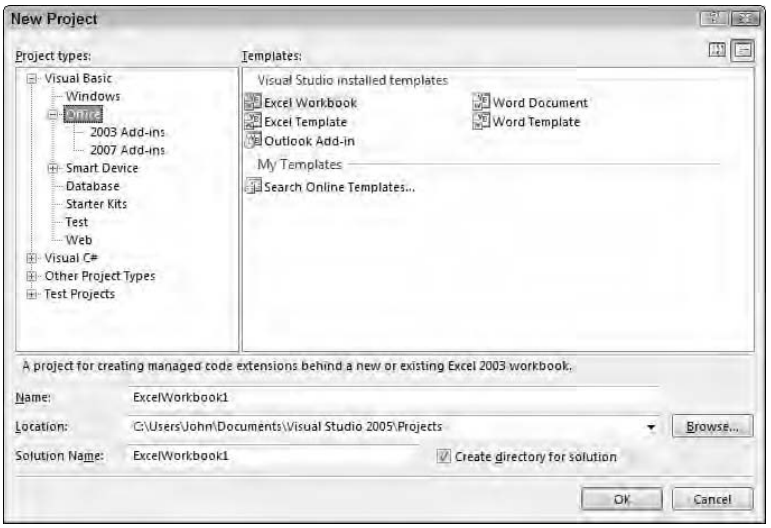


Figure 5-3: Choose between documents and templates in the Office folder.

The offerings in the Office folder are Office 2003–specific. You must have a copy of Office 2003 installed to use them. If you want to create a Ribbon-specific solution, you must use an add-in. Consequently, this book generally considers the add-in solution when working with Visual Studio 2005, because it focuses on working with the Ribbon and not with older versions of Office. You can, however, work at the document-and-template level for compatibility solutions, as considered in Chapter 14.

Select the Visual Basic\Office\2007 Add-ins folder when you want to work with add-ins. Remember that an add-in lets you create an application that works across documents and templates. An add-in is always available until the user either removes it or disables it. You can easily add and remove add-ins as needed.

Figure 5-4 shows the list of add-ins that Office 2007 provides. These solutions include Excel, InfoPath, Outlook, PowerPoint, Visio, and Word. The Office 2007 setup provides a template for creating an InfoPath add-in. This template is a new feature in Office 2007.

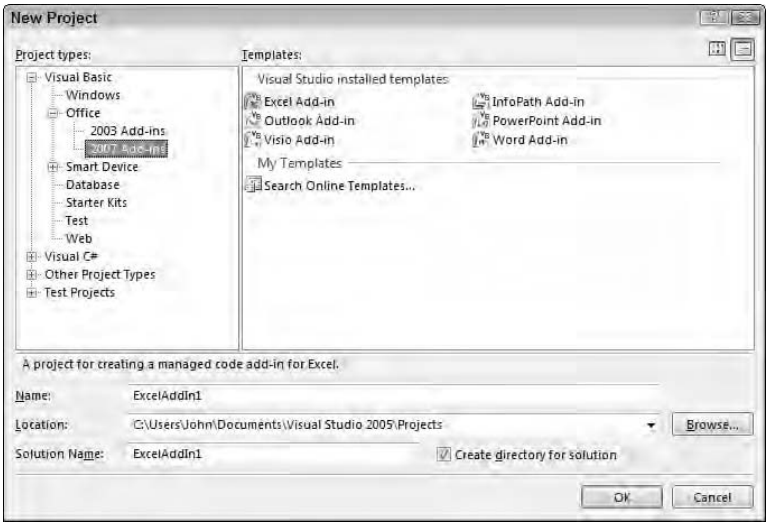


Figure 5-4: Select a template project from the 2007 Add-ins folder.

The templates you see in Figure 5-4 represent the support that Microsoft provides at the time of writing. At some point, Microsoft is going to add more templates to Visual Studio 2005. To determine whether Microsoft has provided additional templates, choose Search Online Templates and click OK. Visual Studio searches online for additional templates and (if it finds them) lets you install them to your system. The new templates provide additional project types that make your Office 2007 development tasks easier.

Choosing Between VBA and Visual Studio

Visual Studio and VBA are complementary tools when it comes to the Ribbon. Visual Studio has many advantages that VBA doesn't enjoy; likewise, you must use VBA to accomplish other tasks. Chapter 4 points out that you must use VBA if you want to create a RibbonX solution for a document or template, except in the case of a compatibility scenario. Visual Studio, as mentioned earlier in the chapter, is your only option when you want to create an add-in.

However, the choices aren't simply a matter of choosing the kind of implementation. You may have noticed that there isn't an add-in template for Access in Figure 5-4. That's because Visual Studio doesn't provide Access support (Microsoft may provide this support later). Consequently, you can't use Visual Studio to create an add-in to modify Access. It turns out that you can't use the process defined in Chapter 4 for VBA either. Access is a special situation, in which you use XML in a different way — with VBA — to define changes to the Ribbon. Chapter 8 provides complete details on this Access-only RibbonX methodology.

Visual Studio also has its specialized application. Outlook doesn't provide standard files you can modify (using the Custom UI Editor) either. In this case, you must create an add-in using Visual Studio to modify the Ribbon. Given that Outlook doesn't really lend itself to documents in the same way that Excel and Word do, the use of add-ins makes sense in this situation. You'll always modify Outlook as a whole, rather than individual files.

Creating a Basic Tab

You can create add-in code using a number of techniques. For example, when the application allows it, you could add the required XML directly to a document or template, and then implement the callbacks using an add-in. Every document or template with the required custom controls could provide access to the functions within the add-in. Of course, you have to exercise extreme care when using this approach because you might inadvertently call the wrong add-in should you give two functions in different add-ins the same name.

The second method is safer because you define the XML directly in the add-in or as part of an external file that you load into the add-in during runtime. The example uses this second approach because it provides an easier path to Ribbon modification.

Defining the project

Unlike VBA, Visual Studio provides a complete solution for working with the Ribbon. You never have to leave the IDE to create anything, not even the XML. The easiest way to work with the Ribbon is to start with one of the Add-in project templates, shown in Figure 5-4. This example works with Excel, but the same concepts work for the other Office products. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) The following steps get you started:

1. **Open Visual Studio.**
2. **Choose File➪New➪Project.**

   You see the New Project dialog box, as shown in Figure 5-4.
3. **Open the 2007 Add-ins folder, shown in Figure 5-4, for your favorite language.**

   The screenshots show C# as the example language, but the same processes work in Visual Basic.
4. **Choose the add-in template for your application, type a name for the application, and click OK.**

   The example uses an application name of BasicTab and uses the Excel Add-in project template. Whichever settings you choose, Visual Studio creates the project for you.

Adding the RibbonX files

It may seem counterintuitive, but the project that you create isn't complete. In order to work with the Ribbon, you must add another piece to the puzzle. The Ribbon files include a code file that contains the code behind for the controls you create, and it includes the XML file that tells Office how to configure the Ribbon. The following steps tell how to create this part of the example:

1. **Right-click the project entry (BasicTab) in Solution Explorer and then choose Add➪New Item from the context menu.**

   You see the Add New Item - BasicTab dialog box, as shown in Figure 5-5.

Figure 5-5: This dialog box lets you add the Ribbon functionality your application needs.

2. **Highlight the Ribbon Support entry, optionally type a name for the file (the example uses the default name), and click Add.**

   Visual Studio adds a new code file and an XML file. The `Ribbon1.XML` file already contains a simple tab, group, and button. So, the example relies on these default entries. However, before you can actually see changes to the Ribbon, you add the XML file to the application resources.

3. **Open the `<Project>\Properties` folder in Solution Explorer. Double-click `Resources.RESX` to open it.**

   You'll see the Resources window.

4. **Drag and drop the `Ribbon1.XML` file onto the Resources window.**

   Visual Studio adds the file to the resources, as shown in Figure 5-6.

Adding some code

You may have noticed that most of the code required to make the example complete also appears as part of the project you created. Of course, you do need to make a few changes to complete the example. The first task is to uncomment the `ThisAddIn` partial class found in the `Ribbon1` code file. This code creates a connection between your Ribbon and the Office application.

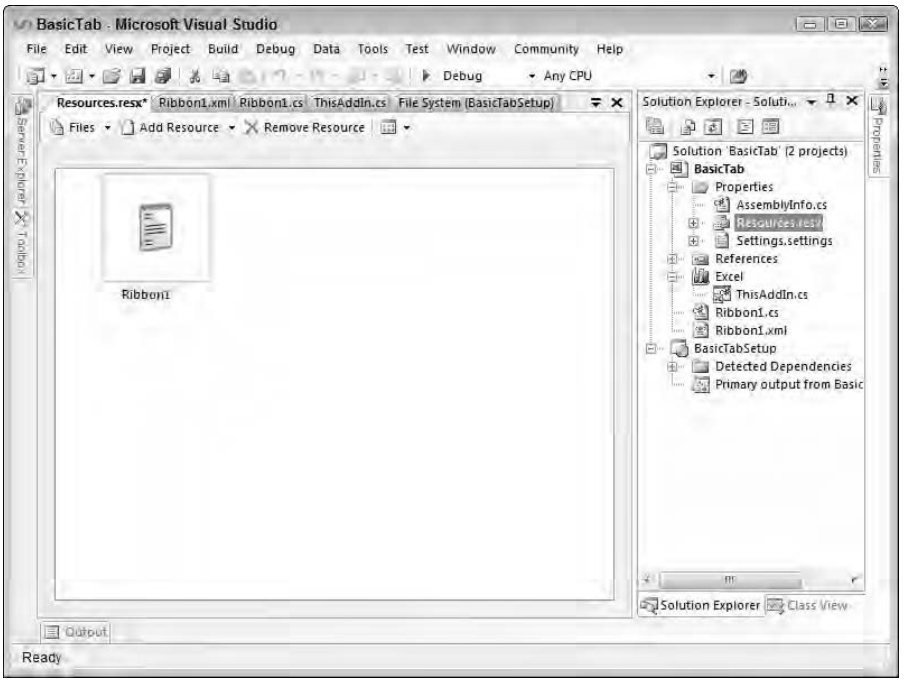


Figure 5-6: Add the XML file to the project as a resource.

The next change appears in the `IRibbonExtensibility` region. Open this region and change the `GetCustomUI` method so it looks like the one shown here:

```
public string GetCustomUI(string ribbonID)
{
    Return Properties.Resources.Ribbon1;
}
```

The example code obtains the XML from the `Properties.Resources.Ribbon1` resource and sends it to the Office application, which then uses it to change how the Ribbon appears. At this point, you can build and run the application. Simply click Debug to start the application. When the application starts, choose the Add-Ins tab and you'll see a new button on-screen, as shown in Figure 5-7. When you click this button, you'll see a message box telling you the button state (pressed or released).

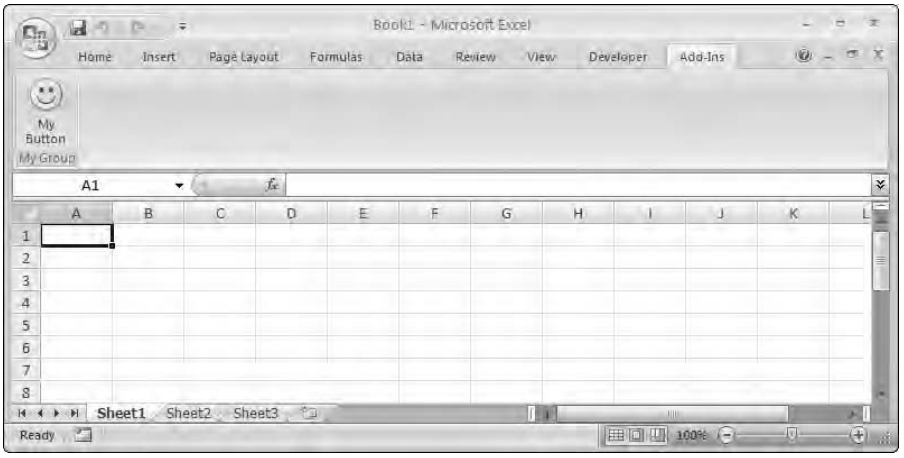


Figure 5-7: The new button appears in the default position on the Add-Ins tab.

Creating a package for end users

The add-in project template also creates a setup program for you. You can use this setup program to make the add-in you create available for end users. They install the add-in just as they would any other program. The setup program isn't part of the build process at the outset, though; Microsoft wants to ensure that you don't have to wait too long to test your add-in.

To create the setup program, choose Configuration Manager from the System Configurations drop-down list box on the Standard toolbar. You'll see a Configuration Manager dialog box, like the one shown in Figure 5-8. Place a check mark next to the BasicTabSetup entry and click Close. Then build your project and you'll have an installation program you can distribute to end users for your add-in.

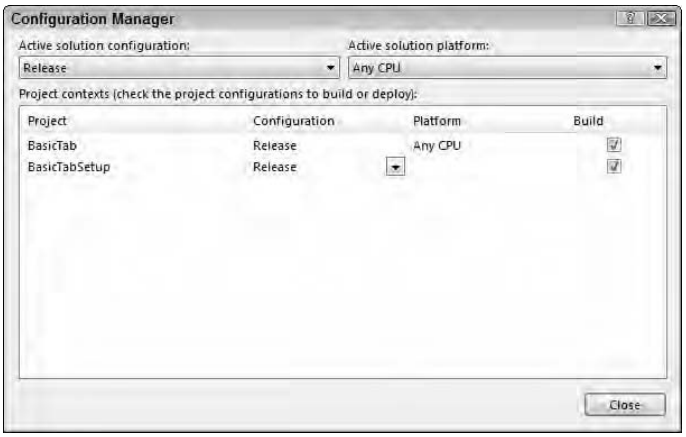


Figure 5-8: Add a setup program to the build sequence after you debug your add-in.

Removing the add-in

When you create code for a document or template, the code stays with it and you don't have to worry about very much. However, when you create an add-in, the code isn't associated with a document or template, so it affects the application as a whole. Consequently, you can't simply delete a document or template to get rid of it. You may notice that after you run the BasicTab add-in that it appears every time you start Excel. In fact, if you delete the project without deleting the add-in reference, you'll find that Excel continues to look for it and may even display an error message.

Fortunately, it isn't too hard to get rid of the add-in when it's done its job. The following steps tell how to get rid of an add-in you no longer need:

1. **Choose Office Menu➪Excel Options.**

   You see the Excel Options dialog box.

2. **Select the Add-Ins folder.**

   You'll see the BasicTab (or other) add-in listed in the Active Application Add-ins list, as shown in Figure 5-9. Notice that the Type column tells you that this is a COM Add-in type.

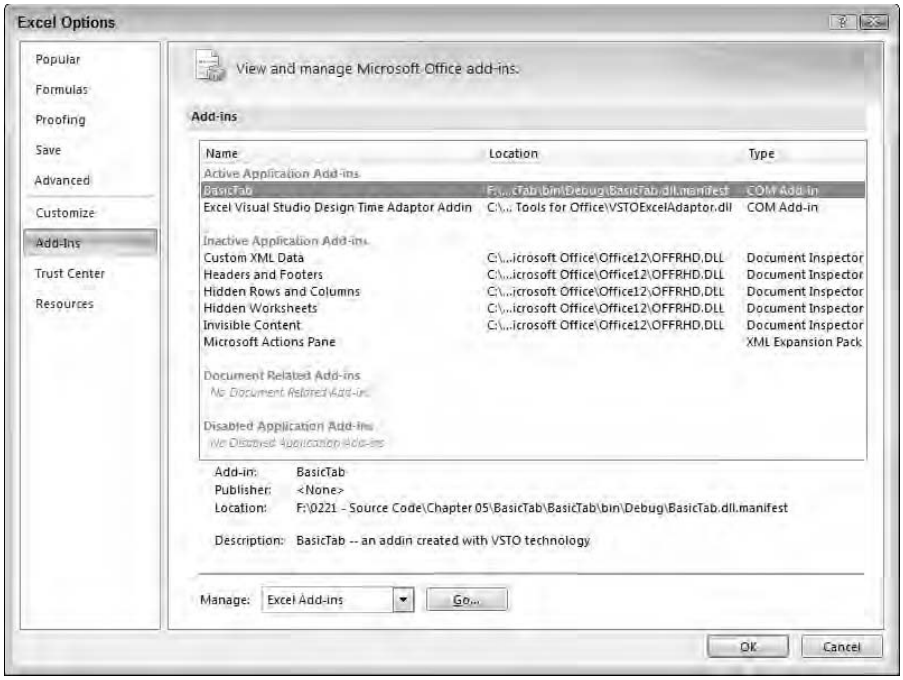


Figure 5-9: Locate your add-in in the list of active add-ins for the application.

3. **Choose COM Add-ins in the Manage field, and click Go.**

   The option you choose in the Manage field must reflect the add-in type. Visual Studio always produces COM add-ins. However, you might see other add-in types. You'll see the COM Add-Ins dialog box, as shown in Figure 5-10.

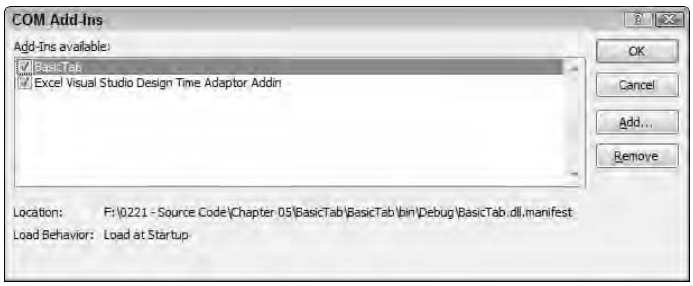


Figure 5-10: The add-in you want to remove will appear in this list.

4. **Clear the check mark next to the add-in to deactivate it. Highlight the entry and click Remove.**

   The Office application removes the application from the add-in list.

5. **Click OK.**

   You no longer see the changes the add-in supplied to the Ribbon.

Unfortunately, these steps don't actually remove the add-in from your Registry. You also have to remove the COM entries for the DLL. To perform this task, you can either uninstall the add-in (assuming you installed it using the setup program that you created in the "Creating a package for end users" section of the chapter), or you can use a special utility to remove the Registry entries. The following steps tell how to use the RegAsm (Register Assembly) utility to remove the entries.

1. **Choose Start➪Programs➪Microsoft Visual Studio 2005➪Visual Studio Tools➪Visual Studio 2005 Command Prompt.**

   You see a command prompt. If you're using Vista, you'll want to right-click this entry and choose Run As Administrator from the context menu.

2. **Locate the DLL used for your add-in and use the CD (change directory) command to see it in the command prompt.**

3. **Type** RegAsm /Unregister <*Name of DLL*> **at the command prompt and press Enter.**

   Make sure you include the full DLL name, such as **BasicTab.DLL**. The RegAsm utility tells you that it has unregistered the types successfully.

After you ensure that all of the Registry entries are gone — using the Registry Editor (RegEd) utility or a similar tool — you can finally delete the project from your system. Nothing bad will happen if you *don't* remove the Registry entries, but the Registry does eventually get clogged with these unnecessary entries and your system slows down.

Writing Code Behind for RibbonX

Although the basic tab example gets you going, you can do a lot more with RibbonX in Visual Studio. You can break down these tasks into five major groups:

- Create a new tab with groups and controls
- Add new controls to an existing tab
- Add new controls to the Office menu
- Repurpose controls on an existing tab
- Repurpose controls on the Office menu

Repurposing is the act of using an existing control for a new purpose or augmenting the existing purpose. For example, you can change the Office 97-2003 document save feature to display a dialog box that asks whether the user really wants to save the document in the older format. The following sections describe how you can perform each of these tasks. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) The steps in the following section get you started.

Handling graphics in Visual Studio

When you're working with VBA, you normally embed the graphics you want to use directly in the document or template file. When you program with Visual Studio, however, you work with an add-in that isn't attached to a particular document or template — so embedding the graphics won't work. In this case, you must serve up the graphics as part of the add-in, which means creating additional code. The XML for this particular need appears as part of the `customUI` element, as shown in Listing 5-1.

Listing 5-1: Creating the `<customUI>` Element

```
<customUI onLoad="OnLoad"
          xmlns="http://schemas.microsoft.com/office/2006/01/customui"
          loadImage="GetImage">
```

The `loadImage` attribute provides a call to the `GetImage()` method in your code. Unfortunately, the `GetImage()` method never knows which icon or image it must send to the Office application — so you have to create it in such a way that it can handle any need. Listing 5-2 shows an example of the code you might create for `GetImage()`.

Listing 5-2: Handling Graphics in Visual Studio

```
public stdole.IPictureDisp GetImage(string ImageName)
{
   // Holds the bitmap to pass to Office.
   Bitmap ThisBitmap = new Bitmap(20, 20);

   // Detect the image name and corresponding resource.
   switch (ImageName)
   {
      case "Colorblk2":
         ThisBitmap =
            new Bitmap(Properties.Resources.Colorblk2);
         break;
      case "TIME":
```

(continued)

Listing 5-2 *(continued)*

```
            ThisBitmap =
               new Bitmap(
                  Properties.Resources.TIME.ToBitmap());
            break;
      }

      // Convert the bitmap to an IPictureDisp.
      return
         PictureConverter.ImageToPictureDisp(ThisBitmap);
}
```

This example works with two different kinds of images. The first, `Colorblk2`, is a standard bitmap, so you can use it directly. The second, `TIME`, is an icon; you have to convert it to a bitmap before you can use it by calling the `ToBitmap()` method. Using a case statement ensures that no one tries to use a nonexistent bitmap — but you could also create a generic method to handle all possible bitmaps and icons the user might request.

Notice that the code requests the images from `Properties.Resources`. You can add images to the embedded add-in resources by choosing Add Resource➪Add Existing File in the Resource.RESX window. Visual Studio displays a dialog box where you can locate the image and add it directly to the add-in.

After this first level of selection and conversion, the `GetImage()` method calls the `PictureConverter.ImageToPictureDisp()` method. You might initially wonder why you have to make this special call to an internal class you create, but Microsoft has *protected* certain `AxHost` features — which means you can't access them directly. Listing 5-3 shows the internal class you have to create to convert the image to a `stdole.IPictureDisp` type.

Listing 5-3: Converting an Image into the `stdole.IPictureDisp` Type

```
internal class PictureConverter : AxHost
{
   private PictureConverter() : base(String.Empty) { }

   // This method converts a bitmap to an Office
   // compatible bitmap.
   static public stdole.IPictureDisp
      ImageToPictureDisp(Image image)
   {
      // Here's the reason for the separate class. The
      // GetIPictureDispFromPicture() is protected, so
      // you can't access it directly.
      return
(stdole.IPictureDisp)GetIPictureDispFromPicture(image);
   }
}
```

The code is very simple, in this case, because it's doing something you should be able to perform directly: The `PictureConverter` class inherits `AxHost`. It then adds a single new method, `ImageToPictureDisp()`, and calls the protected `GetIPictureDispFromPicture()` method to convert a .NET image into an object, which you then cast into the `stdole.IPictureDisp` type.

Performing tasks when the Ribbon loads

The default project that you create with Visual Studio includes an `OnLoad()` method. This method automatically executes when Office loads the Ribbon. Of course, you'll overwrite the original XML and will probably remove the default callbacks at some point, so it's important to know how this feature works. The XML for this need appears in Listing 5-1. The `onload` attribute determines which method Office calls for any initialization needs. You can modify the method, the default name is `OnLoad()`, to meet any need. Listing 5-4 shows the code for this example.

Listing 5-4: Performing Tasks When the Ribbon Loads

```
public void OnLoad(Office.IRibbonUI ribbonUI)
{
   // Save the Ribbon reference.
   this.ribbon = ribbonUI;

   // Initialize the underline state.
   UnderlineState = false;

   // Display a loaded message.
   MessageBox.Show("Ribbon Loaded");
}
```

You always have to save a reference to the Ribbon in order to perform some required tasks such as dynamically updating the Ribbon content. The project template provides the `ribbon` variable for this purpose, so the example performs the default action of saving the reference to it.

This example provides an example of a control that works in a certain way depending on a selection a user makes with another control. In this case, the user can control whether the application allows underlining or not. Because Excel needs to determine the state of this control when the Ribbon loads, the code includes an initialization variable named `UnderlineState`.

It's helpful to include a status message in the code while you're debugging. This one simply tells you that the Ribbon has loaded. If you don't see the message box, you know something has gone wrong with your add-in.

Creating new tabs, groups, and controls

One of the tasks you'll perform most often is creating new tabs, groups, and controls for your application. The process begins by adding XML for this purpose to your application. You saw one example of this technique when creating the default application provided with the project template. Here's the XML used for this example:

```
<tab id="myTab" label="My Tab">
  <group id="myGroup" label="My Group">
    <button id="myButton"
                label="My Button"
                image="Colorblk2"
                size="large"
                onAction="myButton_ClickHandler"/>
  </group>
</tab>
```

This example provides a tab called `My Tab`, a group named `My Group`, and a large button named `My Button`. The example uses a custom image, so it adds this information to the `image` attribute. The "Handling graphics in Visual Studio" section of the chapter describes the special techniques you use to work with graphics in Visual Studio.

When the user clicks My Button, Office calls the `myButton_Click Handler()` method in the add-in. At this point, your code gains control and you can perform any task needed. In this case, the code performs the simple task of displaying a message box. Notice that the method provides the control as input — so you could write one method to address the needs of multiple controls, but it's always better to handle each control on the Ribbon individually to avoid potential confusion.

```
public void myButton_ClickHandler(
   Office.IRibbonControl Control)
{
   // Display a simple message box.
   MessageBox.Show("My Button Clicked");
}
```

It's important to note that this method doesn't return a value to Office because you don't expect Office to do anything. When working with an existing control, you have to tell Office whether to perform a default action.

Modifying or repurposing existing tabs, groups, and controls

Sometimes you'll want to modify or repurpose an existing tab, group, or control, rather than create something new. For example, you might want to change the functionality of the Format Painter to meet internal formatting requirements or hide some tabs, groups, or controls completely. Before you can perform a modification of this sort, you need to know which tab, group, or control to modify. The "Obtaining an identifier for an existing tab, group, or control" section of Chapter 4 tells you how to perform this task.

When you modify an existing element, you work with it as you would when working with a new feature. You add the required information to the existing tab or group. Listing 5-5 shows an example of adding a control to the existing Home tab.

Listing 5-5: Adding a Group and Control to the Home Tab

```
<tab idMso="TabHome">
  <group id="BehaviorChange"
         label="Behavior"
         insertAfterMso="GroupFont">
    <toggleButton id="StopUnderline"
                  label="Stop Underlining"
                  onAction="StopUnderline_ClickHandler"
                  getPressed="StopUnderline_GetPressed"
                  size="large"
                  imageMso="ShapeFillColorPicker"
                  insertBeforeMso="UnderlineGallery"/>
  </group>
</tab>
```

Whenever you want to change an existing feature, you use the `idMso` attribute, in place of the `id` attribute. You supply the identifier that Office uses for this feature. In this case, the `TabHome` identifier points to the Home tab of Excel.

The new group, `Behavior`, includes the `insertAfterMso` attribute. You use this attribute to control positioning of the new group on the Home tab. You can only position groups and controls using an existing group or control or a qualified group or control as a reference point. A *qualified* group or control is one that has its own namespace so Office can ensure that the group or control is unique. The four attributes that control positioning are

- insertAfterMso
- insertAfterQ
- insertBeforeMso
- insertBeforeQ

Most controls require only that you provide an `onAction` attribute so that something happens when the users interact with the control. The `<toggleButton>` in this example also requires a `getPressed` attribute entry so Office knows how to set the state of the button.

The process of repurposing an existing control is different from modifying one. In this case, you must add `<command>` entries to the `<commands>` element, as shown in Listing 5-6.

Listing 5-6: Repurposing Controls

```
<commands>
  <command idMso="Underline"
           onAction="myUnderline"/>
  <command idMso="FileSaveAsExcel97_2003"
           onAction="FileSaveAs_ClickHandler"/>
</commands>
```

This example has two repurposed controls. The first is the Underline button that appears on the Home tab. The second is the Excel 97-2003 Workbook entry found on the Save As menu of the Office menu. In both cases, the example overrides the `onAction` attribute, which is the most common override you'll perform. However, you should look at other callbacks as needed, such as `getPressed` when you're working with a toggle button.

The callback code for implementing all of the features discussed in Listings 5-5 and 5-6 appears in the Ribbon Callbacks region of the `Ribbon1.CS` file. The example must implement methods for the new control, as well as the underline. Listing 5-7 shows the code you'll need in this case.

Listing 5-7: Handling a Change in Tab Control Functionality

```
public void myUnderline(
   Office.IRibbonControl Control,
   Boolean Pressed,
   ref Boolean CancelDefault)
{
   // Check the underline control state.
   if (UnderlineState)
   {
```

```
      // Display an error message.
      MessageBox.Show("No Underline Allowed");
      // Set the control so it isn't pressed.
      Pressed = false;

      // Display the correct state on screen.
      ribbon.InvalidateControl(Control.Id);

      // Tell Office not to perform the default action.
      CancelDefault = true;
   }
   else
      // Otherwise, tell Office to perform the default
      // action.
      CancelDefault = false;
}

public void StopUnderline_ClickHandler(
   Office.IRibbonControl Control,
   Boolean Pressed)
{
   // Store the current button state.
   UnderlineState = Pressed;

   // Display the correct state on screen.
   ribbon.InvalidateControl(Control.Id);
}

public Boolean StopUnderline_GetPressed(
   Office.IRibbonControl Control)
{
   // Return the current pressed state.
   return UnderlineState;
}
```

The code begins with the `myUnderline()` method, which controls whether the Underline control works as normal or provides alternative functionality based on whether the user has the Stop Underlining toggle button pressed. During normal operation, the method simply sets `CancelDefault` to `false` (which means that Office performs the default tasks) and exits.

On the other hand, when the user presses Stop Underlining, the method displays a message box stating that the Underline control won't work. It then sets `Pressed` to `false`, which means that the Underline control won't appear depressed as it would if the command proceeded normally. Many interface changes appear even when you cancel the default activity, so often

you must reset the control to the desired setting. Changes the code makes to a control won't appear unless it calls `ribbon.InvalidateControl()` with the `Control.Id` argument. The code then sets `CancelDefault` to `true` and exits.

Something seemingly odd happens in this case. When you run the code, you'll see the message box appear twice, even though you only clicked Underline once. The first appearance of the error dialog box does occur when you click Underline. The second appearance occurs because Office pressed the button, but the code set `Pressed` to `false`. The result is the same and you wouldn't notice the second call normally (or you could add code to keep from displaying it by checking the status of `Pressed`).

Modifying or repurposing the Office menu

All of the items on the tabs relate somehow to a task. However, you might not always need to add task-related features to an application. The Office menu contains a wealth of configuration and file entries that don't relate directly to a task. When you want to create a nontask entry, add the `<OfficeMenu>` entry, as shown in Listing 5-8.

Listing 5-8: Modifying Office Menu Functionality

```
<officeMenu>
  <menu idMso="FilePrepareMenu">
    <button id="NewPrepButton"
            label="My Prepare Button"
            description="Prepare Time"
            image="TIME"
            insertBeforeMso="FileProperties"
            onAction="NewPrepButton_ClickHandler"/>
  </menu>

  <splitButton idMso="FileSaveAsMenu">
    <menu idMso="FileSaveAsMenu">
      <button id="SayHello"
              label="Say Hello"
              description="This button says hello!"
              image="Colorblk2"
              onAction="SayHello_ClickHandler"/>
    </menu>
  </splitButton>

</officeMenu>
```

Notice that you use buttons in the same way as normal, but that you may need to implement special handling in some cases. The `WordRibbonControls.xlsx` file is found in the `2007OfficeControlIDsExcel2007.exe` file that you can download from the 2007 Office System Document: Lists of Control IDs site at

`http://www.microsoft.com/downloads/details.aspx?FamilyId=4329D9E9-4D11-46A5-898D-23E4F331E9AE`

This file tells you how to handle the various controls. For Excel, the `FilePrepareMenu` entry appears as a menu, and the `FileSaveAsMenu` entry appears as a split button, despite the fact that they appear physically the same in the application. You'll need to exercise caution when working with some Ribbon features for this reason.

Repurposing an Office menu control works precisely the same as it does on the Ribbon. The `FileSaveAsExcel97_2003` entry in Listing 5-6 actually appears on the Office menu. Listing 5-9 shows the contents of the code-behind file for these Ribbon changes.

Listing 5-9: Handling a Change in Office Menu Control Functionality

```
public void NewPrepButton_ClickHandler(
   Office.IRibbonControl Control)
{
   // Display a simple message box.
   MessageBox.Show("Are You Prepared?");
}

public void FileSaveAs_ClickHandler(
   Office.IRibbonControl Control,
   ref Boolean CancelDefault)
{
   // Holds the user's response.
   DialogResult Response;

   // Ask the user about saving the file.
   Response =
   MessageBox.Show(
      "Saving as an older version. Are you sure?",
      "Old File Version Warning",
      MessageBoxButtons.YesNo);

   // Check the response.
   if (Response == DialogResult.Yes)
      CancelDefault = false;
   else
      CancelDefault = true;
}
```

(continued)

Listing 5-9 *(continued)*

```
public void SayHello_ClickHandler(Office.IRibbonControl
          Control)
{
   // Display a simple message box.
   MessageBox.Show("Hello");
}
```

The `NewPrepButton_ClickHandler` entry handles the My Prepare button that appears on the Prepare menu of the Office menu. As you can see, it displays a simple message box. The Say Hello button on the Save As menu works the same way.

The repurposed code for the Excel 97-2003 Workbook entry on the Save As menu is a bit different. This code begins by asking the user whether Excel really should save the file in an older format. Only if the user clicks Yes does the code permit the default action. This code has a very practical application for organizations that are trying to wean users off the older Office file formats.

Creating a Ribbon Using startFromScratch Mode

This section appears in the chapter mostly as a warning. You really *don't* want to create a Ribbon from scratch using an add-in. It may seem like a good idea, but it really isn't — because your organization will suffer a deluge of support calls. It's possible that you could come up with a good reason to create the Ribbon from scratch when working with a single document, or you might even come up with a convincing argument (in rare cases) for a template.

If you create the Ribbon from scratch with an add-in, it affects every document that the Office application loads. The result is chaos because users will almost certainly need the "lost" standard controls to perform their work.

You won't find an example of how to create the Ribbon from scratch for Visual Studio in the source code for this book. However, in the interest of completeness, you'll probably want to know how to check other people's code for this possible mistake. The essential change to the XML you'd use to

begin from scratch appears in the `<ribbon>` element. You simply add the `startFromScratch` attribute, as shown here:

```
<ribbon startFromScratch="true">
```

If you do decide that you absolutely must start a Ribbon from scratch, use VBA to do it. The "Creating a Ribbon Using startFromScratch Mode" section of Chapter 4 tells you how to perform this task using VBA. The technique works well for either single documents or templates. In fact, you'll even find an example of this technique in the Chapter 4 source code.

Part III

Creating New RibbonX Applications

The 5th Wave By Rich Tennant

"Once I told Mona that Access was an 'argument' based program, she seemed to warm up to it."

In this part...

Every Office product that supports the Ribbon has something special to offer when it comes to creating a RibbonX application. The chapters in this part show you the unique characteristics of each Office product. Chapter 6 discusses Word applications and shows you an interesting letter-writing application. Chapter 7 discusses Excel and shows a new way to work with equations. Chapter 8 discusses Access and shows how to create multiple Ribbons for a single database. Chapter 9 discusses Outlook and demonstrates how to manage your e-mail more effectively. Chapter 10 discusses PowerPoint and shows how you can automate presentation tasks.

Chapter 11 is special because it shows how to work with Web services from within a Ribbon application. You could use this information to work with any Web service in any Office product. In this case, you'll work with Amazon Web Services using Excel.

Chapter 6

Developing Business Applications for Word

In This Chapter

- Understanding how to work with Word
- Designing a letter and memo tab
- Using automation to create envelopes
- Designing labels
- Defining and automating forms

Microsoft Word is the tool of choice for most documentation tasks in organizations of every size. In fact, Word sees use in more than just the corporate world; you'll see it used in everything from home businesses to college campuses. The kind of documents that people create using Word is also nothing short of amazing. In many cases, developers think about Word as a means of creating a letter or providing documentation to others. However, Word also sees use as a means of filling out forms and even accessing Web services. Some people use Word exclusively to create reports, while others use it to formulate presentations. You could possibly make a career out of finding new uses for Word alone — and that's not even considering the other elements of the Office suite.

This chapter discusses a number of major application categories for Word that include the typical letter or memo and more unusual uses such as reports. In between, you'll find applications to create envelopes, design labels, and automate forms. The last application in the chapter even describes how to use Word when working with graphics. Although this chapter hardly exhausts every possible use for Word, you'll find enough variety here to make it significantly easier to create a Word application of your own for your personal or organizational use.

You'll also find that the chapter divides the examples by language. The simpler examples (letter and memo, envelopes, and labels) rely on VBA as a language because they lend themselves to that particular environment. Yes, you could easily create these examples using Visual Studio, but these examples don't really require such an advanced product. Likewise, the chapter uses Visual Studio to create some of the more complex examples (forms, reports, and graphics applications). Even though you could create these examples using VBA, the complexity of doing so outweighs other benefits you receive by using Visual Studio.

Getting Started with Word Applications

Word is one of the most used of the Office applications. The following list tells some of the reasons that Word is so popular:

- Provides a broad range of uses
- Offers a significant amount of flexibility
- Customizes well to a variety of needs
- Lets you produce output in final format

You have a choice of using VBA or Visual Studio when working with Word, but the choice isn't simply one of personal taste or capabilities. Because of the way that Microsoft has added Ribbon functionality to Word, you might find your choice dictated more by the needs of the application than any other factor.

Understanding Word and VBA

VBA offers the best choice for document- or template-centric tasks. For example, most companies rely on a particular template to create letters. Some companies may have several templates for the purpose. However, it's unlikely that a company will have employees whose sole purpose is to type letters (and absolutely nothing else), so it's unlikely that a Ribbon add-in for letters will work very well. Using VBA to produce a template for letter add-ins is a significantly better choice.

Any application that affects only a subset of the documents produced by Word is a candidate for VBA. However, don't get the idea that VBA is always the best choice. For example, it might appear at first that VBA is the only choice when creating a forms application. The way the user employs the form is the determining factor in this case. If you create a form that the user must fill out for every project, such as a routing sheet, then the form may work better as an add-in, which means you should use Visual Studio instead of VBA to complete the task.

Unfortunately, you can't combine VBA and Visual Studio with any ease unless you plan carefully. For example, you could create an add-in that provides basic functionality, and then augment that capability based on the document or template using VBA. For example, thinking about the forms example again, you could create general forms capability using a Visual Studio add-in, and then augment that functionality using VBA. Each document or template could include special form features for that particular need.

Understanding Word and Visual Studio

The add-ins you create using Visual Studio affect every document that the user opens with Word. Consequently, Visual Studio is the best tool to use when you want to create an application that affects the user all of the time. If there's a company policy in place that the user can't underline anything in a document or use strikethrough without employing revision marks, then you could add that restriction quite easily, using an add-in. The policy would remain in effect no matter which document the user opened.

As mentioned in the "Understanding Word and VBA" section of the chapter, some forms lend themselves to add-ins. You might also want to offer final output options as part of an add-in. For example, a user might always have to create output in a report format no matter what document they work with. The report format could be something as simple as adding the company's logo to the output or including a special notice. Of course, reports can take other forms and you can make the output as simple or as fancy as needed. The point is that reports often encompass more than one document type and lend themselves to implementation as an add-in as a result.

A more common global application need is graphics. Word comes with a good set of graphics primitives, which works fine if you happen to be an artist. However, for those who don't draw well, the graphics primitives are probably a little too primitive. A graphics tab could provide access to company-approved icons and clip art. The tab could even include guidelines for adding the art to a document.

Creating a Letter/Memo Tab

The Letter/Memo tab, shown in Figure 6-1, demonstrates the perfect use of a custom tab to promote workflow. In this case, the user moves from left to right across the Ribbon. When the user reaches the last group, the user has a new, perfectly formatted, letter in place. The major concept, in this case, is to keep the user moving and yet provide complete assistance. The template ensures that the user doesn't miss anything and that every element appears

as it should according to company guidelines. Most important of all, the user doesn't have to think too hard about anything; the template helps the user create a letter without forcing the user to think about menus, toolbars, special formatting, or anything else of that nature. The tab doesn't hide anything, either; it doesn't have to in order to make things easy.

Figure 6-1: Use custom tabs to promote a workflow within your organization.

One of the more important issues to consider is whether a particular set of features applies to a document or a template. A user is quite likely to create multiple letters and memos, so this example works best as a template. However, when you create the Ribbon and associated VBA macros, it's actually easier to work with a document because you can see the results of any changes faster. Consequently, this example began with a document and ended as a template to make development considerably easier.

The Letter/Memo tab example demonstrates several techniques, and the chapter simply can't hold all the required source code. For example, you won't see all of the source code for the Ribbon because the book already has several examples of this code. The text also skips repetitive coding examples and presents only one version of a particular coding technique. The sections that follow do provide you with complete information about all of the essential techniques for working with this example and modifying it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the `Letter and Memo.DOTM` template, and the `Sample Letter.DOCM` document.

Setting the style

The user begins all the way to the left of the Letter/Memo tab in the Style group. The Split button defaults to the Letter style for new documents. The other styles include a memo, an invitation, and a past due notice. You could add as many styles as needed to support your organization. Each of the styles sports a special icon, label, and appearance. Of course, RibbonX provides no obvious means of changing either the icon or the label as the user

works with the application. The trick is to provide the required functionality as a callback. Listing 6-1 shows the required XML for the Style group.

Listing 6-1: Creating a Style Group

```
<group id="Style" label="Style">
   <splitButton id="UseStyle" size="large">
      <button id="StSelected"
              onAction="StyleDefault"
              getImage="StyleGetImage"
              getLabel="StyleGetLabel"/>
      <menu id="OtherStyles">
         <button id="StLetter"
                 label="Letter"
                 onAction="StyleLetter" />
         <button id="StPastDue"
                 label="Past Due Notice"
                 onAction="StylePastDue" />
         <button id="StMemo"
                 label="Memo" onAction="StyleMemo" />
         <button id="StInvitation"
                 label="Invitation"
                 onAction="StyleInvitation" />
      </menu>
   </splitButton>
</group>
```

Notice that the `StSelected` selected button has no image or label assigned to it. This button uses the `getImage` and `getLabel` attributes instead to assign a VBA callback to handle the image and label needs. The default button actually shows the current document selection, so it doesn't really do anything, in this case, when you press it. This is an alternative use for a Split button; it acts as a status indicator. The buttons that actually perform a change appear as part of the `OtherStyles` menu. The code for performing the image and label updates appears in Listing 6-2.

Listing 6-2: Modifying Labels and Images at Runtime

```
'Callback for StDefault getLabel
Sub StyleGetLabel(control As IRibbonControl, _
                  ByRef returnedVal)

    ' Return the currently selected document style.
    returnedVal = DocType
End Sub

'Callback for StDefault getImage
Sub StyleGetImage(control As IRibbonControl, _
```

(continued)

Listing 6-2 *(continued)*

```
                    ByRef returnedVal)

    ' Choose an image based on the document type.
    Select Case (DocType)
        Case "Letter"
            returnedVal = "EnvelopesAndLabelsDialog"
        Case "Past Due Notice"
            returnedVal = "PermissionRestrict"
        Case "Memo"
            returnedVal = "Paste"
        Case "Invitation"
            returnedVal = "FileCreateDocumentWorkspace"
        Case Else
            returnedVal = "EnvelopesAndLabelsDialog"
    End Select
End Sub
```

The `StyleGetLabel Sub` simply returns a string called `DocType`. The `DocType` variable is global, and the application uses it to track the document type. You can use this information to add text to the document, change the styles based on document type, and to control the actual document content. In this case, the example adds special text for each of the document types and modifies the labels and images.

The image change occurs when Word calls the `StyleGetImage` callback. The code uses a `Select Case` statement to choose the image based on the document type. All these images are built in rather than custom. If you use a custom image, you must embed it as part of the document and then use the name you supplied for the image.

You've seen the physical interface and the code that modifies the appearance of the split button. However, the application still requires some code to trigger the change. Listing 6-3 shows a typical example of one of the button callbacks.

Listing 6-3: Changing the Document Type

```
'Callback for StLetter onAction
Sub StyleLetter(control As IRibbonControl)

    'Set the new default document style.
    DocType = "Letter"

    ' Change the custom document properties.
    ActiveDocument.CustomDocumentProperties( _
        "DocType").Value = "Letter"

    ' Update the Ribbon.
```

```
    Rib.InvalidateControl "StSelected"

    ' Set the document headers.
    SetHeaders
End Sub
```

The code begins by changing the `DocType` variable to reflect the change in document type. It then stores this information in a custom document property using the `CustomDocumentProperties()` method. This step is extremely important because without it, the document won't know what type it is when you open it later. The custom property acts as a permanent memory of the current document type. You can see this custom property by performing the following steps.

1. **Choose Office Menu➪Prepare➪Properties.**

   You see the Document Properties window.

2. **Click the down arrow next to Document Properties and choose Advanced Properties from the menu.**

   You see a Properties dialog box.

3. **Select the Custom tab.**

   Word displays a list of custom properties, including the `DocType` property, as shown in Figure 6-2.

4. **Click Cancel to close the Properties dialog box. Click the close box to close the Document Properties window.**

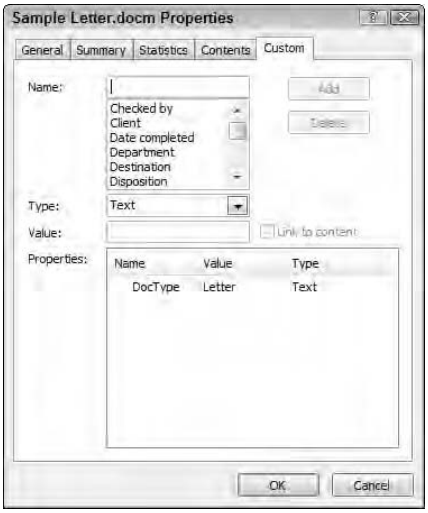


Figure 6-2: Word stores the custom properties you create as part of each document.

The next code step in Listing 6-3 is the most important because it demonstrates a RibbonX principle you need to remember. Notice that the `InvalidateControl` method doesn't use the control identifier for the current control. Instead, the control identifier refers to the default button for the Split button. By invalidating the default button, the code triggers the `getLabel` and `getImage` methods described in Listing 6-2. Even though you can't modify the Ribbon controls directly, you can use these roundabout methods to perform a change in appearance.

The code finally calls a `Sub` called `SetHeaders`. The `SetHeaders Sub` is an example of business logic. You could use the same code in earlier versions of Office to modify the content of the document. In this case, the `Sub` changes the document style features including some heading text. It relies on search techniques to find text that belongs to the older style and then updates the document with any new style-specific entries.

The final step in the document type process is to ensure the document type is correct when the user opens the document. Listing 6-4 shows the document type code that appears in the `OnLoad` callback.

Listing 6-4: Restoring the Document Type After the Ribbon Loads

```
' Check the document type and add a custom property
' for the document if necessary.
DocType = ""
Dim CurrProp As DocumentProperty
For Each CurrProp In _
    ActiveDocument.CustomDocumentProperties

    If CurrProp.Name = "DocType" Then
        DocType = CurrProp.Value
    End If
Next
If DocType = "" Then
    ActiveDocument.CustomDocumentProperties.Add _
        Name:="DocType", _
        LinkToContent:=False, _
        Type:=msoPropertyTypeString, _
        Value:="Letter", _
        LinkSource:=False
    DocType = "Letter"
End If
```

The code begins by initializing `DocType` to an empty string. It then looks for the `DocType` custom property. If this property doesn't appear in the document, the code sets the property and the `DocType` variable to a default type of Letter. Because so many parts of the application rely on `DocType`, you must ensure that new documents receive the proper initialization.

Adding a recipient

The Style group is an example of a modification that can take several forms, but the user can select only one form at a time. The Recipient group provides another sort of entry. The output of this group is positional. The recipient's name is normally going to appear at the top of any document you create, so it always appears first in the document no matter when the user adds the name. Of course, the name does appear below any headings. The recipient's address always appears after the name. Again, it doesn't matter when the user adds the address. The technique that the example employs to ensure the document fidelity is to use special styles throughout the document. The code can then search for these styles and place content correctly regardless of position within the document.

The layout of the Recipient group also shows a preference for some options over others. The Recipient Name button is large to emphasize its importance. Every document type requires a recipient name. The Address and Account Number buttons appear next as smaller buttons because the user will need them for most, but not all, documents. A separator keeps optional buttons apart from the standard buttons. The user probably won't need to include a telephone number or office location, in most cases, so these buttons appear separately and at the lowest priority.

Obtaining a recipient name is a little tricky; you have to consider where the user is most likely to obtain the required recipient information. Because many companies have Outlook (not Outlook Express), the example uses the address book from Outlook as a source of information. To begin working with Outlook, create a reference to the required object library in VBA. Select Tools➪References, and you'll see the References dialog box, as shown in Figure 6-3. Check the Microsoft Outlook 12 Object Library entry and click OK.

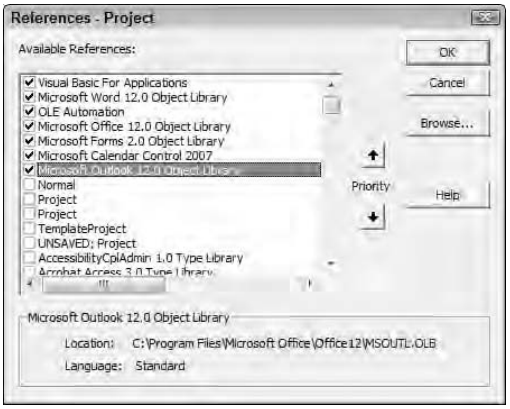


Figure 6-3: Add a reference to Outlook so your Word code can interact with it.

After you have the Outlook reference, you can begin interacting with Outlook using standard VBA programming techniques. Listing 6-5 shows the callback code for obtaining the recipient name. This code also creates the `ThisRecipient` object that the rest of the recipient features use to obtain information about the recipient.

Listing 6-5: Obtaining Recipient Information from Outlook

```
'Callback for RcptName onAction
Sub RecipientName(control As IRibbonControl)

    ' Create a reference to the Outlook Address list.
    Dim SelName As Outlook.SelectNamesDialog
    Set SelName = _
       Outlook.Application.Session.GetSelectNamesDialog

    ' Obtain the current pane object.
    Dim CurrPane As Pane
    Set CurrPane = Application.ActiveWindow.ActivePane

    ' Allow the user to select only one name.
    SelName.AllowMultipleSelection = False

    ' Remove the standard email fields.
    SelName.SetDefaultDisplayMode olDefaultSingleName

    ' Display the selection dialog box.
    SelName.Display

    ' Make sure the user selected one and only one
    ' name.
    If SelName.Recipients.Count = 1 Then

        ' Store the recipient information for later
        ' use.
        Set ThisRecipient = SelName.Recipients.Item(1)
    Else

        ' Display an error message.
        MsgBox "You must select one recipient."

        ' Leave the Sub.
        Exit Sub
    End If

    ' Go to the beginning of the document.
    CurrPane.Selection.GoTo wdGoToLine, wdGoToFirst

    ' Check for a special header.
    If CurrPane.Selection.Style = "Special Header" Then

        ' Move past the special header.
```

```
        CurrPane.Selection.GoToNext wdGoToLine
    End If

    ' Add a Special Heading paragraph.
    CurrPane.Selection.InsertParagraph
    CurrPane.Selection.Style = "Recipient Name"

    ' Insert the recipient's name.
    CurrPane.Selection.EndKey
    CurrPane.Selection.Text = ThisRecipient.Name
    CurrPane.Selection.GoToNext wdGoToLine
End Sub
```

The code begins by creating an instance of the `SelectNamesDialog`, which provides access to the Outlook address book from inside Word. Because the application is interested in obtaining a single name only, it sets the `AllowMultipleSelection` property to `False`. In addition, it removes the e-mail fields by setting the `SetDefaultDisplayMode` property to `olDefaultSingleName`. When using the default settings, the user sees the security dialog box, shown in Figure 6-4, when the code calls `SelName.Display`.

Notice that you can check Allow Access For and set a time interval to avoid seeing the message more than once. The longest interval you can set is 10 minutes, which provides sufficient time to perform all of the configuration tasks for this application. After the user approves the Outlook Access, the application displays the Select Name: Contacts dialog box, as shown in Figure 6-5. (The figure shows just one name, a test contact used for this example.)

Even with all of the precautions taken with the Select Name: Contacts dialog box, the user can still choose to click OK without actually selecting a contact. Consequently, the code verifies that the user has selected precisely one entry.

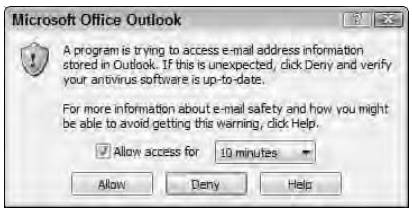


Figure 6-4: Outlook displays this security dialog box when using the default settings.

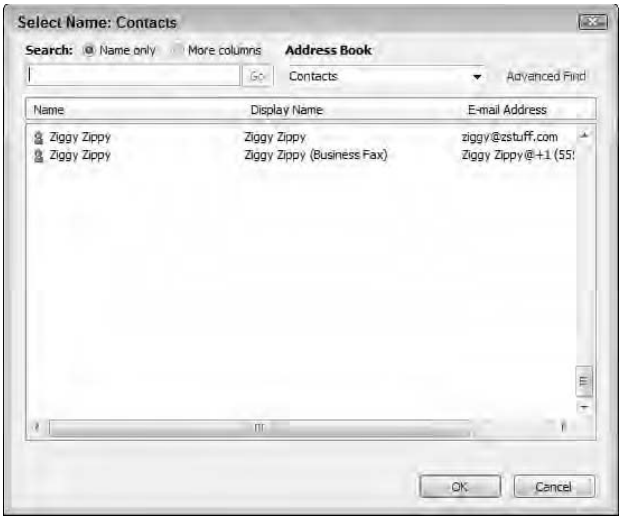


Figure 6-5: The user can select any contact in their address list.

The data placement comes next. The code begins by placing the insertion point at the beginning of the file. It then looks for the Special Header style in the document. The document can have only one (or no) Special Header paragraph when the user uses the application. That's because selecting another style automatically removes any other paragraph that uses the Special Header style. When the code detects a Special Header paragraph, it moves past it.

The code inserts a new paragraph at this point and selects the Recipient Name style for it. You have to move the insertion point because Word selects the entire paragraph when you insert a new one, so the code calls `CurrPane.Selection.EndKey`. The actual data appears as part of `ThisRecipient.Name`. The code ends by placing the insertion point on the next line.

Using `ThisRecipient` as global storage for the recipient information saves time. The other entries that the user can select work with this variable, rather than accessing Outlook directly. Consequently, adding other entries is simply a matter of accessing the correct data in `ThisRecipient`. For example, when the user wants to insert the recipient address, the code accesses the `ThisRecipient.AddressEntry.GetContact.BusinessAddress` property. All of the positioning depends on searching for particular styles and then manipulating the insertion pointer as needed to ensure accurate placement of new data. Figure 6-6 shows how typical recipient information might appear. Notice the use of styles for each data type in the style bar on the left side of the display.

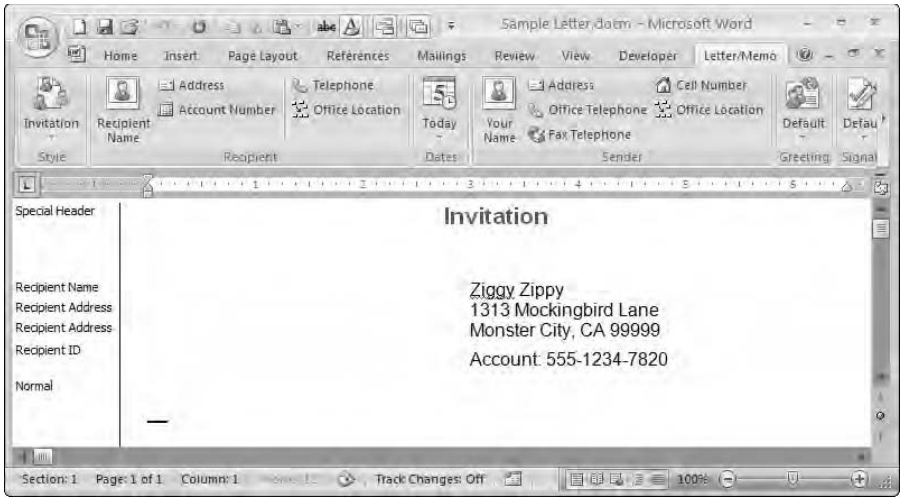


Figure 6-6: Styles play an important part in positioning data using this application.

Working with dates

Unlike other elements in this application, dates can appear anywhere in the document and the user will very likely need more than the current date. However, the application can still help the user with correctness, formatting, and placement. The application accomplishes correctness by displaying several common date options as part of the split button, as shown in Figure 6-7. Notice that in this case (unlike the Style split button mentioned earlier), the default option does have an actual purpose: inserting today's date.

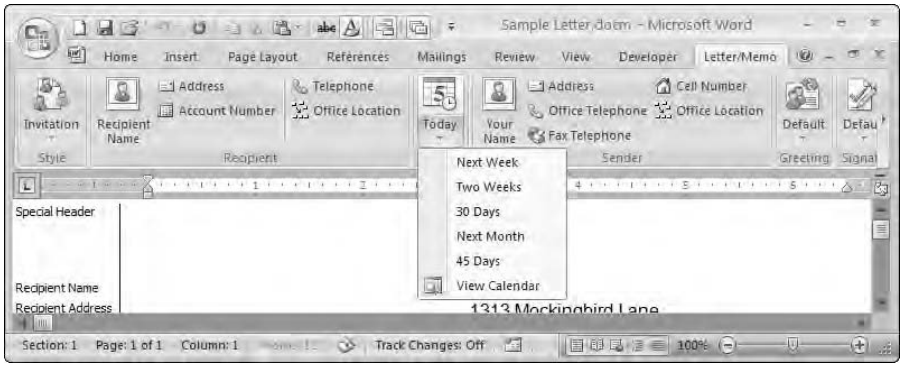


Figure 6-7: Effective use of menu options can serve to reduce user error.

The options don't have icons, for the most part, because they insert the date directly. After all, the application can make the required calculation for the user. However, the View Calendar option does display a special selection dialog box, as shown in Figure 6-8, because the user must choose a date. The use of this dialog box does, however, ensure that the result is a correctly formatted date, and tends to reduce errors (such as choosing a Sunday when your business isn't open on Sunday).

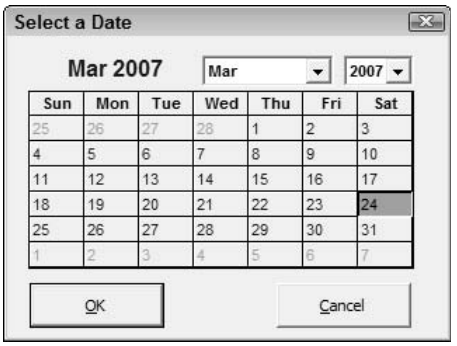


Figure 6-8: Use a calendar control to reduce user errors.

No matter which date option the user selects, the application displays a positioning dialog box, as shown in Figure 6-9. The positioning dialog box offers the user three alternatives for inserting the date. The code automatically places the date and formats it as required after the user makes a selection.

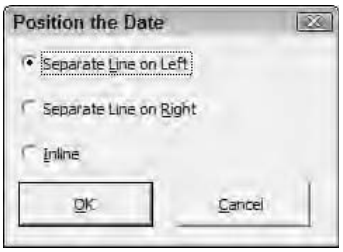


Figure 6-9: Careful planning ensures that the workflow isn't broken by formatting issues.

The callbacks for the date are relatively simple. All you really need to do, in most cases, is add the appropriate time to the existing `DateTime.Date` property. Listing 6-6 shows the code for the most complex example, figuring out precisely one month from the current date.

Listing 6-6: Creating, Formatting, and Inserting a Date

```
'Callback for NextMonth onAction
Sub DateNextMonth(control As IRibbonControl)

   ' Obtain the current month
   Dim Month As Integer
   Month = DateTime.Month(DateTime.Now) + 1

   ' Create a date string.
   Dim DateStr As String
   DateStr = _
      CStr(Month) + "/" + _
      CStr(DateTime.Day(DateTime.Now)) + "/" + _
      CStr(DateTime.Year(DateTime.Now))

   ' Create a new date.
   Dim NewDate As Date
   NewDate = DateTime.DateValue(DateStr)

   ' Insert the required date.
   InsertDate CStr(NewDate)
End Sub

Sub InsertDate(TheDate As String)

   ' Lets the user set the date position.
   Dim GetPosition As ChoosePosition
   Set GetPosition = New ChoosePosition

   ' Obtain the current pane object.
   Dim CurrPane As Pane
   Set CurrPane = Application.ActiveWindow.ActivePane

   ' Get the user input.
   GetPosition.Show

   ' Determine the date position.
   Select Case GetPosition.Result
      Case PositionResult.Left

         ' Add a Date Left paragraph.
         CurrPane.Selection.InsertParagraph
         CurrPane.Selection.Style = "Date Left"

         ' Insert the date.
         CurrPane.Selection.EndKey
         CurrPane.Selection.Text = TheDate
         CurrPane.Selection.GoToNext wdGoToLine

      Case PositionResult.Right

         ' Add a Date Left paragraph.
```

(continued)

Listing 6-6 ***(continued)***

```
            CurrPane.Selection.InsertParagraph
            CurrPane.Selection.Style = "Date Right"

            ' Insert the date.
            CurrPane.Selection.EndKey
            CurrPane.Selection.Text = TheDate
            CurrPane.Selection.GoToNext wdGoToLine

        Case PositionResult.Inline

            ' Insert the date.
            CurrPane.Selection.Text = TheDate
            CurrPane.Selection.EndKey

        Case PositionResult.Cancel

            ' Don't do anything.

    End Select
End Sub
```

The example begins by obtaining the current month, adding a month to it, and using the result to create a date string. It then uses the `DateTime.DateValue()` method to create a date and pass it to the `InsertDate Sub`.

The `InsertDate Sub` begins by creating and displaying the Position the Date dialog box, as shown in Figure 6-9. After the user makes a choice, the code inserts the date using one of three methods. The first two methods create a new paragraph and assign a style to it. All three methods insert the date at the insertion point.

Adding the sender

From the user's perspective, the buttons in the Sender group act much like those in the Recipient group. The user clicks Your Name first, and then clicks any of the other informational buttons as needed. All of the output is positional, like the recipient information. In this case, the application assumes that the user is moving from left to right; it inserts the sender name at the end of the document because that's the next position in the workflow to place data. You could base the position on other criteria, such as placing the username immediately after the date. Using unique styles for each data type means you have flexibility in determining where to place the next data item.

The code is different from the recipient code for a number of reasons. The first reason is that you already know the user's name because it appears as

part of the Word configuration information. Accessing the name and address is very easy, as shown in Listing 6-7. However, since Word only provides access to the user's name and address, you must supply some other means of obtaining other user information such as telephone number and office location. The example relies on a user Outlook address-book entry, which seems a reasonable choice because the user will probably want to add his or her address entry to e-mails and other Outlook objects.

Listing 6-7: Obtaining the Sender Information

```
'Callback for SendName onAction
Sub SenderName(control As IRibbonControl)
   ' Obtain the current pane object.
   Dim CurrPane As Pane
   Set CurrPane = Application.ActiveWindow.ActivePane

   ' Go to the end of the document.
   CurrPane.Selection.GoTo wdGoToLine, wdGoToLast

   ' Add a Sender Name paragraph.
   CurrPane.Selection.InsertParagraph
   CurrPane.Selection.Style = "Sender Name"

   ' Insert the sender's name.
   CurrPane.Selection.EndKey
   CurrPane.Selection.Text = Application.UserName
   CurrPane.Selection.GoToNext wdGoToLine

   ' Look up the sender in Outlook. Begin with the
   ' first name and go from there.
   Dim CheckSender As AddressEntry
   Set CheckSender = _
      Outlook.Application.Session.AddressLists. _
      Item(1).AddressEntries.GetFirst

   ' Check the name.
   If CheckSender.Name = Application.UserName Then
      Exit Sub
   End If

   ' If this isn't the right user, keep searching.
   For Each CheckSender In _
      Outlook.Application.Session.AddressLists. _
      Item(1).AddressEntries

      ' Check the entry name.
      If CheckSender.Name = Application.UserName Then
         Exit For
      End If
```

(continued)

Listing 6-7 ***(continued)***

```
    Next

    ' Determine whether we have an ID to use.
    If Not CheckSender Is Nothing Then
       Set ThisSender = _
          Outlook.Session.GetRecipientFromID( _
             CheckSender.ID)
    End If
End Sub
```

The code begins by inserting the paragraph at the end of the document and setting it to use the Sender Name style. The code then accesses the user's name with the `Application.UserName` property. Using this approach means that you don't have to ask the user for any of the required sender information for a properly configured system.

As previously mentioned, however, you can't obtain the rest of the sender information that the application requires from Word because Word tracks only the username and address. The code creates a new Outlook `AddressEntry` object, `CheckSender`, to locate the user's information in the Outlook address list.

The `Outlook.Application.Session.AddressLists.Item(1).AddressEntries` collection contains all of the `AddressEntry` objects for the selected address book (as indicated by the `Item(1)` index). The `For Each` loop checks each collection entry in turn until it locates an entry where the `Name` property matches the `Application.UserName`.

The check depends on a user creating an address entry with precisely the same name as his or her username in Word. This is a potential point of failure, and you'll probably want to set up configuration policies to guard against it.

When the code finds an address entry, it creates a `Recipient` object, `ThisSender`, that functions precisely the same as `ThisRecipient`. The code uses the `GetRecipientFromID()` method to obtain the `Recipient` object. Notice that it relies on the `CheckSender.ID` property to obtain the required entry ID value.

Greeting the recipient and adding a signature

The Greeting and Signature groups are the easiest part of this application to understand. All they do is insert standard text snippets, using the same techniques as many of the other entries described for this article.

In fact, some developers would probably question the need for these entries. However, they serve an important role when you create a workflow application. It's essential to maintain the perception that everything the user needs appears on the Ribbon. In addition, your organization may actually have standardized greetings and signatures that it requires authors to use.

Of course, you might also make the argument that these bits of text could actually appear in the template. That's true if you have only one greeting and signature, but again, you run into the problem of maintaining the workflow. You don't want the user to see the text until it's time to add it to the document. Consequently, adding the two groups and their associated controls to the Ribbon is the best way to approach the problem.

Considering the CC, routing, and approval requirements

The example has shown many kinds of entries so far. When you select a style, the application removes all of the entries from the previous style first, and then adds the new style changes. Some elements are positional and others appear at the current insertion point. There are good reasons to use each of these entries. The CC, Routing Slip, and Get Approvals check boxes show another kind of entry. In this case, the entries must always appear at the bottom of the document. In addition, the user will expect some type of on/off functionality. The application provides all this required functionality.

The basic functionality requires two callbacks. The first callback performs the application-specific tasks of adding the required data to the document. The second callback changes the appearance of the check box on the Ribbon. Remember that you can't change the check yourself, but must ask Office to perform the task for you. Listing 6-8 shows the code for these two callbacks. Although this code shows the CC check box, the other two check boxes work the same.

Listing 6-8: Adding CC Information to the Document

```
'Callback for RtCC onAction
Sub RouteCC(control As IRibbonControl, _
            pressed As Boolean)

    ' Obtain the current pane object.
    Dim CurrPane As Pane
    Set CurrPane = Application.ActiveWindow.ActivePane

    ' Perform an action based on the current value.
```

(continued)

Listing 6-8 ***(continued)***

```
    If CCPressed Then
        ' Create a search variable.
        Dim DoSearch As Find
        Set DoSearch = CurrPane.Selection.Find

        ' Look for the lines with the correct style.
        DoSearch.Style = "CC"

        ' Look everywhere in the document.
        DoSearch.Wrap = wdFindContinue
        DoSearch.Text = ""

        ' Keep looking for CC lines until they're gone.
        While DoSearch.Execute()

            ' Remove the CC lines.
            CurrPane.Selection.Delete
        Wend

    ' Add a CC.
    Else

        ' Go to the end of the document.
        CurrPane.Selection.GoTo wdGoToLine, wdGoToLast

        ' Add a CC paragraph.
        CurrPane.Selection.InsertParagraph
        CurrPane.Selection.Style = "CC"

        ' Insert the default text.
        CurrPane.Selection.EndKey
        CurrPane.Selection.Text = "CC: "
        CurrPane.Selection.EndKey
    End If

    ' Change the pressed value.
    CCPressed = pressed

End Sub

'Callback for RtCC getPressed
Sub RoutePressedCC(control As IRibbonControl, _
                   ByRef returnedVal)

    ' Return the current pressed status.
    returnedVal = CCPressed
End Sub
```

The `RouteCC` code begins by obtaining a reference to the current document pane. It then detects whether the user has checked the check box or cleared it. When the user checks the check box, the code searches for the end of the document and adds the required text. Likewise, when the user clears the check box, the code locates the existing entry by style, and removes it. Notice that the code handles the situation where the user has created multiple paragraphs with the required style. The user could create such an entry accidentally or to separate the list of names.

One mistake that you might make when searching for a style is forgetting to clear the text entry. You must set `DoSearch.Text = ""` to clear the text or the search could fail in unexpected ways. Worse yet, it could succeed in unexpected ways and produce odd results without displaying an error message.

The `RoutePressedCC` code may almost look too simple when contrasted to other callbacks for this application. Check Listings 6-1 through 6-3 for comparison purposes. However, unlike the Style group, the Ribbon knows that the user has interacted with the CC check box, so you can simply add a `getPressed` attribute, as shown here:

```
<group id="Routing" label="Routing">
   <checkBox id="RtCC" label="CC"
             onAction="RouteCC"
             getPressed="RoutePressedCC"/>
   <checkBox id="RtRouting" label="Routing Slip"
             onAction="RouteRouting"
             getPressed="RoutePressedRouting"/>
   <checkBox id="RtApproval" label="Get Approvals"
             onAction="RouteApproval"
             getPressed="RoutePressedApproval"/>
</group>
```

Simply returning the current state of the check box is enough in this case. Office doesn't require that you invalidate the control separately or perform any other odd programming tasks. However, you do have to consider the startup state of the Ribbon for these controls, which means adding code to the `OnLoad` callback. Listing 6-9 shows the code used to detect the current checkbox state.

Listing 6-9: Determining the Initial Checkbox State

```
'Create a search variable.
Dim DoSearch As Find
Set DoSearch = ThisSelect.Find

' Preset the routing checkboxes.
```

(continued)

Listing 6-9 ***(continued)***

```
CCPressed = False
RoutingPressed = False
ApprovalPressed = False

'Determine the state of each routing checkbox.
DoSearch.Wrap = wdFindContinue
DoSearch.Text = ""
DoSearch.Style = "CC"
If DoSearch.Execute() Then
    CCPressed = True
End If
DoSearch.Wrap = wdFindContinue
DoSearch.Text = ""
DoSearch.Style = "Routing"
If DoSearch.Execute() Then
    RoutingPressed = True
End If
DoSearch.Wrap = wdFindContinue
DoSearch.Text = ""
DoSearch.Style = "Approval"
If DoSearch.Execute() Then
    ApprovalPressed = True
End If
```

The example uses the same technique as always to locate particular bits of information in the document. The content doesn't matter, but the style does because the style determines the information type. The code begins by creating a search and setting each of the global check-box status variables to `false` (meaning the user hasn't checked them).

When the code finds a paragraph of the right type, it sets the associate variable to `true`. Notice that each checkbox has its own style so the code doesn't confuse the entries. In addition, you must reset the `DoSearch.Wrap` property to `wdFindContinue` every time you begin another search.

Automating Envelopes

Many of the Office 2007 features are generic and some are a little unintuitive for the complete novice to use. Two such examples are envelopes and labels. You can see the buttons for each on the Mailings tab with the Create group. You have a number of decisions to make when you change or augment an existing feature. Look again at the Letter/Memo tab in Figure 6-1. You may simply decide to place the Create group on the Mailings tab there and leave the original in place as well. Adding an existing group, with complete functionality, to a custom tab is easy. Simply add the group name, as shown here:

```
<group idMso="GroupEnvelopeLabelCreate" />
```

Remember, however, that you have to keep the whole workflow issue in mind when you work out the details of an application that uses existing features. You may decide that you really don't want users creating labels within the Letter/Memo tab because they won't ordinarily perform that task for a single letter. It's always possible to use just part of a group. For example, you might decide to display the Envelopes button using XML like this.

```
<group id="MailIt" label="Create">
   <button idMso="EnvelopesAndLabelsDialog"
           size="large"/>
</group>
```

Notice that you still must define a size for the button. Otherwise you can't change anything about the Envelopes button — which means you still can't control the behavior. Unfortunately, there's a problem with the Envelopes dialog box. When you click the button, the Delivery Address field is blank unless you know the secret of highlighting the text you want to see in the output. The user also has to know details — such as the kind of envelope loaded in your printer and whether that envelope has your address pre-printed on it. In fact, there are a hundred ways in which the user can create a pile of partially used and useless envelopes.

To overcome this problem (and keep the user from choosing the Envelopes option on the Mailings tab), you can repurpose the control. Normally you won't want to repurpose the default behavior of a control, but in this case you're actually making things easier for the template user. Here's the `<commands>` element you add to override the existing envelope functionality:

```
   <commands>
      <command idMso="EnvelopesAndLabelsDialog"
               onAction="CreateEnvelope" />
   </commands>
```

Of course, now you have to duplicate the functionality that the Envelopes and Labels dialog box provides using other techniques. Fortunately, Word provides the `Application.ActiveDocument.Envelope` object you can use to provide programmatic support for creating an envelope. Listing 6-10 shows how you can use this feature with the Letter/Memo tab.

Listing 6-10: Creating Custom Envelope Output

```
'Callback for EnvelopesAndLabelsDialog onAction
Sub CreateEnvelope(control As IRibbonControl, _
                   ByRef cancelDefault)

   'Create an envelope reference.
   Dim ThisEnvelope As Envelope
   Set ThisEnvelope = _
```

(continued)

Listing 6-10 ***(continued)***

```
      Application.ActiveDocument.Envelope

   ' Set the envelope size.
   ThisEnvelope.DefaultSize = "Size 10"

   ' Omit the return address.
   ThisEnvelope.DefaultOmitReturnAddress = True

   ' Obtain the current pane object.
   Dim CurrPane As Pane
   Set CurrPane = Application.ActiveWindow.ActivePane

   ' Define a search.
   Dim DoSearch As Find
   Set DoSearch = CurrPane.Selection.Find

   ' Holds the address range start.
   Dim AddressStart As Long

   ' Search for the start of the range.
   DoSearch.Wrap = wdFindContinue
   DoSearch.Text = ""
   DoSearch.Style = "Recipient Name"
   If DoSearch.Execute() Then
       AddressStart = CurrPane.Selection.Range.Start
   End If

   ' Search for the end of the range.
   DoSearch.Wrap = wdFindContinue
   DoSearch.Text = ""
   DoSearch.Style = "Recipient Address"
   While DoSearch.Execute()
   Wend

   ' Reset the address start.
   CurrPane.Selection.MoveStart _
      wdCharacter, AddressStart - _
      CurrPane.Selection.Start

   ' Output the envelope.
   ThisEnvelope.PrintOut

   ' Go to the beginning of the document.
   CurrPane.Selection.GoTo wdGoToLine, wdGoToFirst
End Sub
```

The example begins by creating an `Envelope` object, `ThisEnvelope`, based on the default envelope for the active document. The resulting object contains all of the system defaults (which may or may not meet your printing

requirements). You should always check essentials such as the envelope size because you don't know whether the user has changed these entries.

In this case, the example sets the envelope size and omits that return address. The actual power of workflow applications, however, is that you can more easily control the data that the application puts out. As shown in the listing, the application finds the beginning and ending point of the recipient information based solely on the document styles. It then selects the entire range and calls `ThisEnvelope.PrintOut` to print the envelope.

If you decide to include the envelope information as part of the document, you can also use `ThisEnvelope.Insert`. The envelope information appears at the top of the document in a separate section.

Creating Labels

You override the Labels button on the Mailings tab for some of the same reasons as you override the Envelopes button. Generally, you have a need to control the output of labels on a system. When working with Ribbon applications, it's important not to interrupt the workflow or you won't gain all the benefits that come from the new method of creating user applications.

Working with labels is almost the same as envelopes. You still need to create an object and set any defaults, as shown here:

```
' Create the MailingLabel object.
Dim MyLabel As MailingLabel
Set MyLabel = Application.MailingLabel

' Set the output type.
MyLabel.DefaultLabelName = "30 Per Page"
```

Notice that the label isn't part of the document; it's part of the application, which does make for some differences in printing. You select the text to print just as you would for an envelope. However, when it comes time to print, simply highlighting the text isn't enough. The code must provide the text to print as input to the output method, as shown here:

```
' Output the label.
MyLabel.PrintOut Address:=CurrPane.Selection.Text
```

As with envelopes, you can print labels to a document instead of to the printer. However, the labels appear as a separate document, not as part of the original document. You use the `MyLabel.CreateNewDocument` method to accomplish the task. As with the printed output, you must provide text as input using the `Address` argument of the call.

Filling Out Forms

Forms fulfill a number of purposes — everything from requesting services to documenting work accomplished — in organizations. In fact, many organizations have so many forms that people create duplicates simply because they don't know the original exists. Your organization may have four or five versions of a single form right now, and the redundancy causes a wealth of problems. One method of overcoming this problem is to make forms instantly available so that users can peruse them and choose the forms they need. Figure 6-10 shows the Forms tab described in this section.

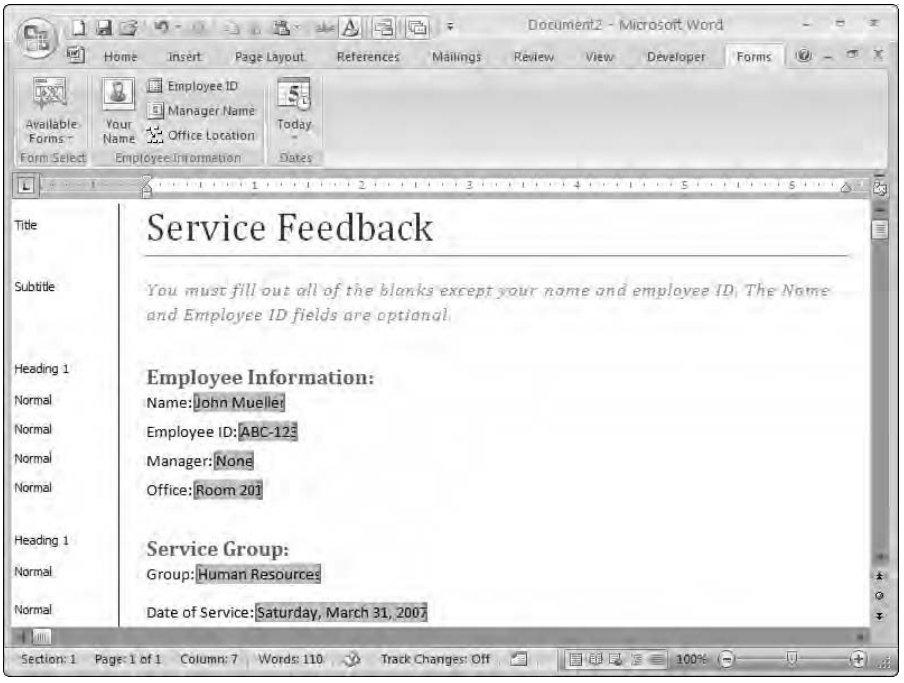


Figure 6-10: Filling out forms isn't a chore when the application does much of the work.

Notice that this example, like the letter example in this chapter, follows a left-to-right flow of events: The user selects a form, fills out the personal data, and then provides a date. Using this application produces three results:

- The user doesn't have to search for forms.
- The forms contain the correct user information (making it easier to find the user to ask questions).
- The forms have a better chance of containing complete information.

The sections that follow describe the major activities needed to make this application work. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Creating the forms

This example includes a new feature: The forms appear as part of a gallery, as shown in Figure 6-11. The amazing thing is that you don't have to rely on any odd programming techniques to achieve this goal. The application relies on some simple coding techniques and good directory organization to make this gallery possible. The coding technique also makes it possible to add forms to the gallery at any time without doing anything special. The application automatically updates the gallery to reflect any new form templates added to the list.

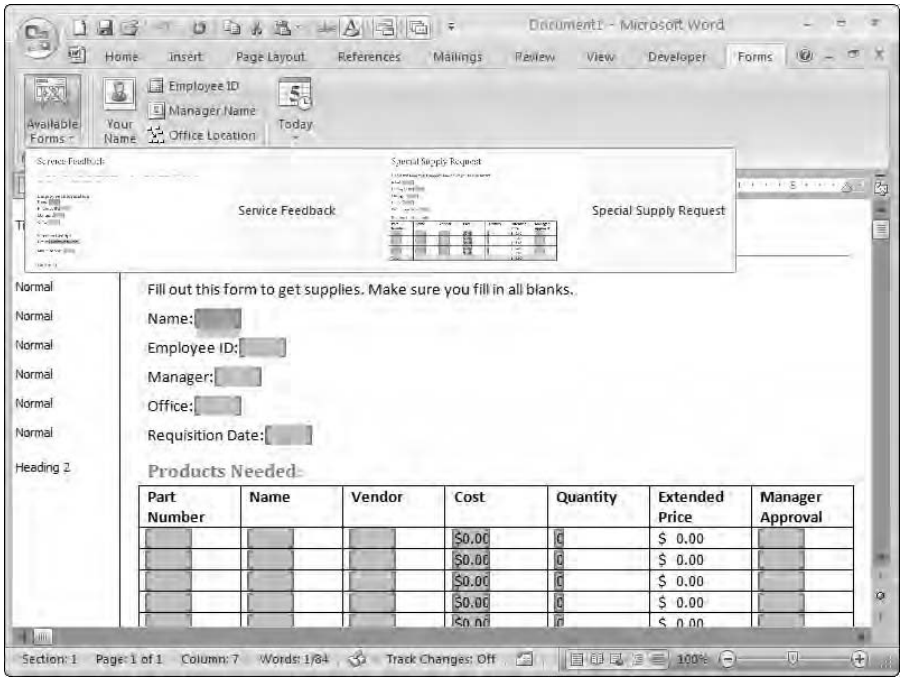


Figure 6-11: The user begins by choosing one of the forms from the Forms directory.

Creating a form in Word 2007 is similar to older versions of Word, but there are some important differences. The first difference is that the controls you add to a form appear in a special section of the Developer tab, as shown in Figure 6-12. You select the location of the control on-screen, and then choose one of the controls from the list.

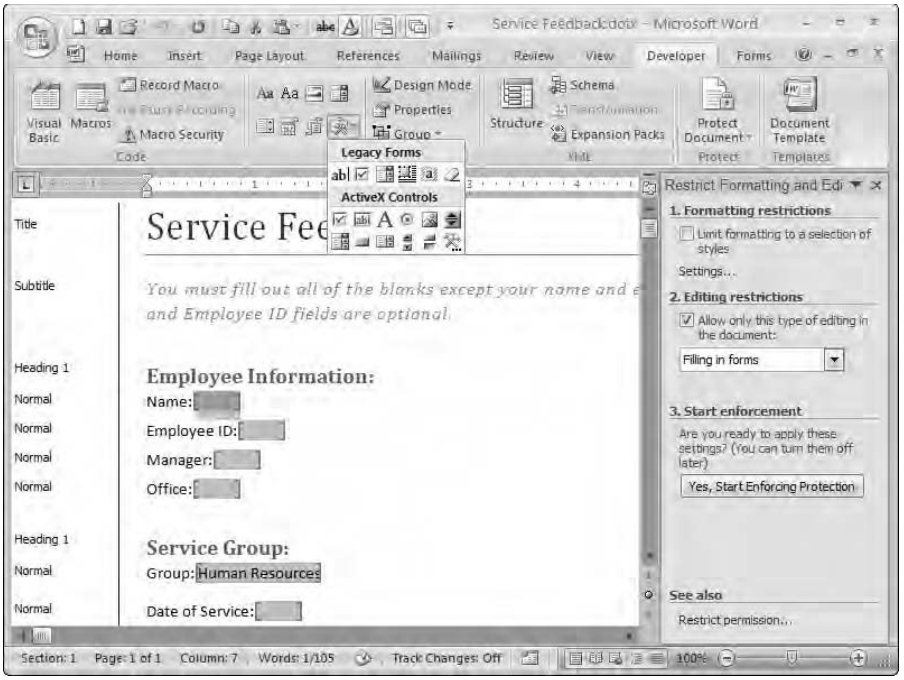


Figure 6-12: Choose the controls you want to use from the special Legacy Forms gallery.

The naming of controls is important when you want to create a Ribbon application with them. For example, every form in this example that has a date field calls it `ReqDate`. To set the field properties, right-click the control and choose Properties from the context menu. The Bookmark field of the control's Options dialog box determines the name you'll use for that control in your code. If you want to save time and effort, using consistent names is essential.

After you add and configure all of the controls on your form, you need to test it. You have to protect the form in order to activate the form fields. Microsoft has changed this functionality in Office 2007 as well. Use the following steps to protect a form:

1. **Click Protect Document on either the Developer or Review tab.**

   You see a list of protection options.

2. **Choose Restrict Formatting and Editing from the list.**

   Word displays a Restrict Formatting and Editing task pane like the one shown in Figure 6-12.

3. **Check the Allow Only This Type of Editing in the Document option.**

   Word activates the option and lets you choose the level of restriction.

4. **Select the Filling in Forms option.**

5. **Click Yes, Start Enforcing Protection.**

   Word displays the Start Enforcing Protection dialog box, as shown in Figure 6-13.

6. **Type passwords to protect your document, and then click OK.**

   Word activates the fields on the document so that you can test the form functionality.

Figure 6-13: Type passwords to protect your document from change.

You can remove document protection by clicking Stop Protection at the bottom of the Restrict Formatting and Editing task pane. Word asks you to provide the password you supplied earlier. After you enter the password, you can make changes to the document again.

Make sure you save your form as a template, and not as a document. If you save the form as a document, the application won't recognize it; even if you do open it, the form won't create a new version of itself. The form will act as a one-time fill-in, rather than as a means of creating multiple copies.

One of the tasks that you must perform for this example, in addition to creating a form, is to take a screenshot of the form. The example uses the screenshot to show how the form will appear when the user accesses it using the gallery. The screenshot must have the same filename as the form, but with the image extension. The example code relies on the Portable Network Graphic (PNG) file format. However, you can easily modify the code to meet any need.

Understanding differences between Content Controls and Legacy Controls

Anyone who worked with forms in older versions of Office knows about the controls that Microsoft provides for creating forms. You simply add the controls you want to the page, configure them, and then add some VBA code as needed to perform required tasks in the background. It's still possible to use these controls in Office 2007, but you must look for them in a different location. The controls that you're used to working with appear on the Developer tab in the Controls group in a small button called Legacy Controls. When you click this button, you'll see a small window containing two areas: Legacy Forms and ActiveX Controls. To use a control, simply place the cursor where you want the control to appear on the form and then click on the control as you normally would.

Microsoft also makes it possible to add new ActiveX controls. For example, the form in the "Filling Out Forms" section of the chapter includes a Calendar control. To add this control to your form, click the More Controls button that appears at the end of the list in the ActiveX Controls section. You'll see the More Controls dialog box, where you can choose the Calendar Control 12.0 entry and click OK. Adding the Calendar control to your form doesn't add it to the ActiveX Controls list, so you must perform this action every time you want to use the special control.

Content Controls are new to Office 2007 and Microsoft has given them a prominent location in the Controls Group. These controls don't work like the form controls you used in the past. In fact, these controls are part of a new technology called Building Blocks that you can read about at

`http://blogs.msdn.com/microsoft_office_word/archive/2006/11/21/building-blocks-part-i.aspx`

You can read more about how Building Blocks and Custom Controls work together at

`http://blogs.msdn.com/microsoft_office_word/archive/2006/11/22/inserting-and-swapping-building-blocks.aspx`

You can even create your own Building Blocks using information from the article at

`http://blogs.msdn.com/microsoft_office_word/archive/2007/01/03/creating-building-blocks.aspx`

The special editor found at

`http://www.codeplex.com/Wiki/View.aspx?ProjectName=dbe`

helps you manage and work with Custom Controls. Because this topic is so complex and is outside the scope of this book, you won't see any further references to either Building Blocks or Custom Controls in this book.

Selecting a form

The user's first task is selecting a form. Because the user could have multiple forms in progress at any given time, the application closes only the current document if it's obvious that the user has opened Word for this specific document. Consequently, the user will have to close any documents that aren't needed for a particular session.

Creating the physical presentation

The user sees the gallery control when choosing a form. However, you don't provide any content for this control at design time. That's because you can't predict which forms the user's machine will contain. All content appears at runtime. Consequently, the XML for this example contains a lot of callbacks, as shown in Listing 6-11.

Listing 6-11: Defining the Gallery Content

```
<gallery id="FormList"
         label="Available Forms"
         imageMso="BusinessFormWizard"
         size="large"
         columns="2" rows="2"
         itemHeight="100" itemWidth="180"
         getItemCount="GetItemCount"
         getItemID="GetItemID"
         getItemImage="GetItemImage"
         getItemLabel="GetItemLabel"
         onAction="ItemClicked"/>
```

You do need to consider a few configuration issues when creating the `<gallery>` element. Notice how the code uses the `columns` and `rows` attributes to control the number of templates the user sees at any time. You need to provide some amount of control or the gallery could very well fill the screen, making a selection more difficult, rather than easier.

It's also important to control the item size. Otherwise Word displays each of the templates at the full size you used to capture the image. You may have to spend some time figuring out the right size for the image. The user has to see what the form looks like, but you don't want the form full-size either. Maintaining the aspect ratio also helps in the recognition process. For example, if the screenshot height is 500 and the width is 900, and you reduce the height to 100 for display purposes, then you should set the `itemWidth` attribute to 180.

Creating the Templates variable and interacting with Word

Many of the callbacks rely on a global variable named `Templates` that contains a list of the templates for the current user. The example places the code to create this variable in the `OnLoad()` callback. Here's the code that makes everything else work:

```
// Obtain the user's path to templates.
string TemplatePath =
   Globals.ThisAddIn.GetTemplatePath();

// Build a list of form templates.
Templates = Directory.GetFiles(
   TemplatePath + @"\Forms", "*.DOTX");
```

The code begins by obtaining the location of the templates for the current user. You could use any other location on the hard drive as a starting point, but using the current user's folder ensures that Word can find the templates it needs. All of the forms reside in a special folder named Forms and have a `DOTX` file extension, so using the `GetFiles()` method makes it easy to fill the `Templates` array with a list of template filenames. Of course, the big question is where the `Globals.ThisAddIn.GetTemplatePath()` method resides.

You can't interact directly with Word from the `Ribbon1.CS` file; all you can do is interact with the Ribbon. Consequently, if you want the add-in to perform any useful work, you must call on public methods in the `ThisAddIn` class located in `ThisAddIn.CS`. Unfortunately, you can't access this class directly. If you try to create an instance of the class or create static members, you quickly find that you can't achieve any results, even if the code compiles. You always access the members of the `ThisAddIn` class using the `Globals` object as shown. When you type `Globals`, instantly you see a list of add-ins; after selecting a list of add-ins, you'll see the public members for that add-in. Here, the reason for the special emphasis is that Microsoft isn't particularly good about documenting these interactions — you almost need to know already that they exist, without Microsoft's help. The `GetTemplatePath()` method simply accesses Word and obtains the current path, as shown here:

```
public String GetTemplatePath()
{
   // Obtain the template path for Word and return it.
   return Application.NormalTemplate.Path;
}
```

Obtaining the template information for display

The remaining attributes define callbacks that Word needs to define the gallery content. Each image requires the `getItemID` and `getItemImage` attributes as a minimum (the `getItemLabel` attribute is optional, as are many of the other callbacks). The code must also implement the `getItemCount` attribute or the gallery won't know how many templates to display. Finally, you need to implement `onAction` to provide some means of reacting to user selections. Listing 6-12 shows the code required to implement the callbacks for this part of the example.

Listing 6-12: Providing the Gallery Items

```
public long GetItemCount(Office.IRibbonControl control)
{
   // Return the number of items based on the template
   // length.
   return Templates.Length;
```

```
}

public string GetItemID(Office.IRibbonControl control,
                        int index)
{
   // Obtain the template name.
   string ThisID =
      Path.GetFileNameWithoutExtension(
         Templates[index]);

   // Remove any extra spaces.
   ThisID = ThisID.Replace(" ", "");

   // Return the item id based on the template name.
   return ThisID;
}

public Bitmap GetItemImage(
   Office.IRibbonControl control, int index)
{
   // Obtain a pointer to the current template image.
   String TemplateLocation = Templates[index];
   TemplateLocation =
      TemplateLocation.Replace(".dotx", ".png");

   // Create a bitmap based on the image.
   Bitmap TemplateImage = new Bitmap(TemplateLocation);

   // Return the bitmap.
   return TemplateImage;
}

public string GetItemLabel(
   Office.IRibbonControl control, int index)
{
   // Return the template name.
   return
      Path.GetFileNameWithoutExtension(
         Templates[index]);
}

public void ItemClicked(Office.IRibbonControl control,
                        string selectedId,
                        int selectedIndex)
{
   // Use this path to supply information for the
   // template.
   Globals.ThisAddIn.CreateNewDocument(
      Templates[selectedIndex]);
}
```

The `GetItemCount()` method simply returns the length of a special variable named Templates that contains an array of template path strings. For right now, all you need to know is that the array shows where each template in the list resides. The "Creating the Templates variable and interacting with Word" section of the chapter describes how the application creates this variable. The `Templates.Length` property tells how many items the array contains, which therefore tells you how many elements the gallery has.

Every item in the gallery must have a unique ID. The `GetItemID()` method provides a unique ID based on the template filename. It's unlikely that two templates will have a name variation so close that it results in an ID collision. Notice how the code uses the `Path.GetFileNameWithoutExtension()` method to obtain just the filename, and then takes the spaces out of the filename to produce the ID.

Using the filename approach to creating IDs also makes it easier to debug your application. If a particular template causes problems, you can discover its name quickly and perform any required changes.

Remember that every one of the templates has an associated `.PNG` file that contains the screenshot of that template. The `GetItemImage()` method uses this feature to obtain an image for the gallery. The code begins by obtaining a template location, and then replaces the template file extension with the `PNG` extension. The code can then create a `Bitmap` for the image and return it to Word. Notice how none of the resources in this example require an absolute directory location — the code automatically compensates for differences in system setup.

The templates for this example have descriptive names; that way they're easier to locate if they require changes. In fact, there's no reason not to give them descriptive names. The `GetItemLabel()` method uses this template feature to create the labels for each gallery item. You can't count on the user seeing enough of the template to make a choice, so the distinctive name is a requirement to ensure the user makes the right selection at the outset.

The `ItemClicked()` method provides that last piece of the puzzle. When a user clicks on one of the templates, Word calls this callback with the selected item's ID and index. The code makes a call to the `Globals.ThisAddIn.CreateNewDocument()` method to actually create a new form, basing the form on the template location provided in the `Templates` variable. You'll always need to add this extra code because Office doesn't let you access the application directly through the Ribbon callback code.

Creating the new document with a particular template

In many respects, working with Word in Visual Studio looks like merely a different kind of VBA application. Of course, there are differences, but the essential coding tasks are the same. Listing 6-13 shows the code required to create a new form based on one of the templates you created earlier in this example.

Listing 6-13: Creating the new form based on the selected template

```
public void CreateNewDocument(string ThisTemplate)
{
   // Get rid of Document1 if the user has recently
   // opened Word and this is the first document
   // created.
   if (Application.ActiveDocument.Name == "Document1")
   {
      // Use wdPromptToSaveChanges to ensure the user
      // still has a chance to save any changes.  The
      // prompt won't appear if the user hasn't made
      // any changes to Document1.
      object SaveChanges =
      (object)Word.WdSaveOptions.wdPromptToSaveChanges;

      // Save the document in its original format.
      object OriginalFormat =
(object)Word.WdOriginalFormat.wdOriginalDocumentFormat;

      // Don't route the document anywhere.
      object RouteDocument = (object)false;

      // Close Document1.
      Application.ActiveDocument.Close(
         ref SaveChanges, ref OriginalFormat,
         ref RouteDocument);
   }

   // Create the new document.
   // Define the template to use to create the
   // document.
   object objTemplate = (object)ThisTemplate;

   // Don't create a new template based on the document
   // template.
   object objNewTemplate = (object)false;

   // Define the kind of document to create.
   object objDocumentType =
     (object)Word.WdNewDocumentType.wdNewBlankDocument;

   // Make sure the document is visible so the user can
   // see it.
   object objVisible = (object)true;

   // Create the document.
   Application.Documents.Add(ref objTemplate,
                             ref objNewTemplate,
                             ref objDocumentType,
                             ref objVisible);
}
```

The code begins by checking for `Document1`, the default name of the document that appears when you open Word. Notice that the code makes this check without creating any special objects. If you try to type `Application` and don't see the IntelliSense for the relation objects, you're probably working in the wrong file.

The code begins by creating some variables for the `Application.ActiveDocument.Close()` method. Unlike VBA, this code won't let you simply skip arguments. You must provide every required argument for the method calls, even if that means supplying a default value.

Most of the calls you'll make refer to the `object` class — they don't provide much in the way of information about the kind of input the call expects. The example doesn't take any chances with the user's data. If you have any doubt about the data type, look up the method call in the VBA documentation. Notice how the example uses type coercion to document the values for each argument. The code uses the actual VBA types, but then coerces them to the `object` type. VB.NET users will find that they need to do less work than C# developers because VB.NET performs some of the type coercion for you.

The example calls the `Application.ActiveDocument.Close()` method using the `ref` keyword in C#. VB.NET developers don't have to add the extra keyword. The reason you must include this keyword in C# is to allow Word to pass back information (it happens rarely — make sure you understand how the method call works before you assume that Word will provide any return values). Notice that the example uses `Word.WdSaveOptions.wdPromptToSaveChanges` to ensure that the user doesn't lose any data. Word won't display a dialog box unless the user has made changes to the document.

Creating the new document comes next. As with the `Application.ActiveDocument.Close()` method, the `Application.Documents.Add()` method requires not only that you pass arguments of the `object` type, but also that you pass them by reference. The `objTemplate` argument always contains the name of the template you want to create. The remaining arguments define how Word creates the document. You can choose to create the document as a template, to use a document type other than a blank document (such as an e-mail), and even to create invisible documents so you can work in the background.

Adding the user information

When the user reaches the Employee Information group, the Ribbon-handling code becomes a little boring. All that it does is pass the request to the `ThisAddIn` class, using code like this:

```
public void EmpName(Office.IRibbonControl control)
{
   // Insert the data into the field.
   Globals.ThisAddIn.InsertUserName();
}
```

Of course, if you fail to include any of these little glue-code calls, a feature of your application will fail to work. However, most of the activity occurs in the `ThisAddIn` class; there you begin by adding an Outlook reference. As with the letter-and-memo example, this example relies on a user entry within Outlook to supply user information that Word can't. To add the required reference, right-click References in Solution Explorer and choose Add Reference from the context menu to display the Add Reference dialog box, shown in Figure 6-14.

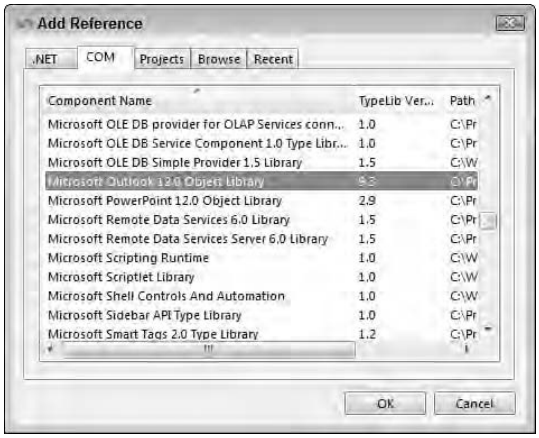


Figure 6-14: Add a reference to the Outlook object library.

Locate the Microsoft Outlook 12.0 Object Library entry, shown in Figure 6-14, and click OK. You'll see the Outlook reference added to the References folder of your project. Rather than type the long list of object names for Outlook in your code, however, make sure your `Imports` or `using` statement looks like the one shown here:

```
using Outlook = Microsoft.Office.Interop.Outlook;
```

After you have the required reference in place, you can begin creating the methods for handling the user entries. These entries fall into two categories. The username is a value that you can obtain from Word, so the example uses this information directly. In addition, you have to have the username before you can locate the remaining data in Outlook. The remaining entries all come from the user's address book entry in Outlook. Listing 6-14 shows examples of both kinds of code.

Listing 6-14: Obtaining User Information and Placing It in Fields

```
public void InsertUserName()
{
   // Obtain the username.
   string UserName = Application.UserName;

   // Define the field to fill.
   object SelectEmpReq = (object)"EmpName";

   // Insert the data into the field.
   Application.ActiveDocument.FormFields.get_Item(
      ref SelectEmpReq).Result = UserName;

   // Locate the user in the Outlook address book.
   Outlook._Application ThisOutlook =
      new Outlook.Application();
   foreach (Outlook.AddressEntry UserEntry in
    ThisOutlook.Session.AddressLists[1].AddressEntries)
   {
      if (UserEntry.Name == Application.UserName)
      {
         UserOutlookEntry =
            ThisOutlook.Session.GetRecipientFromID(
               UserEntry.ID);
         return;
      }
   }
}

public void InsertUserID()
{
   // Get the user identifier, which is stored in the
   // CustomerID field.
   string UserID =
 UserOutlookEntry.AddressEntry.GetContact().CustomerID;

   // Define the field to fill.
   object SelectEmpReq = (object)"EmpID";

   // Insert the data into the field.
   Application.ActiveDocument.FormFields.get_Item(
      ref SelectEmpReq).Result = UserID;
}
```

The code begins by obtaining the username using a technique that looks very similar to VBA. The code also creates the `SelectEmpReq` variable, which contains the name of the field to work with on the form. Notice that this code isn't designed to work with a specific form — it works with any form that contains a field with the right name. Consequently, you can create any number of forms and use the same code to modify them all — as long as you use consistent field names. This single application could end up working with hundreds of forms.

After the code creates the required variables, it calls the `Application.ActiveDocument.FormFields.get_Item()` method to access the required field. The `Result` property lets you obtain or modify the value of the field. This particular piece of code is very much different from VBA, which points out a problem with simply moving your VBA code to Visual Studio and assuming you can make minor changes to it.

At this point, the code begins locating the user's information in Outlook. As a point of interest, compare this code to the VBA version of the code in Listing 6-7. You'll notice that the C# code is actually more compact and it executes faster than the VBA equivalent. The code begins by using a `foreach` loop to look at the `UserEntry.Name` property for each user in the address book and compare it to the `Application.UserName` property value. When the code finds a match, it assigns the information to the `UserOutlookEntry` using the `ThisOutlook.Session.GetRecipientFromID()` method, just as you would in VBA. When this code executes, you may see the dialog box shown in Figure 6-4, just as you would when the same code executes in VBA.

Now that the code has access to the `UserOutlookEntry` variable, it can use a simple technique to place values from Outlook into the form fields. The code relies on the same `UserOutlookEntry.AddressEntry.GetContact()` method call that you do in VBA; then it assigns the resulting value to the appropriate field, as it does for the username.

Including a date

One of the problems that occurs most often with forms is that users enter the date incorrectly. People are forever forgetting the current date (or they don't have a calendar handy for looking up the precise date when they do remember the approximate date). The code in this section helps a user enter an accurate date using one of two methods. Either the user lets the application calculate the date (based on some static criterion such as the time interval) or the user uses a calendar to choose the date. Listing 6-15 shows examples of both scenarios.

Listing 6-15: Choosing the Correct Date

```
public void DateLastMonth(
   Office.IRibbonControl control)
{
   // Obtain the correct date.
   int DaysInMonth =
      DateTime.DaysInMonth(DateTime.Today.Year,
                           DateTime.Today.Month);
   DateTime TargetDate = DateTime.Today.Subtract(
```

(continued)

Listing 6-15 ***(continued)***

```
      new TimeSpan(DaysInMonth, 0, 0, 0));
   string InsertDate = TargetDate.ToShortDateString();

   // Insert the date into the field.
   Globals.ThisAddIn.InsertDateInEmpReqField(
      InsertDate);
}

public void DateCalendar(Office.IRibbonControl control)
{
   // Create and display the dialog box.
   Select_Date ChooseDate = new Select_Date();
   if (ChooseDate.ShowDialog() == DialogResult.OK)
   {

      // Obtain the correct date.
      string InsertDate =
ChooseDate.SelectDate.SelectionStart.ToShortDateString();

      // Insert the date into the field.
      Globals.ThisAddIn.InsertDateInEmpReqField(
         InsertDate);
   }
}
```

The `DateLastMonth()` callback shows an example of a date calculation (the most complex for the example). The code begins by creating a variable that contains the number of days in the current month. When the application subtracts this value from the current date, it receives the proper date from the previous month. The calculation relies on the `DateTime.Today.Subtract()` method, which requires a `TimeSpan` as input. As always, the code must call the `ThisAddIn` class to make the actual entry.

The calendar method relies on the form shown in Figure 6-15. The user chooses a date in the dialog box and clicks OK to make the date entry. The code obtains the information directly from the `MonthCalendar` control (make sure you make the control public). Notice that the code calls the same `ThisAddIn` class method as the `DateLastMonth()` callback to add the date to the form.

Figure 6-15: Select a date from the calendar when a static choice won't work.

The final method appears as part of the `ThisAddIn` class. The application uses the following method to add all of the dates to the form:

```
public void InsertDateInEmpReqField(String InsertDate)
{
   // Define the field to fill.
   object SelectEmpReq = (object)"ReqDate";

   // Insert the date into the field.
   Application.ActiveDocument.FormFields.get_Item(
      ref SelectEmpReq).Result = InsertDate;
}
```

As with the other form methods, the `InsertDateInEmpReqField()` method begins by creating a variable to represent the name of the field on the form. It then uses the `Application.ActiveDocument.FormFields.get_Item()` method to add the date to the form.

Chapter 7

Developing Business Applications for Excel

In This Chapter

- Understanding how to work with Excel
- Working with nonstandard equations
- Working with redundant calculations
- Performing data entry using forms

Most people use Excel for calculations of various kinds — everything from mundane accounting to complex scientific needs. A few people use Excel to create presentation graphics, and you'll even find a few people who use Excel for rudimentary word processing or database needs. In short, Excel is an extremely flexible application that helps you perform a wide range of tasks. The number of customizations you'll find for Excel is extraordinary. It would be very hard to say whether Word or Excel has the most number of applications written for it. One thing is certain: The variety of applications for Excel users does exceed that of Word.

Needless to say, the examples in this chapter provide you with only a basic view of some application types you can create in Excel. The examples in this chapter help you explore common applications, such as nonstandard equations and redundant calculations. You'll also find an example of using forms. The result is that you'll work with enough application types in this chapter to perform just about any business task you can imagine and a few of those extraordinary tasks that Excel users love to create.

This chapter also provides examples in both VBA and Visual Studio. The VBA examples focus on document-oriented tasks, such as working with a particular kind of calculation that would normally appear on a single worksheet. The Visual Studio examples create add-ins you could use anywhere in Excel. For example, nonstandard calculations can affect just about any worksheet you create, so you'll probably use Visual Studio to define them as add-ins.

Getting Started with Excel Applications

It's relatively easy to create a workflow scenario for most Office applications. For example, when you create a letter in Word, there's a definite process in play; you can use that process as the basis for your Ribbon tab. Likewise, Access developers can point to a definite process for adding and removing records. However, when it comes to Excel, you must consider both the process and the individual action scenario. When processes occur in Excel, they generally lend themselves to a workflow that you can exploit as a workflow-based tab. An example of a process in Excel is creating a chart or performing an analysis according to strict criteria.

Individual actions occur in Excel when the process required to obtain data isn't understood or the process is so complex that it isn't possible to create a workflow. Anyone performing experimental or what-if analysis falls into the first group because the data by its very nature isn't well understood. An example of an extremely complex workflow is one where the individual performs an analysis of a natural process or a living entity. Creating workflow might not even be worthwhile when the user performs the analysis only once.

Most business applications can rely on a workflow strategy. For example, entering accounting data or creating a database both require strict processes. On the other hand, most scientific applications perform individual actions that focus on the task, rather than the workflow. Because science is experimental in nature, you'll often find that you create individual buttons to perform tasks such as working with particular equations.

Understanding Excel and VBA

You'll generally use VBA for processes that relate to a specific action such as calculating mortgage interest or creating a chart based on this week's data. VBA works well when you create a single workbook to hold data from an external source. The VBA application can perform the required analysis, generate a report, and output graphics based on the ever-changing output of the external source. However, VBA isn't always the optimal choice. For example, it's often easier to work with Web services and databases using Visual Studio. In this case, the essential criterion is whether the user will work with only one workbook.

Understanding Excel and Visual Studio

Visual Studio is the only choice when it comes to certain kinds of Excel applications. For example, if you create a tab containing specialized applications for your company, you probably want to create an add-in, rather than rely on VBA to perform the task. The specialized applications are then available wherever you need them, rather than as part of a single workbook.

Combining VBA and Visual Studio in Excel applications

Excel is one of the few Office applications where you might actually combine VBA and Visual Studio to create a complete solution. The Visual Studio portion of the application can include any features, such as equation or graphics support, that must appear with every application you create. The VBA portion of the application can enforce workflow requirements. Combining general features with specific implementations of those features can help you create extremely flexible applications with a minimum of code.

Creating a Nonstandard Equations Tab

Many businesses use nonstandard equations. For example, insurance companies use specialized equations to determine rates. The equations aren't standardized across the industry; in fact, some equations are proprietary because each insurance company feels that its own equation gives it a business advantage. These equations are so closely guarded that it's unlikely you'll find them online.

Another kind of nonstandard equation is one that doesn't see common use except within a particular industry. For example, circuit analysis is a common need for the electronics industry, but your bank probably won't require any of the equations used for such analysis. You can read an article on statistical circuit analysis with Excel at

`http://www.maxim-ic.com/appnotes.cfm/appnote_number/2878/`

A third kind of nonstandard equation is one that provides some type of public-use benefit. For example, you might need an Excel worksheet to help you compare the benefits of one car loan over another, or help you determine the mortgage rate on your home (you can find a loan-calculator template for Excel at `http://office.microsoft.com/en-us/templates/TC062062871033.aspx`). Some equations help you perform tasks such as determining the interest on a credit card or selecting the best repayment plan for your current credit cards. A special worksheet might help you convert units of measure or determine a physical characteristic, such as percentage of body fat. You can find an entire assortment of templates at

`http://office.microsoft.com/en-us/templates/FX100595491033.aspx`

All three of these examples define times when you might want to create a nonstandard equations tab for Excel. However, just creating nonstandard equations doesn't really use the Ribbon all that well. You could create such

an application quite easily using previous versions of Excel. The focus of the Ribbon is to make more efficient workflows possible. Consequently, this example demonstrates how you can use multiple versions of the same Ribbon tab to facilitate a workflow within Excel.

This example demonstrates a considerable number of techniques; the chapter simply can't hold all the source code required to create it. The sections that follow do provide you with complete information about all the essential techniques for working with this example. They also show how to modify it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `NonStandardEquation` folder.

Creating a starting element

Figure 7-1 shows one form of the example application; the first thing most people are going to notice is that the Start group apparently doesn't contain any controls. Many developers are used to the idea that every feature in an application must perform a task. In this case, the Start group simply serves as a starting point. It's a reminder, and the application uses three labels to provide the information. The whole idea behind a workflow is ensuring the user can follow a process. In this case, the first step of the process is simply to place the cursor in the correct cell.

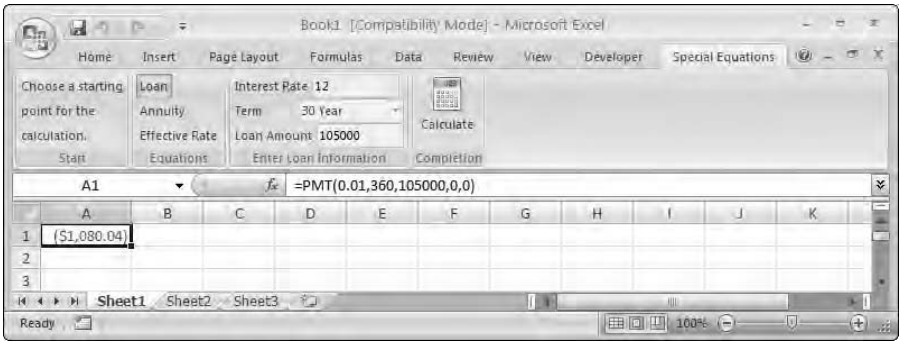


Figure 7-1: Calculating a loan is no longer an error-prone process.

Choosing the correct equation

The list of equations in Figure 7-1 isn't extensive, but it does provide a simple view of what you might do with this application. You could also choose to use the split button or gallery approaches used with the Word examples in Chapter 6. No matter which approach you use, you have to provide some means for the user to select an equation. The toggle-button approach used for this example works very well in many cases. The example even highlights the selected equation. Of course, you have to coordinate all three toggle buttons — which means performing a little extra coding. The linkage begins with the XML shown here:

```
<toggleButton id="Loan"
              label="Loan"
              onAction="SetupLoan"
              getPressed="SelectedEquation"/>
```

When the user initially clicks one of the toggle buttons, the code calls the methods pointed to by the `onAction` attribute. In this case, the code calls the `SetupLoan()` method that appears in Listing 7-1.

Listing 7-1: Choosing an Equation Type

```
public void SetupLoan(Office.IRibbonControl control,
                      bool pressed)
{
   // Set the calculation type.
   CalcType = "Loan";

   // Set the pressed state.
   pressed = true;

   // Invalidate the entire Ribbon.
   ribbon.Invalidate();
}
```

The `CalcType` variable is a global string that tracks the equation in use. This particular variable appears in quite a few places because it determines many of the application actions and even the final appearance of the tab.

Setting `pressed` to `true` changes the condition of the target control, but it doesn't do anything for the other controls on the tab. This application changes quite a few of the controls when the user chooses a different equation, so simply setting the target button won't work. That's why the code ends with a call to `ribbon.Invalidate()` to invalidate the entire Ribbon. You could use individual calls to controls, but there are too many of them to change in this example — invalidating the Ribbon as a whole works far better.

Now that the system has pressed the control, Excel calls the method pointed to by the `getPressed` attribute. In this case, all three toggle buttons use the same method because the method performs essentially the same task for all of them. You can see the `SelectedEquation()` method in Listing 7-2.

Listing 7-2: Setting the Equation Control State

```
public bool SelectedEquation(
   Office.IRibbonControl control)
{
   // Determine the pressed state based on the current
   // equation.
   switch (CalcType)
   {
      case "Loan":
         if (control.Id == "Loan")
            return true;
         else
            return false;
      case "Annuity":
         if (control.Id == "Annuity")
            return true;
         else
            return false;
      case "Effective Rate":
         if (control.Id == "EffectiveRate")
            return true;
         else
            return false;
      default:
         return false;
   }
}
```

The code uses a `switch` to choose a calculation type based on the content of `CalcType`. Once the code chooses a particular `case`, it uses an `if` statement to determine whether to pass `true` or `false` to the caller. The return value determines whether Excel presses the control.

Defining the multiple Ribbon elements

This example relies on a single Ribbon tab, but multiple designs give the application the appearance of providing multiple functionality on a single tab. When the user selects a particular equation, the tab content changes to reflect the needs of that equation. The user doesn't even worry about the worksheet. The controls across the Ribbon let the user move from left to right to solve a specific problem. The worksheet shows the result of the entries that the user makes as part of the workflow. Figure 7-1 shows an example of one of the equations with the data entered into it.

The three equations are relatively simple financial equations that Excel actually provides on the Formulas tab as PMT, FV, and EFFECT. The problem is that using the Formulas tab requires a good knowledge of the equations, and there's a very good chance that a user could type incorrect data. Figure 7-2 shows what happens when you choose the PMT option. Excel displays a dialog box containing a number of potentially confusing entries. For example, to provide the entry in the Rate field, you must calculate the interest rate per period first. A 12% interest rate for monthly loan payments translates to 0.12 / 12 or 0.01 per period. However, the dialog box doesn't make the required information apparent.

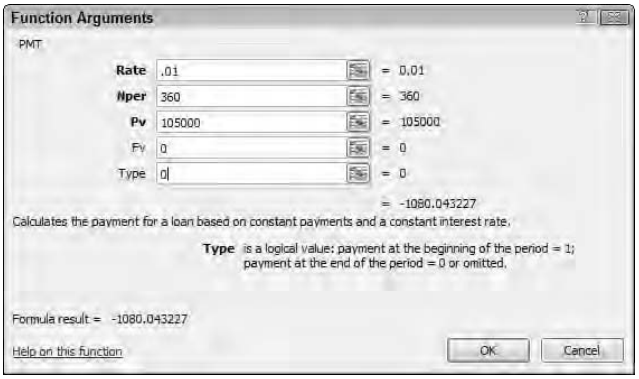


Figure 7-2: Even when Excel provides an equation, you have to know how to use it.

Compare the process for using PMT in Figure 7-1 against that in Figure 7-2. Using the custom tab is clearly less error prone and enforces company policies without much effort on the part of the user.

Of course, creating the tab shown in Figure 7-1 requires some effort on the part of the developer to create a design that works for all of the equations. The PMT function requires the periodic interest rate, number of periods, and loan amount as required input for this application. The optional future value amount is 0 since the borrower achieves a 0 value when the loan reaches maturity. The type value is also 0 since the bank calculates the loan amount at the end of the accounting period. Consequently, the PMT function requires three inputs, as shown in Figure 7-1. Notice that the number of periods appears as a drop-down list box so the user can choose only loan-term lengths that the bank actually supports.

The FV function requires the most entries on the tab. These entries appear in Figure 7-3. In this case, the user must provide an interest rate, the amount of time for the annuity, the initial deposit the client makes, and the amount of money the client pays into the annuity each month. Although this calculation uses one additional control, you don't have to get rid of any existing controls from the PMT function.

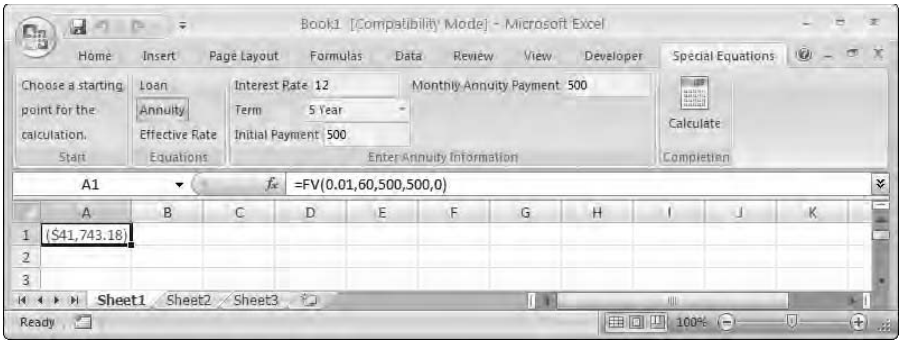


Figure 7-3: The FV function requires an additional field to work properly in this application.

The least control-intensive function is EFFECT. All you really need is a single field to hold the current interest rate. You might think that the various configurations would be problematic, but you control it using standard Ribbon features. Listing 7-3 shows the XML required to present the different view of the tab.

Listing 7-3: Providing Flexible Data Entry

```
<group id="DataEntry" getLabel="GetDataEntryLabel">
   <editBox id="Rate"
            label="Interest Rate"
            onChange="GetRateText"/>
   <dropDown id="Term"
             label="Term"
             getVisible="TermVisible"
             getItemCount="TermCount"
             getItemID="TermItemID"
             getItemLabel="TermItemLabel"
             onAction="GetSelectedTerm"/>
   <editBox id="Payment"
            label="Initial Payment"
            getVisible="PaymentVisible"
            onChange="GetPaymentText"/>
   <editBox id="Amount"
            getLabel="AmountLabel"
            getVisible="AmountVisible"
            onChange="GetAmountText"/>
</group>
```

Notice how the application uses the various attributes to see each of the controls as needed. The `Rate` control appears in every application, so it has a default label and doesn't require the `getVisible` attribute. All the other controls do have a `getVisible` attribute that determines whether the control appears on the Ribbon, depending on the current equation selection. In all cases, the `getVisible` attribute method uses a simple selection method (such as the one shown in Listing 7-4).

Listing 7-4: Showing or Hiding Controls

```
public bool TermVisible(Office.IRibbonControl control)
{
   // The application doesn't use this field for the
   // effective rate calculation.
   if (CalcType == "Effective Rate")
      return false;
   else
      return true;
}
```

Notice that this choice, like many others, depends on the `CalcType` variable. You can keep your code simple by maintaining a single variable of this kind to control the application's on-screen appearance.

Compare Figures 7-1 and 7-3 and you'll notice that group and `Amount` control labels change to differentiate the equation types. In both cases, the example relies on the `getLabel` attribute to provide the required connectivity. Listing 7-5 shows the code used for this example.

Listing 7-5: Changing the Group and Control Labels

```
public string GetDataEntryLabel(
   Office.IRibbonControl control)
{
   // Determine the group label by the kind of
   // calculation selected.
   switch (CalcType)
   {
      case "Loan":
         return "Enter Loan Information";
      case "Annuity":
         return "Enter Annuity Information";
      case "Effective Rate":
         return "Enter Effective Rate Information";
      default:
         return "Not Implemented";
   }
}

public string AmountLabel(
   Office.IRibbonControl control)
{
   // Determine the amount label by the kind of
   // calculation selected.  Since the effective rate
   // calculation doesn't use this control, the
   // application doesn't supply a label for it.
   switch (CalcType)
   {
```

(continued)

Listing 7-5 ***(continued)***

```
        case "Loan":
           return "Loan Amount";
        case "Annuity":
           return "Monthly Annuity Payment";
        default:
           return "Not Implemented";
     }
}
```

You'll notice that many of the `switch` statements used in this application include a `default` option. Using this approach is going to save you a lot of debugging time at some point — because someone will almost certainly come behind you and add other options to your application. In some cases, the changes result in a broken application unless you include a `default` option to handle unanticipated modifications.

In both cases, the application returns a string that redefines how the control appears on-screen. You might wonder why the application doesn't include a `ribbon.Invalidate()` call in this and other callbacks. In many cases, careful placement of the `ribbon.Invalidate()` call means that you have to include it only once in the application. Because clicking one of the equation options begins all the changes noted in this section, placing the `ribbon.Invalidate()` call there makes the most sense.

Always use the `ribbon.Invalidate()` call with care. If you include it more than once in an application, you might cause significant problems. The application could end up in a long loop of repetitive updates. Theoretically, the application could even crash — or cause the user to terminate the application when the Ribbon-updating process takes too much time.

Obtaining the data entered in the Ribbon

The Ribbon doesn't allow any direct interaction, so you can't obtain the information the user types into the Ribbon controls directly. The answer to this problem, as with many other problems, is to create a callback. Listing 7-3 shows the two answers needed in this case. Most controls provide an `onChange` attribute that you can use to detect changes in the control data. One of the exceptions to this rule is the drop-down list box, which requires that you use the `onAction` attribute to detect a change in selection. Listing 7-6 shows a typical example of an `onChange` implementation and the `onAction` implementation for the `Term` control.

Listing 7-6: Obtaining Data from the Ribbon

```
public void GetRateText(Office.IRibbonControl control,
                        string text)
{
   // Save the input value of the text.
   Rate = Int32.Parse(text);
}

public void GetSelectedTerm(
   Office.IRibbonControl control,
   string selectedId, int selectedIndex)
{
   // Store the default value.
   Term = 0;

   // Save the terms for loans.
   if (CalcType == "Loan")
      switch (selectedIndex)
      {
         case 0:
            Term = 10;
            break;
         case 1:
            Term = 15;
            break;
         case 2:
            Term = 20;
            break;
         case 3:
            Term = 30;
            break;
      }

   // Save the terms for annuities.
   if (CalcType == "Annuity")
      switch (selectedIndex)
      {
         case 0:
            Term = 5;
            break;
         case 1:
            Term = 7;
            break;
         case 2:
            Term = 10;
            break;
         case 3:
```

(continued)

Listing 7-6 *(continued)*

```
            Term = 15;
            break;
         case 4:
            Term = 20;
            break;
      }
}
```

Converting the numeric input of the edit boxes is relatively easy because the `Int32.Parse()` method does all that work for you. The best part is that this method call also safeguards your application to some extent. If a user puts anything other than a number in the edit box, the `Int32.Parse()` method outputs a 0. In short, if someone tries to input a script or other nasty into your application, the application will simply see a 0.

The drop-down list box requires considerably more processing. In this case, the code must interpret the information based on the selections available for each of the equations. Although a drop-down list box prevents the user from inputting inaccurate data, you can cause problems for yourself if you don't check the interpretation code carefully.

Performing the calculation

It's finally time to perform the calculation. The application requires two pieces of code: The first reacts to the click of Calculate on the Ribbon; the second converts all the data into a string, and then places the string in Excel before performing the calculation. Listing 7-7 shows the code that reacts to user input.

Listing 7-7: Reacting to the Calculate Button

```
public void Calculate(Office.IRibbonControl control)
{
   // Choose a calculation and call it.
   switch (CalcType)
   {
      case "Loan":
         Globals.ThisAddIn.CalculatePMT(Rate, Term,
                                        Amount);
         break;
      case "Annuity":
         Globals.ThisAddIn.CalculateFV(
            Rate, Term, Payment, Amount);
         break;
      case "Effective Rate":
         Globals.ThisAddIn.CalculateEFFECT(Rate);
         break;
   }
}
```

The call is simple, at this point, because the application has already placed the required data in global variables. All the code has to do is call the appropriate add-in function and provide the required input. The actual calculation requires a bit more effort because you have to construct the same string that the Excel formula commands create. Listing 7-8 shows the code for the loan calculation.

Listing 7-8: Performing the Actual Calculation

```
// Calculate a loan amount.
public void CalculatePMT(double Rate, int NPer, int PV)
{
   // Compute the rate.
   double PeriodicRate = (Rate / 100) / 12;

   // Compute the number of periods.
   int Periods = NPer * 12;

   // Perform the actual calculation.
   Application.ActiveWindow.ActiveCell.Cells[1, 1] =
      (object)"=PMT(" + PeriodicRate.ToString() + "," +
      Periods.ToString() + "," + PV.ToString() +
      ",0,0)";
   Application.ActiveWindow.ActiveCell.Calculate();
}
```

Notice how the code performs a subtle, yet essential, conversion of the `int Rate` value to a `double Rate` value using a change in argument type. If you don't perform this conversion, the first calculation in the example fails because Visual Studio insists on performing it as an integer value.

The first calculation converts the annual percentage rate as an integer into a periodic rate that the equation can actually use to perform the calculation. The second calculation converts the years provided by the user into the required number of periods for the equation. In both cases, the application assumes a monthly payment schedule, which is the default for most banks (even though many now offer a host of alternatives that dizzy the mind of the most savvy financial expert).

Creating the string comes next. The application uses the `Application.ActiveWindow.ActiveCell.Cells[1, 1]` property to place the string in the upper-left corner of the worksheet selection. Notice that it's working with `ActiveCell.Cells`, which is the currently selected cell range. Even if the user selects a range of cells, the answer will appear in the upper-left corner of that range. The actual equation looks something like this (the numbers will vary based on what the user provides as input):

```
=PMT(0.01,360,105000,0,0)
```

Simply adding the text to the worksheet doesn't assure that the calculation takes place. The code ends by calling `Application.ActiveWindow.ActiveCell.Calculate()` to perform the calculation.

The effective rate calculation requires special formatting, as will many other calculations you perform. Excel defaults to the General number format. However, the example formats the cells to the Percentage number format with four places of accuracy after the decimal point for easier reading. Here's the code you need to perform the formatting:

```
// Format the cell.
Excel.Range ThisRange;
ThisRange = (Excel.Range)Application.Cells[(object)1,
                                           (object)1];
ThisRange.NumberFormat = "0.0000%";
```

Although you can modify the value of a cell directly by using the `Application.ActiveWindow.ActiveCell.Cells[1, 1]` property, you can't use this property to change the formatting. To perform this task, and any other formatting-related tasks such as changing the font, you must create an `Excel.Range` object, use it to obtain the cell you want to modify, and then change the formatting information. Notice how you must coerce the type of the output of `Application.Cells` to make it a range (Excel returns a simple `object`).

The number formats aren't always easy to discover because the VBA help file doesn't list them. The easiest way to obtain the required `NumberFormat` (or other formatting) value is to open a copy of Excel, format a cell the way you want it, and then use a simple VBA macro to return the required strings. You obtain the `NumberFormat` string using the following VBA macro:

```
Public Sub TestNumberFormat()
   Dim ThisRange As Range
   Set ThisRange = Application.Cells(1, 1)
   MsgBox ThisRange.NumberFormat
End Sub
```

Performing Redundant Calculations

Redundant calculations are those you must set up and perform more than once in a particular worksheet. For example, you might want to compute a range of mortgage values, rather than a single value, to provide a client with a number of options. Using the example in the "Creating a Nonstandard Equations Tab" section of the chapter, you could guide a user into performing the same calculation multiple times without any problem. However, this approach is going to drive a seasoned user nuts — not to mention waste

considerable time, which the Ribbon is supposed to prevent. Consequently, you need something different to make this application work a little better.

The sections that follow provide you with complete information about all the essential techniques for working with this example and modifying it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `RedundantCalculations` folder.

Defining the problem solution

You could solve the problem by performing the same calculation with different values multiple times, using a number of techniques, including (say) adding ranges to the current tab. It's important to remember, though, that the Ribbon is supposed to promote a workflow, so adding these ranges directly to the tab is going to prove confusing when the user really does need to perform a single calculation. A better idea is to make the range calculation part of a dialog box and to use a dialog-box launcher to access it. Consequently, the new version of the nonstandard equations application has one minor change, as shown in Figure 7-4.

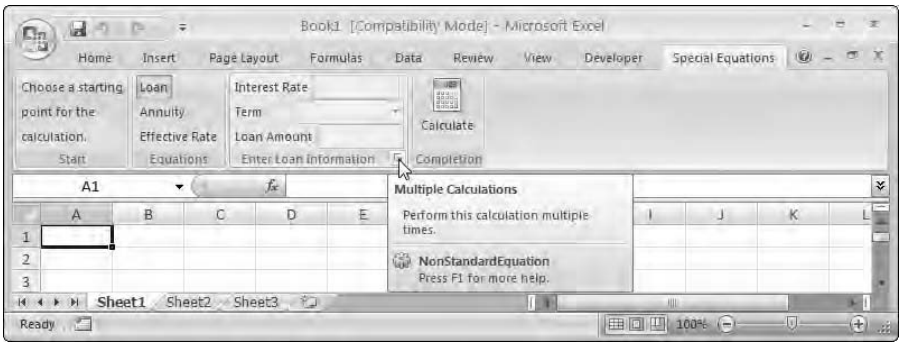


Figure 7-4: Adding functionality sometimes means using a dialog-box launcher.

The XML for this addition is relatively simple. All you need is the standard dialog-box launcher information added to the `DataEntry` group, as shown here:

```
<dialogBoxLauncher>
   <button id="RedundantCalcsLaunch"
           screentip="Multiple Calculations"
           supertip="Perform this calculation multiple
                     times."
           onAction="DisplayRedundantCalc"/>
</dialogBoxLauncher>
```

Implementing the dialog-box launcher is another story. Here are the criteria for implementing this example:

- You must create three different dialog boxes, one for each equation.
- The code must distinguish between the different equations.
- Any dialog boxes used to interact with the user will require linkage with the information on the Ribbon; the Ribbon controls will likewise require updates from the dialog boxes.
- The code must make multiple calls to the required `ThisAddIn` method, once for each cell in the range.
- Depending on your requirements, you might need to provide additional information, such as column and row headings, to make the output usable.

Designing the dialog boxes

The dialog box design is important. You must weigh the advantages of user flexibility against the complexities of coding a usable design. The example specifically limits the user to two range choices, which means that you can display the output as a table. Of course, this means you also have to decide which two ranges the user is likely to need. The example uses the simple design shown in Figure 7-5 for the loan calculation. Notice that the two range values are those that the user is most likely to need — the interest rate and the payoff period. The user is going to know the loan amount at the outset, so keeping this value fixed probably won't present a problem.

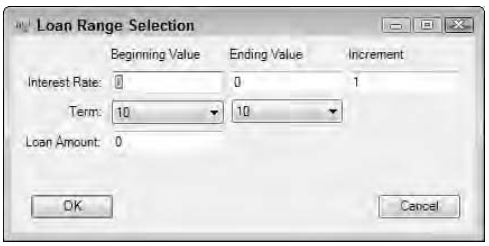


Figure 7-5: The choice of range variables is relatively easy with the loan calculation.

The annuity dialog box, shown in Figure 7-6, is different from the loan dialog box because you have to make some hard choices when designing it. The user may actually need four ranges in this case. In addition to the interest and deposit terms, the user may also require the payment amount and even the initial deposit. If you provide ranges for three variables (the periodic payment amount is the likely candidate), you could present the output as a series of tables. However, when you start working with four variables, it becomes difficult to provide the user with data at a glance because you'll likely need to start using multiple worksheets to display the data, which means the user is going to become easily confused.

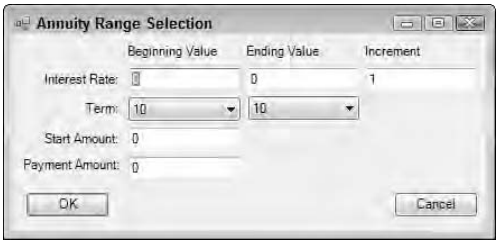


Figure 7-6: The annuity dialog box presents tough tradeoffs that you'll have to consider.

The purpose of moving an application to the Ribbon is to reduce complexity. If you end up adding more complexity due to some unfortunate design choices (even for the expert user), then the user doesn't receive any benefit from using the Ribbon. In general, your design of dialog boxes should move in the same direction as the Ribbon as a whole — look for ways to reduce the number of decisions the user must make to use the application.

Of the three calculations, creating a range dialog box for the effective rate calculation is the easiest. In this case, all you need to supply is a single range for the interest rate, so the dialog box design is almost trivial.

Creating the calculation code

Performing the required calculations is essentially the same whether you perform one or many calculations. In fact, the differences are subtle. Listing 7-9 shows the code for this example. Compare this code to Listing 7-8.

Listing 7-9: Performing Multiple Calculations

```
// Calculate a loan amount and include positional data.
public void CalculatePMT(double Rate, int NPer, int PV,
                         int X, int Y)
{
   // Compute the rate.
   double PeriodicRate = (Rate / 100) / 12;

   // Compute the number of periods.
   int Periods = NPer * 12;

   // Perform the actual calculation.
   Application.ActiveWindow.ActiveCell.Cells[X, Y] =
      (object)"=PMT(" + PeriodicRate.ToString() + "," +
      Periods.ToString() + "," +
      PV.ToString() + ",0,0)";
   Application.ActiveWindow.ActiveCell.Calculate();
}
```

The most noticeable difference between the two methods is that this section's version requires two additional positioning arguments. The version in Listing 7-8 assumes that you want to put the output in a particular place. This version lets you move the output to any location on the worksheet.

The `Application.ActiveWindow.ActiveCell.Cells[X, Y]` property performs the positioning. Instead of providing specific numbers, this version uses the input arguments. Other than this small change, the code works precisely as before. Even though the change is subtle, it's important; you'll probably want to maintain both versions of the method to make coding easier.

Defining linkages to existing data

When you start creating dialog boxes for your Ribbon application, you need to create linkage between each of the dialog boxes and the appropriate Ribbon controls. Otherwise the two elements will get out of sync and you'll find your application displays erroneous data. The linkage requires three elements:

- Getting information from the Ribbon
- Setting information on the Ribbon
- Defining the required callbacks in XML

You've already seen the first element in the list. Listing 7-6 shows how to get text from the Ribbon and use it in your application. Listing 7-10 shows the third element. Normally, you'll create the XML for the linkage before you create the code to ensure that you have the methods defined properly.

Listing 7-10: Creating XML Linkages

```
<editBox id="Rate"
         label="Interest Rate"
         onChange="GetRateText"
         getText="SetRateText"/>
<dropDown id="Term"
          label="Term"
          getVisible="TermVisible"
          getItemCount="TermCount"
          getItemID="TermItemID"
          getItemLabel="TermItemLabel"
          onAction="GetSelectedTerm"
          getSelectedItemIndex="SetSelectedTerm"/>
```

The attribute you use to implement the linkage depends on the control type. Most controls use the `getText` attribute, which actually draws information from your application and displays it in the control (an `editBox`, in this case). Some controls require that you use an alternative attribute, such as the `getSelectedItemIndex` attribute shown in Listing 7-10.

The example already has global variables defined for the various controls on the Ribbon. For example, `Rate` contains the value of the `Rate` edit box on the Ribbon. All you need to do, in many cases, is convert the value and output it to the Ribbon as a string, as shown in Listing 7-11.

Listing 7-11: Setting Data Values on the Ribbon

```
public string SetRateText(
   Office.IRibbonControl control)
{
   // Return the current value of the Rate variable.
   return Rate.ToString();
}

public int SetSelectedTerm(
   Office.IRibbonControl control)
{
   // Set the term for loans.
   if (CalcType == "Loan")
      switch (Term)
      {
         case 10:
            return 0;
         case 15:
            return 1;
         case 20:
            return 2;
         case 30:
            return 3;
```

(continued)

Listing 7-11 *(continued)*

```
      }

   // Set the term for annuities.
   if (CalcType == "Annuity")
      switch (Term)
      {
         case 5:
            return 0;
         case 7:
            return 1;
         case 10:
            return 2;
         case 15:
            return 3;
         case 20:
            return 4;
      }

   // Provide a default return value.
   return 0;
}
```

All that the `SetRateText()` method requires is a single line of code to perform the required work. To make this code work, keep the `Rate` variable updated and invalidate the associated control as needed.

The `SetSelectedTerm()` method shows the work required to keep the drop-down list box synchronized. You can't simply base a value on the current value of `Term` and send it to the control. Since the value can vary depending on the drop-down list box items, you can't perform a direct translation. Using `switch` to perform the task works well.

Performing the redundant calculations

The `<dialogBoxLauncher>` has only one `onAction` attribute, so the starting point for any calculation is the method that this attribute points to, `DisplayRedundantCalc`. Of course, the calculations for each of the equations are different, so you need to provide some means of calling these unique implementations. Listing 7-12 shows the code used for this purpose.

Listing 7-12: Choosing a Redundant Calculation Procedure

```
public void DisplayRedundantCalc(
   Office.IRibbonControl control)
{
   // Select the correct procedure.
```

```
   switch (CalcType)
   {
      case "Loan":
         PerformLoanRangeCalc();
         break;
      case "Annuity":
         PerformAnnuityRangeCalc();
         break;
      case "Effective Rate":
         PerformEffectiveRateRangeCalc();
         break;
   }
}
```

As you can see, the code is a simple switching network for a more complex process. The actual calculation loop takes up much of the additional code for this example; it can become significantly more complex when you allow for additional ranges. Listing 7-13 shows a typical example of the redundant calculation loop.

Listing 7-13: Performing the Redundant Calculation Loop

```
private void PerformLoanRangeCalc()
{
   // Create the dialog box.
   LoanRangeSelection ThisSelection =
      new LoanRangeSelection();

   // Add the existing variables to it.
   ThisSelection.txtIntBeg.Text = Rate.ToString();
   ThisSelection.txtIntEnd.Text = Rate.ToString();
   ThisSelection.txtIntInc.Text = "1";
   ThisSelection.cbTermBeg.Text = Term.ToString();
   ThisSelection.cbTermEnd.Text = Term.ToString();
   ThisSelection.txtLoanAmt.Text = Amount.ToString();

   // Display the dialog box and process the data if
   // the user clicks OK.
   if (ThisSelection.ShowDialog() == DialogResult.OK)
   {
      // Convert the data values to integers.
      Rate = Int32.Parse(ThisSelection.txtIntBeg.Text);
      Term = Int32.Parse(ThisSelection.cbTermBeg.Text);
      Amount =
         Int32.Parse(ThisSelection.txtLoanAmt.Text);

      // Create local variables to hold the calculation
      // data.
      Int32 EndRate =
         Int32.Parse(ThisSelection.txtIntEnd.Text);
```

(continued)

Listing 7-13 ***(continued)***

```
        Int32 IncRate =
           Int32.Parse(ThisSelection.txtIntInc.Text);
        Int32 EndTerm =
           Int32.Parse(ThisSelection.cbTermEnd.Text);

        // Update the values on the Ribbon.
        ribbon.InvalidateControl("Rate");
        ribbon.InvalidateControl("Term");
        ribbon.InvalidateControl("Amount");

        // Add the initial heading.
        Globals.ThisAddIn.SetHeading("Interest", 1, 1);

        // Perform the calculations.
        for (int i = Rate; i <= EndRate; i += IncRate)
        {
           // Calculate the X and Y positioning value.
           int X = i + 2 - Rate;
           int Y = 2;

           // Print the interest rate.
           Globals.ThisAddIn.SetHeading(
              i.ToString() + "%", X, 1);

           // Use a series of if statements to determine
           // the year settings.
           if ((Term == 10) && (EndTerm >= 10))
           {
              // Perform the calculation.
              Globals.ThisAddIn.CalculatePMT(
                 i, 10, Amount, X, Y);

              // Print the heading.
              Globals.ThisAddIn.SetHeading(
                 "10 Year", 1, Y);

              // Increment Y if we've used it.
              Y++;
           }

           if ((Term <= 15) && (EndTerm >= 15))
           {
              // Perform the calculation.
              Globals.ThisAddIn.CalculatePMT(
                 i, 15, Amount, X, Y);

              // Print the heading.
              Globals.ThisAddIn.SetHeading(
                 "15 Year", 1, Y);

              // Increment Y if we've used it.
              Y++;
```

```
            }

            if ((Term <= 20) && (EndTerm >= 20))
            {
               // Perform the calculation.
               Globals.ThisAddIn.CalculatePMT(
                  i, 20, Amount, X, Y);

               // Print the heading.
               Globals.ThisAddIn.SetHeading(
                  "20 Year", 1, Y);

               // Increment Y if we've used it.
               Y++;
            }

            if ((Term <= 30) && (EndTerm >= 30))
            {
               // Perform the calculation.
               Globals.ThisAddIn.CalculatePMT(
                  i, 30, Amount, X, Y);

               // Print the heading.
               Globals.ThisAddIn.SetHeading(
                  "30 Year", 1, Y);

               // Increment Y if we've used it.
               Y++;
            }
         }
      }
   }
```

The code begins by creating the requisite dialog box. Figures 7-5 and 7-6 show two examples of these dialog boxes. However, the code doesn't display the dialog boxes immediately. Instead, it fills the various dialog boxes with the values the user needs to create a range with the least amount of work. After the dialog box contains the required information, the code displays the dialog box using the `ShowDialog()` method and then tests for the `DialogResult` value. When the user clicks OK, the code begins processing the data the user provides.

The first task is to update the global variables so that the code can update the Ribbon later. The code also needs to create local variables to hold the range data. The ranges don't appear on the Ribbon, so you don't have to maintain global variables for them. At this point, the code calls `ribbon.InvalidateControl()` to update the Ribbon. The code updates the individual Ribbon controls, rather than the Ribbon as a whole, because there are only three controls to consider. Using this approach makes the application work more efficiently and reduces the potential for problems such as screen blinking.

The calculations occur in what amounts to a double loop. The `Rate` loop is easy to see because it relies on a standard `for` structure. The second loop is harder to see because you have to unroll it as a series of `if` statements. Remember that `Term` has no standard increment, so you can't use a `for` structure to handle it. Instead, the code must use a series of `if` statements to test the potential `Term` and `EndTerm` values.

Of course, you also don't know where the range will begin or end, so you can't assign definite values to the positioning variables either. The code begins with a starting point and then updates the variables for each range entry. When the code encounters one of the `Term` values in the specified range, it calls `Globals.ThisAddIn.CalculatePMT()` to perform the calculation and display the result on-screen.

The code has one potential inefficiency: Because you don't know where the range starts or ends, you can't determine where to place a heading to describe the range entry. Consequently, the code makes several calls to `Globals.ThisAddIn.SetHeading()` to set the heading information. This call is very short and won't have much impact on the application's performance, but you should consider avoiding it when you can. The "Considering the data identification requirements" section of the chapter describes the `Globals.ThisAddIn.SetHeading()` method in detail.

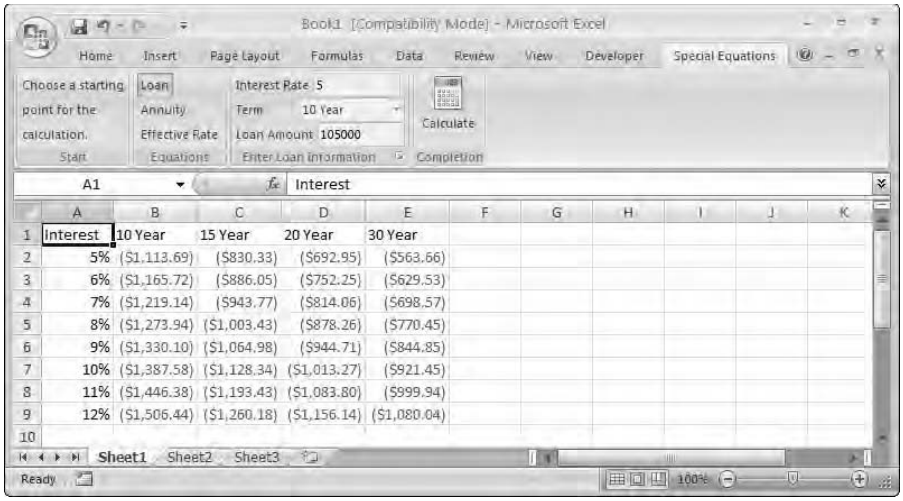
Considering the data identification requirements

The final piece of this example provides a means of adding headings to the output. It's easy to argue that a single calculation doesn't require any headings. However, to make the data meaningful when you perform multiple calculations, you must provide headers. The code in Listing 7-14 performs this task.

Listing 7-14: Creating a Worksheet Heading

```
public void SetHeading(
   String Heading, Int32 X, Int32 Y)
{
   // Add the requested heading.
   Application.ActiveWindow.ActiveCell.Cells[X, Y] =
      (object)Heading;
}
```

Now that all the pieces are in place, you can run the code, create a range, and see the output. Figure 7-7 shows an example of the output for a loan with a starting amount of $105,000, payoff times between 10 and 30 years, and interest rates between 5% and 12%. Creating the output this way is definitely easier than using the individual entries in the Formulas tab (see Figure 7-2 for an example of the dialog box used for calculations on the Formulas tab).



	A	B	C	D	E
1	Interest	10 Year	15 Year	20 Year	30 Year
2	5%	($1,113.69)	($830.33)	($692.95)	($563.66)
3	6%	($1,165.72)	($886.05)	($752.25)	($629.53)
4	7%	($1,219.14)	($943.77)	($814.06)	($698.57)
5	8%	($1,273.94)	($1,003.43)	($878.26)	($770.45)
6	9%	($1,330.10)	($1,064.98)	($944.71)	($844.85)
7	10%	($1,387.58)	($1,128.34)	($1,013.27)	($921.45)
8	11%	($1,446.38)	($1,193.43)	($1,083.80)	($999.94)
9	12%	($1,506.44)	($1,260.18)	($1,156.14)	($1,080.04)
10					

Figure 7-7: The output from this example produces a table of values based on the input ranges.

Automating Data Entry with Forms

Forms are one of the more common business applications for both Word and Excel. While forms in Word tend to have a universal appeal (such as a form you use to requisition supplies), those in Excel often have more limited uses. Consequently, you might find that you want to create an Excel form as a template or even as a document, rather than as an add-in. (You may want to contrast the use of forms in Word with the use of forms in Excel by reviewing the example in the "Filling Out Forms" section of Chapter 6.)

The sections that follow provide you with complete information about all the essential techniques for working with this example and modifying it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `ExcelForm.xltm` file.

Creating the form

Many people don't realize that you can create forms with Excel (they don't have the popularity of the Word variety because they're not as easy to create in an eye-pleasing form) or that you can create Excel templates. The form controls for Excel don't follow precisely the same conventions as those for Word. You'll still find them on the Developer tab, but they appear as part of the Insert split button, as shown in Figure 7-8. If you need to choose additional controls, click the More Controls button in the lower-right corner of the ActiveX Controls section to display the More Controls dialog box.

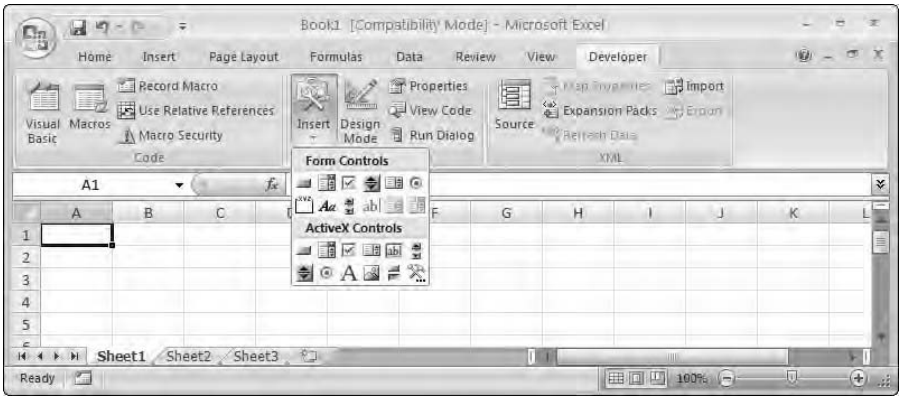


Figure 7-8: Excel provides form controls on the Developer tab.

As with Word, you must protect the form before you can work with the controls. However, unlike Word, Microsoft places the Excel document-protection feature on the Review tab, as shown in Figure 7-9. The following steps tell how to protect the document:

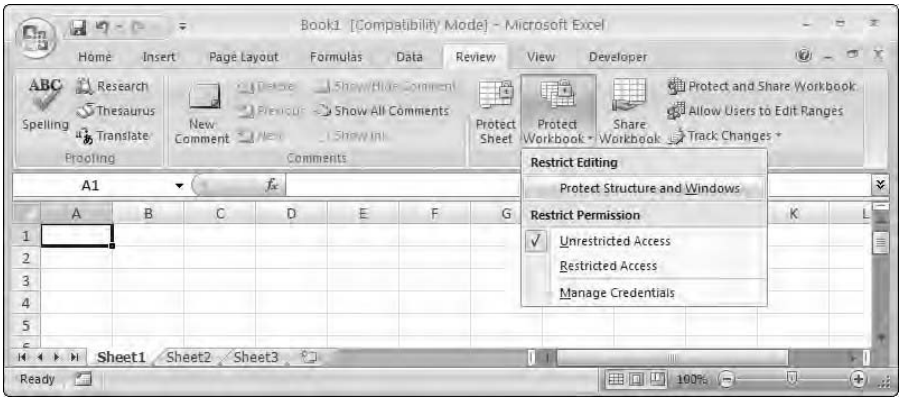


Figure 7-9: Excel places document protection on the Review tab.

1. **Select the Review tab.**
2. **Click Protect Workbook and choose Protect Structure and Windows.**

   You see the Protect Structure and Windows dialog box, shown in Figure 7-10.

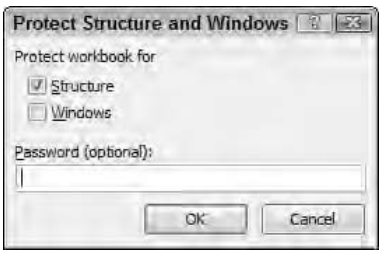


Figure 7-10: Protect the document to enable the fields that you place on the form.

3. **Check the protection options you want to use.**

   Make sure you choose structure protection to enable the form controls (this is the default option). Protect the windows when you also want to prevent the user from modifying the worksheet layout.

4. **(Optional) Type a password for the document and click OK.**

   Excel enables protection in the document and activates the controls you've placed on the form. If you don't provide a password, anyone can remove the protection you've put in place.

You'll need to decide whether you want to create your form as a document or a template. The example uses a template. When creating a template, you need to use the `XLTM` extension. Excel automatically places files in the appropriate folder:

In Windows XP:

```
\Documents and Settings\<User Name>\Application Data\Microsoft\Templates
```

In Windows Vista:

```
\Users\<User Name>\AppData\Roaming\Microsoft\Templates
```

Excel doesn't provide a direct option to save a document as a template, though, so you'll need to use the following procedure to save it:

1. **Choose Office Menu➪Save As➪Other Formats.**

   You see the Save As dialog box, shown in Figure 7-11.

2. **Select the Excel Macro-Enabled Template (*.xltm) option from the Save as Type drop-down list box.**

3. **Optionally, right-click any blank area of the file listing area and choose New➪Folder to create a new folder for the template.**

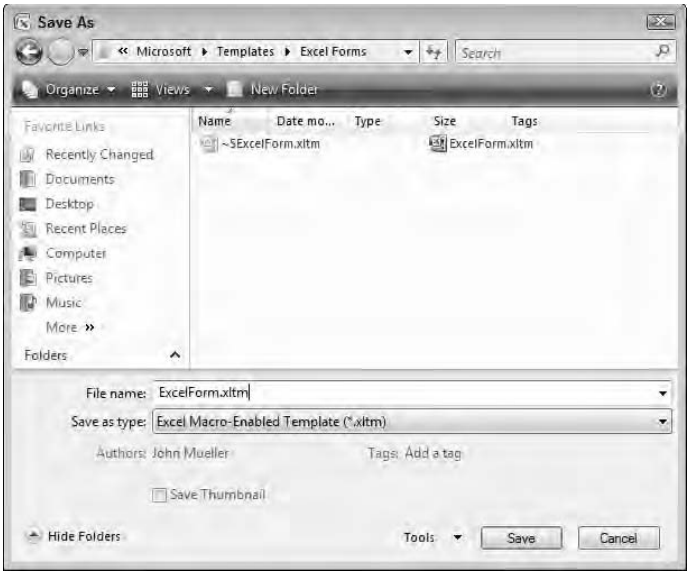


Figure 7-11: Use the options in the Save As dialog box to save your worksheet as a template.

4. **Check the Save Thumbnail option.**

   Excel saves a picture of the form. Saving a thumbnail lets you see the template when you want to create a new document based on it later.

5. **Type a name for your template and click Save.**

   Excel saves a copy of the template.

When you want to use the template later, choose Office Menu➪New to display the New Workbook dialog box. Select the My Templates option and you'll see the template on the Excel Forms tab of the resulting New dialog box. If you've saved a thumbnail for your template, you'll see a picture of the form in the Preview window, as shown in Figure 7-12.

Remove the grid lines from your worksheet when you're working with forms. Doing so provides a nicer appearance and also makes it easier for the user to work with the form. Figure 7-13 shows the Employee Expense Report worksheet used for this example. You'll find the gridlines option on the Page Layout tab within the Sheet Options group. Clear the check next to the View option in the Gridlines area of the group.

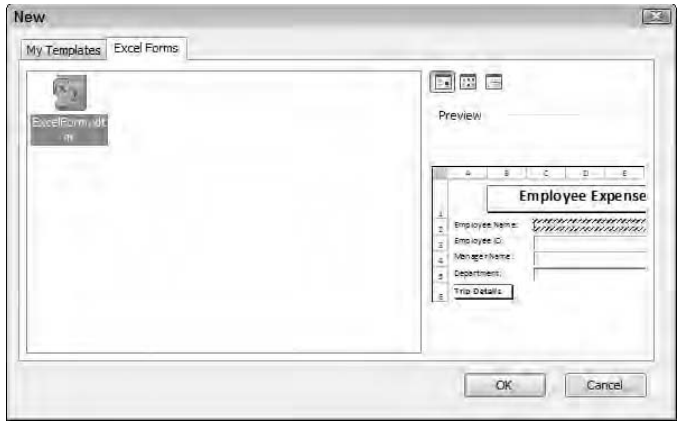


Figure 7-12: Save a thumbnail for your form so that you can see how it appears later.

Figure 7-13: Removing the gridlines makes this form easier to see and use.

If you're experiencing problems in working with the fields, then choose the Developer tab and click Design Mode. The Design Mode toggle button must remain depressed while you modify features such as field properties. Excel automatically resets the Design Mode button when you perform certain tasks. For example, when you make changes to your VBA code, Excel normally takes the application out of Design mode.

Adding the Ribbon code

The Ribbon code is relatively simple for this example, despite the number of items you see in Figure 7-13. Every group has an identifier and label associated with it. The employee buttons use the standard button setup shown here:

```
<button id="EmpName"
        label="Your Name"
        size="large"
        onAction="OAEmpName"
        imageMso="ContactPictureMenu" />
<button id="EmpID"
        label="Your ID"
        onAction="OAEmpID" />
```

The only button that includes an image is the large Your Name button. You'll sometimes find that a descriptive label provides everything that a user needs and that the images simply end up using space. That's the case with this example. Images normally help more with tasks that the user might not perform every day or that require decisions by less skilled workers. Each of the check boxes also uses a relatively simple setup, as shown here:

```
<checkBox id="TripPurpose"
          label="Trip Purpose"
          onAction="OATripPurpose"
          getPressed="GPTripPurpose" />
```

Notice that you must provide both `onAction` and `getPressed` attributes. Otherwise the check box won't provide the correct status information when the user reopens the file. Even though the Ribbon code is simple in this case, it still performs everything expected from a Ribbon application, including reducing complexity and maintaining a workflow.

Performing content sleight-of-hand

This example may seem as if it should perform almost magical manipulations of the form. Yet, the entire form is always in place with this example. All the fields you see in Figure 7-13 are always available. However, the person filling out the form sees only the fields checked in the Ribbon. This content sleight-of-hand is easy to perform when you follow a few rules.

The example uses a simple technique where each row is dedicated to one kind of data. You make each row large enough to hold just one set of controls. For example, the Employee Name field shown in Figure 7-13 is 18 pixels high. The row containing it is 20 pixels high, which allows one pixel on each side of the control. By hiding the row, you can hide the field within the row. Yes, the

field still exists, but the user can't see it and it won't print out either. As far as the user is concerned, the field doesn't exist. The field must reside entirely within the row to make this feature work.

In addition to sizing and placement considerations, you must also change one of the field properties. Right-click the field and choose Format Control from the context menu. Select the Properties tab. You'll see the Format Control dialog box, as shown in Figure 7-14. Select the Move and Size with Cells option or the field won't hide automatically when you hide the row.

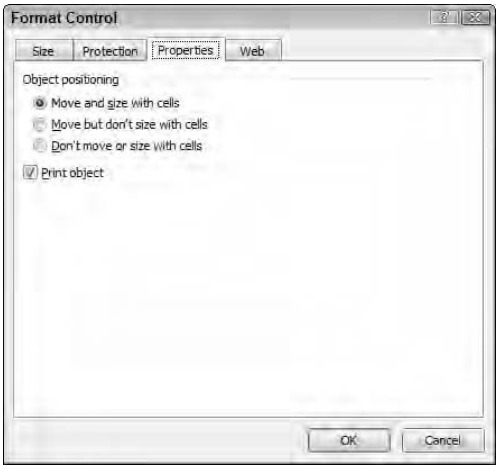


Figure 7-14: Setting the field properly ensures the user only sees it when needed.

Now that you have the required form structure in place, you can write code to hide and show the various form elements as needed. The `onAction` attribute Subs perform the task of showing and hiding the fields, as shown in Listing 7-15.

Listing 7-15: Showing and Hiding the Form Fields

```
'Callback for TripPurpose onAction
Sub OATripPurpose(control As IRibbonControl, _
                  pressed As Boolean)

   'Change the screen appearance.
   If pressed Then
      Rows("7:7").Select
      Selection.EntireRow.Hidden = False
   Else
      Rows("7:7").Select
      Selection.EntireRow.Hidden = True
   End If
End Sub
```

The code simply selects the appropriate row in this case, and then it hides or displays the entire row. Because you set the controls to change with the rows, showing or hiding a row also shows or hides the associated controls.

To maintain the illusion of disappearing and reappearing fields, the application must include some type of status information. You could save custom data to perform this task, but the easiest way to accomplish the goal is to ask Excel to track the information for you. The `getPressed` attribute Subs request the information from Excel, as shown in Listing 7-16.

Listing 7-16: Returning the Control Pressed State

```
'Callback for TripPurpose getPressed
Sub GPTripPurpose(control As IRibbonControl, _
                  ByRef returnedVal)

   'Detect the row condition.
   If Rows("7:7").Hidden Then
      returnedVal = False
   Else
      returnedVal = True
   End If
End Sub
```

Excel saves the status of each row in the worksheet. If the row holding the field controls is hidden, so are the controls. A hidden control means that the associated Ribbon check box isn't checked, so when `Rows("7:7").Hidden` is true, the application returns `False`.

Creating the worksheet linkage

The controls you create on the worksheet won't do much because you can't work with the values they contain. You must add a cell range to the `LinkedCell` property of each control. Every control must have a unique link. You assign the property by right-clicking the control and choosing Properties to display the Properties dialog box, shown in Figure 7-15. (The screenshot shows the field highlighted.) Notice that you provide a letter column and a numeric row as input.

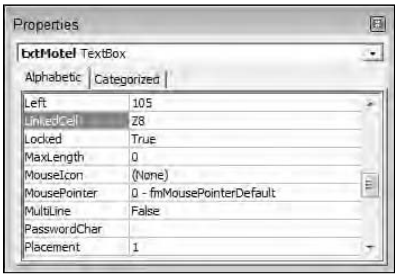


Figure 7-15: Provide a LinkedCell field input so the control can communicate with VBA.

After you assign a unique cell location for every control (it pays to put these cells out of sight), it helps to assign a named range to each cell. Although this may seem like a lot of extra work, using the named ranges makes your code considerably easier to read. You set a named range by clicking the cell in question and choosing Name a Range from the context menu. Excel displays the New Name dialog box, as shown in Figure 7-16. As a matter of convenience and self-documentation, you'll want to give the named range the same name as the control.

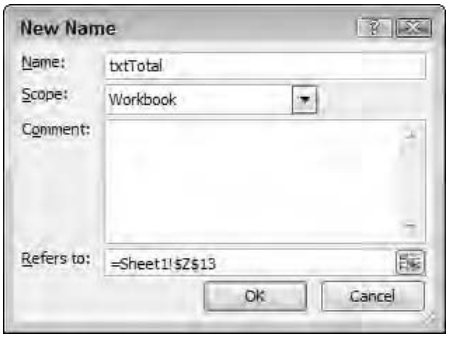


Figure 7-16: Add named ranges for each of the control links.

Defining the employee selections

Most of the fields on the form require explicit input from the user. For example, you can't automatically generate a tome on the purpose for the trip. However, you can provide some essential information and ensure that it's correct. The employee information is an essential part of the expense report, and you need it in order to ask the employee questions later. Consequently, automating this input can have a big impact on the validity of essential form data.

Before you can add the employee information, you need a source. In many companies, you could access the information from a database. This example uses Outlook as a source of information because many people have it installed on their systems; it would be a viable information source in most organizations. Before you can use Outlook in Excel, you must add a reference to the appropriate Outlook library. Choose Tools➪References to display the References – VBAProject dialog box, shown in Figure 7-17. Check the Microsoft Outlook 12.0 Object Library entry and click OK to complete the reference.

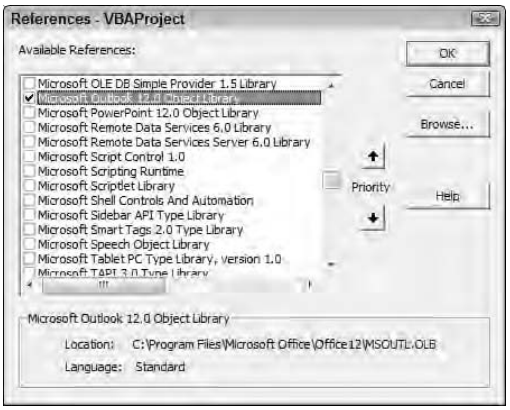


Figure 7-17: Adding an Outlook library reference to your application.

Excel does provide one critical piece of information for the employee — the employee name. With this piece of information, you can look up any other required information in Outlook. Listing 7-17 shows the code required to complete two typical employee inputs (the rest of the inputs use the same technique as the `OAEmpID()` method).

Listing 7-17: Obtaining and Displaying Employee Information

```
'Callback for EmpName onAction
Sub OAEmpName(control As IRibbonControl)

   'Obtain a reference to the worksheet.
   Dim ThisSheet As Worksheet
   Set ThisSheet = Application.ActiveSheet

   'Place the username in the appropriate linked cell.
   ThisSheet.Range("txtEmpName") = Application.UserName

   'Look up the sender in Outlook. Begin with the
   'first name and go from there.
   Dim CheckSender As AddressEntry
   Set CheckSender = _
      Outlook.Application.Session.AddressLists. _
```

```
      Item(1).AddressEntries.GetFirst

   ' Check the name.
   If CheckSender.Name = Application.UserName Then
      Exit Sub
   End If

   ' If this isn't the right user, keep searching.
   For Each CheckSender In _
      Outlook.Application.Session.AddressLists. _
        Item(1).AddressEntries

      ' Check the entry name.
      If CheckSender.Name = Application.UserName Then
         Exit For
      End If
   Next

   ' Determine whether we have an ID to use.
   If Not CheckSender Is Nothing Then
      Set ThisSender = _
Outlook.Session.GetRecipientFromID(CheckSender.ID)
   End If
End Sub

'Callback for EmpID onAction
Sub OAEmpID(control As IRibbonControl)

   'Obtain a reference to the worksheet.
   Dim ThisSheet As Worksheet
   Set ThisSheet = Application.ActiveSheet

   'Get the required information from Outlook
   'and place it in the appropriate cell.
   ThisSheet.Range("txtEmpID") = _
      ThisSender.AddressEntry.GetContact.CustomerID
End Sub
```

The code begins by creating a reference to the current worksheet. It then assigns the username (as defined by the application settings) to the cell range assigned to the `txtEmpName` textbox. The linkage you put in place earlier automatically transfers the value in this cell to the control, so the user sees the information on-screen as if you had placed it there directly.

The code creates an Outlook `AddressEntry` object next and uses it to begin searching for the user's name. The code initially sets the collection pointer to the beginning of the list using `AddressEntries.GetFirst()`. If the first name isn't the one that contains the user information, then the code uses a `For Each` loop to continue looking for it. When the code finally locates the user, it stores the user's information in a global `Recipient` object, `ThisSender`. Notice how the code uses the `Outlook.Session.GetRecipientFromID()` method to perform the user data conversion.

At this point, all the hard work is done. The `OAEmpID Sub` begins by creating a worksheet reference. It then uses the same technique to store information for the `txtEmpID` control as the `OAEmpName Sub`. All the code has to do is obtain the correct information from the `ThisSender.AddressEntry.GetContact` property. In this case, the application stores the `CustomerID` property value, which contains the employee ID.

Calculating the cost

The application can also ensure that the user provides an accurate total. Clicking the Calculate Total button calls the `OACalculate Sub` shown in Listing 7-18.

Listing 7-18: Performing the Final Expense Report Calculation

```
'Callback for Calculate onAction
Sub OACalculate(control As IRibbonControl)

   ' Holds the total cost.
   Dim Total As Single
   Total = 0

   'Obtain a reference to the worksheet.
   Dim ThisSheet As Worksheet
   Set ThisSheet = Application.ActiveSheet

   'Perform the calculations based on the rows that
   'appear on screen.
   If Rows("11:11").Hidden = False Then
      Total = CSng(ThisSheet.Range("txtMotel").Value)
   End If
   If Rows("12:12").Hidden = False Then
      Total = Total + _
         CSng(ThisSheet.Range("txtCarRental").Value)
   End If
   If Rows("13:13").Hidden = False Then
      Total = Total + _
         CSng(ThisSheet.Range("txtMeals").Value)
   End If
   If Rows("15:15").Hidden = False Then
      Total = Total + _
         CSng(ThisSheet.Range("txtParkingTolls").Value)
   End If
   If Rows("16:16").Hidden = False Then
      Total = Total + _
         CSng(_
            ThisSheet.Range("txtMiscellaneous").Value)
```

```
    End If

    'Display the total on screen.
    ThisSheet.Range("txtTotal") = Total
End Sub
```

One of the problems with the expense report is that the user might encounter some expenses and not others. The user could make mistakes and add entries into fields that aren't required. If the user makes those fields invisible later, the code could come up with the wrong total if it simply looks for a value in the field. Consequently, the code has to provide a means of adding only the visible fields to the total. To perform this task, it relies on the `Rows(<Range>).Hidden` property.

Whenever the code encounters a visible field, it obtains the current value of the associated cell for that field on the worksheet. It's important to remember that you can't retrieve the required information directly. The code simply adds the value to `Total` and then moves on to the next field.

The technique used in this example has an interesting protection factor: The user can enter only numbers into the associated fields and obtain anything in return. The `CSng()` function used to convert the text into a number automatically ignores any non-numeric value. If the user has entered text or special characters, the `CSng()` function outputs a 0.

The code ends by placing the total in the cell associated with the `txtTotal` control. The user sees the output as you'd expect.

Chapter 8

Developing Business Applications for Access

In This Chapter

- Understanding how to work with Access
- Defining an XML file for Access
- Letting Access see the changes you make to the Ribbon
- Installing the sample database for the examples
- Working with temporary tables or filtered results

Access is a different kind of application when it comes to the Ribbon — but then, you use Access differently from other Office applications to create new applications. The reason: Access isn't self-contained. After all, a data source that doesn't communicate with the outside world isn't much good. In fact, you can use Access without ever opening the Access application. For example, you can interact with Access just fine by creating a Word application to provide the user interaction — all of the database activity can occur in the background and the user might not even know it happens.

You can create standalone applications for Access. For example, you might design a virtual "Rolodex" to maintain your contact list. Certainly you don't have to create such an application in Word or Excel. A standalone application works fine for data entry and contact access. Should you decide to access the information from Word to type a letter or the data from Excel to perform a calculation, you can access the database as you normally would. Most businesses do use the combination approach with users relying on the user interface that makes the most sense for their particular need. In almost every case, the database administrator (DBA) will require some level of access to the database through the Access interface for management tasks. Users often require access to the interface for generating reports as well.

The method of interacting with the Ribbon in Access is similar to other Office applications, but it's also significantly different. For example, you can't use the Custom UI Editor to make changes to the XML for the Ribbon — you use an entirely different technique in Access. On the other hand, you still use the

same XML entries to create entries on the Access Ribbon. In addition, all the same techniques you normally use to create callbacks in other Office products also work with Access. Consequently, you'll find some old and some new techniques in this chapter.

Getting Started with Access Applications

The most important consideration for Access applications is that you can't create an add-in for Access. You won't find a project for Access in Visual Studio. Any application you create must interact with a specific Access database or group of databases. For example, you can't create a new Ribbon tab for Access that promotes specific formatting procedures for comments across all databases. If you want that kind of functionality, you need to install it for every Access database (including every new database). Consequently, every application presented in this chapter relies on VBA to interact with the Ribbon.

Working with Access also requires that you perform the Ribbon changes using special techniques. The changes don't appear in the `CustomUI` folder as they do in other Office applications. In fact, you have three options available to you for loading Ribbon changes in Access (all of which appear in the "Loading the Ribbon Changes" section of the chapter):

- `USysRibbons` table
- Standard user table
- External XML file

The approach you use to work with the Ribbon depends on how you use the database. In some cases, it's actually counterproductive to create custom Ribbon applications for Access. For example, if the mode of access is external, then you probably won't want to invest the time in creating a custom Ribbon application. The biggest benefit of creating a Ribbon application is to provide access to a series of product features in a workflow-based configuration. In some cases, you'll also want to include access to custom applications or less accessible features through a Ribbon application, but the DBA managing your Access database may not receive much of a benefit from the improved access, which means the additional programming time that the Ribbon requires is lost.

Creating the XML File

Access doesn't let you use the CustomUI Editor to create the XML file you need. Consequently, unless you're willing to trust your fate to typing the required XML in Notepad, you'll need a different XML editor for Access. You could use the XML editing capabilities in Visual Studio, but that seems like

overkill unless you use Visual Studio for other tasks. It's also possible to download one of the Express products for free from

```
http://msdn.microsoft.com/vstudio/express/
```

You'll have to register to obtain the products, but at the time of writing, Microsoft isn't charging for them. Any of the Express products will work fine.

A better alternative for creating the Access files is an XML editor because the download is small and you can use it for any editing task. You could invest in a high-end product such as XMLSpy, available at

```
http://www.altova.com/products/xmlspy/xml_editor.html
```

Fortunately, Microsoft provides a free alternative that works great for Access. You can get XML Notepad 2007 at

```
http://www.microsoft.com/downloads/details.aspx?familyid=
          72d6aa49-787d-4118-ba5f-4f30fe913628
```

and it finally provides the full schema support that previous versions of this product didn't. Consequently, you can use the very small, fast, and most importantly, free XML Notepad 2007 to perform all the editing you need. Figure 8-1 shows an example of XML Notepad 2007 with a `customUI.xml` file loaded.

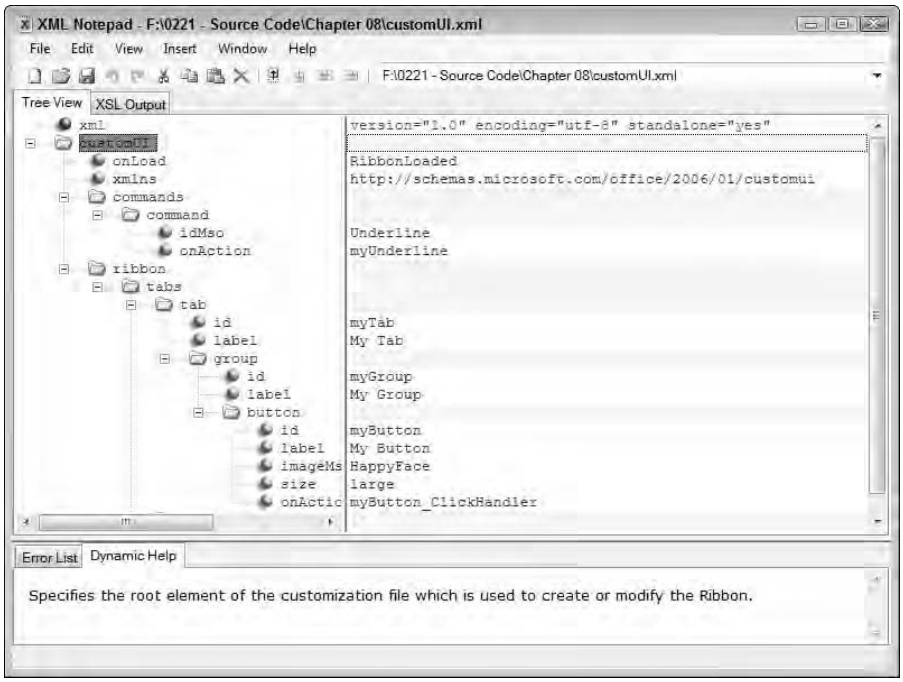


Figure 8-1: Use XML Notepad 2007 to perform edits of your Access XML files.

Notice the dynamic help displayed at the bottom of Figure 8-1. The help won't appear until you add an XML schema (`XSD`) file to your project. You can download the `XSD` file for the Ribbon from `http://officeblogs.net/UI/customUI.xsd`. Place this `XSD` file in your project folder, and then use the following steps to add it to XML Notepad:

1. **Choose View➪Schemas.**

   You see the XML Schemas dialog box, as shown in Figure 8-2.

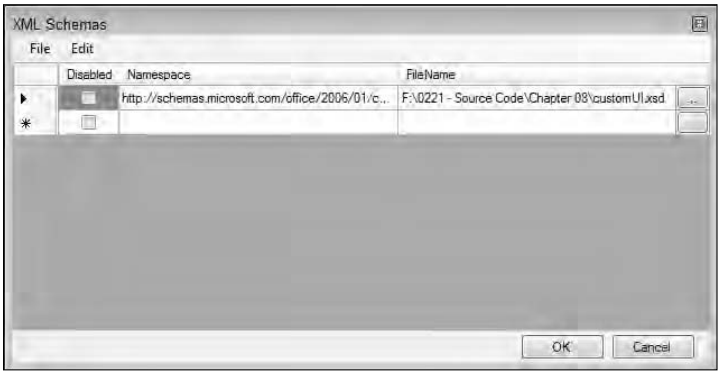


Figure 8-2: The XML Schemas dialog box provides the means for adding XSD files.

2. **Click the ellipsis at the end of the first open row.**

   You see an Open dialog box.

3. **Locate the `customUI.xsd` file you downloaded earlier; click Open.**

   XML Notepad loads the XSD file. You'll now see the dynamic help, shown in Figure 8-1, when you create the Access XML file.

The application will tell you of an error by displaying the following error message in the Dynamic Help window:

```
Dynamic help displays the xsd:documentation for the selected
node. You currently have no associated XML schema or your
selected node has no corresponding xsd:documentation in an
xsd:annotation.
```

In addition, you'll see an error message in the Error List window, shown in Figure 8-3, but only if you load the schema before you type the erroneous entry.

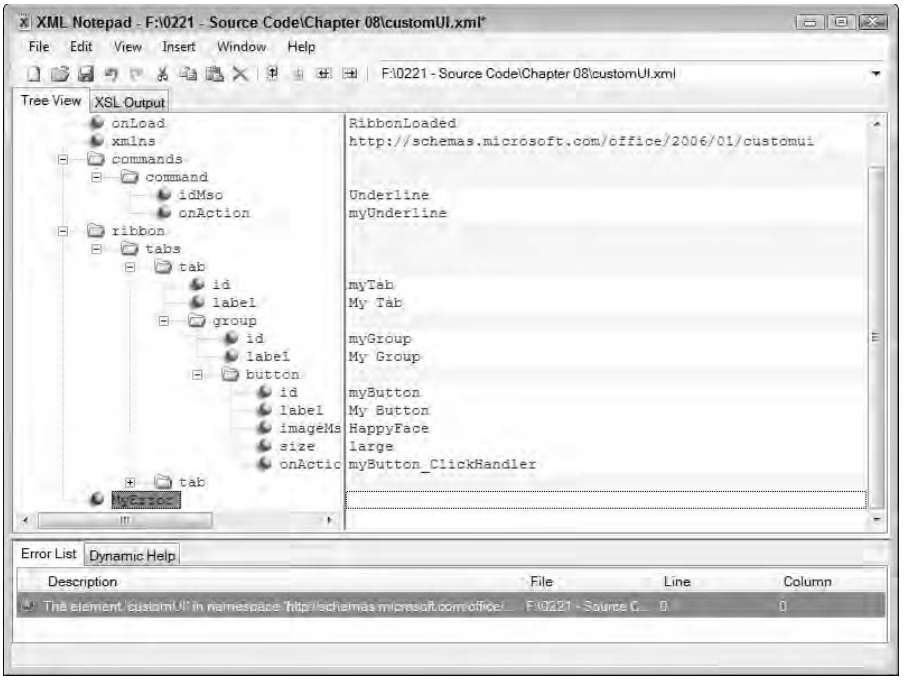


Figure 8-3: XML Notepad shows errant entries using two different techniques.

You can remove a schema from use at any time in XML Notepad by placing a check mark next to the entry in the XML Schemas dialog box (refer to Figure 8-2). The dynamic help window won't display the helpful messages any longer when you remove the schema from consideration. In addition, XML Notepad won't display any error messages.

XML Notepad also helps you make the correct entries. Double-click the entry when you add a new element or attribute. You'll see a list of acceptable entries for that position, as shown in Figure 8-4.

The XML used for this example is very similar to the XML used for the Excel example in Chapter 1. However, it has a couple of important differences:

- Access doesn't always use precisely the same names for the tabs.
- The Ribbon layout is different.

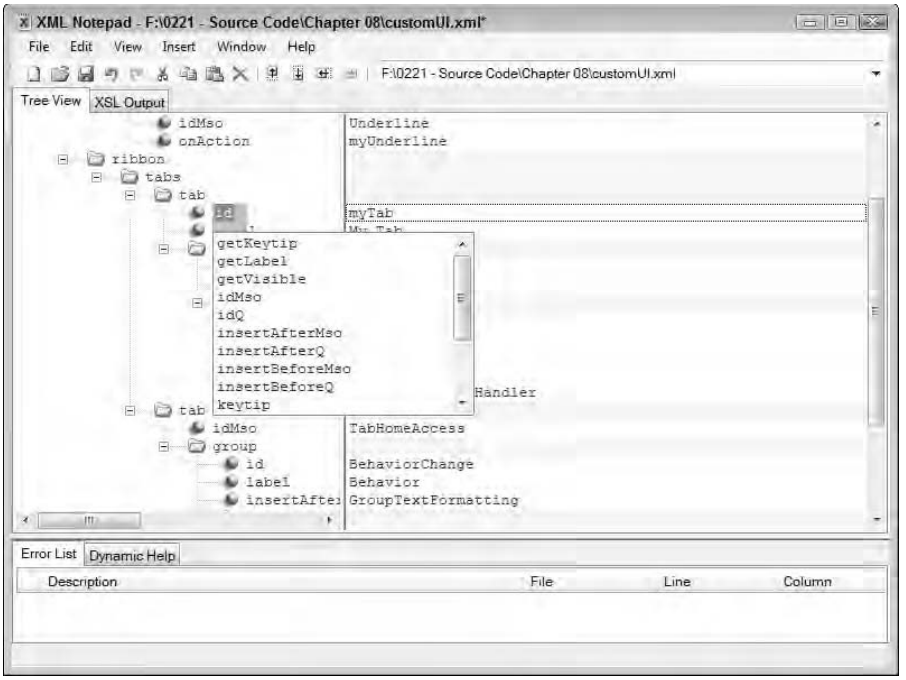


Figure 8-4: Let XML Notepad help you make the correct entries.

Listing 8-1 shows the XML for this example. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Listing 8-1: Defining Some Simple Ribbon Changes

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customUI onLoad="RibbonLoaded"
xmlns="http://schemas.microsoft.com/office/2006/01/customu
          i">
  <commands>
    <command idMso="Underline"
             onAction="myUnderline" />
  </commands>
  <ribbon>
    <tabs>
      <tab id="myTab" label="My Tab">
        <group id="myGroup" label="My Group">
          <button id="myButton"
                  label="My Button"
                  imageMso="HappyFace"
                  size="large"
                  onAction="myButton_ClickHandler" />
        </group>
      </tab>
```

```
      <tab idMso="TabHomeAccess">
        <group id="BehaviorChange"
               label="Behavior"
               insertAfterMso="GroupTextFormatting">
          <toggleButton id="StopUnderline"
             label="Stop Underlining"
             onAction="StopUnderline_ClickHandler"
             getPressed="StopUnderline_GetPressed"
             size="large"
             imageMso="ShapeFillColorPicker"
             insertBeforeMso="UnderlineGallery" />
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

One of the first changes you'll need to make is to ensure that you use `TabHomeAccess`, rather than `TabHome` (as found in the other Office products). Because the `TabHome` identifier doesn't appear in Access, you'll see an error message when you try to load the errant Ribbon.

The other change is more subtle and Access won't complain about it. Notice, however, that the group you add to the Home tab appears in the wrong place. The `GroupFont` identifier changes to `GroupTextFormatting` in Access.

The best lesson you can learn with this example is that the identifiers aren't necessarily consistent across Office applications, so you have to exercise caution when you move code from one application to the other.

The example defines two changes. First, it adds a new tab to the example (named My Tab). The tab contains a single group, `My Group`, with a single large button, `My Button`. The button has a smiley-face icon like the one shown in Figure 8-5.

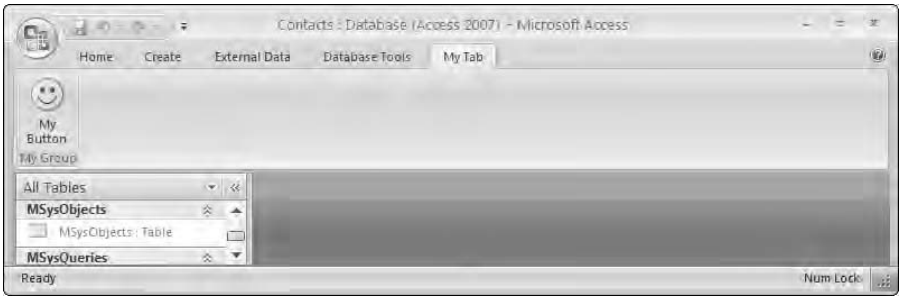


Figure 8-5: The new tab contains a single test button.

The second change is to the Home tab. The example adds the `Behavior` group and places a single large toggle button in it, `Stop Underlining`. The

button has the icon shown in Figure 8-6, which uses an `imageMso` value of `ShapeFillColorPicker`.

Figure 8-6: The first example also includes a button that changes the Underline button's behavior.

Loading the Ribbon Changes

You can choose whether to embed the XML for a change to the Ribbon in your Access file. Here Access differs from other Office products: You can actually choose to load the XML from an external source using VBA. You could use the external-load approach for an application that requires a level of dynamic response to user needs.

The more conventional approaches are to load the XML as part of a table. You use the USysRibbons table to store the XML when you want to modify the Ribbon for an entire database. Load the XML into a standard user table when you want to change the Ribbon only when working with that table. The following sections describe your options in detail.

Defining the Ribbon macros

The macros for this example are precisely the same as those found in the "Modifying or repurposing existing tabs, groups, and controls" section of Chapter 4. The fact that you can use the same Ribbon code in multiple applications says a lot about the Ribbon. You still need to consider any differences between applications. Some applications use different identifiers for essentially the same on-screen elements. You also have to consider how each application uses the `Ribbon` element and whether the element exists at all within a particular application. For example, Access doesn't include a `Styles` group, though Word does.

Using the system table USysRibbons

Using the `USysRibbons` approach requires that you create a new system table. The table contains the name of the Ribbon (you can use it to create multiple Ribbons and load the one you want programmatically) and the XML required to create the Ribbon. The following steps help you get started with this approach:

1. **Open the database you want to use for the Ribbon.**
2. **Create a new table.**

   The Ribbon requires that you create a table that contains three fields: ID (`AutoNumber` data type), RibbonName (`Text` data type), and RibbonXML (`Memo` data type), as shown in Figure 8-7. You don't need to change any of the default properties for the table elements.

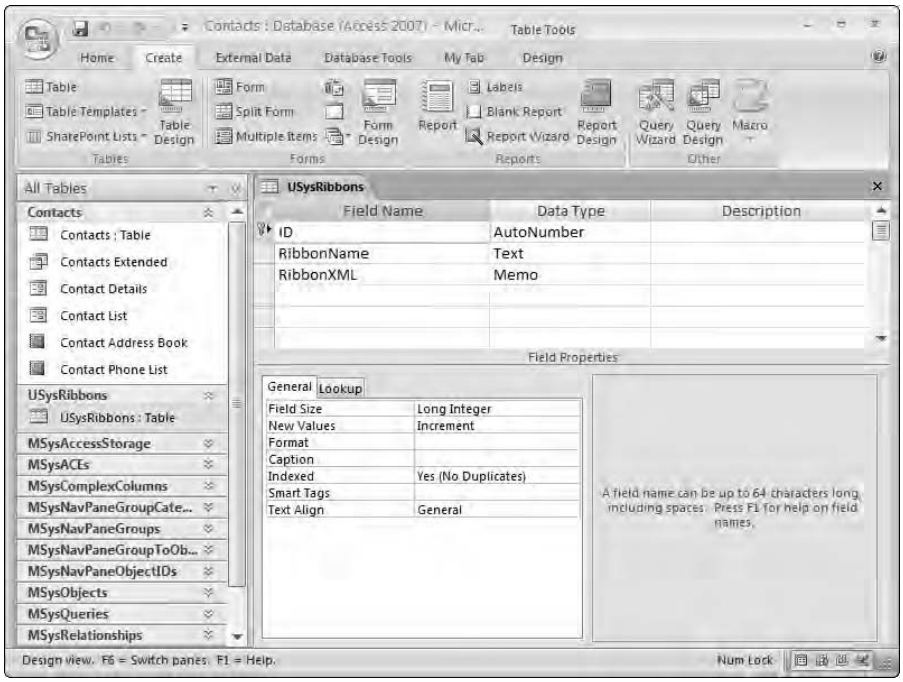


Figure 8-7: Create a table to hold the XML for your Ribbon.

3. **Save the new table with the name** USysRibbons**.**

   Access creates the new system table. Unfortunately, you can't see the new table in the All Tables list.

4. **Right-click the All Tables heading and choose Navigation Options from the context menu.**

   You see the Navigation Options dialog box, shown in Figure 8-8.

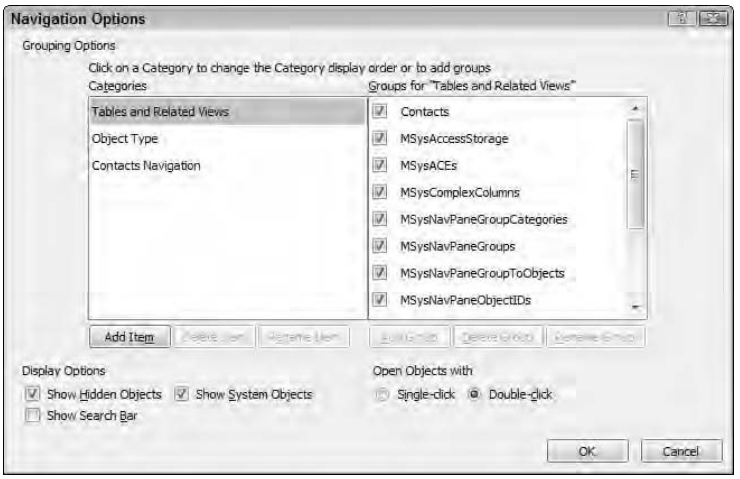


Figure 8-8: Change the navigation options to see the system tables.

5. **Check Show System Objects and click OK.**

   You see the tables for your application and the `USysRibbons` table, along with other system tables, as shown in Figure 8-7.

6. **Select the Data Sheet view from the View split button on the Home tab.**

   You see a data-entry table for the `USysRibbons` table.

At this point, you have a new table to hold any number of Ribbons you might want to create for Access. Unlike other Office applications, you can choose one Ribbon from any of the Ribbons you create. Consequently, even though Access is different from other Office products, it's also more flexible. The following steps help you create a new Ribbon entry and select it for use:

1. **Create a new record for the `USysRibbons` table.**
2. **Type** MyRibbon **in the RibbonName field.**
3. **In XML Notepad, choose View➪Source.**

   XML Notepad opens a copy of Notepad containing the XML that you created.

4. **Copy the Ribbon XML in Notepad and paste it into the RibbonXML field of the `USysRibbons` table in Access.**
5. **Close the Database and then reopen it.**

6. **Choose Office Menu➪Access Options.**
7. **Select Current Database.**

   You see the list of options shown in Figure 8-9.
8. **Choose MyRibbon in the Ribbon Name field of the Ribbon and Toolbar Options group.**
9. **Click OK.**

   You may see a message telling you that you must close the current database and reopen it to see the new Ribbon. Make sure that you close and reopen the database as needed.
10. **Close the current database and reopen it if necessary.**

    You see the new Ribbon shown in Figures 8-5 and 8-6.

If you haven't done so already, you need to create a new VBA module. It's normally a good idea to provide a descriptive name such as `MyRibbon RibbonX` for the new module. You want to include the name of the Ribbon entry as part of the module name so there isn't any confusion when you provide multiple Ribbons in the same database. Provide a VBA sub for each of the callbacks you define in your Ribbon XML entry.

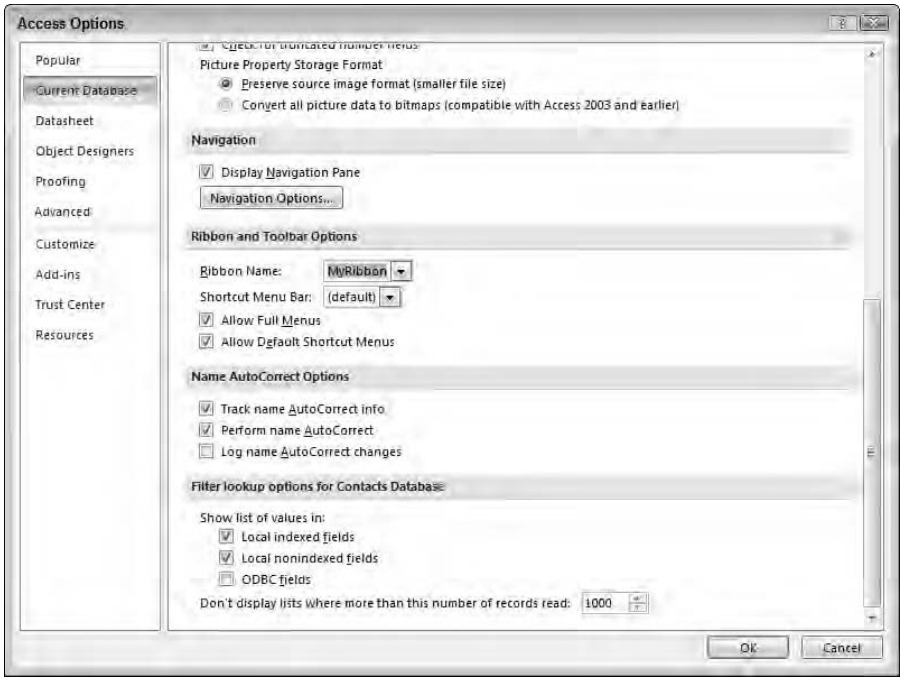


Figure 8-9: Select the Ribbon you want to use from the Ribbon Name list.

Using a standard user table

Access provides a special method that comes in very handy for loading the Ribbon on the fly — the `Application.LoadCustomUI()` method lets you place the XML for the Ribbon in a standard user table, rather than the `USysRibbons` table. This technique doesn't eliminate the need to change the Ribbon manually or close the database after you make the change. However, it does give you two advantages:

- You can store your Ribbons in an easily accessible table.
- You can move both the Ribbons and their associated code between databases with relative ease.

All you need do is export the table and place it in the new database, along with the associated RibbonX modules for each Ribbon (another good reason to use a separate module for each Ribbon).

The example uses a standard user table named `RibbonData`. You create this table the same way that you do the `USysRibbons` table. The example includes a single new Ribbon named `UserTableRibbon`. However, because of the way the code is designed, you can add any number of Ribbons to your table and the code will continue to work as before.

The XML for this Ribbon appears in the `StandardUserDatabase.xml` file supplied with the source code for the book. Since the purpose of this example is to demonstrate another technique for loading a Ribbon, the demonstration code is very simple. Here's the single tab, group, and button that the example uses:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customUI
xmlns="http://schemas.microsoft.com/office/2006/01/customu
          i">
  <ribbon>
    <tabs>
      <tab id="MyTab" label="My Tab">
        <group id="MyGroup" label="User Database">
          <button
             id="TestMe" label="Test Me" size="large"
             onAction="TestMe_ClickHandler"
             imageMso="TentativeAcceptInvitation" />
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

The resulting Ribbon contains a single button with a big purple question mark on it. Clicking the button calls the `TestMe_ClickHandler()` callback method shown here:

```
'Callback for TestMe onAction
Sub TestMe_ClickHandler(control As IRibbonControl)
    MsgBox "Hello from UserTableRibbon!"
End Sub
```

You'll need to create a function to load the XML. Generally, you'll want to place this function in its own module so you don't accidentally move it with the rest of the RibbonX code. The example uses a generic loading technique, as shown in Listing 8-2.

Listing 8-2: Adding Ribbons to the User Interface

```
'Adds Ribbons from a table.
Function AddRibbons()

   'Access the current database.
   Dim CurrentDB As DAO.Database
   Set CurrentDB = Application.CurrentDB

   'Obtain access to the standard user database.
   Dim RibbonData As Recordset
   Set RibbonData = _
      Application.CurrentDB("RibbonData").OpenRecordset

   'Locate the data.
   RibbonData.MoveFirst

   'Holds the name of the Ribbon.
   Dim RibbonName As String

   'Holds the XML for the Ribbon
   Dim XMLData As String

   'Obtain all of the Ribbons in the table.
   While Not RibbonData.EOF

      'Obtain the Ribbon name.
      RibbonName = _
         RibbonData.Fields("RibbonName").Value

      'Obtain the XML.
      XMLData = RibbonData.Fields("RibbonXML").Value

      'Load the new ribbon from a user database.
```

(continued)

Listing 8-2 *(continued)*

```
        Application.LoadCustomUI RibbonName, XMLData

        'Move to the next record.
        RibbonData.MoveNext
    Wend

End Function
```

The code begins by creating a `DAO.Database` object that points to the current database. It then creates a `Recordset` object to hold the records from the `RibbonData` table.

At this point, the code begins loading the `RibbonData` rows one at a time and adding them to the user interface. The code must supply both a Ribbon name and the Ribbon's XML content. In this case, the code places the `RibbonData.Fields("RibbonName")` value in `RibbonName` and the `RibbonData.Fields("RibbonXML")` value in `XMLData`.

One word of caution here: You need to ensure that the Ribbons you create have unique names. Access requires that every Ribbon have a unique name. If you try to load a Ribbon that has the same name as another Ribbon, Access will display an error message.

Calling the `AddRibbons()` function every time you open the database is important because the Ribbon entries have no permanent status within the database. To perform this task, you create an `AutoExec` macro that contains a single `RunCode` action. You provide the `AddRibbons()` function as input to the Function Name field.

Whenever the database opens, it loads all of the Ribbons stored in RibbonData. You can select any of these Ribbons using precisely the same technique as you use for Ribbons found in the USysRibbons table. The only difference is that these Ribbons aren't a permanent part of the database and you can move them around quite easily.

Using an XML file directly

You can gain a significant advantage by using XML files directly with Access. The file resides outside of Access, so you can edit it without even opening Access. The next time someone opens the database, it automatically loads the changes you made to the Ribbon.

Working with an XML file directly follows the same pattern as working with a user database. You begin by creating a function to load the XML files as strings and then use the `Application.LoadCustomUI()` method to load the XML files into Access. Because the technique is essentially the same, you also

have to provide the `AutoExec` macro to perform the required work when the user opens the database.

Before you can begin working with this example, you'll need to create a new VBA module to hold it. As part of the configuration process, you'll need to add two new references: Microsoft Scripting Runtime and Microsoft XML (the example uses version 6.0 of this feature). To add the references, choose Tools➪ References. You'll see the References dialog box, shown in Figure 8-10, where you can check the two required references. Click OK to complete the process.

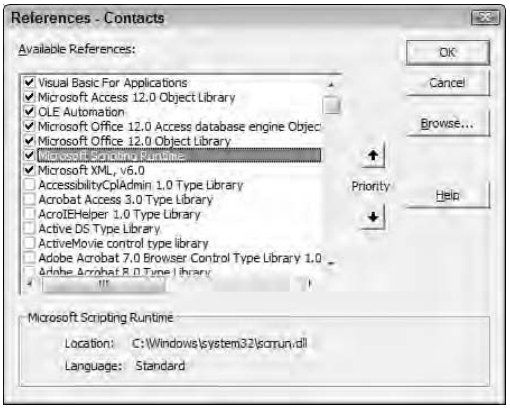


Figure 8-10: Add the references required to make this example work.

It's important to remember that VBA loads the XML file as XML, not as a string. Consequently you have to perform the required conversions before you can use the XML. Listing 8-3 shows the function you have to create in order to work with XML directly.

Listing 8-3: Loading XML Directly

```
'Adds Ribbons from an XML file.
Function LoadXMLRibbon(XMLFileLocation As String)

   'Provides a pointer to the file system.
   Dim ThisFile As FileSystemObject
   Set ThisFile = New FileSystemObject

   'Holds the XML file as a string.
   Dim RibbonXML As String

   'Holds the Ribbon name.
   Dim RibbonName As String

   'Holds the XML version of the file.
   Dim ThisDoc As DOMDocument60
```

(continued)

Listing 8-3 ***(continued)***

```
    Set ThisDoc = New DOMDocument60

    'Locate the XML file.
    If ThisFile.FileExists(XMLFileLocation) Then

       'Set the Ribbon name to the filename.
       RibbonName = _
          ThisFile.GetFileName(XMLFileLocation)

       'Remove the file extension.
       RibbonName = _
          Left(RibbonName, Len(RibbonName) - 4)

       'When the file exists, load it.
       If ThisDoc.Load(XMLFileLocation) Then

          'Place the pure text into the Ribbon string.
          RibbonXML = ThisDoc.XML

          'Load the XML file into Access.
          Application.LoadCustomUI RibbonName, RibbonXML

       'Display an error message.
       Else
          MsgBox "Couldn't read the XML file.", _
                 vbOKOnly And vbExclamation, _
                 "Error Reading XML File"

       End If

    'Display an error messages.
    Else

       MsgBox "The XML File: " + XMLFileLocation + _
              " doesn't exist.", _
              vbOKOnly And vbExclamation, _
              "Error Loading XML File"

    End If
 End Function
```

The code begins by creating a `FileSystemObject` and using it to check for the existence of the file with the `ThisFile.FileExists()` method. Even the best-organized hard drive will occasionally lose a file, so you should always perform this check, even if you're certain that no one would tamper with or remove the file. The `FileSystemObject` object appears as part of the Microsoft Scripting Runtime library, which is why you need to add that reference to the example. If the file doesn't exist, the example displays an error message and exits.

The example assumes that you won't have two XML files with the same name, so it uses the filename to create the `RibbonName` string. If you think you'll have multiple XML files with the same name, you'll need to create another naming convention; Access won't allow two Ribbon entries with the same name.

At this point, the code loads the file into a Document Object Model (DOM) document object (`DOMDocument60` in this case) using the `ThisDoc.Load()` method. You can read the file directly from the hard drive without creating an intermediate `TextStream` object. This approach also frees you from worrying about how to open the file (or how to close it later when you no longer need it). If the XML file won't load because it contains an error, the example displays an error message and exits.

You may wonder why the code loads the text from the file into a `DOMDocument`. The reason you have to perform this extra step is that the file will very likely have a *Byte Order Mark* (BOM) in it. The BOM is a set of extra characters that appear at the beginning of the file to tell what kind of encoding it uses. For example, when a file uses UTF-8 encoding, it includes three hexadecimal characters — `EF BB BF` — at the beginning of the file. The number of characters varies by encoding type, so you can't simply skip a certain set number of characters when you read the text file directly; loading the file into a `DOMDocument` is the best way to remove them safely from the rest of the content. You can read more about the BOM at

```
http://unicode.org/faq/utf_bom.html
```

You still need to convert the XML into a string. The code performs this task by placing the `ThisDoc.XML` property into RibbonXML. Finally, the code calls the `Application.LoadCustomUI()` method to load the new Ribbon into Access.

A Ribbon that you load directly from an XML file behaves precisely the same as one you load using USysRibbons or a standalone table. You still need to select the new Ribbon from the list in the Ribbon Name field of the Ribbon and Toolbar Options group of the Access Options dialog box. After you close and reopen the database, Access displays it as normal. The process that Access uses is very similar to a standalone table — only the source changes.

You also have to add an entry to the AutoExec macro, just as you do for the standalone table. In this case, you must include the location of the XML file on disk. The example contains the location of the file for my system. You'll need to change the path shown in Figure 8-11 to match your system or the example won't work. It's possible to change the code to detect the application path and use that path for the XML file as well, which makes the code less brittle, but then you lose flexibility because the XML file must appear in a specific location.

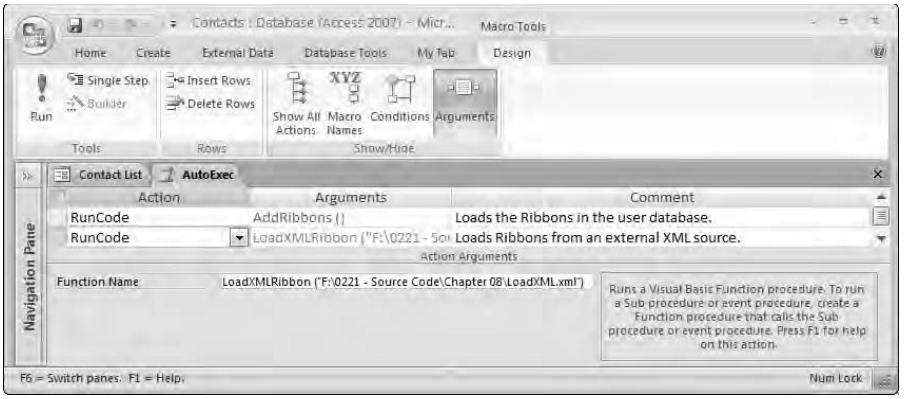


Figure 8-11: Change the path of the XML file to match your system.

Obtaining the Sample Database

Access doesn't come with a standard sample database. Microsoft provides myriad templates, but none of them include sample data you could easily use for testing. Consequently, you need a sample database you can use for quick tests of working with the Ribbon using real data. This book uses the familiar Northwind Traders database that you can download at

```
http://www.microsoft.com/downloads/details.aspx?FamilyID=
          C6661372-8DBE-422B-8676-C632D66C529C
```

After you download and extract the database file, you'll need to convert it to an Access 2007 format. The following steps help you perform the required conversion:

1. **Choose Office Menu⇨Open.**

   You see the Open dialog box.

2. **Locate and highlight the `NWind.mdb` file. Click Open.**

   Access displays the Database Enhancement dialog box, as shown in Figure 8-12.

3. **Click Yes.**

   You see a Save As dialog box where you can choose a new filename for the file.

Because Access 2007 uses a file extension other than `MDB`, saving the file in the new format doesn't overwrite the old file. Consequently, whenever you need to create a new copy of the sample database, you can simply open the original `NWind.mdb` file and convert it into a new Access 2007 file. The new file contains all the unchanged records of the original.

Figure 8-12: Access automatically recognizes the old file format of the sample database.

4. **Click Save to accept the default filename of `NWind.accdb`.**

   You'll see an error dialog box telling you that Access encountered errors converting the database. The only error that you should see in the Conversion Errors table is one that relates to user-level security. Access 2007 no longer supports user-level security, so it removes the user-level protection from the file. Because this database is for testing purposes only, the loss of security won't matter.

5. **Click OK to bypass the error dialog box.**

   Access tells you that the new database is in an updated format that you can't share with users of older Access versions.

6. **Click OK to bypass the warning dialog box.**

   Access converts the file and saves it to the new file extension.

After you perform the required conversion, you'll notice that Access performs a few tasks differently from every other Office product. The custom menus for Access still appear on the Add-Ins tab, but you'll find that they are quite usable. Figure 8-13 shows an example of what you'll see when you open the Alphabetical List of Products report.

Chapter 12 makes you aware of some other Access differences that can actually make it easier to move your application to Access 2007. In some cases, you might find that your old application works fine with a few additional tweaks. Access provides more flexibility in this regard than any other Office product.

However, don't get the idea that Access can fix every potential woe. Not only do you need to compensate for a loss of user-level security in your security plans, but you also need to consider operating system issues. For example, you'll have to rewrite any existing Access HLP (help) files, just as you would for any other application running under Vista because Vista doesn't support the older HLP files. Click Show Me and you'll see the usual Vista error message, shown in Figure 8-14, for HLP files.

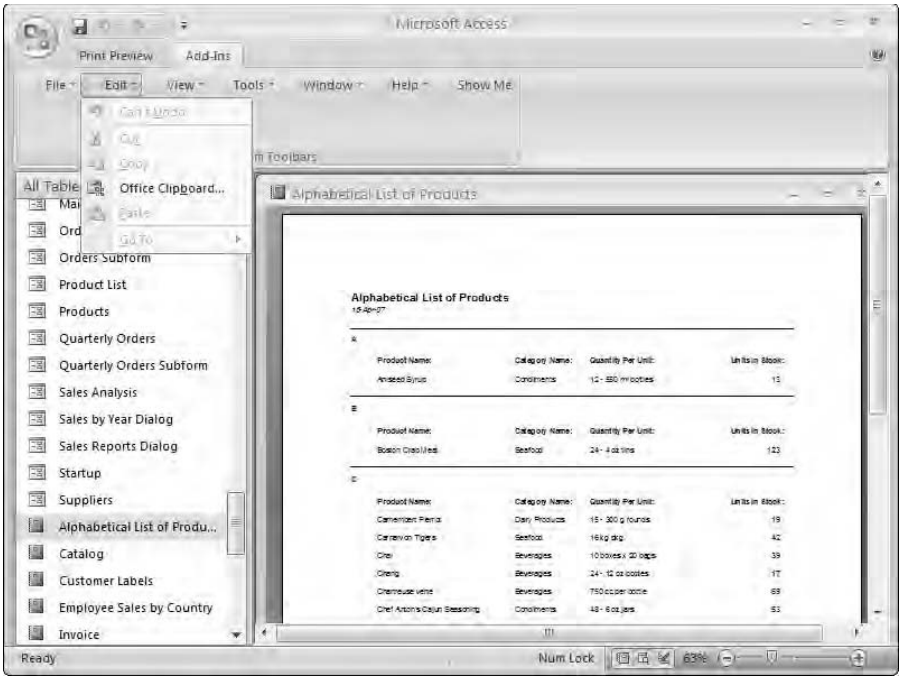


Figure 8-13: Access tends to leave custom menus intact when using the Ribbon.

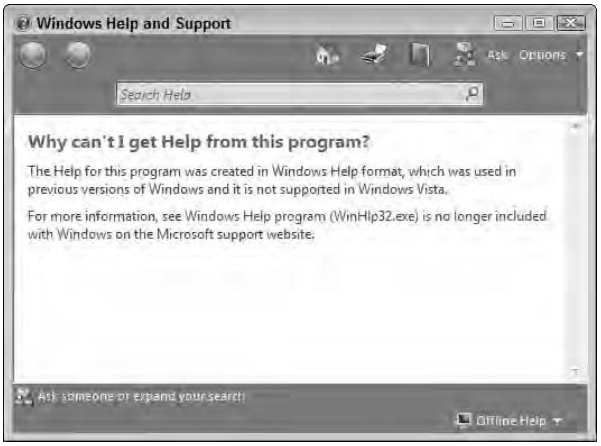


Figure 8-14: Access won't overcome deficiencies of the operating system.

Generating Temporary Tables or Filtered Results

This example provides you with a means of performing several tasks that you'll commonly require in Access. The first is that the Access application

will very likely have to support multiple versions. Unlike other Office products, Access is very capable in this regard as long as you follow a few rules. The following sections describe how to create an application that generates a filtered result or temporary table, while demonstrating some basic Access rules of the road. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.)

Hiding the Add-Ins tab

One of the basic rules for working with Access is hiding the Add-Ins tab unless you really need it. Otherwise, you may find that some commands don't work as intended, which is the problem with the Show Me option in the Northwind menu. Whether you hide the Add-Ins tab depends on how you intend to work with the Access application. You may very well find that it isn't a requirement, but only when every menu item works as advertised in Access 2007.

The XML for hiding any tab is relatively easy. Here's all you need to hide the Add-Ins tab:

```
<tab idMso="TabAddIns" visible="false" />
```

Notice that all you need to do is set the `visible` attribute to `false`. The rules are the same for any Access feature. For example, if you want to hide a group, you'd create a `<tab>` element with the correct `idMso` attribute value, then a `<group>` child element with the `idMso` value for the group you want to hide. Add a `visible="false"` entry and the group will disappear.

Placing the groups in the correct order

Another issue is placing groups correctly. Because of the specific way you work with the Ribbon in Access, you'll probably want to add some new groups to existing tabs or add new controls to existing groups. In this example, you actually add two new groups to the existing Create tab. The first defines the group for temporary tables, while the second defines the group for the filtered results. You could use the following XML for the task:

```
<tab idMso="TabCreate">
  <group id="CreateTemporaryTable"
         label="Temporary Table"
         insertAfterMso="GroupCreateTables" />
  <group id="CreateFilteredResults"
         label="Filtered Results"
         insertAfterMso="GroupCreateTables" />
</tab>
```

The `insertAfterMso` attribute tells Access to place each of these groups after the `GroupCreateTables` group. Unfortunately, this XML places the new groups in reverse order. Yes, the resulting Ribbon will probably still work, but it doesn't appear precisely the way you'd like it to appear. The user sees the `Filtered Results` group first, placing more emphasis on those features.

One way around this problem is to place all the entries in reverse order in the XML file, but that hardly seems the best approach. A better way to control the order is to use your own namespace and then employ the `insertAfterQ` attribute for the `CreateFilteredResults` group. You place the namespace in the `<customUI>` element, as shown in Figure 8-15.

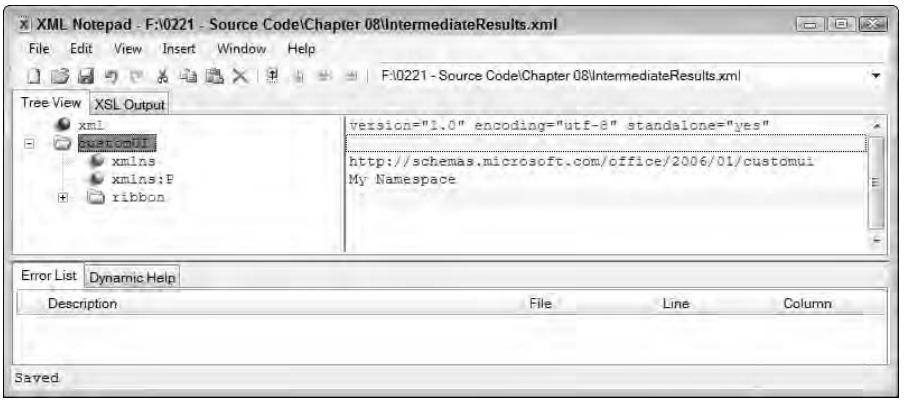


Figure 8-15: Create a custom namespace for your groups and controls.

Notice that `xmlns:P` points to a string, not a URL. Some people think that a namespace always appears within an XSD file that you specify as part of a specific URL. In this case, the namespace serves only to differentiate between your controls and those that Microsoft supplies as part of Access. Consequently, you don't need to provide anything more than a string such as `My Namespace`.

Now that you have a namespace to use, you can qualify the groups and order them properly. The following XML shows the new group XML for `TabCreate`:

```
<tab idMso="TabCreate">
  <group idQ="P:CreateTemporaryTable"
         label="Temporary Table"
         insertAfterMso="GroupCreateTables" />
  <group idQ="P:CreateFilteredResults"
         label="Filtered Results"
         insertAfterQ="P:CreateTemporaryTable" />
```

Notice that both of the groups now use the `idQ` attribute, rather than the `id` attribute. In addition, each of the IDs now has a namespace qualifier. The `CreateTemporaryTable` group still uses the `insertAfterMso= "GroupCreateTables"` entry. However, the `CreateFilteredResults`

group now uses an `insertAfterQ="P:CreateTemporaryTable"` entry to correctly place this group after `CreateTemporaryTable`. Figure 8-16 shows the updated Create tab for this example. Notice that the new tab also leaves out the Add-Ins tab.

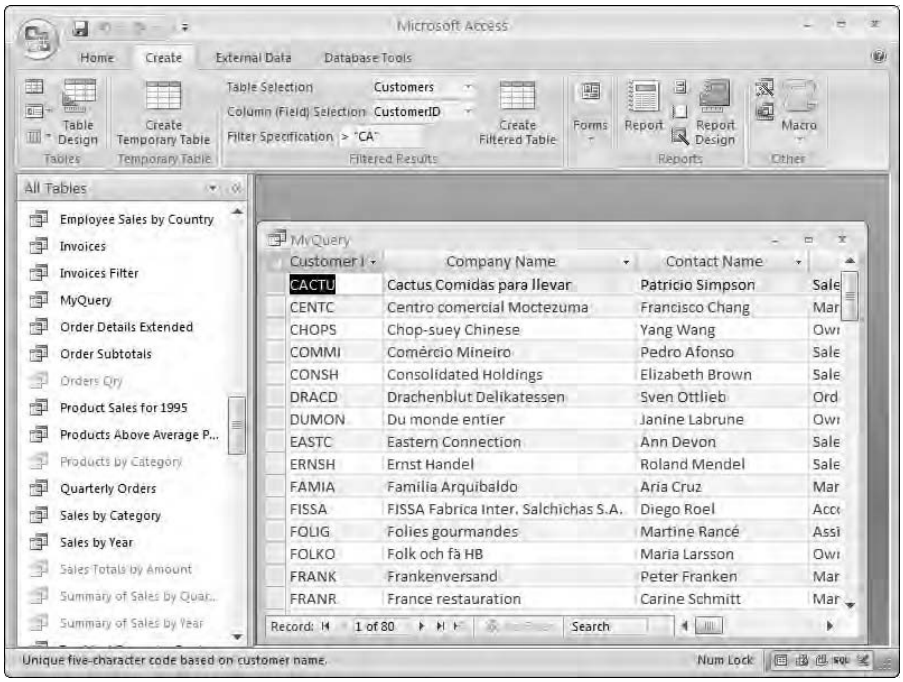


Figure 8-16: The new Create tab includes entries for temporary and filtered result tables.

Creating a temporary table

DBAs use temporary tables for a number of tasks. For example, you might want to see what happens to a table (or set of tables) when you create a certain query, without risking your data. Using a set of temporary tables to perform the task lets you perform the test without risking your data. Temporary tables can also hold copies of records until you complete a given task. You can also use them for other purposes when you don't want to risk the data but you do need to accomplish specific tasks within the database. The example uses the XML shown here to create the single button shown in Figure 8-16:

```
<group label="Temporary Table"
       insertAfterMso="GroupCreateTables"
       idQ="P:CreateTemporaryTable">
  <button id="TempTable" label="Create Temporary Table"
          size="large" imageMso="CreateTable"
          onAction="TempTable_ClickHandler" />
</group>
```

At this point, you're probably wondering how a single button can accomplish the required task. The example relies on a special form to display the selection data. When working with Access, you can either use forms or place the data selection on the Ribbon. You can see an example of the second option in the "Defining a filtered result" section of the chapter; Figure 8-17 shows the form used for this example.

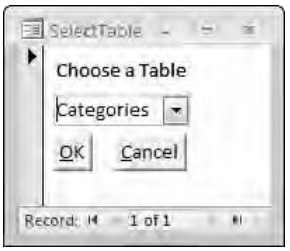


Figure 8-17: Choose a table to temporarily copy using this form.

Normally you'll create a form in VBA, display it, and retrieve the user selections from it all in a single `Sub`. However, when working with Access, you need to create a single `Sub` to display the form and then another `Sub` for each button to perform any special processing. Fortunately, Access provides built-in functionality to handle common tasks such as closing the form. Consequently, this example needs to provide only a `Sub` to handle the OK button. Listing 8-4 shows the code you need for this part of the example.

Listing 8-4: Choosing and Copying the Table

```
Sub TempTable_ClickHandler(control As IRibbonControl)

   ' Open the form.
   DoCmd.OpenForm "SelectTable", View:=acNormal

   ' Create a reference to the table selection form.
   Dim ThisForm As Form
   Set ThisForm = Application.Forms("SelectTable")

   ' Create a reference to the control that holds the
   ' selections.
   Dim TableList As ComboBox
   Set TableList = ThisForm.Controls("cbTables")

   ' Ensure the form is selected.
   ThisForm.SetFocus

   ' Obtain a list of tables and place the names
   ' in the table selection form.
```

```
   Dim SelTable As TableDef
   For Each SelTable In Application.CurrentDb.TableDefs

      ' Add the table name.
      TableList.AddItem SelTable.Name
   Next
End Sub

Function TempTable_OK()

   ' Create a reference to the table selection form.
   Dim ThisForm As Form
   Set ThisForm = Application.Forms("SelectTable")

   ' Create a reference to the control that holds the
   ' selections.
   Dim TableList As ComboBox
   Set TableList = ThisForm.Controls("cbTables")

   ' Holds the current database selection.
   Dim SelectedName As String

   ' Make sure the user actually selected something.
   If IsNull(TableList.Value) Then
      MsgBox "You must select a database!", _
      vbOKOnly And vbExclamation, _
      "Selection Error"

      Exit Function
   End If

   ' Otherwise, save the database selection.
   SelectedName = TableList.Value

   ' Create the temporary table based on the selection.
   DoCmd.CopyObject _
      NewName:="Temp" + SelectedName + _
               DateTime.Date$ + DateTime.Time$, _
      SourceObjectType:=acTable, _
      SourceObjectName:=SelectedName

   ' Close the form.
   DoCmd.Close acForm, "SelectTable"
End Function
```

The code begins by opening the `SelectTable` form in Form view (rather than Layout or Design view). You must have the form open before you can begin populating the `cbTables` combo box. The code obtains references to the form as `ThisForm` and to the `cbTables` control as `TableList`.

After the code displays the form and ensures it has focus (by calling `ThisForm.SetFocus`), it begins filling `cbTables` with data. In this case, the code uses a simple `For Each` loop and copies the names of the tables currently found in the database to the control. At this point, the user can select a database to copy as a temporary table.

The `TempTable_OK()` function performs the required processing when the user clicks OK, rather than Cancel. The code begins by obtaining a reference to the `cbTables` combo box. It uses this reference to access the `TableList.Value` property, which contains the user selection. Because the user could click OK without actually selecting a table entry, you must provide error-detection code that looks for a `Null TableList.Value` property using the `IsNull()` function.

The code places the table selection in `SelectedName`. It then relies on `DoCmd.CopyObject` to create the actual temporary database. The `DoCmd.CopyObject` call will fail when the database already contains the requested target table. Consequently, the example adds a date, `DateTime.Date$`, and time, `DateTime.Time$`, to the table name. If you want to ensure that the system contains only one temporary version of each table, you can always use a `For Each` loop to scan the current table list looking for a duplicate. Simply add code that looks similar to the code shown in the `TempTable_ClickHandler()` method.

Defining a filtered result

Most users complain about seeing too much data. For that matter, data overload is a common problem for DBAs too. This part of the example shows one way to make data filtering easier. It's not a very complicated technique, but you'll find it works well, in many cases, to reduce on-screen clutter when you need to find a few records quickly.

This example demonstrates a considerable number of techniques, and the chapter simply can't hold all the source code required to create it. The sections that follow do provide complete information about all the essential techniques for working with this example and modifying it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `Nwind.accdb` file.

Creating the Ribbon presentation

As previously mentioned, you can choose between forms and controls on the Ribbon to accept input from the user. This example shows how to use the Ribbon controls technique. Consequently, the XML for the Ribbon is a little

more complicated than the code shown in the "Creating a temporary table" section of this chapter. Listing 8-5 shows the code needed to create the physical Ribbon presentation.

Listing 8-5: Defining the Filtered Results Group

```
<group label="Filtered Results"
       idQ="P:CreateFilteredResults"
       insertAfterQ="P:CreateTemporaryTable">
  <comboBox id="FilteredTable" label="Table Selection"
            getItemCount="TableItemCount"
            getItemID="TableItemID"
            getItemLabel="TableItemLabel"
            onChange="TableChanged" />
  <comboBox id="ColumnSelect"
            label="Column (Field) Selection"
            getItemCount="ColumnItemCount"
            getItemID="ColumnItemID"
            getItemLabel="ColumnItemLabel"
            onChange="ColumnChanged"
            getEnabled="ColumnGetEnabled"
            getText="ColumnGetText" />
  <editBox id="FilterCriterion"
           label="Filter Specification"
           onChange="FilterChanged"
           getEnabled="FilterGetEnabled"
           getText="FilterGetText" />
  <button id="CreateFiltered"
          label="Create Filtered Table"
          onAction="CreateFiltered_ClickHandler"
          size="large" imageMso="CreateTable" />
</group>
```

Notice that the `ColumnSelect` and `FilterCriterion` controls both use the `getEnabled` attribute. It's a good practice to disable controls that require input from other controls until the user fills in the prerequisite controls. That's how these controls work. In this case, the `ColumnSelect` control remains disabled until the user provides a value for `FilteredTable`. Likewise, `FilterCriterion` remains disabled until the user provides a value for `ColumnSelect`.

All data-entry controls provide callbacks for the `onChange` attribute. Each control's callback maintains a record of the current control value so the application can process the input later. In addition to the `onChange` attribute, the `ColumnSelect` and `FilterCriterion` controls also implement a callback for the `getText` attribute, which clears the control values when the user selects a new table.

The two combo boxes require lists of items. Because you can't determine what the list will contain at design time (in fact, the content of the `ColumnSelect` control changes to reflect the fields supported by the current table), you must provide callbacks for the `getItemCount`, `getItemID`, and `getItemLabel` controls.

Filling the Table Selection field with data

Now that you have a better idea of how the controls work, it's time to look at the first set of callbacks. One of the first tasks the code has to perform is to add a list of tables to the `FilteredTable` control in the current database. Listing 8-6 shows the code used for this purpose.

Listing 8-6: Populating the Table Selection Combo Box with Data

```
Sub TableItemCount(control As IRibbonControl, _
                   ByRef count)
   ' Obtain the table count and return it.
   count = Application.CurrentDb.TableDefs.count
End Sub

Sub TableItemID(control As IRibbonControl, _
                index As Integer, ByRef ID)
   ' Return the table ID.
   ID = "Table" + CStr(index)
End Sub

Sub TableItemLabel(control As IRibbonControl, _
                   index As Integer, ByRef label)
   ' Return the table name.
   label = Application.CurrentDb.TableDefs(index).Name
End Sub
```

You actually need three `Subs` to perform this task. Each `Sub` handles a different aspect of filling the control with data. The `TableItemCount() Sub` provides a simple count of the number of items in the combo box. The code obtains this information from the `Application.CurrentDb.TableDefs.count` property. The `TableItemID()` provides an identifier for each of the items; in this case, the code uses the simple method of appending the `index` number to the word `Table`. The only requirement is that you provide a unique identifier for each of the items. Finally, the `TableItemLabel() Sub` provides the name of each of the tables. You can use `index` as a reference to the list of `TableDefs`, which makes providing the correct table name easy.

Whenever the user changes a selection in the Ribbon, the code has to react. In fact, it has to do more than react — it must coordinate the user's activity so that only the essential items remain active on-screen and the user can't

accidentally make improper choices. Listing 8-7 shows the code needed to react to user input.

Listing 8-7: Registering a Selection Change

```
Sub TableChanged(control As IRibbonControl, _
                 text As String)
   ' Save the selected value to a global variable.
   TempVars("TableSelected") = text

   ' Set the field selection state.
   If Not IsNull(text) Then
      TempVars("ColumnEnabled") = True
   Else
      TempVars("ColumnEnabled") = False
   End If

   'Verify the Ribbon reference.
   If Rib Is Nothing Then
      MsgBox "Close the Database and reopen it.", _
      vbOKOnly And vbExclamation, _
      "Ribbon Reference Error"

      Exit Sub
   End If

   ' Set the field to an empty value.
   TempVars("ColumnChanged") = ""

   ' Invalidate the control.
   Rib.InvalidateControl "ColumnSelect"
End Sub
```

The code begins by saving the current value found in `text` to the `TableSelected` global variable. Microsoft added the `TempVars()` function to Access 2007 to overcome an issue where Access automatically clears all variables when a macro encounters an error. In times past, this problem would cause macros to stop functioning properly. The user often had to close the database and start over. Unfortunately, this new technique doesn't work with the Ribbon reference, so you still have to use a global variable with it.

The next step is to enable or disable the `ColumnEnabled` control based on the content of text. If the user hasn't selected anything, then the `ColumnEnabled` control remains disabled.

It's important to check the status of the Ribbon reference before you use it; Access can clear this variable at any time. Using `Rib` after Access clears it results in an error. Always verify the status of the Ribbon reference before you use it in your code.

At this point, the code sets the `ColumnChanged` value to a blank and invalidates the `ColumnSelect` control using the `Rib.InvalidateControl()` method. Because of the way that you have to set up the Ribbon controls, it's important to initialize all the variables when the Ribbon loads. As with nearly every other Ribbon application, you set an `onLoad` callback as part of the `customUI` element. Listing 8-8 shows the initialization code for this example. (You can find all of the XML for this example in the `Filtered Results.xml` file supplied with the source code for this book.)

Listing 8-8: Saving the Ribbon Reference and Performing Setups

```
Public Rib As IRibbonUI

Sub TempTableOnLoad(ribbon As IRibbonUI)
   ' Save the Ribbon reference.
   Set Rib = ribbon

   ' Set the state of the column selection.
   TempVars("ColumnEnabled") = False
   TempVars("FilterEnabled") = False

   ' Set the initial text values.
   TempVars("ColumnChanged") = ""
   TempVars("FilterChanged") = ""
End Sub
```

As previously mentioned, `Rib` is a standard public variable because you can't use the `TempVars()` method to store objects. This issue makes the Ribbon problematic in Access because you can obtain a Ribbon reference only by using the `onLoad` callback. Unfortunately, Access calls only the `onLoad` callback when you initially open the database.

Filling the Column (Field) Selection field with data

After a user selects one of the entries in the Table Selection field, the application populates the Column (Field) Selection field. The code in Listing 8-9 shows how this process works.

Listing 8-9: Populating the Column (Field) Selection Combo Box with Data

```
Sub ColumnItemCount(control As IRibbonControl, _
                    ByRef count)
   ' Return the number of fields in the table.
   count = _
      Application.CurrentDb.TableDefs( _
         TempVars("TableSelected")).Fields.count
```

```
End Sub

Sub ColumnItemID(control As IRibbonControl, _
                 index As Integer, ByRef ID)
   ID = "Field" + CStr(index)
End Sub

Sub ColumnItemLabel(control As IRibbonControl, _
                    index As Integer, ByRef label)
   label = _
      Application.CurrentDb.TableDefs( _
         TempVars("TableSelected")).Fields(index).Name
End Sub

Sub ColumnGetEnabled(control As IRibbonControl, _
                     ByRef enabled)
   ' Return the selected item state.
   enabled = TempVars("ColumnEnabled")
End Sub

Sub ColumnGetText(control As IRibbonControl, _
                  text As String)
   ' Output the current value.
   text = TempVars("ColumnChanged")
End Sub
```

The code performs essentially the same tasks as the code for the Table Selection field in Listing 8-6. Notice that you're working with the fields associated with a particular table. Consequently, most of the code in Listing 8-9 will fail unless the user makes a table selection first. Considering the fact that any error in Access will kill your Ribbon reference, it pays to reduce the possibilities for error as much as possible. Generally speaking, if the user makes an entry error, you'll need to ask the user to close the database and then reopen it.

The `ColumnGetText()` method is new for the Column (Field) Selection field. This method normally provides the combo box with an output value based on criteria within your application. For example, you might use it to choose a particular entry within the list. In this case, the method actually provides a way of clearing the user's current selection when that user chooses a new table.

Processing the user's filtering selections

After all the data gathering the application has performed, you might wonder whether the user will ever see a filtered table. The final step is to provide a callback for the `CreateFiltered` control. Listing 8-10 shows the code needed to process the user input and create filtered output.

Listing 8-10: Performing the Filtered Query

```
Sub CreateFiltered_ClickHandler( _
   control As IRibbonControl)

   ' Define the SQL statement.
   Dim SQLText As String
   SQLText = _
      "SELECT * FROM [" + _
         TempVars("TableSelected") + "]" + _
      " WHERE [" + TempVars("ColumnChanged") + "]" + _
         TempVars("FilterChanged")

   ' Holds the query.
   Dim MyQuery As QueryDef

   ' Remove any existing queries.
   For Each MyQuery In CurrentDb.QueryDefs
      If MyQuery.Name = "MyQuery" Then
         CurrentDb.QueryDefs.Delete "MyQuery"
      End If
   Next

   ' Create the new query.
   Set MyQuery = _
      CurrentDb.CreateQueryDef("MyQuery", SQLText)

   ' View the results.
   DoCmd.OpenQuery "MyQuery"
End Sub
```

You might look at the code in Listing 8-10 and wonder why it doesn't use the simple `DoCmd.RunSQL` method. It's true that the `DoCmd.RunSQL` method is very powerful, and you'll probably want to rely on it when you can. Remember, however, that this method *executes* queries; therefore it doesn't work with `SELECT` queries. Since this method does use a `SELECT` query, you must use the rather roundabout method shown here to ultimately display the output shown in Figure 8-16.

The code begins by creating an SQL statement from the input the user provides. This is a standard `SELECT` statement. The `WHERE` clause defines the filtering that the user wants to see. Notice that the code surrounds the table name and selected field in square brackets; otherwise a space in either the table name or selected field name could cause the query to fail.

The code looks for an existing copy of `MyQuery` next. As with the tables, the query you create must have a unique name. In this case, you really don't want to build an application that generates a large quantity of queries, so deleting any existing query makes sense.

Once the code removes any existing query, it creates a new query using the `CurrentDb.CreateQueryDef()` method. The new query has the `MyQuery` name and contains the SQL query that the code created earlier. Now that the query exists, the code can use `DoCmd.OpenQuery` to open the query and display the results on-screen.

Chapter 9

Developing Business Applications for Outlook

In This Chapter

- Understanding how to work with Outlook
- Defining a mail-management tab
- Using user selections to manage incoming mail

Outlook (not to be confused with Outlook Express, which doesn't provide VBA or add-in capability) provides an interesting place in which to write custom applications because you can interact with Outlook at many levels. The applications you create for Outlook are always present. However, these applications might affect only one aspect of Outlook. For example, you could write an application to manage your calendar. The application would always be present, but you'd use it for calendar tasks only.

You'll also find that Outlook uses a mixed environment. It's not a true Ribbon environment, but it doesn't rely exclusively on the menu-and-toolbar system either. The main application still relies on a menu-and-toolbar setup. You use the same options as you always have to set application options and to tell Outlook about your accounts. The Ribbon interface appears when you attempt to perform tasks such as writing an e-mail.

Getting Started with Outlook Applications

Outlook doesn't provide any support for VBA when it comes to the Ribbon. That may seem like a big problem, but when you look at Outlook, you'll quickly find that many of the Outlook features don't actually rely on the Ribbon. For example, you can add and remove toolbars from the main Outlook interface, much as you always have using VBA. The difference for VBA users is that you can't use VBA to modify features such as the toolbar for new messages. When working with Outlook, some of your scripts will fail, but others won't.

To perform any major reconfiguration of Outlook, you need to rely on an add-in written using Visual Studio. As with all other add-ins in this book, you'll create an add-in class and a separate class for the Ribbon. Outlook actually presents very few surprises when it comes to working with add-ins.

You do need to consider how Outlook manages data. Unlike what happens in Word and Excel, you don't create documents as such in Outlook. What you do, instead, is create data that resides within a database. This single file contains everything that Outlook knows about your e-mail — everything from your address book, to your appointments, to the e-mails that you've collected. It pays to think of each kind of data as a separate table within the database. You'll use a different form to work with each kind of data. The following list summarizes how your old code behaves in Outlook 2007:

- `Explorer.CommandBars`: All of this code works precisely as before; you don't need to make any changes.
- `Inspector.CommandBars`: All of this code works, but Outlook places the entries on the Add-Ins tab. The placement depends on the kind of addition as follows:
  - `CommandBarControls` on a built-in menu: The resulting controls appear in the Menu Commands group of the Add-Ins tab.
  - `CommandBarControls` on a built-in toolbar: The resulting controls appear in the Toolbar Commands group of the Add-Ins tab.
  - `CommandBarControls` on a custom toolbar: The resulting controls appear in the Custom Toolbar group of the Add-Ins tab.
- `Word.CommandBars`: This code doesn't work at all. You must move the existing code from Word to Outlook. The code can use the `Inspector.WordEditor` object to return a Word `Document` object that you can use to provide the rich editing environment of Word within Outlook.

Because each of the data forms uses a different context in Outlook, you must perform extra processing within the `onLoad` handler for the Ribbon. You must determine whether the class calling your add-in is actually the class that requires the Ribbon feature. When the wrong class calls, you need to include code to disable the Ribbon functionality. The "Detecting the caller's class" section of the chapter provides details on this requirement.

Outlook also uses a slightly different means of accessing add-ins, compared to how other Office products do it. To remove an add-in you no longer want, choose Tools➪Trust Center. Select the Add-ins folder and you can now remove the add-in as you would in any other Office application.

Creating a Mail-Management Tab

Most corporate users receive a lot of e-mail. In fact, they often receive so much e-mail that it's very hard to manage it effectively. In some cases, the user ends up spending more time trying to figure out what to do with the e-mail, than how to answer it. Providing an organizational aid can help reduce the complexity of working with e-mail. This example looks at the task of putting the e-mail into the right location. A location might be an action file or a particular project file. The user might also need to make copies of it for multiple storage folders.

Interestingly enough, Microsoft provides this feature when you read an e-mail, but not when you respond to it. When you read an e-mail, you'll see Move to Folder as one of the buttons in the Actions group, but this group doesn't appear when you reply to a message. Consequently, your response can end up in the wrong place. This example provides similar functionality for storing responses so you can ensure that the response always appears where you need it.

This example demonstrates a considerable number of techniques, and the chapter simply can't hold all of the source code required to create it. The sections that follow do provide you with complete information about all of the essential techniques for working with this example and modifying it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `ManageMessage` folder.

Trying the default project

You create an Outlook add-in just as you would any other add-in, but the default project doesn't buy you very much. Outlook presents a very different programming problem for the developer from other Office products. The default project compiles and you can even start it in the debugger. Unfortunately, the results are anything but what you may have expected. Open a new message and you'll immediately discover that something is wrong. Figure 9-1 shows the default output.

At this point, you're probably asking yourself why the button doesn't appear in the right place — after all, it appears in a separate tab in the other default applications. Actually, it turns out that the default location for all Outlook add-ins is the Add-Ins tab. If you try to set the `<tab>` element to a custom tab, the tab won't appear at all within Outlook.

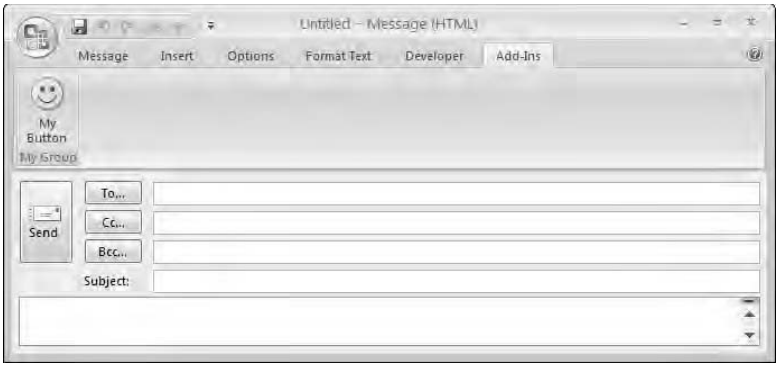


Figure 9-1: The default project output is less than inspiring.

You can change the new content to another tab if you want to: Simply change the `idMso` value to something like `TabNewMailMessage`. Now, when you create a new message, you'll see the new group and any associated buttons. However, the lack of new tabs means that you always have to select one of the existing tabs. This fact means you can't really create a new workflow for Outlook unless you want to grab the tab and clean everything else out before you populate it with new content.

Outlook developers also have another problem to overcome. As previously mentioned, some of the tabs you work with appear in more than one context. When you create the default example, the Add-Ins tab appears everywhere; you see the new group and button no matter what task you want to perform. If the button did anything more than display a message, the user could become quite confused. Imagine seeing a message-related group when you're working with a new appointment. Consequently, you always want to move your add-in from the Add-Ins tab to some other tab, and you want to monitor the class calling your add-in to ensure the code takes the appropriate action.

The problem becomes a little more complicated when you consider the control IDs that you have to work with in Outlook. The `2007OfficeControlIDsExcel2007` download described in the "Understanding the XML Connection" section of Chapter 2 contains 20 files for Outlook. While Word, Excel, Access, and PowerPoint have a single file each, it takes 20 files to provide all the control IDs for Outlook. It's important to look in the right file for the particular control ID you need. Fortunately, it's relatively easy to determine which file to open. For example, if you want to see the control IDs for a new mail message, you open the `OutlookMailComposeItemRibbonControls.xlsx` file. On the other hand, if you want to change the way the Ribbon appears when you read mail, open the `OutlookMailReadItemRibbonControls.xlsx` file instead. Make sure you open the correct file to obtain the right control IDs for your project.

Avoiding communication problems in Outlook

It's important to consider how Outlook elements communicate when you put your application together in order to avoid potential problems. In general, it's almost best to view the various elements — for example, creating an appointment based on the content of an e-mail — as two separate applications communicating together. This viewpoint means you can't assume certain things about the Outlook environment. For example, you shouldn't expect to share data as part of a variable in memory; instead, pass the data as part of the input to the other element (as part of a method call, for example). Passing data by value or reference, rather than as part of a variable, is always good programming practice, but Outlook tends to enforce this requirement.

The communication problems also mean there are no shortcuts you can use to obtain data from other parts of the application. For example, if you want to obtain an address from the address book, you use the same technique as if you were making the request outside of Outlook. In many cases, this means you're limited to only the data that Outlook makes publicly available. If you try to use internal data, you might be disappointed in the results. Always view Outlook as a collection of mini-applications that just happen to work together to make e-mail and time management possible.

Detecting the caller's class

You can limit the use of a particular Ribbon element in Outlook by choosing the correct tab for placement. For example, if you place a new group on `TabNewMailMessage`, it won't appear when you open a message for reading or create a new appointment. However, something else does happen — you'll see the error message shown in Figure 9-2.

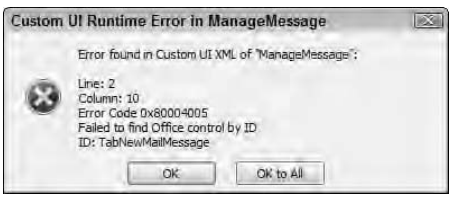


Figure 9-2: Using standard Ribbon development techniques in Outlook results in lots of error messages.

An Outlook application must always be aware of the class that is calling so it can provide the correct Ribbon handling. So far, you haven't had much reason to use the `ribbonID` value provided as part of the `GetCustomUI()` method in the Ribbon Support item you add to your add-in, but in Outlook, you must use it or face the fact your add-in is going to work poorly. Table 9-1 shows all of the message classes that Outlook supports, along with the `ribbonID` value your application receives.

Table 9-1 Outlook Ribbon IDs and Classes

Message Class	*Ribbon Identifier*
IPM.Activity.*	Microsoft.Outlook.Journal
IPM.Appointment.*	Microsoft.Outlook.Appointment
IPM.Contact.*	Microsoft.Outlook.Contact
IPM.DistList.*	Microsoft.Outlook.DistributionList
IPM.Note.*	Microsoft.Outlook.Mail.Read
IPM.Note.*	Microsoft.Outlook.Mail.Compose
IPM.Post.*	Microsoft.Outlook.Post.Read
IPM.Post.*	Microsoft.Outlook.Post.Compose
IPM.Post.Rss.*	Microsoft.Outlook.RSS
IPM.Report.*	Microsoft.Outlook.Report
IPM.Resend.*	Microsoft.Outlook.Resend
IPM.Schedule.Meeting.Request	Microsoft.Outlook.MeetingRequest.Send
IPM.Schedule.Meeting.Request or IPM.Schedule.Meeting.Canceled	Microsoft.Outlook.MeetingRequest.Read
IPM.Schedule.Meeting.Resp.*	Microsoft.Outlook.Response.Read
IPM.Schedule.Meeting.Resp.*	Microsoft.Outlook.Response.Compose
IPM.Schedule.Meeting.Resp.*	Microsoft.Outlook.Response.CounterPropose
IPM.Sharing.*	Microsoft.Outlook.Sharing.Read
IPM.Sharing.*	Microsoft.Outlook.Sharing.Compose
IPM.StickyNote.*	Not Implemented
IPM.Task.* and IPM.TaskRequest.*	Microsoft.Outlook.Task

You use these values to detect the request class and provide the correct Ribbon XML to the caller. Yes, you may even need to provide multiple XML files in your add-in to ensure you can provide the same group to multiple classes. Here's the code you'd need to ensure a Ribbon group appears only as part of the new message window:

```
public string GetCustomUI(string ribbonID)
{
   // If this is a new mail message, then display the
   // new group. Otherwise, don't display anything.
   if (ribbonID == "Microsoft.Outlook.Mail.Compose")
      return
         GetResourceText("ManageMessage.MailRead.xml");
   else
      return string.Empty;
}
```

Because your Outlook add-ins are dealing with specific classes, it's essential that you give your Ribbon Support items a meaningful name. Although you can write a complex Word or Excel application using a Ribbon Support item named Ribbon1, this naming convention won't work in Outlook. Giving the Ribbon Support item a meaningful name is an important part of the documentation process. Otherwise, other people viewing your code will have a hard time understanding what part of Outlook the Ribbon addition affects.

Designing the filing interface

The filing interface requires a location and a button to perform the filing as a minimum. This example also includes the option to save the current e-mail as a draft. When the user checks this option, the application saves the current e-mail in the Drafts folder and then closes the message. Otherwise you can file the e-mail in as many places as needed. Figure 9-3 shows the interface for this example.

Notice that the Filing Options group appears at the end of the Ribbon. This position is natural for this example because the user will normally file the e-mail after completing every other task with it. In some cases, you can still maintain a good workflow in Outlook applications despite the limitations that Microsoft places on good design. Listing 9-1 shows the XML for the interface.

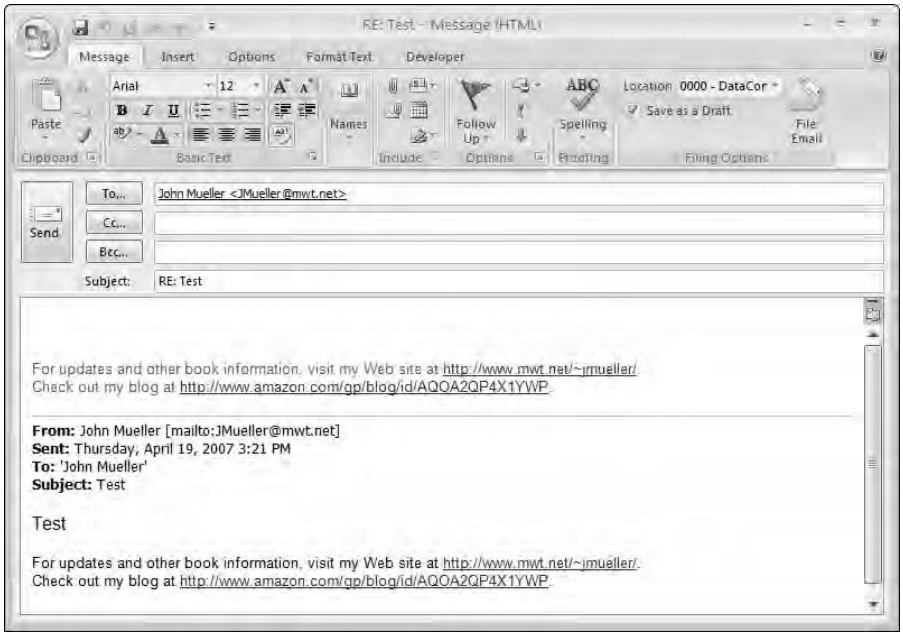


Figure 9-3: The new filing feature appears in the Filing Options group.

Listing 9-1: Creating an Interface for Filing E-mail

```
<tab idMso="TabNewMailMessage">
  <group id="FilingOptions"
         label="Filing Options">
     <comboBox id="Location"
               label="Location"
               getItemCount="LocationCount"
               getItemID="LocationID"
               getItemLabel="LocationLabel"
               onChange="LocationChange"
               getText="LocationText"/>
     <checkBox id="CloseEmail"
               label="Save as a Draft"
               getPressed="ClosePressed"
               onAction="CloseClicked"/>
     <button id="FileIt"
             label="File Email"
             size="large"
             imageMso="CreateMailRule"
             onAction="FileIt_Clicked"/>
  </group>
</tab>
```

The code provides maximum user convenience by including both the `onChange` and `getText` options for the Location combo box. The `getPressed` and `onAction` attributes provide the same functionality for the `CloseEmail` check box. In both cases, the application retains the user's last selections and makes them available for the next application use. The callbacks simply make use of a global variable to store the current setting. The code either stores or returns the required value, as shown here, for the `Location` control:

```
public void LocationChange(
   Office.IRibbonControl control, String text)
{
   // Save the new folder selection.
   SelectedFolder = text;
}

public String LocationText(
   Office.IRibbonControl control)
{
   // Return the current folder selection.
   return SelectedFolder;
}
```

In all other ways, the example uses the standard, required, attributes. For example, in order to make the combo box functional, you must include the `getItemCount`, `getItemID`, and `getItemLabel` attributes.

Obtaining the folder list

Outlook uses a hierarchical folder structure. Normally, Outlook contains a single top-level folder named Personal Folders. (Unless you're using Exchange Server — then it's Mailbox - <user name>.) However, when you open another e-mail file, Outlook normally creates a second top-level folder for it. Consequently, Outlook could have as many top-level folders as needed to hold all of the open e-mail files. The default e-mail file always appears as the first item in the top-level folder list.

Within the top-level folder are all of the child folders associated with standard tasks such as the Inbox, Deleted Items, and Sent Items folders. The children also include special folders such as Notes, Calendar, and Tasks. Any first-level, user-created folders also appear as children of the top-level folder. The hierarchy continues as you see it in Outlook. This example uses only the first-level folders — those that appear as children of the top-level folder. You can create as complex a hierarchy as needed to store everything that Outlook supports.

The folder list appears within the Location combo box. However, you must obtain it in the `ThisAddIn` class. Consequently, the Ribbon callback simply makes a call to `Globals.ThisAddIn.GetFolderList()` to obtain a list of folders. Since calling `GetFolderList()` multiple times would incur a performance penalty, the actual call appears in the constructor and the `LocationLabel` callback relies on a global array to obtain the desired information. Listing 9-2 shows the code used to obtain both a count of folders and the folder list.

Listing 9-2: Creating a List of Folders

```
public Int32 GetFolderCount()
{
   // Return the number of folders.
   return Application.Session.Folders[1].Folders.Count;
}

public String[] GetFolderList()
{
   // Obtain the list of folders.
   Outlook.Folders FolderList =
      Application.Session.Folders[1].Folders;

   // Holds the output list of names.
   String[] FolderNames = new String[FolderList.Count];

   // Place the names in the array.
   for (int I = 1; I <= FolderList.Count; I++)
      FolderNames[I - 1] = FolderList[I].Name;

   // Return the name list.
   return FolderNames;
}
```

The `GetFolderCount()` method simply returns the `Application.Session.Folders[1].Folders.Count` property value. The first use of `Folders` represents the top-level folder. The second use of `Folders` represents the first-level folders you want to add to the list. You use another `Folders` object for each level of folders that you want to obtain. The index array always points to the parent object that you want to work with. Normally you'll use a value of 1 for the top-level array so that you work with the default Outlook e-mail file.

The `GetFolderList()` method begins by creating a reference to the list of first-level folders. The only reason to take this approach is to reduce the amount of typing you have to perform. The next step is to create a local

`String` array (`FolderNames`) of sufficient size to hold all of the entries. You can't perform this task during design time because it's impossible to know how many folders a particular Outlook file will have.

After the code creates the `String` array, it uses a `for` loop to fill the array with the folder names. It then returns this array to the caller for display on the Ribbon.

Creating the copy

At some point, the user will click File Email. At this point, the code has to determine a course of action based on the user selections. Listing 9-3 shows the code required for this task.

Listing 9-3: Determining the Filing Actions

```
public void FileIt_Clicked(
   Office.IRibbonControl control)
{

   // Save a copy of the file in the required location.
   Globals.ThisAddIn.CopyEmail(SelectedFolder);

   // Save a draft of the message when required.
   if (SelectClose)
      Globals.ThisAddIn.SaveDraft();

}
```

Notice that you need to provide the `SelectedFolder` (the folder name that currently appears in the Location combo box) only when copying the e-mail. A draft e-mail always appears in the Drafts folder, so you don't have to worry about a location in this case. Whenever the user clicks File Email, the code copies the e-mail first. If the user also checks Save as a Draft, the code also saves the draft. However, saving a draft closes the e-mail. Consequently, you must always make the copy first, and then save the draft. Otherwise the copying process fails because there isn't anything to copy (the e-mail is closed).

At this point, the code can perform the actual copying process. Listing 9-4 shows how the copying process works.

Listing 9-4: Copying the E-mail

```
public void CopyEmail(String Location)
{
   // Obtain the current email.
   Outlook.MailItem ThisMail =
(Outlook.MailItem)Application.ActiveInspector().CurrentIte
          m;

   // Create the MAPI folder.
   Outlook.MAPIFolder DestFolder = null;

   // Discover the subpath.
   foreach (Outlook.Folder SubPath in
            Application.Session.Folders[1].Folders)
      if (SubPath.Name == Location)
         DestFolder = SubPath;

   // Save a copy of the message.
   Outlook.MailItem NewMail =
      (Outlook.MailItem)ThisMail.Copy();
   NewMail.Move(DestFolder);
}
```

The code begins by obtaining a reference to the current e-mail. Notice that you must coerce the object to the correct type because Outlook returns an `Object`, not an `Outlook.MailItem`.

The next step is to create a destination for the e-mail based on the user's selection. You can't use the user's selection directly, you must convert it to an `Outlook.MAPIFolder` instead. The code relies on a simple `foreach` loop to look through the first-level folders. When it finds a match, it copies the entire folder to `DestFolder`.

After the code obtains a destination for the e-mail, it must create a copy of the existing e-mail (`NewMail` in this case). If you try to move the existing e-mail to a new folder, the system will comply, but then it won't appear in the original folder. The code completes the task by using the `Move()` method of the `NewMail` object to move the copy to the location the user specified.

Saving as a draft

There are many times when a user doesn't have time to complete a particular e-mail immediately. Although Outlook provides a method for saving the e-mail as a draft, adding it as part of the Filing Options group can make the task easier. Listing 9-5 shows the code for this task.

Listing 9-5: Saving a Message Draft

```
public void SaveDraft()
{
   // Obtain the current email.
   Outlook.MailItem ThisMail =
(Outlook.MailItem)Application.ActiveInspector().CurrentIte
          m;

   // Save the message draft and close the file.
   ThisMail.Close(Outlook.OlInspectorClose.olSave);
}
```

The code begins by obtaining a reference to the current e-mail. In this case, it works directly with the e-mail, rather than creating a copy first. The application creates a draft save by using the `ThisMail.Close()` method with the `Outlook.OlInspectorClose.olSave` option.

Processing Incoming Mail Based on User Selections

Users often have a need to perform tasks other than simply organizing their e-mail. For example, you might receive an e-mail and need to use it as a basis for a meeting later. Unfortunately, many users have to rely on cut and paste to create the meeting information, and often don't get all of the details. Sometimes a task or meeting entry requires input from multiple e-mails, making the task even harder. The example in this section reduces the need to move around trying to create tasks based on an e-mail. Instead, the user highlights the text associated with the task and clicks a few buttons to add a new task to the Tasks list. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `CreateTask` folder.

Considering multiple Outlook class issues

As described in the "Detecting the caller's class" section, Outlook add-ins can affect one or more classes. Now, here's where things can get very interesting. Let's say you have a group that you want to appear for both reading and composing messages. You can't simply send the same XML file to both windows and expect it to work because the tabs you attach the groups to are different for each window. The solution to this problem is to create a separate XML file for each of the tabs and then send the appropriate XML to the tab. The following steps describe how to use multiple XML files in a single add-in.

1. **Right-click the project entry in Solution Explorer and choose Add➪ New Item from the context menu.**

   Visual Studio displays the Add New Item dialog box shown in Figure 9-4.

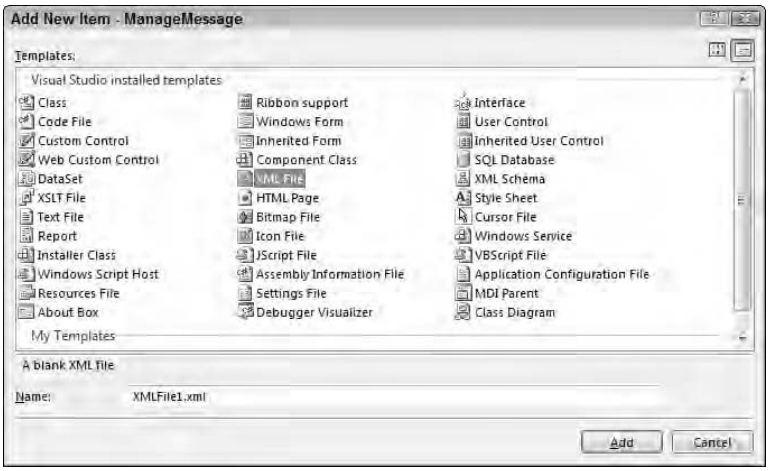


Figure 9-4: The Add New Item dialog box contains entries for XML files.

2. **Highlight the XML File entry.**

3. **In the Name field, type a meaningful name (such as** TaskEdit**) for the XML file and click Add.**

   Visual Studio adds the new XML file to Solution Explorer and opens it for you to edit.

4. **Delete the** `<?xml version="1.0" encoding="utf-8" ?>` **processing instruction.**

5. **Type** `<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui" onLoad="OnLoad">`**.**

   Look at how Visual Studio formats the XML file when you add the Ribbon Support item to a project. The file doesn't include the XML-processing instruction, but it does include the `<customUI>` element.

6. **Click the ellipsis in the Schemas field of the Properties dialog box.**

   You see the XSD Schemas dialog box shown in Figure 9-5.

7. **Locate the CustomUI.xsd file show in Figure 9-5 and add it to your project.**

8. **Click OK.**

   Visual Studio adds the appropriate IntelliSense to your Ribbon XML file. Make sure you follow all of the standard Ribbon XML conventions as you create the new XML file.

Figure 9-5: Add the appropriate XSD file to the new Ribbon XML file.

9. **Highlight the new XML file in Solution Explorer.**
10. **Select Embedded Resource in the Build Action property found in the Properties window.**

    The new XML file is now ready for inclusion in your project.

Simply creating the XML file and embedding it in your project isn't sufficient to make it useable. You also have to provide code in the `GetCustomUI()` method. Listing 9-6 provides a typical example of loading the appropriate Ribbon XML for a particular class.

Listing 9-6: Making Multiple XML Files Access to Outlook

```
public string GetCustomUI(string ribbonID)
{
   // Determine an action based on the Outlook class.
   switch (ribbonID)
   {
      // If the user is reading a message, then display
      // the new group.
      case "Microsoft.Outlook.Mail.Read":
         return GetResourceText(
            "ManageMessage.MailRead.xml");

      // If this is a new mail message, then display
      // the new group.
```

(continued)

Listing 9-6 ***(continued)***

```
        case "Microsoft.Outlook.Mail.Compose":
           return GetResourceText(
              "ManageMessage.MailCompose.xml");

        // When the user performs a task-related task,
        // display the new group.
        case "Microsoft.Outlook.Task":
           return GetResourceText(
              "ManageMessage.TaskAdd.xml");

        // Otherwise, don't display any special Ribbon
        // features.
        default:
           return string.Empty;
     }
  }
```

The code begins by determining which class requires input. When the class doesn't appear as part of the `switch` statement, the add-in ignores it. Otherwise the code returns the correct XML for the class in question, using the `GetResourceText()` method. For example, when the user opens a task, the code sees the `Microsoft.Outlook.Task` Ribbon identifier associated with the `IPM.Task.*` and `IPM.TaskRequest.*` classes, and it returns the `ManageMessage.TaskAdd.xml` file.

Of course, this code doesn't answer the question of what happens to your special group and buttons. The problem is figuring out what to do when you want the same group to appear in several locations, but you don't want to write the code three separate times. It turns out that the separate XML files don't hinder you in any way. If you have a particular callback in one XML file, it executes just the same as when it appears in another XML file. Consequently, if you have a button named Click Me with a callback of `ClickMe_Handler` in the `MailCompose.XML` file, you don't have to write anything new for the hander in the `MailRead.XML` file.

Unfortunately, the lack of special differentiation can also cause problems. You might want to perform a little special processing based on the class that issued the callback. Outlook doesn't provide any special information to handle this situation, so the `tag` attribute comes into play here. Say you're working with the default application again and have placed the group on the first tab of whatever class you want to work with. The example for using that tag might appear like this:

```
<toggleButton id="toggleButton1"
              size="large"
              label="My Button"
              screentip="My Button Screentip"
              onAction="OnToggleButton1"
              imageMso="HappyFace"
              tag="MailRead"/>
```

In this case, the `tag` attribute identifies the XML file as `MailRead.xml`. You could then process the tag attribute in the corresponding callback like this:

```
public void OnToggleButton1(
   Office.IRibbonControl control, bool isPressed)
{
   if (isPressed)
      MessageBox.Show("Pressed", control.Tag);
   else
      MessageBox.Show("Released", control.Tag);
}
```

The name of the XML file (and therefore the XML class) appears in the `control.Tag` property. You'll see the name in the title bar of the message box that appears when the user clicks the button. Because you know precisely which class issued the callback, you can now perform any required special processing based on class.

Designing the task-creation interface

The idea of taking selected text from an e-mail and turning into a task entry is compelling. It makes an error-prone task very simple. The example doesn't provide all of the bells and whistles that some people will want, but it does do the job. You could probably implement additional features, such as setting the start and due dates as part of a dialog-box launcher feature. Figure 9-6 shows the interface for the example. The new Create a Task group contains a button, several check boxes, and a combo box you can use to set up a task based on the highlighted text in the message.

One feature you should notice about this example is that the button is on the left side of the selections. That's because the selections are optional. The user can simply highlight the pertinent text and click Create Task if desired. Now compare this with the example shown in Figure 9-3. In this case, the button appears to the right of the options. That's because the Location option is required in this case; the user must select a Location prior to clicking File Email. The Ribbon is a good tool, but you need to consider how the user is going to interact with the controls. These examples show two cases where the user must click a button, but the positioning of the button changes to reflect the importance of the options.

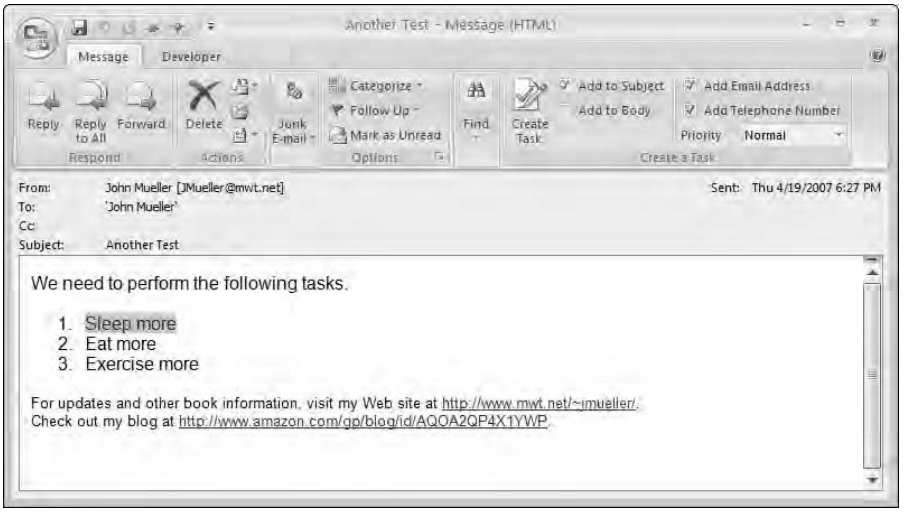


Figure 9-6: Creating tasks from e-mails is easy using this add-in.

As previously noted, this example adds the group shown in Figure 9-6 to the Message tab when the user either composes or reads a message. Consequently, the application code actually contains two XML files with slightly different content. Listing 9-7 shows typical code for the interface shown in Figure 9-6.

Listing 9-7: Defining a Task-Creator Interface

```
<tab idMso="TabNewMailMessage">
  <group id="CreateTaskGroup"
         label="Create a Task">
    <button id="CreateTask"
            label="Create Task"
            size="large"
            imageMso="ReviewAcceptChange"
            onAction="CreateTaskClicked"/>
    <checkBox id="AddSubject"
              label="Add to Subject"
              getPressed="AddSubjectPressed"
              onAction="AddSubjectClicked"/>
    <checkBox id="AddBody"
              label="Add to Body"
              getPressed="AddBodyPressed"
              onAction="AddBodyClicked"/>
    <separator id="Separator1"/>
    <checkBox id="AddEmail"
              label="Add Email Address"
```

```
                getPressed="AddEmailPressed"
                onAction="AddEmailClicked"/>
      <checkBox id="AddTelephone"
                label="Add Telephone Number"
                getPressed="AddTelephonePressed"
                onAction="AddTelephoneClicked"/>
      <comboBox id="AddPriority"
                label="Priority"
                getText="AddPriorityText"
                onChange="AddPriorityChanged">
         <item id="PriorityLow" label="Low"/>
         <item id="PriorityNormal" label="Normal"/>
         <item id="PriorityHigh" label="High"/>
      </comboBox>
   </group>
</tab>
```

Many of the features in this Ribbon XML have appeared in other examples. For example, you always include an `onAction` attribute for buttons if you want them to perform any useful work. The check boxes also include the `getPressed` and `onAction` attributes as normal, along with the normal code behind. (See the explanation in the "Designing the filing interface" section of the chapter for details.)

The most significant difference, in this case, is that the `<comboBox>` element has the `<item>` elements defined, so you don't need callback to populate the `<comboBox>` in this case. Make sure you use predefined entries whenever you can to reduce application complexity.

Defining the task

Before you can do anything with the task, it's important to define a procedure for creating one. Listing 9-8 contains the Ribbon callback that handles creating a task when the user clicks Create Task.

Listing 9-8: Defining the Task Features

```
public void CreateTaskClicked(Office.IRibbonControl
          control)
{
   // Obtain the email subject.
   String Subject =
      Globals.ThisAddIn.GetEmailSubject();

   // Obtain the email text.
```

(continued)

Listing 9-8 *(continued)*

```
    String Body =
       Globals.ThisAddIn.GetEmailSelectedText();

    // Add the selected text to the subject if
    // necessary.
    if (AddSubject)
       Subject = Subject + " - " + Body;

    // Use the whole body when requested.
    if (AddBody)
       Body = Globals.ThisAddIn.GetWholeBody();

    // Create the task.
    Globals.ThisAddIn.CreateATask(
       Subject, Body, AddPriority);

    // Add the Email address to the body if asked.
    if (AddEmail)
       Globals.ThisAddIn.AddEmailToBody();

    // Add the telephone number when asked.
    if (AddTelephone)
       Globals.ThisAddIn.AddTelephoneToBody();

    // Make sure you close the task when finished.
    Globals.ThisAddIn.CloseTask();
}
```

Even though the code calls a number of `Globals.ThisAddIn` methods, it provides the structure required to make the task creation flexible. The user has a number of options, all of which appear as part of the Ribbon, so this is the best place to make the content decisions. Every task must have a subject, body, and priority, so the code begins by obtaining the information for those three elements. The subject and body come from the e-mail, while the priority is a Ribbon selection. Notice that the body can contain just the selected text or the entire contents of the e-mail message.

The whole subject and e-mail body are relatively easy to get. Listing 9-9 shows that you can perform this task with just a few lines of code.

Listing 9-9: Getting the Subject and Entire Body

```
public String GetEmailSubject()
{
    // Obtain the current email.
    Outlook.MailItem ThisMail =
```

```
(Outlook.MailItem)Application.ActiveInspector().
          CurrentItem;

   // Return the subject.
   return ThisMail.Subject;
}

public String GetWholeBody()
{
   // Obtain the current email.
   Outlook.MailItem ThisMail =
(Outlook.MailItem)Application.ActiveInspector().
          CurrentItem;

   // Return everything.
   return ThisMail.Body;
}
```

In both cases, the code begins by creating a reference to the e-mail. It then returns the appropriate property value from the e-mail to the caller. In the first case, the code uses the `Subject` property; in the second case, it uses the `Body` property.

Microsoft doesn't make it easy to obtain the selected text in an e-mail. Part of the problem is that you have to discover the editor used to create the e-mail before you can do anything else. To provide even basic functionality, you must include both an HTML and a Word editor for Outlook. You can use the Word editor for text and Rich Text Format (RTF) messages as well.

Before you can work with either Word or HTML, you must create references for them. This example uses the `Microsoft.mshtml` library for the HTML editor and the Microsoft Word 12.0 Object Library for the Word editor. It helps if you include using statements at the beginning of the `ThisAddIn.cs` file, as shown here:

```
using mshtml;
using Word = Microsoft.Office.Interop.Word;
```

Once you have the references in place, you can begin coding. Listing 9-10 shows a typical example of how to obtain selected text from using an editor. You can also use this technique for a wide range of other purposes. For example, by creating your own selections (using the objects appropriate for the editor in use), you can add highlighting and other special effects to your e-mail messages.

Listing 9-10: Obtaining the Selected Text

```
public String GetEmailSelectedText()
{
   // Obtain the current email.
   Outlook.MailItem ThisMail =
(Outlook.MailItem)Application.ActiveInspector().
           CurrentItem;

   // Create an HTML document for the HTML editor.
   mshtml.HTMLDocument ThisHTMLDoc;

   // Create a Word document for the Word editor.
   Word.Document ThisWordDoc;

   // Holds the selected text.
   String SelText = "";

   // Look for an HTML document.
   if (ThisMail.GetInspector.EditorType ==
       Outlook.OlEditorType.olEditorHTML)
   {
      // Get the selected text.
      ThisHTMLDoc =
(mshtml.HTMLDocument)ThisMail.GetInspector.HTMLEditor;
      SelText = ThisHTMLDoc.selection.ToString();
   }

   // Look for a Word document.
   if (ThisMail.GetInspector.EditorType ==
       Outlook.OlEditorType.olEditorWord)
   {
      // Get the selected text.
      ThisWordDoc =
       (Word.Document)ThisMail.GetInspector.WordEditor;
      SelText = ThisWordDoc.Application.Selection.Text;
   }

   // Return the selected text.
   if (SelText.Length > 0)
      return SelText;
   else
      return ThisMail.Body;
}
```

The code begins by creating some basic objects. It obtains a reference to the current e-mail. It then creates two documents — one for an HTML document and a second for a Word document — because you can't tell which editor the current e-mail uses. The `SelText` String contains the output from the editor if the user has selected any text and the application can detect the editor type.

The key to detecting the editor type is to check the `ThisMail.GetInspector.EditorType` property. This is an enumerated value that

can detect plaintext, RTF, HTML, and Word documents. When the document is an HTML document, the code begins by creating the `ThisHTMLDoc` object. Notice that the code uses the `ThisMail.GetInspector.HTMLEditor` object to perform this task. When you create a Word, plaintext, or RTF document, you use the `ThisMail.GetInspector.WordEditor` object instead.

In both cases, the code uses the selection features of the editor to obtain the selected text. In the case of the HTLM editor, the `ThisHTMLDoc.selection.ToString()` method accomplishes the task. The Word editor relies on the `ThisWordDoc.Application.Selection.Text` property. If the `GetEmailSelectedText()` method fails to obtain the selected text (there are many causes for that; you can't write error-correcting code for them all), the method returns the entire e-mail body using the `ThisMail.Body` property. Even though the user will see the entire body, at least the application won't fail.

At this point, the code has retrieved a subject and a body. It's time to create the task. Listing 9-11 shows how to create a basic task.

Listing 9-11: Creating a Basic Task Entry

```
public void CreateATask(
   String Subject, String Task, String Priority)
{
   // Create the task.
   ThisTask = (Outlook.TaskItem)Application.CreateItem(
      Outlook.OlItemType.olTaskItem);

   // Add the subject and body.
   ThisTask.Subject = Subject;
   ThisTask.Body = Task;

   // Set the priority.
   switch (Priority)
   {
      case "Low":
         ThisTask.Importance =
            Outlook.OlImportance.olImportanceLow;
         break;
      case "Normal":
         ThisTask.Importance =
            Outlook.OlImportance.olImportanceNormal;
         break;
      case "High":
         ThisTask.Importance =
            Outlook.OlImportance.olImportanceHigh;
         break;
      default:
         ThisTask.Importance =
            Outlook.OlImportance.olImportanceNormal;
         break;
   }
}
```

The code begins by obtaining a reference to a new task. It performs this task by calling on the `Application.CreateItem()` method. You must define the kind of object to create using the `Outlook.OlItemType.olTaskItem` enumeration. Because this technique returns an `Object` in all cases, you must always coerce the return value into the correct data type.

Adding the subject and body consists of providing the information sent by the caller. The priority requires special handling. You must use a `switch` to convert the string into the correct `Outlook.OlImportance` enumerated value and place it within the `ThisTask.Importance` property.

Adding supplemental information

The task isn't completed until you close it. Consequently, you can add any amount of supplemental information to a basic task to refine the task information. The example provides two basic pieces of information as part of the task body. The first is the sender's e-mail; the second is a telephone number. Listing 9-12 shows how to add this information to the task.

Listing 9-12: Defining the Supplemental Information

```
public void AddEmailToBody()
{
   // Obtain the current email.
   Outlook.MailItem ThisMail =
(Outlook.MailItem)Application.ActiveInspector().
           CurrentItem;

   // Obtain the email of the sender.
   String SenderEmail = ThisMail.SenderEmailAddress;

   // Add the email address to the body of the task.
   ThisTask.Body = ThisTask.Body +
                   "\r\nEmail: " + SenderEmail;
}

public void AddTelephoneToBody()
{
   // Obtain the current email.
   Outlook.MailItem ThisMail =
(Outlook.MailItem)Application.ActiveInspector().
           CurrentItem;

   // Get the sender's name.
   String SenderName = ThisMail.SenderName;
```

```
    // Obtain the name from the contact database.
    String UserTelephone = "";
    foreach (Outlook.AddressEntry ThisAddress in
     Application.Session.AddressLists[1].AddressEntries)
       if (ThisAddress.Name == SenderName)
          UserTelephone =
             ThisAddress.GetContact().
                BusinessTelephoneNumber.ToString();

    // Check the telephone number.
    if (UserTelephone.Length > 0)
       ThisTask.Body = ThisTask.Body +
          "\r\nTelehone Number: " + UserTelephone;
}
```

In both cases, the code begins by obtaining a reference to the current e-mail. You might be surprised at the amount of information you can derive from a simple-looking e-mail. In this case, the code can obtain the sender's e-mail address directly using the `ThisMail.SenderEmailAddress` property. Adding the information to the body of the task is simply a matter of concatenating the appropriate strings.

E-mails don't come with telephone numbers, in most cases, unless someone sends one on a business card or as part of a signature. The example assumes that the sender hasn't provided this option. In this case, the code relies on the sender's name (found in the `ThisMail.SenderName` property).

The code then relies on a `foreach` loop to look through each of the address entries in the user's contact database. The information appears in the `AddressEntry` as part of the `ThisAddress.GetContact().BusinessTelephoneNumber` property.

Closing the task

It's finally time to close the task. Up until this point, you can continue to add as much or little information as you want to the task. Once the task is closed, you have to obtain a reference to it again and reopen it before you can do any work. Fortunately, closing the task requires a single call, as shown here:

```
public void CloseTask()
{
    // Close the task.
    ThisTask.Close(Outlook.OlInspectorClose.olSave);
}
```

Notice that you must provide one of the `Outlook.OlInspectorClose` enumerated values. You can choose to discard the task, prompt for a save, or save the task automatically. The code uses an automatic save. Figure 9-7 shows the output from this example, based on the selections shown in Figure 9-6.

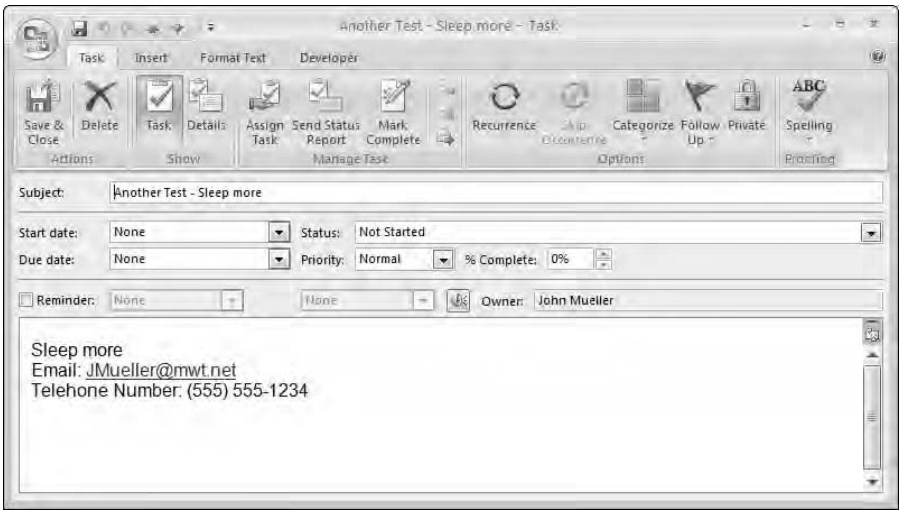


Figure 9-7: The output of this task shows its origins in the e-mail message.

Chapter 10

Developing Business Applications for PowerPoint

In This Chapter

- Understanding how to work with PowerPoint
- Defining custom-feature tabs for presentations
- Designing the initial slide
- Considering the use of optional slides
- Providing constants for the user
- Ending the presentation
- Saving and using a template

Like Word and Excel, PowerPoint uses the Ribbon exclusively — you won't find any menu-and-toolbar holdovers in this application. In addition, when you're working with PowerPoint, you have access to all three levels of Ribbon customization: add-in, template, and document. In fact, PowerPoint provides a broad range of flexibility when it comes to the Ribbon; PowerPoint may actually provide the simplest Ribbon programming experience in some respects because the documents it creates are straightforward.

Of course, you still have a considerable number of tasks you can perform with PowerPoint. The fact that it provides a straightforward document setup actually makes it easier to customize this application; this setup also provides opportunities to speed up the user experience. All these advantages work to the user's favor because most users don't want to spend a lot of time learning PowerPoint features. Rather, they know that they must get a presentation created and that the presentation is usually due in a hurry.

The following sections explore PowerPoint functionality that can help just about anyone who needs to create a presentation quickly. How much you customize your presentation depends on your users, of course. If you're working with engineering staff, providing a feature to look for keywords within slides is going to be very helpful because engineers tend to use complex terms. Likewise, if you work for a large company, creating presentations

with a precise look and feel is going to be a concern, which makes the Custom Presentation tab very useful. This chapter contains a little something for everyone.

Getting Started with PowerPoint Applications

PowerPoint provides a standardized approach to working with the Ribbon. Like other Office products, PowerPoint provides both standard and macro-enabled versions of the documents and templates it creates. To work with the Ribbon in VBA, you must save a document using the `PPTM` extension. Likewise, if you want to save a template, you must use the `POTM` extension.

Unlike many other Office applications, the focal point of PowerPoint is the template. You'll create many presentations using the same basic layout, but it's unlikely that you'll rework a single presentation for more than one topic. After you finish a particular presentation, you might tweak it a little here or there, but presentations aren't like Word documents or Excel workbooks (in which you can reuse a single document to perform more than one task). It's very likely that you'll create more templates than documents; you may never have to create a document with a modified Ribbon for PowerPoint.

As with most other Office applications, it's often easier to create a document in PowerPoint first, and then save the document as a template when the document contains all the features you need to create a class of documents. The VBA examples in this chapter rely on the document-to-template technique because it's the most practical method of creating a template. Some people do create templates directly, though, so you might want to try both approaches to see which one works best for you.

When you're working with an add-in (created using Visual Studio and the C# or VB.NET language), you can save the output in any format. The format you choose depends on whether the document contains any VBA. Always choose the more secure non-macro-enabled versions of the file whenever possible. These safer formats use the `PPTX` extension for documents and the `POTX` extension for templates. Avoid using macro-enabled document formats whenever possible to improve security for your system as a whole.

You can also save documents as PowerPoint shows. However, in this case, changes to the Ribbon won't be of the presentation creation sort. Instead, you might make changes for the benefit of the PowerPoint show viewer. Because the controls that PowerPoint provides are adequate for viewing a presentation, it's unlikely that you'll ever need to change the Ribbon for this file format (which is why the chapter doesn't discuss such a change).

Choosing between the Ribbon and the template

Presentations have a formal style that is unlike other kinds of Office documents. The decision between what you include on the Ribbon and what you include as part of the template is more precise. Some elements, such as the heading that appears on every slide, belong in the template, even if you modify these elements later with optional features on the Ribbon. Likewise, content features always appear on the Ribbon when you can define them during design time.

The keyword for the Ribbon in PowerPoint is optional. Normally, you include only optional information as part of the Ribbon. All of the mandatory information appears as part of the template. This approach differs from an application such as Word. Even though the recipient address is a required part of a letter, you don't include it in the template because the information changes every time you write another letter. Contrast this content with the header for a presentation that includes your company name and logo. Because the company name and logo won't change, including this content as part of the template makes perfect sense.

Defining the Custom Presentation Tab Interface

Every presentation includes some piece of required content that changes according to presenter or other criteria. For example, every presentation should include an opening slide with the name of the presentation, the presenter's name, and the contact information for the presenter. Even though the format and layout of this information don't change, the actual content does, so having this material as a Ribbon feature works well. The example includes these required, but changeable, content items. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `Custom Presentation.potm` file.

The interface you design for your application requires a great deal of thought, especially when you want to enforce corporate policies and make it simple enough for all of your users to work with. The default PowerPoint interface assumes that the user is knowledgeable in creating presentations and has no corporate guidelines to hinder creativity. Consequently, the sample application removes some features, consolidates others, and creates new elements to meet the needs of this specific user. Figure 10-1 shows the overall layout of this application.

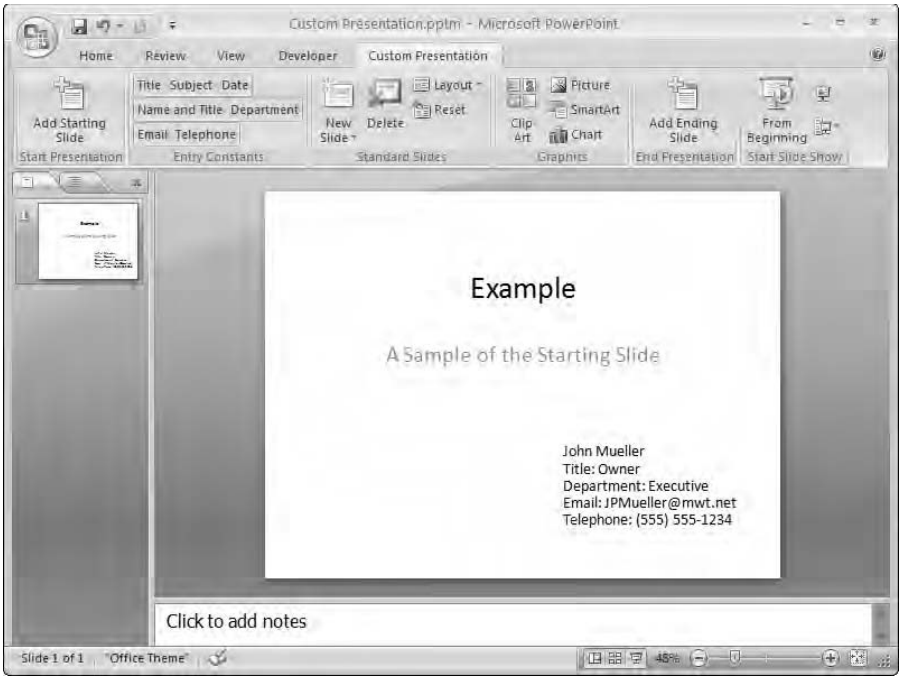


Figure 10-1: The interface for the Custom Presentation tab includes optional content items.

The new features all appear on the Custom Presentation tab. Even though this layout is basically a workflow layout, the user will need to move back and forth between some of the groups. For example, after creating a new slide using the options in the Standard Slides group, the user will provide content for the slides using the items in the Entry Constants group. Listing 10-1 shows the code used to create the Custom Presentation tab.

Listing 10-1: Defining the Custom Presentation Interface

```
<tab id="CPresent" label="Custom Presentation">
   <group id="Start" label="Start Presentation">
      <button id="StartSlide"
              label="Add Starting Slide"
              size="large"
              imageMso="QueryShowTable"
              onAction="StartSlideClick" />
   </group>
   <group id="Constants" label="Entry Constants">
      <buttonGroup id="PresentationIdentification">
         <button id="PresentTitle"
                 label="Title"
                 onAction="PresentTitleClick" />
         <button id="PresentSubject"
```

```
                label="Subject"
                onAction="PresentSubjectClick" />
         <button id="PresentDate"
                label="Date"
                onAction="PresentDateClick" />
      </buttonGroup>
      <buttonGroup id="PresenterID">
         <button id="PresentName"
                label="Name and Title"
                onAction="PresentNameClick" />
         <button id="PresentDept"
                label="Department"
                onAction="PresentDeptClick" />
      </buttonGroup>
      <buttonGroup id="PresenterContact">
         <button id="PresentEmail"
                label="Email"
                onAction="PresentEmailClick" />
         <button id="PresentTel"
                label="Telephone"
                onAction="PresentTelClick" />
      </buttonGroup>
   </group>
   <group id="AddSlide" label="Standard Slides">
      <gallery idMso="SlideNewGallery" size="large" />
      <button idMso="SlideDelete" size="large" />
      <gallery idMso="SlideLayoutGallery" />
      <button idMso="SlideReset" />
   </group>
   <group id="GraphicContent" label="Graphics">
      <toggleButton idMso="ClipArtInsert"
                    size="large"/>
      <button idMso="PictureInsertFromFilePowerPoint"/>
      <button idMso="SmartArtInsert" />
      <button idMso="ChartInsert" />
   </group>
   <group id="End" label="End Presentation">
      <button id="EndSlide"
              label="Add Ending Slide"
              size="large"
              imageMso="QueryShowTable"
              onAction="EndSlideClick" />
   </group>
</tab>
```

All of the action features of this example rely on pre-existing controls or on buttons. For example, look at the `GraphicContent` group and you'll see that all of the controls are pre-existing controls that the example repurposes for this workflow. Although it may seem like cheating, using pre-existing controls whenever you can makes sense; doing so reduces development time. The Ribbon makes it easy to move controls from anywhere to anywhere else.

The buttons rely on the usual XML. The larger buttons include the `size`, `imageMso`, and `onAction` attributes, in addition to the standard `id` and `label` attributes. Smaller buttons require only the `id`, `label`, and `onAction` attributes.

Notice how the application uses the `<buttonGroup>` element to organize the buttons. Using the `<buttonGroup>` element can help you imply functionality to the user. In this case, the various `<buttonGroup>` elements show

- Presentation information
- Presenter identification
- Presenter contact information

If you've used PowerPoint 2007, you'll immediately notice that this interface is significantly simplified from the standard product. The Insert, Design, Animations, and Slide Show tabs are gone. The Developer tab should be missing in action too, but I left it in place to make it easier for you to work with the example. The View tab is still in place, but many of its features are gone as well, as shown in Figure 10-2. (You'll also notice that the Drawing group is missing from the Home tab.) What remains is a significantly simplified interface that won't allow the user to make most changes to the interface that interfere with the creation of corporate approved presentations.

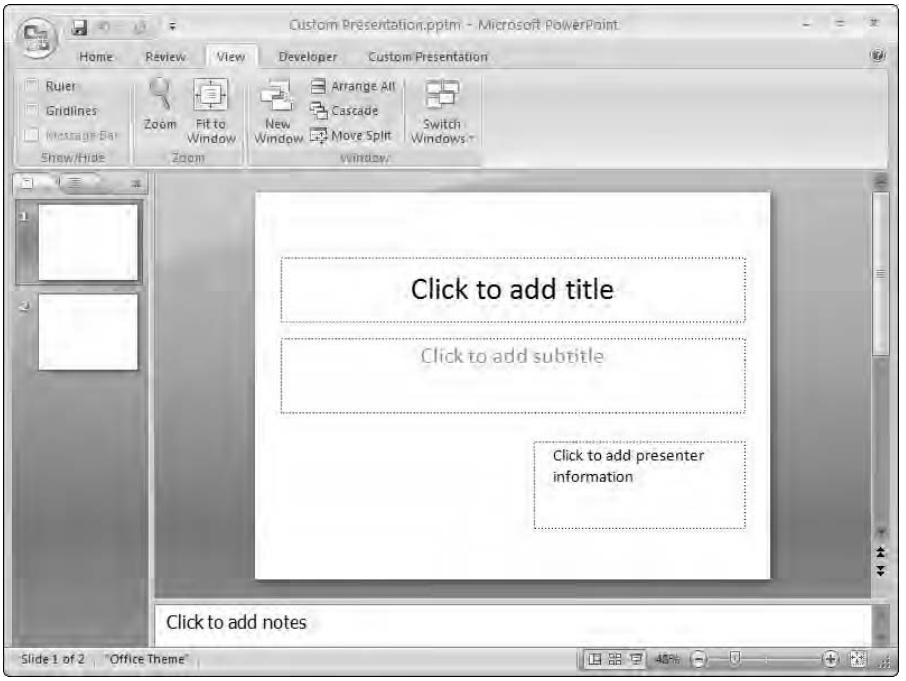


Figure 10-2: Remove the application features that tend to confuse, rather than help, the user.

Removing the extraneous features does make the task easier for inexperienced users. However, you'll also hear complaints from experienced users when their favorite feature is missing. You'll need to balance the user experience based on the actual application requirements. In some cases, you may want to consider returning some advanced user functionality (in the form of dialog-box launchers) or even create two template versions and issue the one that's most appropriate for a given user. Listing 10-2 shows the XML used to remove the extra features.

Listing 10-2: Removing Extraneous User Interface Elements

```
<tab idMso="TabHome">
   <group idMso="GroupDrawing" visible="false" />
</tab>
<tab idMso="TabInsert" visible="false" />
<tab idMso="TabDesign" visible="false" />
<tab idMso="TabAnimations" visible="false" />
<tab idMso="TabSlideShow" visible="false" />
<tab idMso="TabView">
   <group idMso="GroupPresentationViews"
          visible="false" />
   <group idMso="GroupColorGrayscale"
          visible="false" />
   <group idMso="GroupMacros" visible="false" />
</tab>
```

All you need to remove a feature is the identifier of the application element, such as `TabInsert`, and the `visible="false"` entry. You can remove features at any level. However, if you want to remove a group, you must first provide the `<tab>` tag. Likewise, when you're removing a control, you must provide both the `<tab>` and `<group>` tags.

In addition to creating the slideshow, the user needs to perform one additional task — viewing the slideshow. PowerPoint also provides this feature, but it isn't part of the custom tab. Adding the Start Slide Show group requires one last piece of XML, as shown here:

```
<group idMso="GroupSlideShowStart" />
```

Creating the Initial Slide

Every presentation has to start somewhere, and that first slide is important. When you view some presentations, you wonder what happened because the first slide is less than appealing or it doesn't contain all of the right information. This section shows how you can automate most of the task and provide

a wizard-like interface that asks for the rest of the information from the user. By combining these two processes, the initial slide is filled out before the user really gets to do anything with it, which ensures that the slide has a consistent appearance no matter who creates the presentation.

Starting the process

Creating a complex interface requires quite a bit of code when working with the Ribbon; it's important to break the process into manageable pieces. This section describes the overall process — what happens when the user clicks Add Starting Slide on the Ribbon (shown in Figure 10-1). Listing 10-3 shows the code needed to perform this task.

Listing 10-3: Creating the Initial Slide

```
'Callback for StartSlide onAction
Sub StartSlideClick(control As IRibbonControl)

   ' Locate the layout for the slide.
   Dim ThisLayout As CustomLayout
   For Each ThisLayout In _
      Presentations(1).SlideMaster.CustomLayouts

      If ThisLayout.Name = "Starting Slide" Then

         ' After locating the layout, exit the loop.
         Exit For
      End If
   Next

   ' Add the required slide to the presentation.
   Dim ThisSlide As Slide
   Set ThisSlide = _
      ActivePresentation.Slides.AddSlide(1, ThisLayout)

   ' Add the presentation title.
   PresentTitle = InputBox( _
      "Provide a title for the presentation", _
      "Presentation Title", "My Presentation")
   ThisSlide.Shapes(1).TextFrame.TextRange.Text = _
      PresentTitle

   ' Add the presentation subject.
   PresentSubject = InputBox( _
      "Provide a subject for the presentation", _
      "Presentation Subject", _
      "My Presentation Subject")
```

```
    ThisSlide.Shapes(2).TextFrame.TextRange.Text = _
       PresentSubject

    ' Save the data for later use.
    SetCustomProperty "PresentationTitle", PresentTitle
    SetCustomProperty _
       "PresentationSubject", PresentSubject

    ' Build the presenter information string.
    Dim PresenterInfo As String

    ' Get the username.
    PresenterInfo = CStr(GetBuiltInProperty("Author"))

    ' Get the additional user information.
    PresenterTitle = GetCustomProperty("PresenterTitle")
    PresenterDept = GetCustomProperty("PresenterDept")
    PresenterEmail = GetCustomProperty("PresenterEmail")
    PresenterTel = GetCustomProperty("PresenterTel")

    ' Make sure all of the values are available.
    If PresenterTitle = "" Or PresenterDept = "" Or _
       PresenterEmail = "" Or PresenterTel = "" Then

       ' Obtain the required information.
       GetUserData

    End If

    ' Add the additional information to the presenter
    ' string.
    PresenterInfo = PresenterInfo + _
       vbCrLf + "Title: " + PresenterTitle + _
       vbCrLf + "Department: " + PresenterDept + _
       vbCrLf + "Email: " + PresenterEmail + _
       vbCrLf + "Telephone: " + PresenterTel

    ' Add the presenter information to the slide.
    ThisSlide.Shapes(3).TextFrame.TextRange.Text = _
       PresenterInfo

End Sub
```

The code begins by locating the custom layout used with this example. The code can't access the layout directly by using its name as an index, so the example relies on a `For Each` loop to perform the task. The layout appears as one of the members of the `Presentations(1).SlideMaster.CustomLayouts` collection.

After the code locates the required layout, it uses the `ActivePresentation.Slides.AddSlide()` method to add the new slide as the first slide in the presentation.

You must provide both a slide number and the `ThisLayout` object as input to the `ActivePresentation.Slides.AddSlide()` method.

The next step is to begin filling the new slide with information. You can obtain a lot of information for these slides from other sources, such as Outlook. However, only the user can provide the presentation title and subject. The example uses a simple `InputBox()` to accomplish the task. The output looks like the dialog box shown in Figure 10-3.

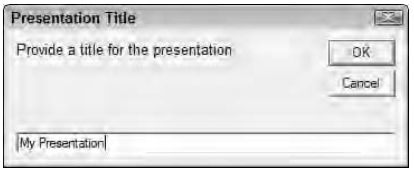


Figure 10-3: Request the title and subject of the presentation from the user.

The user hasn't touched the slide at this point, so you can make certain assumptions about the slide content based solely on the layout you created. Consequently, the code can use the `ThisSlide.Shapes(1).TextFrame.TextRange.Text` property to assign the title to this slide. If the user had accessed the slide, you couldn't make this assumption because the title might not appear as the first shape. The code performs the same sequence of events for the subject.

Because the title and subject can appear almost anywhere in the presentation, the code has to save the information for later use. The best way to accomplish this task is to save the information as part of the document properties. The code uses the `SetCustomProperty()` function (explained later) to perform this task.

At this point, the slide has a title and subject. It's time to take care of the third blank, the presenter information. In many cases, you can assume the person creating the presentation is the one who will ultimately give it. In this case, you can simply draw all of the presenter information from PowerPoint and Outlook.

Although you can probably safely assume that the user will change the title and subject every time the application creates a new starting slide, you shouldn't assume that the user will have a name change (or other change of personal identity). Consequently, the example tries to draw as much information as it can from local sources without bothering the user for input.

The first task is to obtain the username. Unfortunately, PowerPoint doesn't provide an `Application.UserName` property, so you can't simply get the

name directly from the application. Consequently, the example retrieves the username using the `GetBuiltInProperty()` function (explained later). Because built-in properties can use any data type, the return type is an `Object` that you must convert to a string using the `CStr()` function.

The code must locate the other user information (such as department and telephone) next. The code calls the `GetCustomProperty()` function four times, once for each piece of user information required in addition to the username. At this point, the code has to check the values for blanks. If the blanks exist, it calls the `GetUserData() Sub` (explained later) to obtain the data from Outlook.

Now that the application has user data to use, the code builds a string and inserts it into the third blank on the new slide. At this point, the slide is ready to use. You can see typical slide content in Figure 10-1. Except for asking the user two questions, the code obtains all the other information in this form and places it in the correct location.

Saving custom properties for later use

The example saves the information the user supplies to custom properties in the document. Because the document contains a number of custom properties, the example uses a special function named `SetCustomProperty` to perform the task. This function requires two inputs, the name of the property and its value. Listing 10-4 shows the code for `SetCustomProperty`.

Listing 10-4: Saving Custom Properties

```
Function SetCustomProperty(PropertyName As String, Value
          As String)
   ' Determines whether the property exists.
   Dim Found As Boolean

   ' Contains the current property count.
   Dim Counter As Integer

   ' Check for the property value.
   For Counter = 1 To _
      ActivePresentation.CustomDocumentProperties.Count

      ' Change the existing property.
      If ActivePresentation.CustomDocumentProperties( _
         Counter).Name = PropertyName Then

         ActivePresentation.CustomDocumentProperties( _
            PropertyName) = Value
```

(continued)

Listing 10-4 *(continued)*

```
            ' Indicate that the code has found the
            ' property.
            Found = True
            Exit For
         End If
      Next

      If Not Found Then

         ' Add the new property.
         ActivePresentation.CustomDocumentProperties.Add _
            Name:=PropertyName, _
            LinkToContent:=False, _
            Type:=msoPropertyTypeString, _
            Value:=Value
      End If
   End Function
```

The example can actually handle two scenarios with one call. In the first case, the property already exists; the code merely updates its value. In the second case, the property doesn't exist yet in the document; the code adds it. The code looks for the existing property using a `For` loop. When the code finds a match between the `PropertyName` input and the `ActivePresentation.CustomDocumentProperties(Counter).Name` property value, it assigns the new value and exits the loop.

When the code doesn't find the property, it adds a new property using the `ActivePresentation.CustomDocumentProperties.Add` method. This method requires four inputs for PowerPoint — the name, data type, value, and a special indication of whether the property is linked to any content.

Getting built-in property values

Depending on the application you create, it's likely that you'll need to obtain built-in property values, such as the username. Listing 10-5 shows the `GetBuiltInProperty()` method.

Listing 10-5: Obtaining a Built-in Property Value

```
Function GetBuiltInProperty(PropertyName As String) _
   As Object

   ' Keeps track of the current property.
   Dim Counter As Integer

   ' Locate the requested property.
   For Counter = 1 To _
```

```
    ActivePresentation.BuiltInDocumentProperties.Count

    If ActivePresentation.BuiltInDocumentProperties( _
        Counter).Name = PropertyName Then

        Set GetBuiltInProperty = _
  ActivePresentation.BuiltInDocumentProperties(Counter)

        ' Exit the loop when the property is found.
        Exit For
      End If
   Next
End Function
```

The `GetBuiltInProperty()` function uses a `For` loop to locate the required property in the `ActivePresentation.BuiltInDocumentProperties` collection. When the code finds the property, it assigns its value to the output of the function. It's essential to remember that this output is an `Object`, and not a `String`. Because the built-in properties can contain any value, you must handle them as objects, rather than as the desired data type.

Getting custom property values

It doesn't pay to save custom property values if you can't use them later. The `GetCustomProperty()` function shown in Listing 10-6 retrieves any data that you saved earlier.

Listing 10-6: Obtaining a Custom Property Value

```
Function GetCustomProperty(PropertyName As String) As
          String
   ' Keeps track of the current property.
   Dim Counter As Integer

   ' Locate the requested property.
   For Counter = 1 To _
      ActivePresentation.CustomDocumentProperties.Count

      If ActivePresentation.CustomDocumentProperties( _
         Counter).Name = PropertyName Then

         GetCustomProperty = _
   ActivePresentation.CustomDocumentProperties(Counter)

         ' Exit the loop when the property is found.
         Exit For
      End If
   Next
End Function
```

The `GetCustomProperty()` function uses a `For` loop to locate the required property in the `ActivePresentation.CustomDocument Properties()` collection. When the code finds the property, it assigns its value to the output of the function. Otherwise the function has an empty-string output.

Interacting with Outlook to obtain user data

The example requires more than a username, but the built-in properties can't provide any information. In this case, the application retrieves the current user information values from Outlook using the `GetUserData() Sub`. Obviously, you can use Active Directory, a database, or any other source for the user information.

If you use the approach shown in the example, you must add a reference to the Outlook library. Choose Tools➪References and you'll see the References dialog box shown in Figure 10-4. Check the Microsoft Outlook 12.0 Object Library entry, as shown, and click OK.

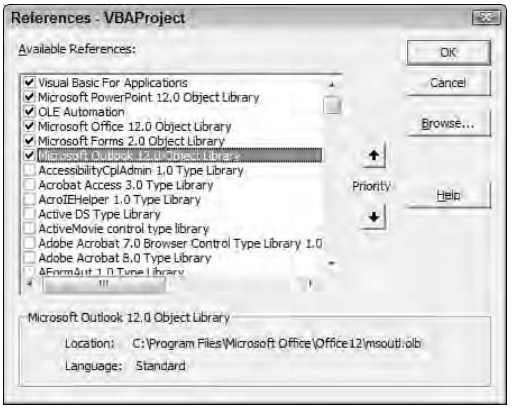


Figure 10-4: Set the Outlook reference for this example.

After you create the reference, you can use Outlook to retrieve the required user information, as shown in Listing 10-7. The reason that the example places this code in a separate `Sub` is that the application might need to retrieve the user information at several points.

Listing 10-7: Obtaining the User Data from Outlook

```
Private Sub GetUserData()

   ' Search for the user information in Outlook.
   Dim CheckSender As AddressEntry
   For Each CheckSender In _
      Outlook.Application.Session.AddressLists. _
         Item(1).AddressEntries

      ' Check the entry name.
      If CheckSender.Name = PresenterInfo Then
         Exit For
      End If
   Next

   ' Save the data to the variables.
   PresenterTitle = CheckSender.GetContact.JobTitle
   PresenterDept = CheckSender.GetContact.Department
   PresenterEmail = _
      CheckSender.GetContact.Email1Address
   PresenterTel = _
      CheckSender.GetContact.BusinessTelephoneNumber

   ' Save the results for later use.
   SetCustomProperty "PresenterTitle", PresenterTitle
   SetCustomProperty "PresenterDept", PresenterDept
   SetCustomProperty "PresenterEmail", PresenterEmail
   SetCustomProperty "PresenterTel", PresenterTel

End Sub
```

The code begins by locating the current user in the `Outlook.Application.Session.AddressLists.Item(1).AddressEntries` collection. The application assumes that the user information appears in the first (default) address list. The `For Each` loop exits when the PowerPoint username matches the `CheckSender.Name` property value.

The code performs two tasks with the data:

- It sets the values of all of the global variables so that any `Sub` or `Function` can access the user data.
- It uses the `SetCustomProperty` method to store the data for later use.

The second user data processing step is very important because you can't depend on the user having access to the required information at all times. The user might use a presentation machine that doesn't have Outlook installed, so saving the user information as part of the document is important.

Adding the Optional Slide Elements

Sometimes Microsoft does part of the work for creating a custom application, but the element appears in the wrong place. Look at the Home tab, and you'll see the Slides group shown in Figure 10-5.

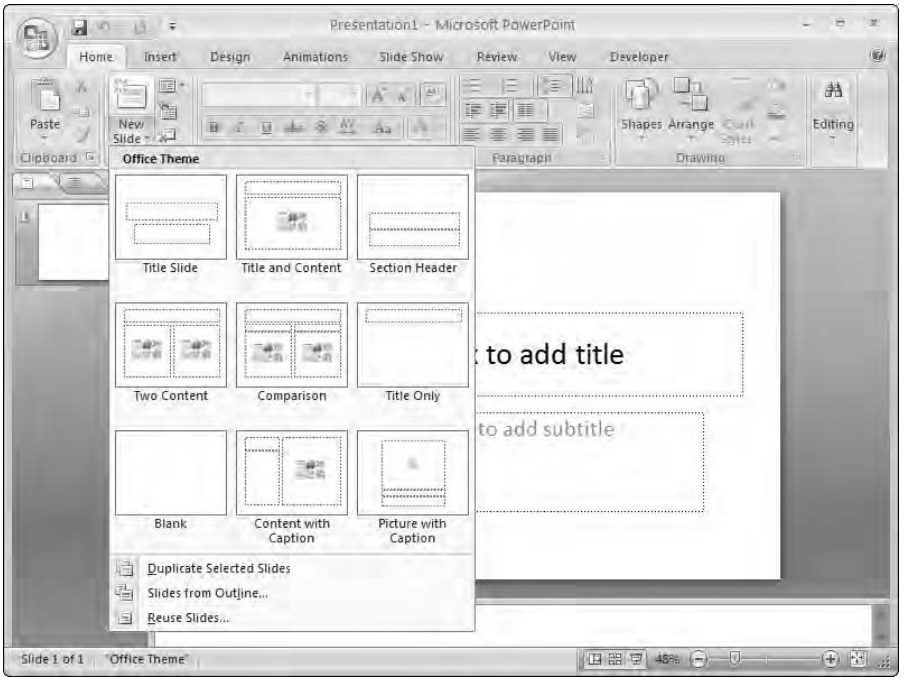


Figure 10-5: Reuse application elements as needed to develop your application faster.

In this case, the Slides group provides the functionality you need to add optional slides to a presentation, but having the user go to the Home tab to use it breaks up the workflow of the custom application. The group contains four buttons that let you perform the following tasks:

- Add a slide
- Change the layout of the current slide
- Reset slide formatting
- Delete a slide

In general, the buttons do everything you need them to do in the custom application. You should also ask yourself these questions:

- Are there enough buttons to perform every task?
- Does the current button layout provide the correct emphasis?
- What do you need to do to reuse the existing Ribbon features?

The custom application requires layouts other than those provided by the default PowerPoint installation. You can fix that problem by creating new layouts. Select the View tab and click Slide Master. PowerPoint adds a new Slide Master tab where you can add, replace, modify, and delete layouts and master slides, as shown in Figure 10-6. Consequently, you can use the existing PowerPoint features to customize the buttons as required. When you finish adding the custom slides, click Close Master View on the Slide Master tab.

Make sure you create any new templates that your application requires *before* you remove this functionality from the View tab. Otherwise you might have the tendency to overwrite your template additions while working on the Ribbon elements. Always be sure to close the file in PowerPoint before you reopen it using the Custom UI Editor.

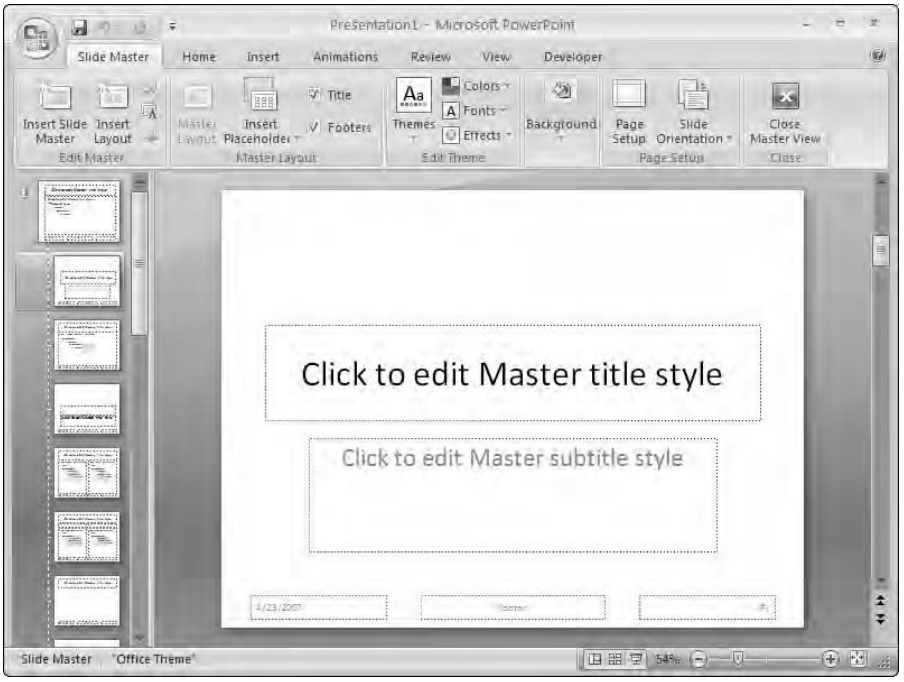


Figure 10-6: Modify the layouts and slide masters as required for the application.

You've now answered the question of modifying the gallery to meet your needs. However, the Slides group still may not provide everything needed. Certainly, the user needs to add, change, reset, and delete slides, but is there anything else the user needs? In this case, the user probably wants to add some graphics to the slides and check the presentation for spelling. Given that this is a workflow application, you want to make most tasks straightforward. Fortunately, you can move some other elements around to make sure the slides are easy to create.

Because you're adding and removing some functionality for this custom application, it's important to ask about emphasis. Adding a slide still requires a strong emphasis. However, you might also want to give deleting a slide more emphasis; the application layout shown in Figure 10-1 does so. The resulting XML for all this movement, reorganization, and reemphasis appears in Listing 10-1.

Supporting Constant Data

The application has a lot of constants stored that the user could possibly employ when creating slides. For example, most of the slides will need the presentation title, and the user will want to add personal information liberally. The example provides the Entry Constants group to let the user access this information. Listing 10-8 shows a typical example of how you can implement this functionality (see the source code on the Web site for a complete code listing for this part of the example).

Listing 10-8: Supporting Typical User Information

```
'Callback for PresentName onAction
Sub PresentNameClick(control As IRibbonControl)

   ' Verify the global variable contains good
   ' information.
   If PresenterTitle = "" Then

      ' If not, obtain the value from the custom
      ' property, when avaialble.
      PresenterTitle = _
         GetCustomProperty("PresenterTitle")

      ' If the custom property is blank, then obtain
      ' the information from Outlook.
```

```
      If PresenterTitle = "" Then
            GetUserData
      End If
   End If

   ' Remove any highlighted material.
   If ActiveWindow.Selection.TextRange.Count > 0 Then
      ActiveWindow.Selection.Delete
   End If

   ' Add the requested text at the insertion point.
   ActiveWindow.Selection.TextRange.InsertAfter _
      CStr(GetBuiltInProperty("Author"))
   ActiveWindow.Selection.TextRange.InsertAfter _
      ", " + PresenterTitle

End Sub
```

The code begins by checking for a value in the global variable. The user might have exited the application between creating the initial slide and clicking the constant button. Consequently, you can't assume that the value is available.

The code looks in the document data first using the `GetCustomProperty()` function. When the data doesn't appear as part of the document, the code obtains it from another source. In this case, the code calls on Outlook for the data using the `GetUserData()` function. However, it might also ask the user for the information or obtain the information in other ways.

After the code obtains the required data, it removes any data that the user has highlighted in the current location using the `ActiveWindow.Selection.Delete()` method. You can check for the existence of a highlight by checking the `ActiveWindow.Selection.TextRange.Count` property.

Finally, the code adds the new data at the current insertion point using the `ActiveWindow.Selection.TextRange.InsertAfter()` method. PowerPoint provides a number of methods for working with data, so it's easy to insert the data in a way that the user expects.

Not every constant requires elaborate storage methods. Sometimes the user wants to insert something simple, such as a date. Listing 10-9 shows an example of how you could insert a date into any text area on the slide.

Listing 10-9: Providing Date Information

```
'Callback for PresentDate onAction
Sub PresentDateClick(control As IRibbonControl)

   ' Display the date dialog box.
   Dim ThisDate As ChooseDate
   Set ThisDate = New ChooseDate
   ThisDate.Show

   ' Obtain the selected date.
   If ThisDate.Result = vbOK Then

      ' Insert the date.
      ActiveWindow.Selection.TextRange.InsertAfter _
         ThisDate.SelectedDate
   End If

End Sub
```

The code begins by displaying the Select a Date dialog box shown in Figure 10-7. This is a custom form created for the application. The reason you want to insert the date using this approach is that the user might need any date, not just the date of the presentation. By making the date-insertion technique flexible, the code can serve many needs. After the code verifies that the user clicked OK in the Select a Date dialog box, it inserts the date information, using the `ActiveWindow.Selection.TextRange.InsertAfter()` method.

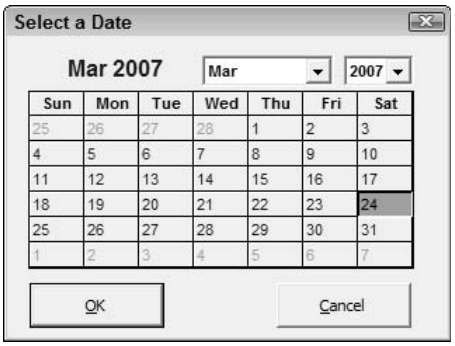


Figure 10-7: The user can choose any date to appear on the slide.

Providing a Presentation Ending

As with the beginning slide, the ending slide for a presentation is particularly important because it often remains displayed for a long time after the presentation. The user leaves with the impression created by the ending slide.

Fortunately, you can automate a considerable portion of the ending slide — perhaps not as much of it as the beginning slide, but certainly enough to make a difference. Listing 10-10 shows the code used for this task.

Listing 10-10: Defining the Ending Slide

```
'Callback for EndSlide onAction
Sub EndSlideClick(control As IRibbonControl)

   ' Locate the layout for the slide.
   Dim ThisLayout As CustomLayout
   For Each ThisLayout In _
      Presentations(1).SlideMaster.CustomLayouts

      If ThisLayout.Name = "Ending Slide" Then

         ' After locating the layout, exit the loop.
         Exit For
      End If
   Next

   ' Add the required slide to the presentation.
   Dim ThisSlide As Slide
   Set ThisSlide = ActivePresentation.Slides. _
         AddSlide( _
            ActivePresentation.Slides.Count + 1, _
            ThisLayout)

   ' Check the presentation title.
   If PresentTitle = "" Then

      ' Get the information from the custom variable.
      PresentTitle = _
         GetCustomProperty("PresentationTitle")

      ' If the title is still missing, get it from the
      ' user.
      If PresentTitle = "" Then
         PresentTitle = InputBox( _
            "Provide a title for the presentation", _
            "Presentation Title", "My Presentation")

         ' Save the data for later use.
         SetCustomProperty _
            "PresentationTitle", PresentTitle
      End If
   End If

   ' Add the presentation title.
   ThisSlide.Shapes(1).TextFrame.TextRange.Text = _
```

(continued)

Listing 10-10 ***(continued)***

```
      PresentTitle

   ' Get the username.
   PresenterInfo = CStr(GetBuiltInProperty("Author"))

   ' Get the additional user information.
   PresenterTitle = GetCustomProperty("PresenterTitle")
   PresenterDept = GetCustomProperty("PresenterDept")
   PresenterEmail = GetCustomProperty("PresenterEmail")
   PresenterTel = GetCustomProperty("PresenterTel")

   ' Make sure all of the values are available.
   If PresenterTitle = "" Or PresenterDept = "" Or _
      PresenterEmail = "" Or PresenterTel = "" Then

      ' Obtain the required information.
      GetUserData

   End If

   ' Add the additional information to the presenter
   ' string.
   PresenterInfo = PresenterInfo + _
      vbCrLf + "Title: " + PresenterTitle + _
      vbCrLf + "Department: " + PresenterDept + _
      vbCrLf + "Email: " + PresenterEmail + _
      vbCrLf + "Telephone: " + PresenterTel

   ' Add the presenter information to the slide.
   ThisSlide.Shapes(3).TextFrame.TextRange.Text = _
      PresenterInfo
End Sub
```

Many of the techniques used for this part of the example mirror those used for the starting slide. For example, the code begins by locating the appropriate layout and using it to create a new slide. However, you'll notice important differences in this code as well. The slide appears at the end of the presentation, so the code uses the `ActivePresentation.Slides.Count` property to place it in the proper location.

Because the ending slide should already have access to all the data it requires, the code simply checks for the presence of the data and uses it immediately. Only when the code can't find the required data (for example, because the user closed the file before completing the presentation) does the code look up the data using custom properties or other means. Figure 10-8 shows typical output from this portion of the application.

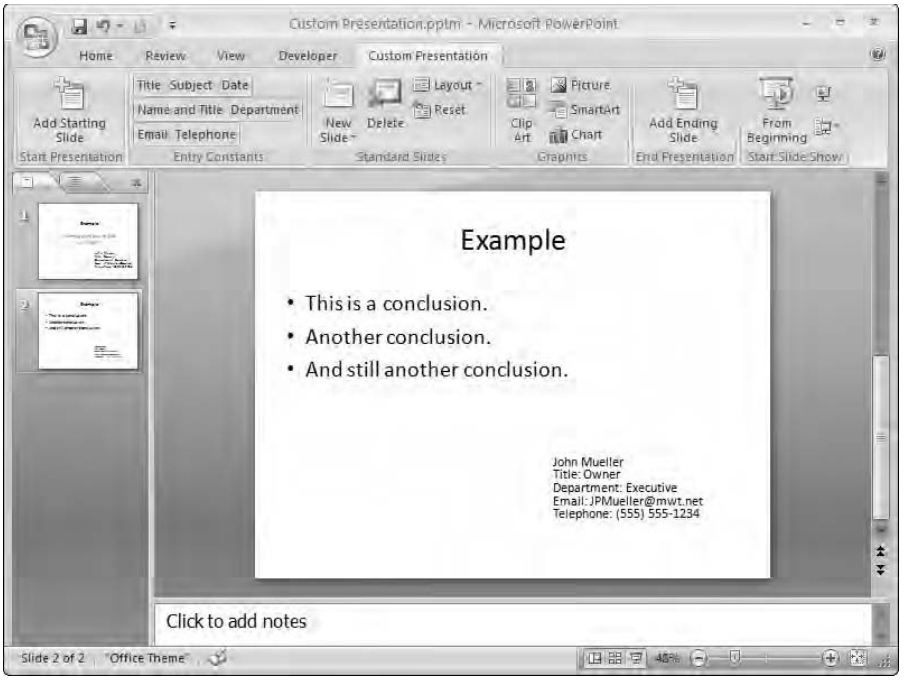


Figure 10-8: The ending slide includes the presentation title, a summary, and the presenter's contact information.

Saving and Using the Template

The last part of this example is turning the presentation into a document. To perform this task, follow these steps:

1. **Choose Office Menu⇨Save As⇨Other Formats.**

   You'll see a Save As dialog box.

2. **Choose the PowerPoint Macro Enabled Template (*.potm) option in the Save As Type field, as shown in Figure 10-9.**

   As soon as you choose the template option, PowerPoint changes the folder to the default template-storage location. Normally the path to the appropriate folder on your hard drive is

   ```
   \Users\<User Name>\AppData\Roaming\Microsoft\Templates\PowerPoint Templates
   ```

3. **Give your template a meaningful name and save it by clicking Save.**

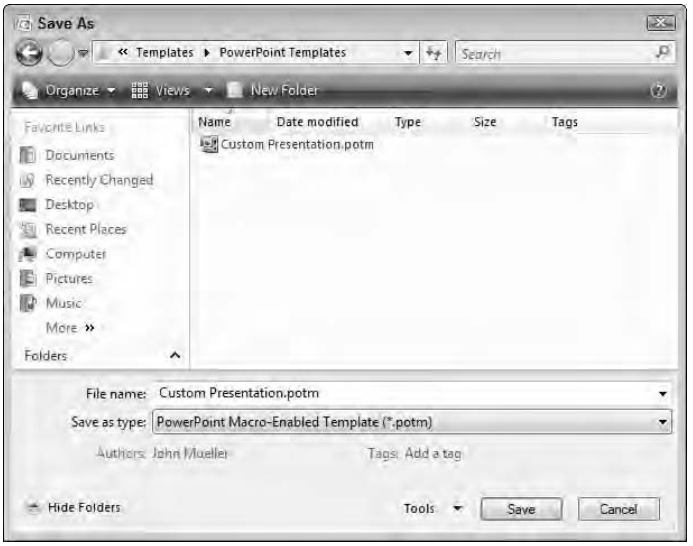


Figure 10-9: Make sure you save your work as a template.

When you want to use the template, follow these steps:

1. **Choose Office Menu➪New.**
2. **Click My Templates in the New Presentation dialog box.**

   You'll see the New Presentation dialog box.
3. **Select the PowerPoint Templates tab shown in Figure 10-10.**

   You'll see a list of the templates you've created. Whenever you highlight a template that includes a starting slide, you'll see the first slide in the Preview pane.

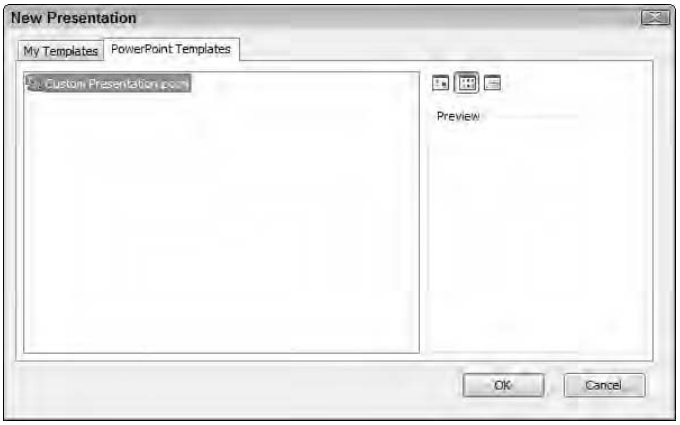


Figure 10-10: Choose the template you created from the list of available templates.

Chapter 11

Working with Web Services

In This Chapter

- Understanding how to work with Web services
- Considering differences between public and private Web services
- Working with Amazon Web Services

Web services have become an essential part of most business plans today because they offer significant flexibility — offering services without the problems of platform dependence. For example, when you access Amazon Web Services (AWS), you have no idea of whether the system is running Linux, and you don't have to care. All that matters is sending a properly formatted request and receiving a response with the data you requested. Public and private Web services are springing up at an amazing rate because they truly are so useful. A Web service does incur a small performance penalty (compared to other technologies), but in a world of high-speed computers, the penalty is hardly noticeable.

Of course, the question is what Web services have to do with the Ribbon. When working with the toolbar and menu system, developers of Web services often had to resort to kludges to create a connection between the Web service and the Office application. The connection between the two was quite noticeable, and it didn't seem as if the Web service would ever truly integrate with the application. Some developers actually resorted to creating specialized documents to work with Web services, such as the sample Excel document created for use with the eBay Web service as part of their Software Development Kit (see `http://developer.ebay.com/` for details).

Many people view the Ribbon as an impediment to working with Office, and it's true that your older applications will require updates. However, Web services are newer sources of information for Office users, so they don't have the incredible number of applications associated with them that other Office applications do. A Web service application may actually present an opportunity for you to start fresh with the Ribbon using a new technology, which is the reason that I chose the topic for this book. Because Web services also offer such a great value to businesses, you'll find that your Ribbon learning experiences also pay real dividends to your company. This chapter explores both public and private Web services so you get a better idea of how to work in both environments.

Getting Started with Web Service Applications

Web service applications require a different perspective than some of the other applications in this book. Normally, you'll work with Web service applications in an integration mode. The Web service isn't part of a workflow, nor is it part of a specific task. Rather, the Web service will integrate into some existing feature or appear integrated into the application in some other way. Even though, from the developer's perspective, a Web service is a different kind of service from the normal network service, the user sees the Web service as simply another form of data storage. It's unlikely that the user will even care that the Web service exists as such.

It's important to consider how a Web service works in comparison with other kinds of services that your application may use. Most Web services require an Internet connection, which can prove unreliable. You need to provide some sort of alternative form when the Web service is inaccessible, or at least degrade the application functionality gracefully. For example, if the Web service receives data from a user's machine, you could simply cache it on the local drive until such time as the service is restored. In one case, I actually saw a developer create a mini-Web service on the user's machine that the application could use in the event of a failure. Of course, it could only serve cached data from the user's own machine, but at least the application didn't fail completely.

The best part about the Ribbon and Web services is that you can hide Web service features all over. You can even repurpose some controls, such as the research features of Office, to use resources you provide instead of those Microsoft normally recommends. For example, you could build a connection to Google Web Services to perform information lookups online, rather than rely on the functionality that Office normally uses. A clip-art feature in Word could actually rely on your private Web service to dish up approved art for company purposes.

It's not always necessary to hide the Web service completely. You might choose to augment existing features in some cases. For example, you could add a map button to the art features of an office application that uses Google Maps (`http://www.google.com/apis/maps/`) to provide maps for your Office application. In fact, you'll find a whole list of Google APIs (Web services) at `http://code.google.com/`. The point is that the Web service shouldn't be noticeable — the user should see it as a natural part of the Office application.

Understanding Public and Private Web Service Differences

Web services come in public and private flavors. The public version is something like Amazon Web Services (AWS) where you obtain data from Amazon for a particular need, such as locating products you want or working with your Amazon store. Of course, public infers that everyone can access the Web service. A private Web service is generally something your company puts together or you access as a third party on someone else's machine. For example, if you work with an ad service, you might be able to access the Web service to upload new banner ads.

In most cases, public Web services aren't completely public. You have to fill out forms and request access to them. The vendor sends you a developer key that you use to access the Web service. Normally, the vendor doesn't discriminate, so everyone gets to try the Web service. However, the vendor can revoke your license for a number of reasons, including using the Web service too much. Consequently, public doesn't always mean easily accessed or always accessible.

A public Web service may also charge fees and require that you jump through hoops to get your application approved. The eBay Web service relies on both of these techniques to ensure the reliability of its database. Your applications start out on a special server that has no connection to the real eBay database at all. This server is called the sandbox, and it lets you test your application in a safe environment, which is a very good idea. When you feel your application is ready, you pay eBay to test and certify it. Only at that point do you get to run your application on the real Web service. In short, you need to know what the public Web service demands as part of your application-planning process. That's because you might need several additional months of testing to get through all the vendor's Web service hoops.

Private Web services vary considerably. While you'll nearly always contact a public Web service using the Internet, a private Web service might exist on the Internet, an intranet, your network, or even your own machine. The private Web service is always locked down; only those who have an actual need to access it can. Vendors don't typically advertise private Web services, so you won't know they exist unless someone tells you about them. The need for security in a private Web service is so great that you might find difficulty accessing it at all. However, when all is said and done, the biggest difference between private and public Web services is one of scope — one is hidden from just about everyone, while the other is equally visible to anyone who needs it.

From an operational perspective, there's often little or no difference between a public and a private Web service. The scope of the Web service doesn't affect the tasks you must perform to work with it. For example, no matter which kind of Web service you work with, you still need to create a connection to it, send it requests, and do something with the responses that the Web service sends back. Public and private Web services use a variety of methods to communicate, and it's possible that you'll find the same technique used for both public and private Web services. Consequently, when you have a Web service set up for use, you really won't see much of a difference between the two while developing your application.

A public Web service does present more of a challenge for debugging your application than a private Web service does. You have access to only the client code when working with a public Web service, which means that you have only the error messages that the Web service sends back to use as a means for diagnosing errors. When you work with a private Web service, you have access to both the client and the server. Consequently, it's easier to locate communication problems. In addition, you can choose whether you want to fix them on the client or on the server.

Creating an Amazon.com Custom Application

AWS provides you with a lot of opportunities to work with a public Web service that provides a significant amount of features in a relatively safe environment. This Web service is especially nice because it works so well with either VBA or Visual Studio. In addition, you gain access to a number of Web services, including these:

- Amazon E-Commerce Service (Amazon ECS)
- Amazon Elastic Compute Cloud (Amazon EC2) – beta
- Amazon Historical Pricing
- Amazon Mechanical Turk (Beta)
- Amazon Simple Storage Service (Amazon S3)
- Amazon Simple Queue Service (Amazon SQS)
- Alexa Site Thumbnail
- Alexa Top Sites
- Alexa Web Information Service
- Alexa Web Search

All of these Web services have different uses, and you can incorporate them all into an Office application through the Ribbon. You can find a description of all these Amazon Web services at

```
http://www.amazon.com/gp/browse.html/?node=15763381
```

Although this chapter won't make you an AWS expert, you'll gain enough information to make use of all the services AWS provides.

This example demonstrates a considerable number of techniques, and the chapter simply can't hold all of the source code required to create it. The sections that follow do provide you with complete information about all of the essential techniques for working with this example and modifying it to meet your specific needs. (You can find the complete source code for this example on the Dummies.com site at `http://www.dummies.com/go/ribbonxfd`.) This example relies on the code found in the `Amazon Example.xlsm` file.

Getting an AWS developer tag

Before you can do anything with AWS, you need a developer tag. Many public Web services require that you obtain a developer tag so the Web site's owner can track how you use the Web service. In addition, you'll find that many of these public Web services enforce limits on how you can interact with them. A few, such as eBay, offer multiple levels of service (free at the lowest level and paid for at the other levels). The following steps tell how to get an AWS developer tag:

1. **Go to the Web site at**

   `http://www.amazon.com/gp/browse.html/?node=3435361`

2. **Click the Create Your Free Amazon Web Services Account link in the Start Using Amazon Web Services section of the Web page (the precise name of the link and section will vary as Amazon updates the Web site).**

   You'll see a sign-in page where you enter your Amazon account information. If you don't have an account, you can create one by selecting the No, I Am a New Customer option.

3. **Provide the required sign-in information and click Continue.**

   Amazon will ask you to provide some account information.

4. **Enter the account information and click Continue.**

   You'll see a success message.

5. **Open your e-mail program and check for new e-mails.**

   Amazon will send your developer tag as part of an e-mail message, along with some other information and useful links for using their Web services. Obtaining the developer tag can require a few minutes. If you don't receive the new developer tag immediately, keep trying and you'll eventually receive it. Contact the Amazon staff if you wait more than an hour; the developer tag normally arrives within an hour.

You must obtain an AWS developer tag to use the example. The file doesn't include a developer tag, and AWS won't send a response without it. Click the dialog-box launcher in the Search Criteria group the first time you run the example to enter your AWS developer tag. The application automatically stores this value so you won't have to enter it again.

Seeing how queries work in a browser

One of the more interesting features of AWS is that you don't need an application to use it. You can simply view the information you want using a browser. Of course, the output is in XML, but you can still use this simple approach to get what you want. Try typing the following URL into your favorite browser, but keep two things in mind:

- Even though this content appears on multiple lines in the book, be sure you type it all as one continuous line in your browser.
- Make sure you replace *Your-Developer-Tag* with the developer tag that Amazon provides.

Here's the content to type:

```
http://ecs.amazonaws.com/onca/xml?
Service=AWSECommerceService&Operation=ItemSearch&
Author=John%20Mueller&SearchIndex=Books&Sort=salesrank&
AWSAccessKeyId=Your-Developer-Tag
```

The URL points to the basic AWS site (`http://ecs.amazonaws.com/onca/xml`). It provides a number of arguments, including these:

- Service name (Amazon Web Services e-Commerce Service)
- Service operation requested (an `ItemSearch`)
- Author name
- Kind of output requested (for example, `books`)
- Output sort order
- Developer tag (an Amazon Web Services Access Key Identifier)

All of this information defines your request to the Web service. In most cases, all you'll change is one argument when you make another request, such as getting the second page of results (Amazon returns ten at a time by default). When you press Enter to go to the Web page, you'll see XML output for my books, as shown in Figure 11-1. (The screenshot is purposely modified to exclude the author's developer tag.)

The XML isn't completely readable — at least not in a usable way, but you can still obtain a lot of information from it. The topmost entries tell you about the query you made. They include data such as your developer tag, which you do know, and the total number of results pages, which you don't know until you make the call. You can also verify that Amazon sees your browser correctly by checking the `UserAgent` argument.

Immediately after the introductory information, you'll find a series of `<Item>` nodes that contain the book information. This information includes

- Amazon Standard Item Number (ASIN)
- Detail page's URL
- Author list
- Special contributors (such as illustrators)
- Publisher
- Book title

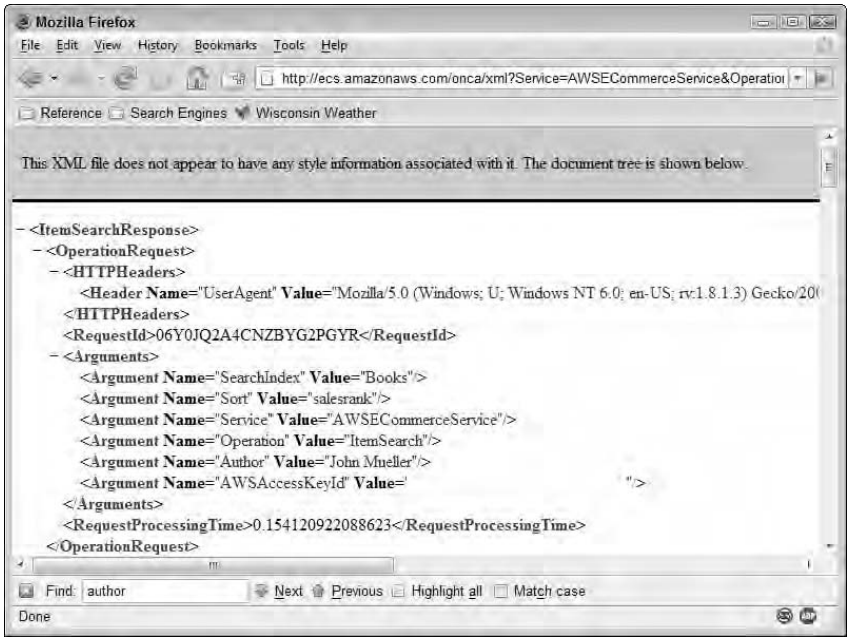


Figure 11-1: Experiment with AWS using a simple browser.

Not all this information appears with every book or other product, but you'll find most of it. The application you create can display any or all of this information. You can combine the output with information from other sources, or use the URLs provided as part of the output to add more information for the user. For example, the book images appear as URLs. Amazon normally provides you with three URLs for three different image sizes, so you can download the image that works best in your application.

Understanding AWS tasks

The AWS Ribbon interface for this example can actually reflect a workflow because you're using it to look for something — much as you would use a search engine. The part of AWS that this example uses will help the user locate a book. AWS uses the term *operation* to reflect a task that you want to perform with the Web service. The operation that this example performs is an `ItemSearch`. You can use the same techniques to locate any product that AWS supports, which includes a broad range of categories called `SearchIndexes`. The following list shows the `SearchIndexes` that AWS supports:

All	Apparel	Automotive
Baby	Beauty	Blended
Books	Classical	DigitalMusic
DVD	Electronics	GourmetFood
HealthPersonalCare	HomeGarden	Industrial
Jewelry	Kitchen	Magazines
Merchants	Miscellaneous	Music
MusicalInstruments	MusicTracks	OfficeProducts
OutdoorLiving	PCHardware	PetSupplies
Photo	Software	SportingGoods
Tools	Toys	VHS
Video	VideoGames	Wireless
WirelessAccessories		

The `SearchIndexes` you can use vary by country. The previous list shows the `SearchIndexes` for the United States. The Web site at

```
http://docs.amazonwebservices.com/AWSECommerceService/2007-04-04/DG/APPNDX_Searc
              hIndexMatricesArticle.html
```

provides `SearchIndexes` for other countries. You'll also find AWS operations for a number of other tasks, including those shown in the following list:

- BrowseNodeLookup
- CartAdd
- CartClear
- CartCreate
- CartGet
- CartModify
- CustomerContentLookup
- CustomerContentSearch
- Help
- ItemLookup
- ItemSearch
- ListLookup
- ListSearch
- SellerListingLookup
- SellerListingSearch
- SellerLookup
- SimilarityLookup
- TransactionLookup

You can find complete documentation for each of these operations at

```
http://docs.amazonwebservices.com/AWSECommerceService/2007-04-04/DG/CHAP_Operati
              onListAlphabetical.html
```

Each of these operations exercises a different portion of AWS. For example, if you want to locate a particular seller, you use the `SellerLookup` operation. The operation is the most important part of the request string for AWS, but you'll often have to combine it with other arguments. For example, a search of any kind will always require a `SearchIndex` value.

All of the AWS operations include optional arguments that you can use to refine your search. In addition, they all include features that help you obtain more information. For example, when you perform a search, you can use the `ResponseGroup` argument to return more than the default level of information.

Defining the AWS Ribbon interface

The interface for this example is a workflow. A workflow is the right choice in this case because the example helps the user search for something. Because you want your searches to provide orderly information, a workflow is the best interface choice. If you were performing another AWS operation, such as obtaining reviews about a particular product, you might use a task-based interface instead. Some operations, such as `CartAdd`, require a hidden interface because you want the user to concentrate on products, not on the interface. Figure 11-2 shows the interface for this example.

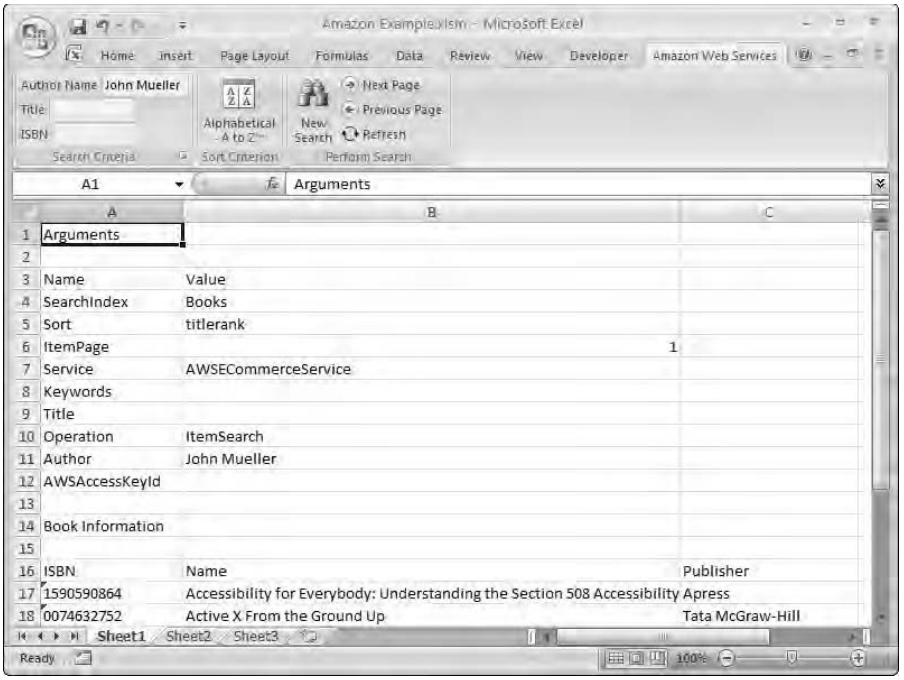


Figure 11-2: A workflow interface works well for searches.

Notice that this example includes a dialog-box launcher. This feature provides important functionality for the example — it provides a way for the user to enter a developer key. You can discover more about this feature in the "Adding the dialog-box launcher" section of this chapter. Listing 11-1 shows the XML for this example.

Listing 11-1: Defining the AWS Interface

```
<tab id="AmazonTab" label="Amazon Web Services">
   <group id="CriteriaGroup" label="Search Criteria">
      <editBox id="AuthorName"
               label="Author Name"
               getText="GetAuthorName"
               onChange="ChgAuthorName" />
      <editBox id="Title"
               label="Title"
               getText="GetTitle"
               onChange="ChgTitle" />
      <editBox id="ISBN"
               label="ISBN"
               getText="GetISBN"
               onChange="ChgISBN" />
      <dialogBoxLauncher>
         <button id="LaunchDialog"
                 screentip="Add Developer Tag"
```

```
                supertip="Enter the AWS tag."
                onAction="ShowSearchDetails"/>
      </dialogBoxLauncher>
   </group>
   <group id="SortGroup" label="Sort Criterion">
      <splitButton id="SortBtn" size="large" >
         <button id="selected"
                getLabel="GetSelLbl"
                imageMso="SortDialog"
                onAction="DefaultSort" />
         <menu id="SortChoice">
            <button id="relevancerank"
                   label="Relevance"
                   onAction="SetRelevance" />
... Other Sort Options ...
            <button id="inverse-titlerank"
                   label="Alphabetical - Z to A"
                   onAction="SetInvAlphabetical" />
         </menu>
      </splitButton>
   </group>
   <group id="SearchGroup" label="Perform Search">
      <button id="NewSearch"
              label="New Search"
              size="large"
              imageMso="FindDialog"
              onAction="DoNewSearch" />
      <button id="NextPage"
              label="Next Page"
              imageMso="ViewGoForward"
              onAction="DoNextPage" />
      <button id="PreviousPage"
              label="Previous Page"
              imageMso="ViewGoBack"
              onAction="DoPrevPage" />
      <button id="Refresh"
              label="Refresh"
              imageMso="RecurrenceEdit"
              onAction="DoRefresh" />
   </group>
</tab>
```

The code begins with the `Search Criteria` group, which contains three edit boxes used to hold the search values. The user need only enter a search criterion in one field to make the example work. However, if the user should request a search without any criteria, the application responds with an empty display (the expected output for no search criteria), rather than with an error. The `<editBox>` elements include both the `getText` and `onChange` attributes. You could rely on just the `onChange` attribute in this case because the application doesn't actually need to output values, but it's better to provide both attributes in case you plan to increase the flexibility of the application. For example, you might choose to add search suggestions at some point.

The code to support the search controls is much like the code for this kind of entry in the rest of the book. All it does is provide a simple exchange with a global variable that contains the current search value.

The Sort Criterion group includes a single split-button control that has all of the sort options that AWS supports for users in the United States. If you choose to provide international support, you'll need to populate this control during runtime and provide only the options that a particular country supports. If you use this approach, you'll need to rely on a `<dynamicMenu>` element, instead of the `<menu>` element shown. You provide the variable XML for the buttons within the menu as part of the `getContent` callback. The code to support the `selected` button appears in Listing 11-2.

Listing 11-2: Defining the Selected Button Label

```
'Callback for selected getLabel
Sub GetSelLbl(control As IRibbonControl, _
              ByRef returnedVal)

   ' Set a default sort label.
   returnedVal = "Invalid Sort"

   ' Set the sort according to sort type.
   Select Case SortType
      Case "relevancerank"
         returnedVal = "Relevance"
      Case "salesrank"
         returnedVal = "Sales"
      Case "reviewrank"
         returnedVal = "Customer Review"
      Case "pricerank"
         returnedVal = "Price - Low to High"
      Case "inverse-pricerank"
         returnedVal = "Price - High to Low"
      Case "daterank"
         returnedVal = "Publication Date"
      Case "titlerank"
         returnedVal = "Alphabetical - A to Z"
      Case "inverse-titlerank"
         returnedVal = "Alphabetical - Z to A"
   End Select
End Sub
```

The `GetSelLbl()` method must always return a value. However, if the `Select Case` statement fails to locate a proper match from the default sort values, then the sort value is invalid. Theoretically, you can avoid letting the user know about the problem at all by providing a default value when none of

the sort values is accurate. However, given the way that the application creates the sort value, an invalid value points to other issues in the application; letting the user know there might be a problem by displaying `Invalid Sort` is better than waiting for the application to crash. Each of the individual buttons in the menu also includes on `onAction` callback. Here's an example of a typical callback:

```
'Callback for relevancerank onAction
Sub SetRelevance(control As IRibbonControl)
   ' Set the sort type.
   SortType = "relevancerank"

   ' Invalidate the control.
   Rib.InvalidateControl "selected"
End Sub
```

The code sets the `SortType` global variable to indicate the kind of sort the user selected. It then calls `Rib.InvalidateControl` to update the `selected` button, so the selected sort appears on the split button.

The Perform Search group consists of four buttons. The `NewSearch` button is larger than the rest because it focuses on the first action the user will perform. All four buttons have icons associated with them to focus attention on their use because the user will access them regularly. The code to support these buttons appears in the "Making a query" section of this chapter.

Not shown in the code is the usual `onLoad="RibbonLoaded"` attribute in the `<customUI>` element. You'll always use this entry to provide a way to save the Ribbon reference and initialize any variables used for the rest of the example. Listing 11-3 shows the code used to initialize this application.

Listing 11-3: Initializing the Application

```
'Callback for customUI.onLoad
Sub RibbonLoaded(ribbon As IRibbonUI)

   ' Save the Ribbon reference.
   Set Rib = ribbon

   ' Set the default search values.
   Author = ""
   Title = ""
   ISBN = ""

   ' Set the default sort type.
```

(continued)

Listing 11-3 *(continued)*

```
    SortType = "salesrank"

    ' Set the default page number.
    PageNumber = 1

    ' Set the total pages.
    TotalPages = 1

    ' Determine whether there is a developer tag.
    If ActiveWorkbook.CustomDocumentProperties.Count = 0
            Then
       DevTag = ""
    Else
       Dim TheseProperties As DocumentProperties
       Set TheseProperties =
            ActiveWorkbook.CustomDocumentProperties
       DevTag = TheseProperties("DevTag")
    End If
End Sub
```

The code begins by saving the Ribbon reference in `Rib` as normal. It also sets all of the variables as you might expect. The `Author`, `Title`, and `ISBN` variables contain the search criteria. The `SortType` variable contains the sort criterion. The `PageNumber` and `TotalPages` variables support the search pages that AWS returns. You must keep track of the beginning and ending of the page list to ensure the user doesn't request a page that doesn't exist.

The user must also provide a developer tag. Note, however, that this tag won't change while using the application, so there's no point entering it more than once. Consequently, the application stores the developer tag as a custom property within the document. Notice how the code checks for the developer `DevTag` custom property. This approach only works if you have one property stored in the document. If you have multiple properties stored in the document, you must first check for a zero property count, and then check for the specific property you need.

Adding the dialog-box launcher

The XML for the dialog-box launcher appears in Listing 11-1. The code is the standard dialog-box launcher setup with the `<dialogBoxLauncher>` element enclosing a button. This particular dialog-box launcher requires only one value, which makes it unique. You can simplify the code considerably by using the technique shown in Listing 11-4 instead of creating a form.

Listing 11-4: Defining the Dialog-Box Launcher

```
'Callback for LaunchDialog onAction
Sub ShowSearchDetails(control As IRibbonControl)

   ' Get the developer tag.
   NewDevTag = _
      InputBox("Provide the AWS Developer Tag", _
               "AWS Tag Input", DevTag)

   ' Verify the user has entered a value.
   If NewDevTag = "" Then
      Exit Sub
   End If

   ' Add it to the custom properties for the document.
   Dim TheseProperties As DocumentProperties
   Set TheseProperties = _
      ActiveWorkbook.CustomDocumentProperties
   TheseProperties.Add Name:="DevTag", _
                       LinkToContent:=False, _
                       Type:=msoPropertyTypeString, _
                       Value:=NewDevTag

   ' Save the new tag.
   DevTag = NewDevTag
End Sub
```

Notice that the code relies on an `InputBox` to request the developer-tag input. You can use this technique only when the dialog box contains a single input. Normally, you'll create a complex dialog box with multiple fields, which means creating a custom form. Of course, this raises the question of why you should use a dialog box launcher at all. It's important to remember that dialog-box launchers hide advanced features from view. Adding the developer tag is a required — though advanced — task that you'll perform only one time. Using a dialog-box launcher is a legitimate way to make the feature accessible, yet hide it from view when it's not in use.

The `InputBox` doesn't provide return value when the user clicks Cancel. The example handles this problem by checking `NewDevTag` for a value after the call returns from `InputBox`. This check also works fine for ensuring the user doesn't accidentally click OK without entering a value.

After the code obtains the developer tag, it stores it as a custom document property. The `Add()` method requires four inputs as normal:

- The name of the variable
- A Boolean that indicates whether the value is attached to document content
- The data type of the variable
- The value

Making a query

The four buttons in the Perform Search group all perform essentially the same task: They request information from AWS. Of course, the difference is in how they make the request. Each button performs the task differently, as shown in Listing 11-5.

Listing 11-5: Creating a Request Based on Specific Needs

```
'Callback for NewSearch onAction
Sub DoNewSearch(control As IRibbonControl)
   ' Set the page number.
   PageNumber = 1

   ' Perform a new search.
   FetchAmazonData
End Sub

'Callback for NextPage onAction
Sub DoNextPage(control As IRibbonControl)

   ' Make sure there are more pages to display.
   If PageNumber < TotalPages Then

      ' Set the page number.
      PageNumber = PageNumber + 1

      ' Perform a new search.
      FetchAmazonData

   Else

      ' Tell the user there aren't any more
      ' pages to display.
      MsgBox "Last Page of Data"
   End If
End Sub

'Callback for PreviousPage onAction
```

```
Sub DoPrevPage(control As IRibbonControl)

   ' Verify that the page number is greater than 1.
   If PageNumber > 1 Then

      ' Set the page number.
      PageNumber = PageNumber - 1

      ' Perform a new search.
      FetchAmazonData

   Else

      ' Tell the user there aren't any more
      ' pages to display.
      MsgBox "First Page of Data"
   End If
End Sub

'Callback for Refresh onAction
Sub DoRefresh(control As IRibbonControl)
   ' Refresh the existing data.
   FetchAmazonData
End Sub
```

All four of the callbacks make a call to the `FetchAmazonData()` method at some point. The `DoNewSearch()` method simply sets the page number to `1` before it makes the call. This button assumes that the user has already changed all the required criteria. The only requirement is that the search start at the first page of the results.

The `DoNextPage()` and `DoPrevPage()` methods perform similar tasks, but in opposite directions of the search results. In the first case, the code verifies that the example isn't already displaying the last page, while in the second case it ensures that the example isn't already displaying the first page. Making this check reduces the number of unnecessary calls the application makes to AWS. Because you have a limit on the calls you can make, it's important to keep the unnecessary calls to a minimum.

Sometimes the user will want to refresh the data from a current search. The `DoRefresh()` method is the most straightforward because it simply calls AWS for an update of the data. You may need to throttle this particular control to ensure that the user doesn't exceed the AWS limits on the number of calls you can make in a given timeframe.

Before you can make a call to AWS, you need to add XML support to VBA. Choose Tools➪References to display the References – VBAProject dialog box shown in Figure 11-3. Locate the Microsoft XML library and check it. You can use any version, 5.0 and greater, for this example (the example actually uses version 6.0, which is the most current at the time of writing). Click OK and VBA will add the required reference to your project.

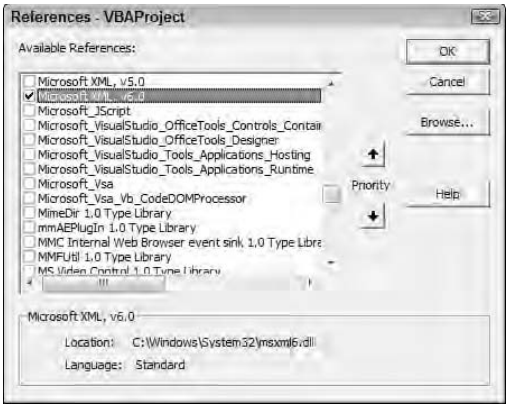


Figure 11-3: Make sure you set a reference for XML support in VBA.

The example uses a technique known as REpresentational State Transfer (REST) to obtain the data it requires from AWS. This technique is simpler for VBA developers to use than to build the actual XML required to make the call. In this case, the code builds a URL and simply loads it for local processing. The languages supported by Visual Studio often work better with Simple Object Access Protocol (SOAP) requests. In this case, the language actually builds an XML message and receives an XML message in return. SOAP offers some flexibility that REST doesn't provide, but it's more complicated to use. Listing 11-6 shows the request code for this example.

Listing 11-6: Making an AWS Request

```
Private Sub FetchAmazonData()
   Dim XMLFile As String
   Dim XMLData As DOMDocument
   Dim CurrentNode As IXMLDOMNode
   Dim DataNode As IXMLDOMNode
   Dim AttributeNode As IXMLDOMNode
   Dim NodeCount As Integer
   Dim CellOffset As Integer

   ' Clear the worksheet data.
   ActiveWorkbook.ActiveSheet.Cells.Clear

   ' Verify the sort type.
   If SortType = "" Then
      SortType = "salesrank"
   End If

   ' Make sure the user has entered a Developer tag.
   If DevTag = "" Then
```

```
      MsgBox "You Must Provide a Developer Tag" + _
             vbCrLf + "Click the Dialog Box " + _
             "Launcher in the Search group" + _
             "to Enter Your Developer Tag", _
             vbInformation And vbOKOnly, _
             "Developer Tag Needed"
      Exit Sub
   End If

   ' Create the string.
   XMLFile = _
      "http://ecs.amazonaws.com/onca/xml?" + _
      "Service=AWSECommerceService" + _
      "&Operation=ItemSearch" + _
      "&SearchIndex=Books" + _
      "&Author=" + AuthorName + _
      "&Title=" + Title + _
      "&Keywords=" + ISBN + _
      "&Sort=" + SortType + _
      "&ItemPage=" + CStr(PageNumber) + _
      "&AWSAccessKeyId=" + DevTag

   ' Load the data.
   Set XMLData = New DOMDocument
   XMLData.async = False
   XMLData.Load XMLFile

   ' Verify the data is good.
   If XMLData.ChildNodes(1).ChildNodes(1). _
      ChildNodes(0).ChildNodes(0).text = "False" Then

      Exit Sub
   End If

   ' Save the total number of pages.
   TotalPages = _
      CInt(XMLData.ChildNodes(1).ChildNodes(1) _
      .ChildNodes(2).text)

   ' Display a heading.
   Sheet1.Cells(1, 1) = "Arguments"
   Sheet1.Cells(3, 1) = "Name"
   Sheet1.Cells(3, 2) = "Value"

   ' Get the ProductInfo/Request/Args node
   Set CurrentNode = _
      XMLData.ChildNodes(1).ChildNodes(0).ChildNodes(2)
   For NodeCount = _
      0 To CurrentNode.ChildNodes.Length - 1
```

(continued)

Listing 11-6 ***(continued)***

```
      ' Display the name. Even if the argument doesn't
      ' have a value, Amazon always returns the name.
      Sheet1.Cells(NodeCount + 4, 1) = _
         CurrentNode.ChildNodes(NodeCount). _
            Attributes(0).text

      ' Some input won't have a value.
      If CurrentNode.ChildNodes(NodeCount). _
         Attributes.Length > 1 Then

         Sheet1.Cells(NodeCount + 4, 2) = _
            CurrentNode.ChildNodes(NodeCount). _
               Attributes(1).text
      End If
   Next

   ' Display the next heading.
   CellOffset = NodeCount + 5
   Sheet1.Cells(CellOffset, 1) = "Book Information"
   CellOffset = CellOffset + 2
   Sheet1.Cells(CellOffset, 1) = "ISBN"
   Sheet1.Cells(CellOffset, 2) = "Name"
   Sheet1.Cells(CellOffset, 3) = "Publisher"

   ' Get the ProductInfo/Details.
   For NodeCount = 3 To _
      XMLData.ChildNodes(1).ChildNodes(1). _
         ChildNodes.Length - 1

      Set CurrentNode = _
         XMLData.ChildNodes(1).ChildNodes(1). _
            ChildNodes(NodeCount)

      ' Process each of the data nodes.
      For Each DataNode In CurrentNode.ChildNodes
         If DataNode.nodeName = "ASIN" Then

            Sheet1.Cells(CellOffset + _
                NodeCount - 2, 1) = _
                   "'" + DataNode.text
         End If

         If DataNode.nodeName = "ItemAttributes" Then
            For Each AttributeNode In _
               DataNode.ChildNodes

               If AttributeNode.nodeName = "Title" Then

                  Sheet1.Cells(CellOffset + _
```

```
                        NodeCount - 2, 2) = __
                           "'" + AttributeNode.text
                End If

                If AttributeNode.nodeName = _
                   "Manufacturer" Then

                   Sheet1.Cells(CellOffset + _
                      NodeCount - 2, 3) = _
                         "'" + AttributeNode.text
                End If
             Next
          End If
       Next
    Next
End Sub
```

The code performs a number of pre-request tasks. First it clears the existing data from the workbook so the user doesn't see any old data. (You can use the `ActiveWorkbook.ActiveSheet.Cells.Clear()` method to accomplish this task.)

Just in case the user makes a call without setting a sort criterion, the code checks `SortType` next. If the code doesn't find a `SortType`, it provides a default value. The assumption is that the user simply didn't choose a sort value. Because the call also requires a developer tag, the code also checks `DevTag` — but there isn't any way to recover from this particular error. The application displays an error message, telling the user what to do to correct the problem at this point.

Even though it looks like a simple string, building `XMLFile` correctly is the most important task in this method. You must provide the values as shown. A good way to avoid problems is to put each variable on a separate line, as shown in the example. Using this approach makes it easier to see errors.

The code performs the actual call to AWS next. The code shows the three essential steps you always perform in this order:

1. Create a `DOMDocument` object to hold the XML data.
2. Set the document's `async` property to `false` to ensure that the call is complete before you begin processing the data.
3. Use the `Load()` method to obtain the data from the Web service.

At this point, `XMLData` contains data, even if the call fails. The example checks the `<IsValid>` element to determine whether the return data is valid. If the `<IsValid>` element contains false, an error has occurred and you need to make the call again. You can also process the error information to determine why the call failed.

VBA isn't particularly friendly when it comes to processing XML data. The data appears in levels. You can count these levels by calling AWS using your browser and then translating the results into something VBA can understand. Every time you go down one level in the XML data, you must add a `ChildNodes` collection to your code. Nodes that appear at the same level are numbered starting at 0. Consequently, when you see `XMLData.ChildNodes(1).ChildNodes(1)` in the code, what you're really seeing is the `<Items>` element for the result, which is the second node under the `<ItemSearchResponse>` element. The "Seeing how queries work in a browser" section of this chapter tells you more about using a browser to see how AWS works.

When the call succeeds, the code saves the contents of the `<TotalPages>` element to `TotalPages`. The code uses this information later to determine how many pages of data the user can request.

At this point, the code knows that `XMLData` contains valid information, so it adds headers for the arguments to the worksheet. It then begins processing the data in the `<Arguments>` node of the response. The arguments always provide a name, but they don't always provide a value. The value only appears when the code provides one as part of the `XMLFile` string. The code relies on variables to track the current position of the attribute arguments and obtains the argument name in all cases. It obtains the value only when one exists.

After the code finishes processing the arguments, it begins with the book data. The output includes the ISBN, book title, and book publisher. You can obtain a considerable amount of data about books by adding more `ResponseGroup` entries. The example shows a minimal implementation to simplify the code.

Much of the data appears at different levels in a single `<item>` element; you'll have to exercise care in obtaining the data. For example, the `<ASIN>` element (the ISBN) appears directly beneath the `<item>` element. However, the `<Title>` element appears within the `<ItemAttributes>` element that appears directly beneath the `<item>` element, so you need to go down an additional level.

The data for each book doesn't appear in any particular order and may not appear at all for a given book. Consequently, the safest processing technique is to use a `For Each` loop, as shown in the example. This technique can also reduce the chance that a change in AWS will break your code. Making your code flexible is an essential technique for working with Web services.

Considering types and uses of private Web services

Private Web services come in an amazing diversity of types. Of course, if you haven't found one that fits your work, you can always create a Web service with your Web server. Most development products, such as Visual Studio, also include project templates for creating Web services. These are traditional forms of Web service development.

You might think there are limits to private Web services, but you can literally create a Web service out of most applications by adding the correct interface elements. For example, it's possible to turn your old COM+ application into a Web service using a few simple options that are part of the COM+ environment. For details on how to perform this task, see the article at

```
http://www.devsource.ziffdavis.com/article2/0,1759,1627474,00.
   asp
```

Many applications also let you create a Web service now. For example, you can create a Web service using SQL Server. (You can see a movie on creating a Web service with SQL Server at `http://channel9.msdn.com/Showpost.aspx?postid=129656` or an article at `http://msdn2.microsoft.com/en-us/library/ms345123.aspx`.) The fact is that an enterprise application you're using today probably has the capability to provide a Web service.

The uses of private Web services are as varied as the means to create them. In many cases, the goal is to share data or to at least provide data to someone, perhaps a third party, but often the employees of the organization. However, Web services need not always output data. In some cases, Web services now gather data. Agents on client machines output data to a Web service in order to help the organization function better.

Most of your users won't understand Web services and don't care to know about them. As mentioned many times in this book, most users care about the task they have to perform and not the tool used to do it. Consequently, the better you integrate your Web service into the Ribbon, the less the user will notice it. For example, if you're using a central document repository, you can simply make using it part of the Save As command for the Office product. Users need never know that the document is whisked away to a server somewhere or that the server is feeding the document back to them when they open it.

Part IV

Converting Existing Toolbars and Macros

The 5th Wave By Rich Tennant

"We're using just-in-time inventory and just-in-time material flows which have saved us from implementing our just-in-time bankruptcy plan."

In this part...

Those guys from Microsoft are such comedians! They think you're just going to drop your current investment (and all the hard work done) in Office applications to create all new RibbonX applications. The reality is that you've worked very hard to create your current application suite and no one is going to convince you to give it up. Then again, you do want to use the Ribbon. This part helps you keep your investment in application code and still use the Ribbon to reduce support costs.

You'll use two of the chapters in this part, depending on which environment you plan to use to support the Ribbon. Everyone needs Chapter 12 because it shows what to do with those pesky menus and toolbars in your old application. Chapter 13 is for VBA developers. It demonstrates how you can make your current applications live with the Ribbon and even thrive in this environment. Chapter 14 describes how to update add-ins that you create using a Visual Studio language such as C# or VB.NET. This chapter also describes some of the issues you need to consider when updating your VBA applications to work in VB.NET.

Chapter 12

Simple Fixes for Older Menus and Toolbars

In This Chapter

- Helping users learn the new Ribbon interface
- Designing forms that ease the transition from menus and toolbars
- Using the Menu control to re-create the interface
- Making the best use of existing Office features
- Defining and storing simple interface changes for later use

Most companies can't simply step into the new Ribbon interface and expect things to run smoothly. Users are going to whine about the change, complain about the hassle of learning a new interface, and tell you how inadequate the new interface is in meeting their needs. Anyone who uses the older versions of Office regularly is going to dislike the new interface because it's different and forces them to work in another way. Okay, migration won't be a picnic; it rarely is. (Let's just say you should stock up on your favorite antacid.) However, you don't have to take the changes lying down — you can do things that not only make your life significantly easier but also ease the transition for your users.

This chapter may not meet everyone's needs, but you'll be amazed at how many needs it can meet. Using the simple fixes in this chapter can probably help you meet most of your transition needs, leaving a few hard nuts to crack later. In fact, in some cases, the solutions in this chapter could provide everything you need if you combine them effectively with the proper training and policies.

Training Users for the New Paradigm

At some point, you're going to need to train users to use the new Ribbon interface. Unless you can create a completely customized experience, the user will have to make the changes required to use the new product. As

you may have noticed, creating custom solutions won't reduce the need to provide training. You can reduce the amount of training needed by creating a well-designed application, but you can't eliminate that need completely. This fix may not seem all that simple, but there's a way to simplify things. Here are some methods you can use to fix the problem of moving to the new Ribbon:

- Provide workflow applications that direct user activity.
- Add task-based application additions for users who need to perform specific tasks.
- Use dialog-box launchers combined with forms to help advanced users make the transition.
- Reduce complexity by hiding Ribbon elements that the user doesn't require.
- Use third party add-ins that mimic the menu-and-toolbar interface.
- Use third party add-ins that help you organize the Ribbon features in a manner that reflects user usage patterns.
- Move new users to the Ribbon first, and then the advanced users, since the new user will have less experience with the menu-and-toolbar interface.
- Provide incentives for using the new Ribbon interface.

Easing the transition with RibbonCustomizer

Even though Microsoft's goal in creating the Ribbon is to reduce user confusion while making support easier, you don't have to stick with the Ribbon in its current form. Of course, you can write custom solutions as described in this book, but a custom solution is only the beginning. You might have power users who agree that the new Ribbon is essentially a good thing, but disagree with the layout. In this case, you can purchase a product that makes it relatively easy to modify the Ribbon, but still stay within the original intent of the Ribbon. The RibbonCustomizer (`http://pschmid.net/office2007/ribboncustomizer/index.php`) lets the power user move items around on the Ribbon, including items from the Add-Ins tab.

Using the RibbonCustomizer, you can reduce the learning curve for the Ribbon, provide some semblance of functionality for moderately complex applications without reprogramming them, and provide power users with the flexibility they crave. The important consideration is that the RibbonCustomizer works with the Ribbon and assumes the user will eventually learn to interact with it. You can learn more about the RibbonCustomizer in the "Working with the RibbonCustomizer" section of Chapter 16.

All these methods can help you ease the user's transition to the new Ribbon, but they can't completely negate the need to train the user. Even though there are many interesting ways to overcome Ribbon issues right now (and you'll find more of them as time goes on), the add-in vendors can only do so much. The Ribbon is here to stay, so you need to create a training plan today, rather than wait until the need to move to the Ribbon is dire. (The need will become dire when Microsoft drops support for Office 2003 and you can't protect your network from Office-based viruses any longer. According to a number of sources, mainstream Office 2003 support ends in January 2009 with extended support to January 2014.)

Substituting Forms for Menus and Toolbars

A number of the examples in this book have used the dialog-box launcher as a means for garnering additional information. When a user is used to seeing custom options on a toolbar or menu, you can help reduce the pain of moving to the Ribbon by placing these elements on a form instead. The user can then launch the form and choose options with something approaching the normal technique. In fact, Microsoft already uses this technique in a number of cases. For example, look at the Styles group on the Home tab of Word. When you click the dialog-box launcher in this group, you see the Styles window shown in Figure 12-1.

Sometimes, however, it's not convenient to use a pure RibbonX solution to the problem. For example, you might want to maintain only a `DOT` file and not have to worry about working with a `DOTX` file as well. In this case, you can create a form that contains the toolbar buttons and/or the menu entries. Add the form to the `DOT` file just as you normally would. Now you can add an `AutoOpen` macro that checks the version of Word that the user is opening. Here's an example of the code you need:

```
Sub AutoOpen()
   ' Load the required forms when using the Ribbon.
   If Application.Version = "12.0" Then
      Dim DlgIcons As Icons
      Set DlgIcons = New Icons
      DlgIcons.Show False
   End If
End Sub
```

The original toolbar for this example looks like the one shown in Figure 12-2. The new form appears in Figure 12-3. Even though they aren't precisely the same, the two are close enough that a user can employ them without making any changes to their usual procedures.

Figure 12-1: Use forms to mimic the elements found in older versions of Office.

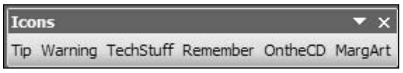


Figure 12-2: It's easy to view a toolbar as simply a series of buttons.

Figure 12-3: Users can still use the form version of the toolbar without extensive training.

Obviously, you can't dock the new toolbar. It has to remain standalone, but most users won't find this new arrangement problematic. In fact, many applications use the toolbars in standalone mode. The biggest benefit of this approach is that you can usually use your existing code without any change. All you need to do is create the same connection between the buttons on the form that you did for the toolbar. In this case, the Tip button uses simple code such as this for connectivity purposes:

```
Private Sub cmdTip_Click()
   ' Create a tip icon.
   Tip.MAIN
End Sub
```

It's also important to note that the code doesn't show the form as a modal form. In order to make the toolbar truly useful, you must set modality to `False` using `DlgIcons.Show False`. Otherwise the Office application simply continues to beep until you close the dialog box.

Be sure to set the `TakeFocusOnClick` property for each of the buttons to `False`. The user expects that clicking a button will simply change an Office feature. Although nothing bad will happen if you don't set this property correctly, the user quickly becomes annoyed with having to click within the document to restore the focus. In addition, you may find that some macros don't work as anticipated when the focus is set incorrectly.

When you know the basic technique, it's easy to add other kinds of standard toolbar features to a form. For example, many toolbars simply provide access to styles in an organized way. You can implement this feature using a form and the appropriate buttons as before. The connectivity code looks like this:

```
Private Sub cmdChapNum_Click()
   ' Set the proper style.
   ActiveWindow.Selection.Style = "Chap #"
End Sub
```

By using this approach, you set the style for whatever the user has selected. If the user hasn't selected anything, then the style naturally affects whatever it was designed to format. For example, a paragraph style will affect the paragraph where the insertion pointer currently resides.

Some toolbar buttons are going to rely on graphics. The easiest way to obtain the icon is to use these steps:

1. **Right-click the toolbar and choose Customize from the context menu.**

   You see the Customize dialog box, shown in Figure 12-4.

2. **Select the template you want to use in the Save In field.**

   The toolbar you want to work with appears on-screen.

Figure 12-4: Use the Customize dialog box to select the template you want to work with.

3. **Right-click the toolbar button containing the image you want to use and choose Copy Button Image.**

   Office places the image on the Clipboard.

4. **Paste the image into your favorite graphics editor.**
5. **Save the image as a `BMP` or other file that VBA supports.**
6. **Click Close in the Customize dialog box.**

   Office closes the Customize dialog box and returns the display to normal.

You can also use applications such as Resource Hacker (`http://www.angusj.com/resourcehacker/`) to obtain images directly from the Office DLLs and EXEs. Unfortunately, the images don't appear in a single `DLL` or `EXE`. For example, you'll find many of them in the `\Program Files\Common Files\microsoft shared\OFFICE12` folder in files such as `MSO.DLL` and `MSOIcons.EXE`.

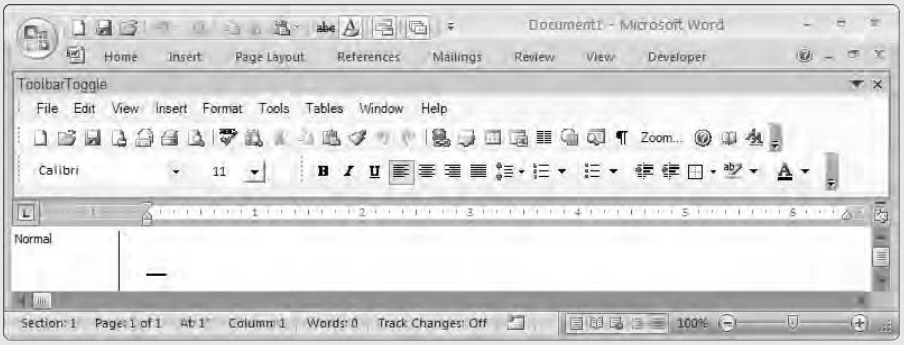
After you obtain the required image, you can add it to the button on the form by clicking the ellipses in the `Picture` property for that control. To make the picture appear to the left of the text (where it normally appears in the toolbar), choose the option labeled `1 - frmPicturePositionLeftCenter` in the `PicturePosition` property for the control.

Even though this solution works very well, some users are going to insist on customizing the setup. Obviously, you'll need to implement your own code for context menus and customization, which isn't a small undertaking. Fortunately, some third-party solutions exist for the context menu part of the solution. For example, you'll find an excellent third-party solution at

```
http://word.mvps.org/FAQS/Userforms/AddRightClickMenu.htm
```

Using third-party tools to regain functionality

You'll find a number of add-in products online that mimic the Office 2003 menus in various ways. The ToolbarToggle add-in from Architects Labs (`http://www.toolbartoggle.com/`) provides an alternative to the Ribbon for Word and Excel (it isn't available for any other Office 2007 product as of this writing). If you hide the Ribbon, you almost feel as if you're working with Office 2003, as shown here (note that this is the product running under Vista; the Windows XP view is slightly different).



Of course, you have to hide the Ribbon because it still exists and you won't get rid of it. The ToolbarToggle application actually appears below the Ribbon as a new entry. That's why it can perform the tasks it does. You can fully customize the menu, just as you do in Office 2003. At the time of this writing, however, it doesn't appear to work with macros that modify the toolbar. For example, if your application depends on a custom toolbar, you'll have to re-create it in ToggleToolbar. The point is that you *can* re-create the toolbar. Simply right-click in the toolbar area and choose Customize from the context menu as you normally would.

ToolbarToggle also provides you with the Standard and Formatting toolbars. If you want other standard toolbars, you have to create them yourself. For example, you won't find the Reviewing toolbar provided as a default feature, even though this feature comes with Office 2003.

Even though the menus are very close to the Office 2003 version and the toolbars are a nice feature, you'll find that the keyboard accelerators don't work the same as before. For example, Alt+F displays the Office Menu, not the File menu (as you might normally expect).

I checked out ToolbarToggle on several machines; it's apparent that it works flawlessly under Windows XP. However, older versions of the software are very buggy under Vista. Make sure you have the latest version of the software (version 2.0.0.8 or higher) before you attempt an installation under Vista. You have to run the target application as an administrator the first time, which means opening the `\Program Files\Microsoft Office\Office12` folder, right clicking the application, and choosing Run As Administrator from the context menu. You must have the User Account Control (UAC) feature enabled for this product to work. I also noticed a definite performance penalty under Vista that I didn't see under Windows XP. The vendor is working to remedy these remaining issues. The vendor's technical support was nearly relentless in helping me solve product issues.

You can also see example code for this task at

```
http://www.vbforums.com/showthread.php?t=402050
```

With a right-click menu in place, you can create an environment where the user will see very little change to a custom application due to the Ribbon. For example, you can use the context menu to dynamically add and remove buttons from the toolbars.

Depending on the complexity of your application and how closely you plan to mimic the menu-and-toolbar system, using the approach outlined in this section can require very little time. Using simple forms, it can require as little as half a day to completely convert an application that contains four or five toolbars, including testing time for each of the buttons.

Relying on the Menu Control

The Menu control can be a best friend when you're working with RibbonX conversions. Using this control lets you create an equivalent of the menus that your custom applications used in the past. What you'll see is a menu setup similar to the one shown in Figure 12-5.

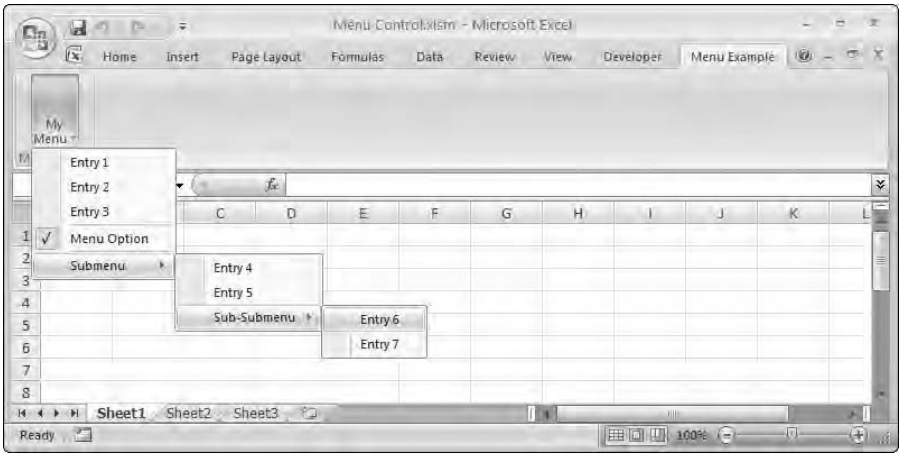


Figure 12-5: Create menus to mimic the menus you used in previous versions of Office.

You can layer the number of levels as deeply as needed. The menus can contain options, so users can check features, separators, and buttons that act as

menu selections. In short, you can obtain all the functionality normally associated with a menu in previous versions of Office. Listing 12-1 shows the code used to create the display in Figure 12-5.

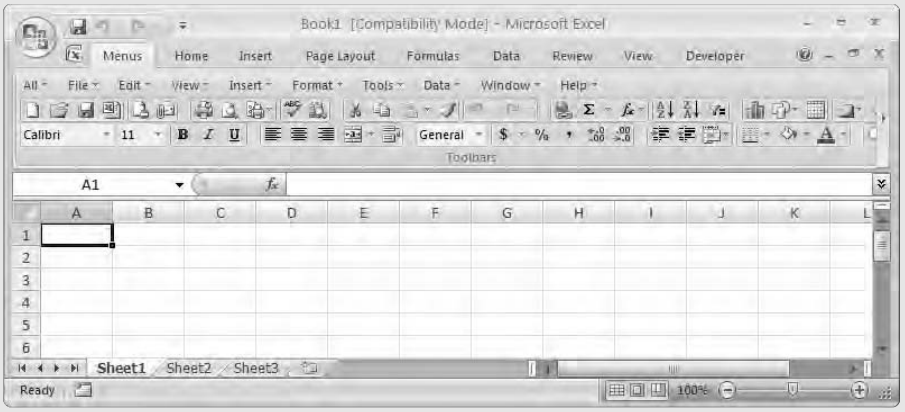
Listing 12-1: Using a Menu Control to Create an Older Office Setup

```
<ribbon>
   <tabs>
      <tab id="MenuTab" label="Menu Example">
         <group id="MenuGroup" label="Menu Group">
            <menu id="ExMenu"
                  label="My Menu"
                  size="large">
               <button id="Entry1" label="Entry 1" />
               <button id="Entry2" label="Entry 2" />
               <button id="Entry3" label="Entry 3" />
               <menuSeparator id="Sep1" />
               <checkBox id="cbOption"
                         label="Menu Option" />
               <menuSeparator id="Sep2" />
               <menu id="SubMenu" label="Submenu">
                  <button id="Entry4"
                          label="Entry 4" />
                  <button id="Entry5"
                          label="Entry 5" />
                  <menu id="SubSubMenu"
                        label="Sub-Submenu">
                     <button id="Entry6"
                             label="Entry 6" />
                     <button id="Entry7"
                             label="Entry 7" />
                  </menu>
               </menu>
            </menu>
         </group>
      </tab>
   </tabs>
</ribbon>
```

When using this kind of setup, you create the menu structure first and ensure that it looks the same as the menu setup used in your Office application. Once you have the setup correct, you can begin adding callbacks to make the menu entries active. The Menu control works just like any other RibbonX feature. The main difference is that it looks like the old menu system to a user. Although you won't normally use this kind of setup when working with the Ribbon (because it doesn't follow the new workflow methodology), you can add menus to help ease the transition to the Ribbon.

The Menu control in action

Sometimes users will definitely want the appearance of the old menu-and-toolbar system when they're working with Office 2007. You can't provide a complete lookalike, but you can improve the appearance of the Ribbon to look more like Office 2003 using some third-party products such as Classic Menu for Office 2007 by Addintools (`http://www.addintools.com/english/menuoffice/`). When you initially install the application, it displays a dialog box that lets you choose how the menu appears. In addition, you can choose which applications (Excel, Word, and PowerPoint) use the add-in. This add-in provides a Ribbon tab that looks similar to the Office 2003 menu, as shown here.



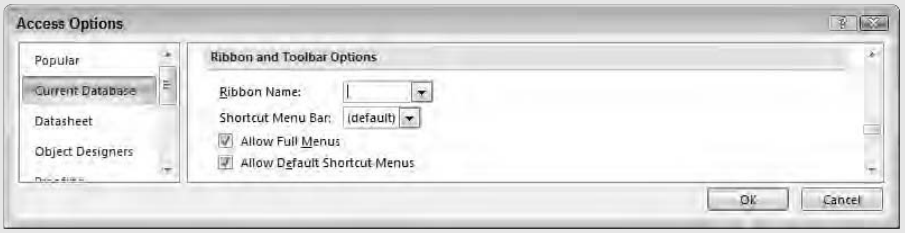
Classic Menu relies on the Menu control described in the "Relying on the Menu Control" section of this chapter to perform its job. In fact, this add-in is probably the best example of the complete Menu control solution for Office 2003. However, you'll find a few differences between Office 2003 and this add-in. For one thing, some of the accelerator keys don't work as expected. Pressing Alt+F still displays the Office Menu, in place of the File menu. You'll also notice some differences in nomenclature, such as the `Header_Footer` command on the View menu (it should appear as the `Header & Footer` command). A final issue is that you can't change the Classic Menu settings unless you also include a product such as RibbonCustomizer (see the "Easing the Transition with RibbonCustomizer" sidebar of this chapter for details). Even so, an experienced user can get up and running quite quickly with this add-in.

Using Existing Office Features

Besides using forms and the Menu control to mimic Office features, you can choose to move components of your application to Office features that exist in all newer versions of Office. The following sections describe two common Office features you can use to your benefit in moving applications from earlier versions of Office to the new Ribbon interface.

Using menu bars in Access to regain menus and toolbars

Access offers a number of features not found in the other Office products. For example, you can choose not to load any Ribbon at all. To remove the Ribbon, choose Office Menu➪Access Options. Select the Current Database folder and clear the Ribbon Name field entry, as shown here.



At this point, you must close and reopen the database to see the change. Access opens without any special Ribbon features, though it does load default Ribbon features and the Office Menu as it would normally.

It's also possible to display your custom menu bar in Access. To use this feature, you must open the Access database as an MDB file, not as an ACCDB file. In this case, the Current Database folder changes, as shown here.

Select the name of the toolbar you want to use, clear the Allow Built-in Toolbars option, and close and reopen the database. You'll see your custom menu in place of the Ribbon, as shown here.

Unfortunately, Access is the only Office product to provide this feature. Consequently, in many respects, Access can provide the easiest move for organizations willing to use the older MDB file format and do without any Ribbon functionality at all.

Relying on context menus

The most reliable of the standard Office features is the context menu because it appears in all versions of Office since the earliest versions. You can create complex menu structures using context menus; you can focus the controls on specific document elements. The user is less likely to make errors when you use this approach. Of course, the downside to using a context menu is that the user must know to right-click in order to use it. Listing 12-2 shows how to add a context menu entry to an Excel workbook. This technique works with any type of Excel workbook, including the new XML formats used by Office 2007.

Listing 12-2: Defining a Context Menu Entry in Excel

```
Private Sub Workbook_Open()

   ' Create a new context menu element.
   Dim NewControl As CommandBarControl

   ' Delete any existing copies of the control.
   For Each NewControl In _
      Application.CommandBars("Cell").Controls

      If NewControl.Caption = "Insert Date" Then
         NewControl.Delete
      End If
   Next

   ' Create a new control.
   Set NewControl = _
      Application.CommandBars("Cell"). _
         Controls.Add(Temporary:=True)

   ' Configure the control for use.
   NewControl.Caption = "Insert Date"
   NewControl.OnAction = "RibbonX.InsertDate"
   NewControl.BeginGroup = True
End Sub
```

The code begins by defining a new context menu entry, which is of type `CommandBarControl`. The code then searches for an existing entry with the required `Caption`. You must perform this search or the Office application could end up with multiple copies of the same context menu entry.

After the code checks to make sure no context menu entry exists, it creates a new entry, using the `Controls.Add(Temporary:=True)` method. The entry isn't configured at this point. The code performs that task next by assigning the control a `Caption` (which is an `OnAction` event) and telling Excel that this context menu entry begins a new group. Every time the user right-clicks a cell, the new context menu entry appears. When the user selects this item, the code calls on the `InsertDate() Sub` shown here:

```
Public Sub InsertDate()
   ' Insert the date.
   ActiveWindow.ActiveCell = DateTime.Now
End Sub
```

Although this is a very simple example, it works well with context menu entries of any complexity. You can use this technique to ensure that your application will work in any of the Office environments, but it does require some additional action on the part of the user, which isn't always ideal.

Relying on task panes

Task panes can access situations in which the user must perform complex actions. As with context menus, task panes work equally well in newer versions of Office. However, you'll find that you lose some compatibility with older versions of Office, such as Office 2000 and Office 97. Despite vigorous efforts by Microsoft, some companies still have copies of these older versions of Office hanging around; a task-pane solution won't work in such a situation.

A complete discussion of how to create a custom task pane is outside the scope of this book. You can find a good article on how to create a custom task pane using Visual Studio at

```
http://msdn2.microsoft.com/en-us/library/aa722570.aspx
```

If you're using VBA, it won't let you create a task pane. Another article, however, provides a great discussion of moving your user-form solutions to a task pane. You can find it at

```
http://msdn2.microsoft.com/en-us/library/aa830702.aspx
```

Performing Simple Interface Changes and Storing Them

It's important not to disregard some simple changes you can make to the interface that Office supports directly. Microsoft has provided methods that let the user perform limited changes to the Office interface setup. Although these changes might seem meager at first, using them carefully can make an intimidating Ribbon friendlier. The following sections describe the changes a user can make to the interface that can help ease the Ribbon pain.

Customizing the Quick Access Toolbar

Clicking between tabs on the Ribbon can quickly become frustrating and time-consuming, especially if you use certain commands often. The Quick Access Toolbar (QAT) answers the need to access a feature quickly. You use it to hold items that you rely on to perform tasks often (such as the Copy, Cut, and Paste commands). You can add either full groups or individual commands to the QAT as needed. Simply right-click the element you want to add and choose Add to Quick Access Toolbar from the context menu. The element you right-clicked appears on the QAT immediately.

Sometimes you want to customize a little more than the context menu allows. In this case, you can right-click anywhere on the Ribbon or QAT and choose Customize Quick Access Toolbar from the context menu. You'll see the Customize folder of the Office application's Options dialog box. Figure 12-6 shows the Word Options dialog box.

At this point, you can find any command and place it on the QAT by clicking Add. If you want to add separators between commands, simply locate the `<Separator>` entry and click Add. When you decide that you no longer want a command on the QAT, highlight its entry and choose Remove. Click Customize when you want to further customize access by using keyboard shortcuts. If you find that the QAT has become too full, you can always display it below the Ribbon and gain some extra space.

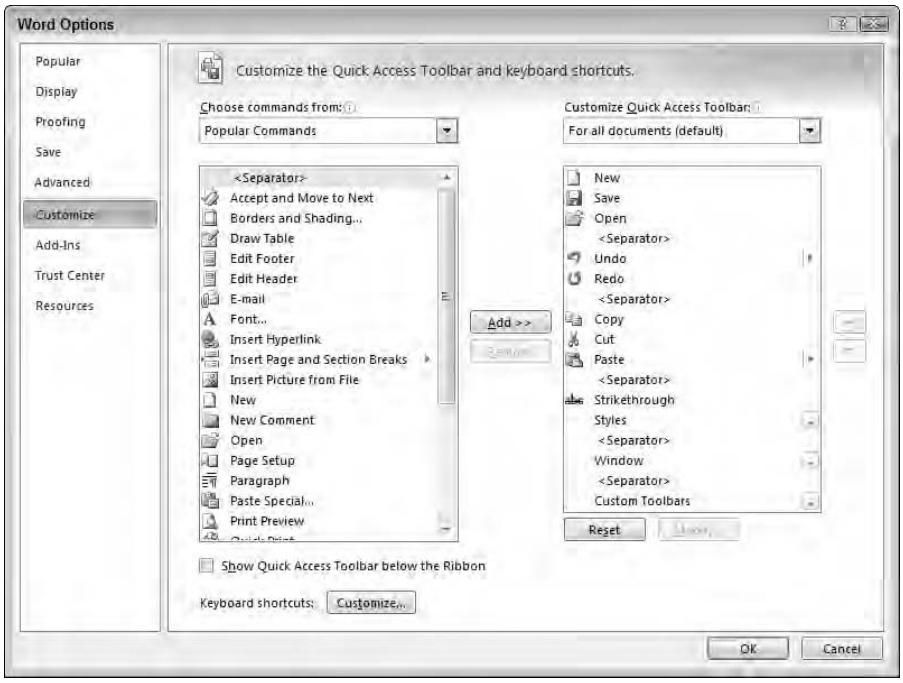


Figure 12-6: Customize the QAT to meet specific needs.

Modifying the Quick Style Set in Word and storing it in the template

The Styles gallery on the Home tab always displays style selections in the order that Microsoft thinks you want to use them. Unfortunately, Microsoft guesses wrong most of the time. Customizing the Styles gallery can save you significant time by making the styles you use most often instantly accessible.

To remove a current style from the list (you have to get rid of all of those entries you'll never use), right-click its entry and choose Remove from Quick Style Gallery from the context menu. Office will remove the entry and move all the other entries up.

To add an entry to the gallery, you must click the dialog-box launcher in the Styles group to display the Styles dialog box. When you see a style you want to add to the gallery, click the down arrow at the right side of the entry and choose Add to Quick Style Gallery from the context menu.

Office always adds entries to the Quick Style Gallery in the order you choose them. You can't reorder the entries once you add them; be sure to choose entries in the order you want them to have when they appear in the gallery.

You may decide you want to have several different setups to meet different needs. Office lets you create Quick Style Sets — files that contain the galleries you want to use. To save a Quick Style Set, follow these steps:

1. **Click the down arrow on the Change Styles button.**
2. **Choose the Style Set⇨Save as Quick Style Set command.**
3. **In the dialog box that appears, enter the name of the set you want to create and then click Save.**

When you want to change the styles, you simply select the style name from the list that Office provides on the Change Styles menu.

Chapter 13

Conversion Techniques for VBA Users

In This Chapter

- Understanding the conversion issues
- Deciding how to perform the conversion
- Creating a list of RibbonX solutions
- Defining required RibbonX changes for Word, Excel, and PowerPoint
- Defining required RibbonX changes for Access
- Defining required RibbonX changes for Outlook
- Working with parallel version solutions

Creating a new solution for RibbonX is easy enough once you understand the techniques. The examples in Chapters 6 through 11 explore many of the methods you can use to create great workflow applications that serve the user better, reduce support costs, and still don't cost you a lot of time to put together. All of these solutions are fine, but they don't consider the thousands of lines of code that you already have and the time you've invested in them.

In some cases, you're going to have to make the code you have now perform the required tasks despite what Microsoft would have you do with RibbonX. You can't simply throw that old code away. Chapter 12 begins exploring the issues of moving from older versions of Office to Office 2007. The generic techniques in that chapter are going to prove very useful to you. This chapter moves on to more detailed solutions that focus specifically on VBA.

One of the issues that you'll need to overcome is that VBA is outdated or inadequate for use with the Ribbon. You've seen too many examples of just how well VBA can work in this book to decide that VBA no longer fulfills its role as an easy-to-use solution. However, Microsoft is starting to work pretty hard at convincing people that they must move to Visual Studio in order to obtain the functionality they need for the Ribbon, and that simply isn't true. You can move your existing VBA applications to the Ribbon, and you can make them work in a dual Office environment where the same template serves both the menu-and-toolbar system and the Ribbon at the same time.

Defining the Issues behind VBA Conversion

Before you can begin the VBA conversion, you need to consider the essential issues behind the conversion. By knowing what will and will not convert easily, you can make a better decision about whether a conversion is worthwhile. It's actually easy to convert many existing templates if you followed good practices in putting them together initially.

You begin by dividing your code into functional areas. Business logic will work no matter which version of Office you use. Consequently, if you kept your business logic separate from everything else when working with Office 2003 and older versions of Office, you should already have a great start on the conversion process.

Most forms will convert quite easily and some won't need any changes at all. To determine whether a form will work, open the UserForm in the Visual Basic Editor and choose Run➪Sub/UserForm from the Visual Basic Editor menu. Office will display the form for you so you can test it. Test each of the buttons on the form to make sure they work in both Office 2003 (or older) and Office 2007.

Menus and toolbars won't convert to the Ribbon. The "Substituting Forms for Menus and Toolbars" section of Chapter 12 shows one method of working through the problems of toolbars. You can simply replace the toolbar with a form and provide connections with VBA code. The code you must write is short, but you have to provide one button for each button that originally appeared on the toolbar. You can replace menus with the Menu control on the Ribbon if desired.

User interface code, anything that works with a `CommandBar`, is likely to cause problems. When possible, convert the code into a physical alternative using a form. For example, if your code creates a special toolbar that the user sees only when performing certain tasks, use a form for the purpose instead. You can still choose to show or hide the form, just as you would with the toolbar, but now the solution will work with all versions of Office.

The big issue for conversions is that you can't use the new extended versions of Office documents. You'll still need to maintain the `DOT` file, rather than move to a `DOTX` file. Even though Microsoft provides a special add-in for Office 2003 that lets you read `DOTX` files, you'll find that the support is for the data only, not for new features that you use. This limitation makes sense when you consider that Office 2003 isn't designed to understand the Ribbon. Of course, using just `DOT` files does limit you in some respects and means that you won't get all of the benefits of working with the new file formats.

All kinds of issues come into play when you perform conversions. If you're in a government agency, you might have a requirement to produce documents that rely on XML to ensure future readability, which means that a conversion won't work in most cases.

Your company may have a good reason to use the new document formats. If so, you'll have to consider whether you can still use your existing business logic and forms in one of the extended document formats. If necessary, export the code first, create the new document, and then import the code. To export your code, right-click the module in the Project window of the Visual Basic Editor and choose Export File from the context menu. You'll see an Export File dialog box like the one shown in Figure 13-1.

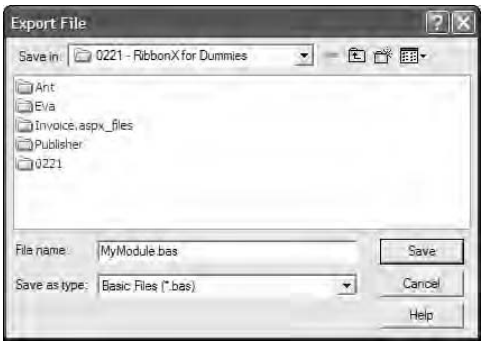


Figure 13-1: Export your code as necessary for use in new files.

Choose a location to store the code and click Save. The process is similar for importing the file later. Right-click anywhere in the Project window and choose Import File from the context menu. Locate the file that contains the code you want to use and then click Open in the Import File dialog box. Some people are advocating a switch to add-ins when you must update, and you might find that making a break to Visual Studio is your best option (although you should consider this approach under extreme conditions only).

Creating a Conversion Strategy

At some point, you'll have a list of problem code and problem features — including (at minimum) menus, toolbars, and user-interface code. You'll need a strategy for converting these items into something the Ribbon can understand. The technique you use depends on many factors, including these:

- Multiple-version compatibility
- User training level

- Time limitations for conversion
- Conversion budget
- Need for new file-format support
- Reasons for updating to Office 2007
- Implementation details of existing code

The reason these criteria are so important is that they define how you convert an existing application. For example, a UserForm works fine in a multiversion environment where you maintain the existing Office 2003 file format, but it might not work as well when you must use the extended file formats. Although a complete conversion to the Ribbon is probably optimal, it's not the best solution when you have time constraints or when the user training level is mediocre. The following sections describe more issues you should consider as part of your conversion strategy.

Using forms

This chapter has already discussed using forms, but it's important to understand that this is probably the best strategy for many situations. If your toolbars are simply providing quick access to styles, then using a form that replicates the toolbar is extremely fast and requires no user retraining at all. This strategy also works great when you simply call a macro that has no user-interface code in it. Code that changes document content, works with external sources, and implements all your business logic will work just fine in Office 2007, so using that existing code makes sense.

Using existing menus and toolbars

Many people are going to tell you that the Add-Ins tab is completely inadequate for updates to the Ribbon. However, the Add-Ins tab is only inadequate for complex setups. If your application consists of a single toolbar or menu (or even two or three), the user probably won't notice the difference after a very short time. All you really need to do is train the user to go to the Add-Ins tab instead of working directly with the toolbar or menu. The only time you need to use a more comprehensive system is when your application contains many toolbars or menus, or it relies on the `CommandBar` object.

Of course, you have to consider the tradeoffs of using the Add-Ins toolbar. If you have a hundred or so users to support and they all have trouble turning their system on in the morning, you're probably better off exploring any solution

other than using the Add-Ins tab. All the support calls needed to work each user through the Add-Ins tab individually (several times, no less) will probably cost a lot more than simply converting the code to some other form.

Before you go to a lot of trouble, however, make sure you check the third-party solutions discussed in Chapters 12 and 16. These solutions can help you overcome many of the obstacles of working with the Ribbon. They can cost quite a bit, but they'll definitely cost a lot less than retraining all those users and may cost less than performing a conversion. You have to consider how the third-party solution is going to fit in with your application and whether it provides enough functionality to keep support costs to a minimum.

Designing toolbars and menus, rather than creating them

At least some of the incompatibilities between Office 2007 and older versions of Office are due to developers creating menus and toolbars programmatically, rather than defining them statically. Using the programmed menus and toolbars means the menu and toolbar entries won't appear on the Add-Ins tab, which means that even if you have a third-party support solution available, it won't work. Whenever possible, consider converting the code that creates the menus and toolbars to a static solution during runtime. Better yet, because you have to perform this conversion anyway, use forms when you need to support a dual-interface environment.

If you're performing a complete conversion to Office 2007, but want to maintain some of the functionality of the existing application, try to create a separate tab for all the menu items and each major toolbar. Place the tabs in the order in which you expect the user to need them. Obviously, this solution runs contrary to creating a task-based or workflow-oriented solution (the *forte* of the Ribbon interface), but it could reduce training costs by giving users something close to what they're used to seeing. The examples in this book show how to create a lot of tabs. You'll also find a discussion of the Menu control in the "Relying on the Menu Control" section of Chapter 12.

Using the Quick Access Toolbar (QAT)

By far, the fastest conversion technique is to place items from the Add-Ins tab on the QAT. The "Customizing the Quick Access Toolbar" section of Chapter 12 tells you how to use this approach to performing a conversion. If you perform the task correctly, the user sees each major group on the Add-Ins tab as

an entry on the QAT, making the features accessible no matter what else the user is doing.

Unfortunately, in many cases the QAT isn't a good conversion solution. The first reason is that the user doesn't obtain any significant cues from the QAT while puzzling out the purpose of an entry (as shown in Figure 13-2). All the user gets for a cue, in this case, is Custom Toolbars — hardly an effective way to help the user find a particular application feature. The user would need to know that all of the toolbars from the old application appear in this one area. Of course, less skilled users will take a lot longer to adjust (making third-party solutions and the forms approach seem a lot better).

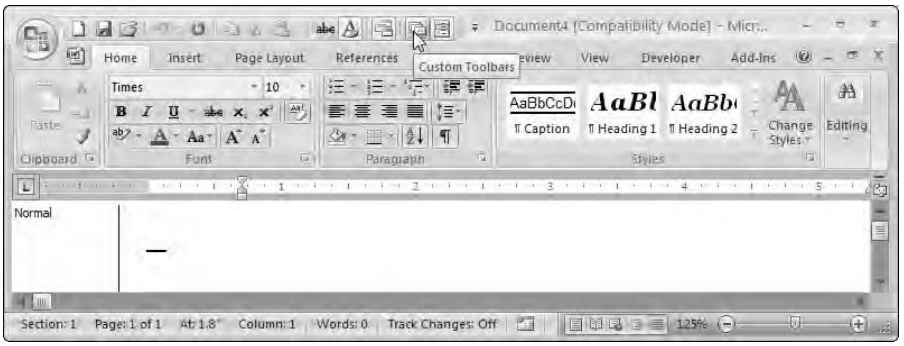


Figure 13-2: The QAT can work as a conversion solution, but only if you know where to look.

The second problem with the QAT is that it tends to show only part of long toolbars, as shown in Figure 13-3. The problem, in this case, is that you have to scroll back and forth to find anything (notice the right-pointing arrow button at the right in the screenshot). It's an annoying way to work and wastes plenty of user time. When you multiply the wasted user time by all the users of an application, it's easy to see why the developer should take the plunge and come up with a better solution.

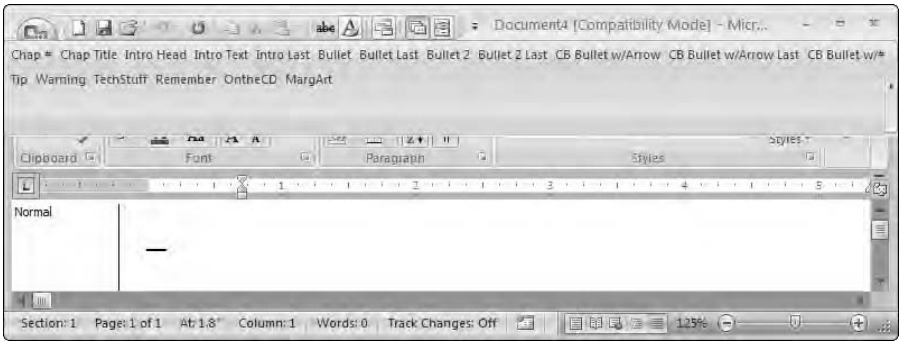


Figure 13-3: Scrolling back and forth to locate a toolbar or menu feature isn't very efficient.

Even with its limitations, however, placing the Add-Ins tab entries on the QAT does make them more accessible, which is useful. At least you can find what you need without having to select the correct tab, too. No matter where you are, you can find the entries, and each group appears as a separate entry, rather than as part of a very long Ribbon on a tab. For example, imagine trying to locate menu commands when the toolbars already require that you scroll through three screens' worth of Ribbon. Placing the menu commands on the QAT gives you single-click access to the group.

Developing a List of RibbonX Changes

At some point, you'll have considered the issues involved in converting your VBA application and will have created a strategy for performing the task. In some cases, you'll use multiple strategies, one to solve each of the issues you've identified. Although (for example) using the QAT may work for some special menu items you've added, it won't work for the toolbars because the toolbars are too long. The point is that you know what you're going to do to overcome each issue.

Now it's time to create a list of the needed changes. The point of this exercise is to ensure that you make all the changes and don't miss anything. Your list of RibbonX changes should consider these elements (in the order shown):

- Menus
- Menu commands
- Toolbars
- Toolbar commands
- UserForms
- Modules
- Class modules

Make sure you go through each element carefully and consider the strategy that works best. However, the important issue is to perform a comprehensive check and to use (or devise) a method for ensuring that you check everything. Perform the check in order or you'll find that you miss items.

Of course, in addition to listing individual items, you need to consider interactions. A toolbar button could open a `UserForm` that calls a `Sub` in a module that relies on a class to create a menu item during runtime. The class might actually service more than one `Sub` and you might have to devise a way to answer the issues for each one of those uses. Searching through the

convoluted hierarchy of some applications can consume a great deal of time, but you have to do it to ensure that your RibbonX solution works as intended.

It pays to perform a sanity check on your list once you complete it. Have another worker (or other independent party) check your list for accuracy. Up until this point, it's relatively easy and painless to correct errors. After you start implementing your solutions, however, it's hard to make changes; they become costly, time-consuming, and error-prone. Consequently, you want to be sure the RibbonX change list you create is completely accurate, follows company policy, considers all the issues, and uses the best solutions for your particular needs.

Creating a Conversion Solution for Word, Excel, and PowerPoint

You'll find that converting a solution for Word, Excel, and PowerPoint is relatively straightforward. Their use of menus and toolbars actually work pretty well with the Ribbon when you use a form or a third-party solution. In fact, it's easy to convert the toolbars to forms in any of these applications.

Of course, if you have a lot of toolbars, you'll need to provide a method for hiding them as needed. What you might end up providing is a master toolbar that shows all of the available toolbars. When a user clicks on a desired toolbar, its form appears on-screen. Even though the master toolbar might not appear in your original application, it provides a means of selection that the Ribbon interface doesn't provide; the training time required for users to adjust to it is minimal. For problem users — those with limited training and experience — you could always provide a set number of toolbar forms that allow access to standard features and leave out the toolbars that show advanced functionality.

Always look for potential problems with a generic solution. For example, Word is especially susceptible to the problem of toolbar forms showing up at the wrong time. When a user opens multiple documents that use the same template, the solution shown in the "Substituting Forms for Menus and Toolbars" section of Chapter 12 opens one set of toolbar forms for each document. The user could close the extra forms, but they really shouldn't have to, and you'll definitely receive complaints if they do. You can modify the solution to set a `Boolean` value to indicate the presence of the form, as shown here.

```
' Create and initialize the check variables.
Dim DlgIconsPresent As Boolean
DlgIconsPresent = False

' Check to determine if the forms are already
' present.
```

```
Dim ThisForm As Object
For Each ThisForm In UserForms

   ' When the Icons form is present, set its
   ' indicator to True.
   If ThisForm.Caption = "Icons" Then
      DlgIconsPresent = True
   End If

Next
```

This code sets an indicator, `DlgIconsPresent`, to `False`, which means that the form isn't present. The code then searches through the VBA `Global` variable, `UserForms`, looking for the form in question based on its `Caption` property. When the code finds the form, it sets `DlgIconsPresent` to `True`, which indicates the form actually is present. Now all you need to do is make displaying the form conditional, as shown here:

```
' Display the Icons toolbar when necessary.
If Not DlgIconsPresent Then
   Dim DlgIcons As Icons
   Set DlgIcons = New Icons
   DlgIcons.Show False
End If
```

This solution takes care of the problem of adding a RibbonX conversion to templates. Remember that you don't need to implement the solution like this when you're working with individual files (such as an individual Excel workbook). Adding the solution when it isn't needed can cause the opposite problem: The toolbar form doesn't show up at all, even if it's needed. When you don't know how to implement the toolbar-form solution, try the easier solution first, and then add selection code as necessary.

When you're creating an actual Ribbon for your conversion solution, applications for these three Office products tend to work best when you can create tabs that reflect the pre-Ribbon configuration:

- Always use a single tab for each toolbar.
- Place all menu additions on a single tab.
- Rely on the Menu control to provide a functional lookalike to your original application.

Creating a Conversion Solution for Access

Access application conversions can rely on many of the same tips found in the "Creating a Conversion Solution for Word, Excel, and PowerPoint" section

of this chapter. However, Access provides the capability to use other solutions as needed; because you don't use Access directly, in many cases you might not need a solution at all.

When you do need a custom solution for Access, consider first all of the easy techniques. For example, the "Using menu bars in Access to regain menus and toolbars" sidebar in Chapter 12 points out a method you can use to display your toolbars directly without any additional work on your part. As pointed out in the sidebar, this solution does come with some risks and tradeoffs; consider whether it's an appropriate solution for you. Sometimes, using what appears to be the old solution denies access to too many new features and makes Access harder to work with.

Unlike other Office applications, Access provides a simple method for using multiple Ribbons. In addition, you can create Ribbons at multiple levels. Consequently, you can customize Access to a degree you won't find in other Office products. However, with flexibility comes complexity. Make sure you look at the various solutions in Chapter 8 and try them with a simple form of your application before you make a decision. It's quite possible that you'll find the multiple Ribbons that Access can support to be more of a nuisance than a help.

Using Access as a Ribbon storage solution

Although you can't use this technique in VBA, storing Ribbons in Access for add-ins is a valuable technique. If the add-in already uses Access (or even SQL Server) for data storage, you already have the required connection in place, so using it to find an appropriate Ribbon is a low-cost solution. A Visual Studio add-in written in VB.NET or C# can query Access for a list of Ribbons for a particular application. The add-in can then choose the appropriate Ribbon to fit the user's role, the document, the environment, and any templates.

The user role is important because you might want to provide solutions to managers that you don't provide to the average user. For example, an authorization button should appear on the manager's Ribbon, but not on the user's. Making this slight change in functionality can make the application more secure.

The document or its content can make a difference. For example, the add-in could check for keywords, the document name, custom properties, or other criteria to determine which Ribbon to load.

The environment can help you choose special Ribbon features. A user calling in from home may not need a printer button when you only want users to print from work. A user calling in over an Internet connection may not need access to sensitive data, especially while working at an Internet café.

The template also makes a difference. You can detect the template and choose an appropriate Ribbon based on the template features. A letter template may not require access to graphics, but a layout template almost certainly will.

The Ribbon(s) you provide for Access won't affect any other application that relies on the database for input. Consequently, you don't have to consider the other applications in your Ribbon solution. Some developers seem confused about this issue because other Access features *do* affect the applications it supports. (For example, a query you create in Access is quite usable in Word.)

Access doesn't provide any means of creating an add-in solution with Visual Studio. Therefore any solution you define has to rely on VBA. The lack of templates in Access means you must create an individual solution for every database. Of course, you can use cut-and-paste techniques to reduce the amount of work involved, but you still have to add more time to your conversion estimate for each database. Make sure you include the time required to create new macros to go with your VBA modules.

Creating a Conversion Solution for Outlook

Outlook presents the most problematic Office product when it comes to many conversions because nothing is tied together. The Microsoft documentation consists of a number of Excel files because you can't get everything you need from a single file. What you'll end up with is a number of individual solution scenarios instead of a single, cohesive scenario (as you do with other Office products). Chapter 9 describes all of the individual pieces you'll need in order to provide a solution to the conversion problem in Outlook.

You do gain some benefits in Outlook. The main application still uses the menu-and-toolbar setup (as shown in Figure 13-4), which means you don't have to do anything to convert applications at this level. You'll also find that form toolbars work very well when you need to replicate an existing toolbar in Outlook. Because there aren't any documents to speak of in Outlook, you have to create only one conversion solution, rather than separate solutions at the application, template, and document levels.

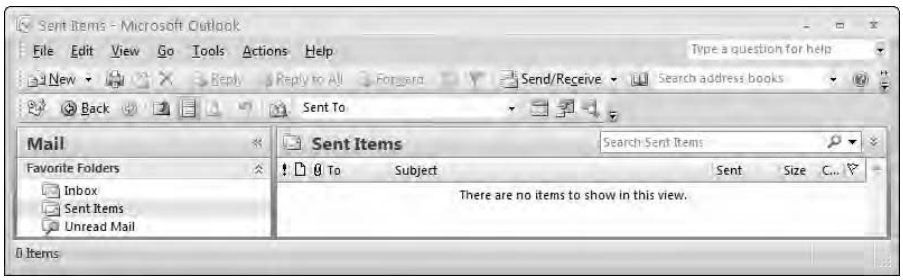


Figure 13-4: Outlook can present a very difficult conversion target because it combines old and new elements.

It's only possible to create RibbonX solutions for Outlook using a Visual Studio add-in. If you're a VBA developer and don't really want to learn Outlook, verify these essentials in the order shown in the following list:

- The solution works as is (test it thoroughly).
- It's possible to use a form to replace the missing functionality.
- Placing the feature in a different location (such as a special toolbar on the main Outlook display) provides the required functionality.
- The user doesn't actually use the missing feature, so you don't replace it.
- It's possible to replace the missing functionality by modifying Outlook data with other applications.

Creating a conversion-time estimate

When you go through the conversion process, one of the first items that someone will ask for is an estimate of the time the conversion will require. Most companies have a policy in place for estimating time; the policy works better in some situations than in others, but it provides a baseline estimate based on the company's experience. Unfortunately, you don't have a policy in place for the conversion (in VBA) from the menu-and-toolbar system to the Ribbon.

Your first estimate is how much time is required to define the issues — it's a look at what you need to convert in the first place. The second estimate involves deciding on a conversion strategy (with a rough timetable), which defines how you plan to perform the conversion. Only when you have these two estimates in place can you do the work to come up with a comprehensive list of the items you need to convert. Of course, the "someone" asking the questions may want a complete estimate that you really can't provide until you do have the list — the only thing you can estimate at the outset is how long creating the list will take.

After you create a list of items you have to convert, you must look at the list in terms of each Office product. Access applications typically take the least time because Access provides so much flexibility, followed by Word, Excel, and PowerPoint, and then by Outlook. When working with Outlook, you have to perform the testing phase very carefully because you may find that more of the original application works than you anticipated.

The solution you choose also affects the time-conversion estimate. In converting three Word templates, I found that I could create form toolbars in a short time. One template took about a half a day, including the time it took to create all the forms required to mimic the toolbar. Performing the same conversion using a RibbonX tab required two days. Even though the RibbonX solution does work better, it required extra time because I had to create an XML file and perform additional testing to ensure everything worked. In addition, completing the linkage from the Ribbon to the existing VBA code required additional lines of code.

This list begs the question of why you should go to such extremes to avoid creating an add-in. After all, add-ins are a perfectly legitimate way to provide RibbonX functionality. The answer is somewhat complex, and it depends on what your company is trying to achieve. The following list describes some of the reasons that using an add-in might be a bad idea with Outlook:

- Your company doesn't currently have any Outlook add-in code and doesn't wish to create any.
- Maintaining two code bases, one for Visual Studio and another for VBA, can make support difficult at best; it reduces the efficiency of your development team.
- Your company doesn't have anyone versed in VB.NET or C# to create the add-in.
- Working with add-in code for Outlook is considerably harder than add-in code for other Office products because of the Outlook structure.
- The use of both VBA and Visual Studio can create unusual debugging situations and make finding some problems nearly impossible.

Designing Parallel Version Solutions

Not everyone's going to start using Office 2007 immediately, and it's unlikely that many companies will perform a wholesale update. In other words, you'll have to work with both Office 2007 *and* some earlier version of Office for quite some time. Of course, the Ribbon interface makes it quite hard to create a dual-interface solution. The "Substituting Forms for Menus and Toolbars" section of Chapter 12 provides one of the best options for parallel solutions. The following sections describe some of the issues you need to consider as part of creating such as solution.

Considering the Office XP/2003 user

The user of an earlier version of Office is accustomed to working with the menu-and-toolbar interface. Changing the interface to some other form, no matter how superior it is, is going to cause a certain amount of confusion, increase support costs, and require extensive training. The Form toolbar is possibly the best solution for this particular user if you must move to Office 2007.

It's important to remember that the Office XP/2003 user already has a particular mindset and established skills. This user won't be interested in throwing

away the old skill set to acquire a new one, and may not be very interested in learning about your workflow or task-oriented solution (even though the new approach will save the user time and effort). Having to learn new ways to use a product that generates little user interest is going to be a stumbling block, no matter how excited you are about it.

One way around the update issue is to implement a forced update and mandatory training. To ensure that no one reinstalls the old version of Office, you can monitor the systems after the updates, using a product such as System Center Operations Manager (SCOM), which you can get at

```
http://www.microsoft.com/systemcenter/opsmgr/default.mspx
```

Even though this approach is extremely painful and some people might actually leave the company over it, at least everyone who's still in-house afterward will be using the same product and templates.

Most companies today don't want to use such draconian measures as enforcing a particular product version. Because the Office XP/2003 user is going to be less than thrilled about updating, you might end up supporting two code bases. The first code base is for the original document and template files (such as `DOT` files). The second code base is for the new Office 2007 document and template files (such as `DOTM`). When you go this route, you'll need to define the common code for both environments so you can use copy-and-paste techniques to move code changes from one environment to another (see the "Defining the common code" section of this chapter for additional details on the topic of common code).

The concept of degrees of parallelism might help you refine the solution to a problem in such a way that it really does benefit your company and help it to provide the best solution for everyone. When thinking about partial parallelism, consider discussing these issues as possible points of parallel development:

- Code base that includes business logic and document manipulation
- Document format
- Wizards and other user aids
- File menu (or Office Menu) commonality
- Toolbar commonality
- Other menus
- Move to workflows and task-oriented solutions

After you explore these issues, you can possibly reduce the number of areas of contention to those that reflect the user interface. If you can also work in forms as part of the solution, you might be able to reduce the problems still further. The point is to keep working at the issues until you have optimized the parallel development. Sometimes, if you keep chipping away at the outstanding issues over time (the process may require months or even years), you'll eventually end up with a completely parallel solution, and then finally end up with a single solution for everyone.

Considering the Office 2007 user

The Office 2007 user is a blank slate. You could train all new Office 2007 users to rely on the same menu-and-toolbar interface that users of earlier versions of Office use, but that's probably a short-sighted and self-defeating approach; it ensures that new users have the same expectations and bad habits that older users have. If you do it that way, you'll never update to Office 2007 in a way that maximizes the productivity gains that this environment can provide. Consequently, even though the new user is a blank slate, you shouldn't use that blank slate to sabotage your own efforts. Treat the blank slate as an opportunity for change, albeit a painful change.

On the one hand, you have the old-version users, and on the other you have the new-version users; except for some remarkable techniques, you can't really make the two groups work together. An alternative solution is to convert the application, but only for new users. That way you can create a workflow solution that makes sense for the new user, but still relies on the old code for business logic and document-content manipulation. Users of the older versions of Office can continue to use the older version of Office that they know how to use. To ensure that both the old and new Office users can use the new file formats, you can install the file-format converters that Microsoft provides for Office 2003. They're available at

```
http://www.microsoft.com/downloads/details.aspx?familyid=941b3470-3ae9-4aee-
              8f43-c6bb74cd1466
```

Figure 13-5 shows how this approach works.

If you decide to use the dual-version method with a single document format, make sure you inform your support staff on how to fix document-compatibility problems. Either the new-version users will have to save their documents using the old file format, or the old-version users will have to use the new file formats. No matter how well you train the users, they're going to make mistakes and save documents using the wrong format from time to time, which means the support staff will need to know to check the file extensions for problems.

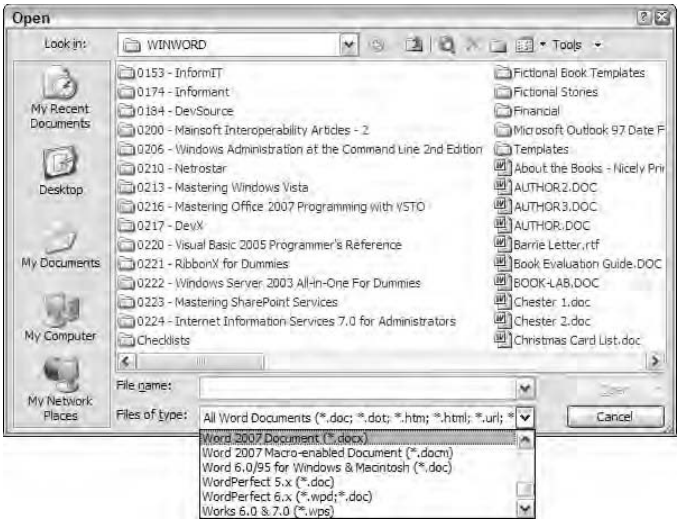


Figure 13-5: Consider using two environments with common file format support.

The Office 2007 user is looking for ease of use, which the workflow and task-based solutions in this book provide. In addition, these new solutions help users perform tasks significantly faster and with fewer errors. One way to promote the new interface is to ensure that everyone hears about these advantages. You can actually document the better performance and use it as a way to help management understand the benefits of moving forward with the rollout of Office 2007.

Defining the common code

When you must develop two code bases, it's important to consider the situations where you might be able to use common code between environments. At least using common code will reduce the amount of dual interface work you must perform to a somewhat manageable level. A parallel solution need not use all of the same code. The parallel portion of the application could include only the business logic and document manipulation code.

When working with common code, you might consider moving code around so that all the common code appears in a single module. Using this approach, you can make changes to the common-code module in one environment, export the module, and then import the resulting module into the second environment. The code is transported between environments without error. Make sure you use care in creating the common code module to ensure that it contains only common code.

The common code should include any global variables or other resources needed to make the code functional. Otherwise you risk introducing errors into the user environment that are hard to detect because they are a global issue. Make sure you keep copious notes in the common code file. Always include a version number and last date of change so you can compare the versions in the two environments when an odd problem pops up. For example, you might think a code change will fix an error in both environments. When you try out the code, however, Office 2007 still has the problem. You can eliminate a failed code transfer as one cause of problems by comparing the code versions.

Never include marginal or incompatible code within the common code module. For example, your old application may require the use of the `CommandBars` object, but you can't include that portion of the code in the common code module because it won't work within Office 2007. You must either convert the `CommandBars` object code into a physical representation or place it in a separate module that exists only within the older application. You'll need to add the interface elements that the `CommandBars` object supported as a new feature to the RibbonX version of the application. For example, you might create a new tab to support the old interface elements in a new way.

Chapter 14

Conversion Techniques for Visual Studio Users

In This Chapter

- Working with existing add-ins
- Understanding the conversion issues
- Obtaining the VBA code
- Defining application-specific conversion requirements
- Developing custom solutions

Most people who are familiar with Visual Studio and VBA see Visual Studio as a way to create add-ins or turn existing VBA code into VB.NET code. This viewpoint probably doesn't match reality completely. Yes, you'll create add-ins with Visual Studio, but they may not be completely new code or they might not be a replacement for existing code. In some cases, you'll use Visual Studio add-ins to merely augment existing code or to provide completely new functionality (such as a connection to a Web server). Although Microsoft would love it if all of those VBA developers out there converted their code to VB.NET, that scenario's unlikely to happen as well. Certainly you'll see some code converted, but the millions of lines of VBA code represent an obstacle too large for most developers to overcome. It's likely that VBA will remain the language of choice for some time to come.

Of course, this doesn't mean you can just skip converting your Visual Studio add-ins to use the Ribbon, or that you won't have to deal with people who use VB.NET as an alternative to VBA. That's one of the interesting things about working with computers — you always have more options to explore. The limiting factor is usually the imagination of the developer, rather than any limitation on the part of the computer. Consequently, this chapter explores a number of solutions and strategies; some may meet your needs and others won't. The point is that you should at least try these strategies to determine whether they'll meet specific needs for your applications and users.

This chapter answers two questions. First, it helps you understand the issues and strategies for converting existing add-ins when necessary. Second, it explores the feasibility and desirability of moving VBA applications to VB.NET.

Using Existing Add-Ins

Visual Studio developers have things both easier and harder than VBA developers do when converting an add-in. The easier part of the equation is that add-ins are more likely to run without much, if any, code conversion because add-ins naturally force the developer to use alternatives to menus and toolbars. Even if you decided to use menus and toolbars, you have considerably more options for performing the required conversion.

The harder part of the equation is that you have to convert the add-in project in most cases. A VBA developer simply opens the same code modules as before. The Visual Studio developer must upgrade to Visual Studio 2005, in most cases, which means updating the project file, modifying references, and changing some of the code so it works with Office 2007. Anyone who's updated a project realizes that some of the upgrade is automatic and some of it is manual. As an alternative, you can choose to re-create the add-in in the new form, which actually works better than converting it in most cases. The following sections explore both parts of the equation.

Existing add-ins from the RibbonX perspective

How well your existing add-in works with Office 2007 depends on what the add-in does. Many add-ins don't include a significant interface because the developer is more interested in manipulating data and presenting it within the document. For example, an add-in that performs calculations based on the selections within an Excel workbook might not have an interface as such. It may actually appear as a single button, which means that the single button would now appear on the Add-Ins tab, rather than in the old location. Some add-ins actually monitor the document for specific conditions and act more or less automatically. (For example, think about a specialized spelling checker for scientific or legal documents that contain a lot of jargon.) Any add-in that has a minimal interface or no interface at all shouldn't require modification to work with Office 2007.

It's important to note that if your add-in works with a context menu or as a SmartTag, you won't have to perform any conversion. Add-ins that rely on forms are also unlikely to experience problems in Office 2007. Even if the

interface is extensive, an add-in that manages to avoid the menus and toolbars of earlier versions of Office is unlikely to require any extra work.

One of the options Visual Studio users have that VBA users don't is the task pane. An add-in can create a task pane to perform its work. Using a task pane can provide a considerable range of interface opportunities for the developer. As shown in Figure 14-1, a task pane can include full input and output, all the controls you might expect, and a means of letting the user work with the task pane exposed in such a way that it doesn't interfere with the current document. Fortunately, task panes don't require any form of conversion.

The only time you'll experience direct issues with Office 2007 and your add-in is when you use the `CommandBars` object to create menus and toolbars. Unlike VBA applications, your Visual Studio add-in project won't let you create menus and toolbars directly; you always create them programmatically. Consequently, this portion of your application will always require conversion. Although you could use form-based menus and toolbars (described in the "Substituting Forms for Menus and Toolbars" section of Chapter 12) as a fix for this problem, Visual Studio developers have a significantly larger number of choices for conversion that are likely to work better.

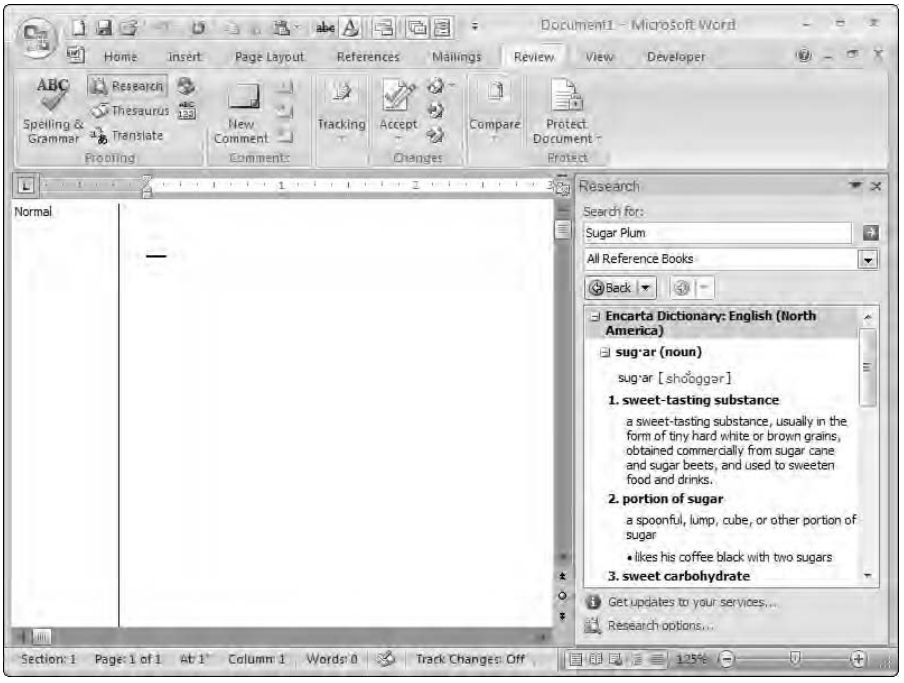


Figure 14-1: Task panes provide a formidable array of features for interacting with the user.

Considering project conversion

Unlike VBA, Visual Studio developers moving from earlier versions of Office to Office 2007 are very likely going to update from Visual Studio 2003 to Visual Studio 2005, or perhaps Orcas when it's released. (Note that Orcas is in beta as of this writing.) You can learn more about Orcas at

```
http://msdn2.microsoft.com/en-us/vstudio/default.aspx
```

or download the latest version at

```
http://www.microsoft.com/downloads/details.aspx?familyid=b533619a-
              0008-4dd6-9ed1-47d482683c78)
```

The point is that with the upgrade of programming tools comes a required upgrade of the add-in project as well.

The first step in creating the project is to bring the project up to date with the new environment. The moment you open an existing add-in project, you'll see the Visual Studio Conversion Wizard. The following steps lead you through the conversion process:

1. **Click Next at the Welcome screen.**

   You'll see the Choose Whether To Create a Backup dialog box (shown in Figure 14-2). Normally you'll want to create a backup of your old project, just in case the conversion fails or you don't have a current backup of the add-in code somewhere else.

Figure 14-2: Decide whether you want to create a backup of the existing project.

2. **Choose a backup option and click Next.**

   You'll see the Ready to Convert dialog box.

3. **Read any notifications. Verify that the Summary text box contains all the information for your project.**

4. **Click Finish.**

   If you requested a backup, Visual Studio performs the backup first. It then performs the conversion of your project. When the conversion is complete, you'll see a Conversion Complete dialog box. Unless something unexpected occurred, the dialog box tells you that all projects converted successfully.

5. **Check the Show the Conversion Log When the Wizard is Closed option. Click Close.**

   Visual Studio opens a conversion report. Check through the conversion report for any problems (it shouldn't list any). More importantly, check through the list for references, and add all these references to your list of items to update for the project.

The conversion report won't tell you everything you need to know. Look in Solution Explorer and you might see one or more references with warnings next to them, as shown in Figure 14-3. You'll need to update or correct every reference that has a warning marker associated with it. At this point, you can begin the compile/debug cycle that all developers know about (but don't necessarily like).

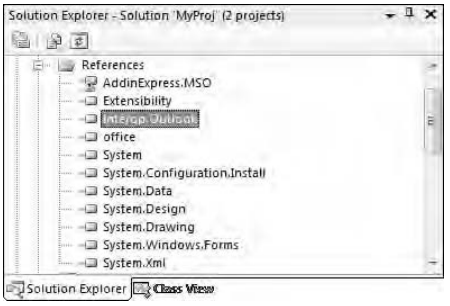


Figure 14-3: Update the assemblies to gain the full benefits of the new Office features.

No matter what you do, some assemblies (such as `Interop.Outlook`, shown in Figure 14-3) will rely on older versions of the .NET Framework. Consequently, you won't obtain access to all of the functionality that the newer assemblies provide. In order to obtain the new features, you must update all your references, which is significantly easier said than done. If your add-in relies on menus or toolbars, then reconstructing the add-in is probably better than working with the older code.

Getting the Visual styles right

Many people complain that their lovely Visual Studio form looks nothing like what they expected when it appears on-screen in Windows XP or Windows Vista. In fact, the form looks very much the way it did in Windows 95, which makes it very obvious that the add-in generated the form and that the form isn't part of the Word product. Some users find the interface change distracting. If you're using forms to make up for problems in the Office 2007 interface, you can't afford the distraction. Fortunately, the fix for this problem is relatively easy: Open the form you created and locate the `Main()` method. Make sure the `Main()` method contains the `Application.EnableVisualStyles()` call to enable the visual styles used by newer versions of Windows, as shown here:

```
Application.EnableVisualStyles();
Application.DoEvents();
Application.Run(new Form1());
```

Visual Studio 2005 automatically inserts the `Application.EnableVisualStyles()` method call for you, but Visual Studio 2003 doesn't, so you'll want to add this code when you convert a project. However, you may notice on some machines that the call still doesn't work as expected. That's when you need to add `Application.DoEvents()` as well. This call ensures that the application processes all Windows messages in the message queue before it goes to the next step, which is to start displaying the form.

You may still run into controls that just don't look right, and it's important that they do look right for an Office add-in. In this case, check the control's `FlatStyle` property. Try setting the `FlatStyle` property to `System` to clear up the display problems. A few controls, such as the `TabControl`, never look quite right, but the differences between the standard control and the newer Windows counterpart is often so small that the user won't notice.

After you test your add-in, you'll find that it works precisely as it did with older versions of Office. That's because you're using the same version of the .NET Framework, the same techniques for accessing Office, and the same methodologies for interacting with the user. In many cases, this result is precisely what you want because it reduces user training time. However, it's not what you want if you eventually plan to move users to Office 2007. To make your add-in Ribbon friendly, you'll need to reconstruct it.

Re-creating a project

In some cases, when your project is simple enough, it's actually easier to re-create the add-in rather than convert it. The re-creation is easier because you spend a lot less time debugging the application. In addition, this is the only

way to obtain easy access to the new RibbonX features provided by Office 2007. Sure, you can attempt to update the references in an older project, but it won't be very easy to make them work because there are so many dependencies that you must also update.

To begin this process, convert your old project (using the information found in the "Considering project conversion" section of this chapter). This step ensures that you can use some older project resources without problems. Close the converted project after you save it (no need to compile it).

Create a new Visual Studio 2005 project using the techniques described in the "Creating a Basic Tab" section of Chapter 5. Add all of your existing forms to the new project by right clicking the project in Solution Explorer and choosing Add Existing Item from the context menu. You'll see the Add Existing Item dialog box shown in Figure 14-4. Highlight the form you want to add and click Add to add it to your project.

Figure 14-4: Save time and effort by using your existing forms whenever possible.

After you've added all the required forms, replace any menus and toolbars with Ribbon equivalents. As previously mentioned, even though you could use form-based menus and toolbars of the same type used by VBA developers, it's really not the best strategy for Visual Studio developers. Create connections between the Ribbon elements and your forms, just as in the original add-in.

Define new `using` or `Imports` statements as needed for your add-in. When that's done, add the required callbacks and use your original add-in code as a basis for handling callbacks. You may need to provide small changes to match the new references you're using and to ensure the code works with the

Ribbon as anticipated. Any business logic, document-content code, or other non-interface code will move with few changes. It's very likely, however, that you'll need to rewrite your user interface code — at least if it works with the `CommandBars` object.

Many developers filled their older add-ins with unsafe code (for example, C# code) and `Platform Invoke` (`PInvoke`) methods. Unfortunately, this code is going to cause problems for you and may not transfer very readily. In addition, because Visual Studio 2005 provides better access to Office 2007 functionality, you may not require the code at all. As part of your rewrite process, make sure you verify and update any older code that relies on dubious methodologies.

At this point, you can compile and test your add-in. The problems you're most likely to encounter include these:

- Graphics features you didn't find and update
- Unsafe code that won't work with Office 2007
- Usage of older assemblies that no longer exist
- Lack of support for new Ribbon features
- Code transferred to the wrong location in the add-in
- Forms that don't work right or look incorrect with Office 2007 (see the "Getting the Visual styles right" sidebar for details)
- Broken external references

Defining a Conversion Strategy

It's important that you look at your current add-in realistically and create a conversion strategy that makes sense. Microsoft hasn't provided a clear update path for add-ins. The assumption is that every developer's going to create a completely new add-in. In some cases, creating an add-in from scratch and simply transferring your business and document change logic to it is going to be the best idea, but this approach is time-consuming and it may not produce the results you need to encourage users to interact with the add-in.

When reviewing your add-in, look for obvious solutions to the problems you're encountering. For example, Chapter 12 describes a number of third-party solutions for moving things around or re-creating a display that looks similar to the one the user is familiar with. These solutions don't always work, but if your add-in has just one or two buttons, it doesn't make sense to completely rewrite it for the Ribbon.

Consider using the form-based toolbar described in Chapter 12. Again, this solution isn't optimal for Visual Studio developers, but it does work. You'll need to create the forms as new Windows forms within your add-in, which can be messy. Make sure you make the forms look as close as possible to the native Office menus and toolbars, using the techniques described in the "Getting the Visual styles right" sidebar in this chapter.

One of the better strategies I've seen is converting an application to use a task pane instead of bothering with menus, toolbars, forms, or other approaches. The task pane is part of the Office environment for both Office 2003 and 2007. You get precisely the same functionality and appearance in both environments. In fact, if you do things right, your users should notice very little difference when working with your add-in. Creating a task pane from scratch is outside the scope of this book. However, you can find a number of good resources for creating task panes online, including those in the following list:

- Creating Custom task panes Using Visual Studio Tools for Office

  `http://msdn2.microsoft.com/en-us/library/aa722570.aspx`

- Paul Ballard's Weblog

  `http://weblogs.asp.net/PaulBallard/`

- Video Tutorial: Creating a VSTO "v3" Custom Task Pane

  `http://blogs.msdn.com/vsto2/archive/2006/03/21/556795.aspx`

- Line-of-Business Data using Visual Studio Tools for Office 2005

  `http://www.microsoft.com/belux/msdn/nl/community/columns/jtielens/lob_vsto2005.mspx`

- Creating Smart Documents with Visual Studio 2005

  `http://www.devx.com/MicrosoftISV/Article/29546`

- Task Pane Resources on Tech Republic

  `http://search.techrepublic.com.com/search/Task+Pane.html`

- MindManager Presenter

  `http://mindjetlabs.com/cs/files/folders/mindjetlabs/entry47.aspx`

These particular resources are exceptionally useful because they show you how to work with task panes in a variety of ways. The technical articles, such as "Creating Custom Task Panes Using Visual Studio Tools for Office," provide

a traditional view of the entire task pane–creation process. (If you want someone to talk with about task panes, check out Paul Ballard's Weblog.)

Some people learn better visually than they do through reading text. In this case, check out the Video Tutorial: Creating a VSTO "v3" Custom task pane site. The article on the Line-of-Business Data using Visual Studio Tools for Office 2005 site tells you how to create a task pane as part of a much larger solution. The article, "Creating Smart Documents with Visual Studio 2005," provides clear and easy steps for working with task panes. There are, in fact, a wealth of task pane resources out there, and you can see a number of them on the Task Pane Resources on Tech Republic site. Finally, the MindManager Presenter is an example of an application that uses two task panes in an interesting way, and you might want to check it out to develop ideas of your own. All these Web sites have something useful to offer.

When you can't use a task pane, another good solution is to make the add-in accessible through the context menu. Using this technique, the user can right-click in any area of the document and use the context menu to access the add-in. This approach actually works better than you might think; users are often looking for a way to interact with data when they need the add-in. The approach that Visual Studio developers use to create an add-in differs enough from those used by VBA developers that a context menu solution actually works pretty well in this case. You can learn more about using the context menu approach in the "Using Existing Office Features" section of Chapter 12.

Some developers have come up with interesting ways to use existing Office features. For example, some developers have asked why they should provide an interface at all if they can do without it. If you're wondering how the user interacts with the application, think about the SmartTag. You can easily create a SmartTag solution with Visual Studio (this option doesn't exist for VBA developers). The add-in simply monitors what the user is doing and presents the appropriate functionality when required. You can learn more about creating a SmartTag with Visual Studio at

```
http://msdn2.microsoft.com/en-us/library/ms178786(VS.80).aspx
```

This is the overview Web site — subfolders contain specific articles on various SmartTag techniques.

A number of third parties make these alternative forms of user interaction easier to create. For example, you'll find a number of useful tools, including one for creating SmartTags, on the Add-in-Express site at

```
http://www.add-in-express.com/
```

Understanding the Visual Studio conversion choice

When you perform a conversion in VBA, you have the option of keeping a lot of the old functionality simply because of the way that VBA interacts with the Office products and because VBA doesn't offer some of the flexibility that you'll find when working with Visual Studio. However, working with Visual Studio is different from working with VBA in important ways. For example, you have the option of sticking with the original add-in when the add-in doesn't break any significant Ribbon rules (as it would if it used menus and toolbars). When you must perform a conversion, the change is significant. You can't simply slide a solution in place and hope it works. As a consequence, you'll find that Visual Studio presents more of an all-or-nothing choice than VBA. You either stick with what you used in the past or you create something completely new.

Many developers are going to view this all-or-nothing approach as a tragedy. However, it's better to view it as an opportunity. Because Visual Studio has so much to offer in the way of flexibility, you really should consider a complete update for your application, or, better yet, create a dual version application that relies on task panes, forms, context menus, SmartTags, and other technologies that go well beyond menus and toolbars. No matter what you think about the Visual Studio environment, it does have a lot to offer, but it would be a mistake to treat it the same way you treat the VBA environment.

Don't forget that you have the full range of Ribbon controls at your disposal when creating a solution for your application. You can use the `Menu` control to re-create essential menus. However, remember that your task pane can include menus too. Using the Ribbon Menu control should be the option of last choice.

Converting VBA Solutions

Most VBA developers aren't very thrilled about moving their applications from VBA to VB.NET. In fact, given the number of lines of VBA code out there, it's unlikely that VBA developers will ever move it all. At some point, the code will become less useful because of changes that Microsoft makes to Office, and some of the code will become unusable — developers will create new solutions. However, the old solutions will remain, and you won't see them disappear for quite some time.

You'll encounter situations where you really do need to convert a VBA solution into a VB.NET alternative. If you really want a compelling reason to make the move based on functionality alone, check out the article at

```
http://www.devx.com/OfficeProDev/Article/28088/2046
```

Even so, most developers will make the move for more pressing reasons, including these:

- VBA doesn't provide a required feature such as task panes.
- Users complain that the VBA application doesn't provide required functionality.
- Your company wants to add new features such as SmartTags.
- The developers are experiencing difficulty in converting the existing VBA code.
- Your company wants to protect the code better than VBA can protect it.

Will this conversion work?

It's the question that many VBA developers are probably going to ask before they begin the conversion process to VB.NET. The answer is that the conversion will work only when VB.NET provides the required functionality. It's important to remember that you can't create documents and templates for Office 2007 products using Visual Studio 2005. When your application requires a document-centric or template-centric approach, VB.NET may not provide a very usable solution. However, the new Visual Studio Orcas (yes, it's still in beta at the time of this writing) will provide document and template capabilities, so you might have to postpone moving your application to VB.NET until the new version of Visual Studio appears. In the meantime, you can still use the VBA conversion techniques described in Chapters 12 and 13 to create a workable solution that will at least allow access to your application.

You also need to consider the question of whether you have someone capable of performing the conversion. A conversion of this type requires someone who speaks both VBA and VB.NET fluently and understands the conversion issues. If you don't have someone who can perform the task at the required level, the conversion will eventually flounder and fail. Make sure you have someone qualified before you begin.

Finally, you need to know whether your company has the will required to complete the conversion. A good part of your code is going to be very easy to convert. If it makes sense to convert that portion of the code first, do so; that way people can at least begin working with the new application and seeing some progress toward the eventual goal of complete conversion. Agile programming techniques work very well with this kind of project.

Converting your application from VBA to VB.NET isn't straightforward. Although some language elements are the same and some are easy to convert, you'll likely run into at least a few areas where you'll need to provide creative solutions. The "Converting Code from VBA to Visual Basic .NET" article at

```
http://msdn2.microsoft.com/en-us/library/aa192490(office.11).aspx
```

provides a lot of generic tips and hints you can use to make the conversion process easier. The following sections describe some considerations for RibbonX conversion.

Performing a VBA walkthrough

In order to create a good VB.NET solution based on your VBA code, you need to understand the VBA code. In many cases, a strict conversion won't produce the results needed; VBA performs tasks differently from VB.NET. You have to understand the logic used within the VBA application to provide a good user experience before you can begin the conversion. In some cases, you may actually have to watch the application work within the debugger in order to understand what the application does today.

It's impossible to overstate the need to create a good baseline, but even more important is knowing where you're going. The application requires an update for some reason. You may simply want to move to VB.NET before Microsoft chooses to end support for it as it did for VB 6.0. Whatever your reason for moving, you must include that reason as part of your walkthrough. The reason for moving provides reference points, so it's clear not only where you want to go, but why.

When you know where you are now and where you want to go, it's important to sit down and discuss your strategy with users, management, the development staff, and any important third parties. The reason for all this preliminary work is that the conversion process is going to be time-consuming anyway. You don't want to compound the problem by starting with flawed assumptions. Make sure you get input from everyone and create a solution that everyone can live with.

Developers often downplay the importance of mockups. However, converting an application from VBA to VB.NET is necessarily going to cause changes in appearance, functionality, viewability, and performance that everyone will notice. It's best to create a mockup to show how the final application will appear (and perhaps work). The goal is to ensure that everyone understands the ways in which your converted application will differ from the original. Sometimes the differences are so subtle that users won't even notice immediately;

but once they do, you can be sure that they'll express their unhappiness with your solution. The most common complaint is going to be that the new version is different from what they're used to using. Spending sufficient time to address user concerns before and during the conversion reduces the amount of rework you need to do later.

Working with menus and toolbars

Whenever possible, convert any code that deals with menus and toolbars to a less cumbersome code in VB.NET. A major problem with the VBA application you want to convert is the habits of the users who work with it. They'll remember that they accessed a certain feature by using the File➪Widget command, and won't realize that the new feature is simply the next step on the Ribbon. They'll complain that it's very hard to select a particular option right now, but then won't like the combo-box control you provide to make the selection easier. The battle you fight when converting a VBA application toVB.NET isn't simply with the code; it's also with the mindset of the user who works with the code.

Something you want to consider before making the conversion is whether the users are actually using all the features of the current application. As part of one update, I actually created a counter for each of the buttons and found that some buttons weren't clicked, not even once, over a month-long test. If a button is so obscure that no one ever uses it, you have to wonder whether you really do need that feature in the new application; why not (for example) place it in something like a dialog box and access it with a dialog-box launcher? Anything you can do to simplify the interface will make the conversion task easier.

When you have to create a Ribbon equivalent of your application, make sure you understand how the users interact with the features of the old application. You'll want to group common controls together. Because you're converting this application to VB.NET, you can also consider adding a login feature and using role-based security to control the features that the user sees. Novice users need not see advanced features designed exclusively for advanced users. Even if a feature receives a lot of use by one group, it doesn't mean that the same feature is equally useful to another group. Consider using roles as a way to customize the interface for each group's use. Again, the goal is to simplify the interface to make it easier to use.

Developing workflows and task-based solutions

Any VBA application you want to convert is going to rely on the menu-and-toolbar mentality, even if it doesn't currently include either feature. One of

the biggest reasons to update to the Ribbon is to gain the benefits of the workflow-and-task-based solution. Of course, you actually have to understand the way the user is presently working with the application before you can create a workflow. Creating a workflow means observing the user's behavior in an unobtrusive way so you can determine the pattern of keypresses and button pushes that the user currently relies on to perform a task.

Unfortunately, simple observation isn't enough. Users often perform tasks in a certain way because the person who taught them used that particular feature. The user might not even know why they use a particularly inefficient set of steps, except that it eventually produces the correct result. Consequently, when you convert your VBA application to VB.NET, you have to learn to ask an important question: "Why?" If you don't know why the users are doing something in a certain way (and they don't know, either), then what you're really doing is building a converted application that perpetuates an archaic and inefficient way of doing things. Someone must know why the users are performing a task in a certain way. If no one does, then you need to ask whether there's a better way to perform the task, and then test the new procedure before you commit it to code.

Considering Application-specific Conversion Requirements

Each Office product has specific needs when it comes to a conversion solution. For example, you'll find that forms work quite well with Word, but you may find that Excel users prefer context menus. As with VBA, you must consider the application as part of your solution. The "Creating a Conversion Solution for Word, Excel, and PowerPoint," "Creating a Conversion Solution for Access," and "Creating a Conversion Solution for Outlook" sections of Chapter 13 describe general issues that you must consider as part of your conversion solution.

However, you'll find that Visual Studio adds some extra constraints to the application-specific conversion scenario. The problem is one of interaction between the `RibbonX` portion of your code and the `ThisAddin` portion of your code. Unlike VBA, Visual Studio requires you to work around class boundaries constantly. That can cause problems if you don't think about the issues in the right way. Most of the examples in this book have looked at the two classes (more for complex applications) as a mere boundary that you must pass in order to provide a complete application. When it comes to a conversion, however, you should consider the `RibbonX` class as the portion of the code devoted to the new interface, and the `ThisAddin` portion as the place for old-interface code and document-manipulation code. In short, you'll probably spend a great deal more time working with the `ThisAddin` class than you might think at first.

If you take extra care with the programming, Visual Studio actually offers you several ways to perform parallel development beyond those offered by VBA. One approach is to create classes for each Office version that you want to support. When the application loads, you can detect the version, and then load the class you require to interact with that specific Office product. This approach might seem to be a very difficult way to accomplish the task, and it is complex, but it does provide the "ultimate" in customization when you have to support several Office versions.

Another technique you can use is selective parallelism. Your add-in can include code for either `CommandBars` or Ribbon implementation as needed. You can even include multiple solutions for the Ribbon. A laptop may display the application options on a custom Ribbon, while a desktop application that has more on-screen real estate may use forms to better mimic an existing application. Obviously, this solution has a tradeoff: It optimizes the user environment in consideration of hardware, but the user sees multiple interfaces, which can make the interface difficult to use.

Creating Custom Conversions When Necessary

In rare cases, you'll find your current application is so different from other applications that it defies easy conversion. For example, a real-time monitor for a warehouse production line that feeds data into Excel and reports progress on building the A1 Multi-widget may not fit within the normal bounds of applications. It may require use of odd COM components, direct access to a serial port, or other strange and exotic code. In this case, you may question whether you really want to change something that's working because changes will undoubtedly introduce interesting problems that will keep you busy for quite some time.

Eventually, you'll have to update that application, and when you make the attempt you'll probably find that the exotic code is nearly impossible to understand, breaks all of the rules, and even relies on undocumented techniques. When this problem occurs, you don't want to try fixing it by using VBA. In fact, you probably don't want to fix it at all. You'll want to create a new application that mimics the functionality of the old application (it probably won't include the same interface — think *workflow* since you have to put so much work into this solution anyway).

One major issue you need to consider is the presence of unmanaged code in the original application. Many exotic applications require use of external

DLLs, which the author could have written in anything from assembly language to C to C++. In fact, you probably don't have code for that module. Any time you create a new managed application, you have to consider the problem of using native language elements as part of it because those native language elements never gain the protections that the managed code possesses. Using the native language module affects your new managed application in the following ways:

- Security
- Memory
- Resource access
- Performance
- Debugging
- Management
- Portability

A native code module affects all these issues negatively. For example, an outsider can use the lax security that many native code modules provide to gain access to your system. Even though the managed VB.NET code is well protected, the native code module isn't, so security is significantly weakened.

Native code modules also have a tendency to develop memory leaks and other memory problems that don't occur with managed code. It's quite possible that your new application will have a memory leak simply because the native code module doesn't release memory as expected.

All these issues are important. When you update your exotic Office application, you need to consider all problems that the application encounters. When you need to rewrite native code modules, you have to include a corresponding increase in development time as part of your estimate. The cost for updating an exotic application can escalate out of control quite quickly; you won't want to undertake such a project lightly.

Part V
The Part of Tens

"They won't let me through security until I remove the bullets from my Word document."

In this part...

This part of the book is all about tens. In Chapter 15 you'll discover ten new tasks you can perform using the Ribbon. You might actually be amazed at just how many ways the Ribbon can improve your life and those of your users. Of course, to get the benefits, you must also perform the work; fortunately, Chapter 15 provides all the necessary details for you.

Everyone likes to have tools and resources that make things easier. Chapter 16 discusses ten tools or resources you can use to make working with the Ribbon significantly easier. Even though the Ribbon has barely appeared in the marketplace, third parties are already working feverishly to make your job easier. Check out these tools and resources when you have a special need for creating better applications.

Chapter 15

Ten New Tasks You Can Perform with RibbonX

In This Chapter

- Devising workflow solutions
- Developing targeted solutions
- Providing flexible alternatives
- Organizing data in new ways
- Letting the user help you integrate applications
- Getting information from Web services
- Developing the hybrid application
- Understanding the user's role
- Transporting code between applications
- Reducing support and training costs

As stated in Chapter 1, the Ribbon isn't a fix for some problems, and will actually cause others. For example, you'll have to update your existing code to use it. However, the Ribbon does provide the means to perform tasks that simply aren't possible using the older menu-and-toolbar methodology. For example, you can't create a workflow using the menu-and-toolbar approach.

The goal of this chapter isn't to convince you that the Ribbon is the best thing that ever happened to developers because (frankly) most developers aren't very thrilled about rewriting their applications. However, this chapter does try to make the Ribbon a little less painful by pointing out the opportunities it can provide. Even if you can't make use of all of these ideas right away, you can at least keep them in mind as you study ways to move your applications to the Ribbon. This chapter summarizes the top ten new tasks that you can perform with the Ribbon, through RibbonX, that a lot of programmers could never imagine doing in the past.

Creating a Workflow Solution

Perhaps the most exciting new task that you can perform with the Ribbon is designing workflows. Most users aren't too interested in learning all about their Office applications. The reason that a user opens Word is to write a letter, not to celebrate the intricacies of table creation. The goal is getting some particular task done; the user isn't particularly concerned with how that happens. A workflow helps the user accomplish a task without placing undue focus on the application itself. In fact, when you view the workflow solutions for Word and Excel in this book, you'll quickly discover that the application appears to dissolve into the background. All the user need do is move from left to right on the Ribbon. It's all about clicking a button or filling in a blank.

The letter-creation example in the "Creating a Letter/Memo Tab" section of Chapter 6 is perhaps the best example in the book of a workflow. In this case, the application doesn't trust the user to do anything except provide the one piece of information that the application can't, which is the body of the letter. Except for typing the body of the letter, the user relies on the Ribbon to perform every other task. Using this approach ensures that the letter follows a standard format, always has the correct information, and doesn't leave anything out. The application works with the user to create a letter. As far as the user is concerned, Word doesn't even exist.

Obviously, you can take the workflow solution much farther than what appears in this book. For example, if you're a law firm and you use WorldDox (`http://www.worldox.com/`), you could ask the application to perform all the required filing based on recipient information and possibly on keywords within the body of the letter. The goal is to automate every possible aspect of the process so that all the user need consider is the content. This level of user assistance is nearly impossible to create using the old menu-and-toolbar approach. Hiding every possible non-application control is impossible with the menus and toolbars — so even if you do succeed in creating a well-designed application, user error will doom your creation to failure.

Of course, you can also take the concept of a workflow too far. When the workflow begins to hinder user activity, rather than work with it, then it becomes a problem. Don't confuse *hindering user activity* with *reduction of tinkering* in this case. Some users want to tinker, and using a workflow tends to interfere with tinkering. The interference is good, in this case, because you really don't want users tinkering with the application. However, when a user needs to address multiple recipients and your letter application doesn't allow for that need, then the workflow becomes a hindrance. Consequently, when designing a workflow, you need to pay particular attention to how users actually perform a task.

The workflow scenario works well with an Agile development strategy. Agile development techniques rely on modular development where you create the application as a series of goals. By using this technique, you can ensure that the application is running faster and that everyone who uses it will have some input. You can read more about Agile development techniques at

```
http://www.cio.com/article/100501/ABC_An_Introduction_to_
          Agile_Programming
```

Targeting Specific User Needs

In some cases, the user need doesn't fall neatly into a workflow. For example, an Access developer might create a temporary table for some other task. The need to create the temporary table for users falls outside the normal workflow. You could view it as a preparatory task — something the developer has to do before performing the actual task of updating the database or testing a new SQL statement.

Most novice users won't have targeted needs. You can use the workflow scenario for most applications designed for novice users. Some intermediate users also rely almost exclusively on workflows because they simply need to get work done quickly. However, as user experience grows, so does the need to provide some targeted solutions. You should consider using a targeted solution in these situations:

- The activity falls outside of a normal workflow.
- The user needs to perform the task intermittently and sometimes not at all.
- You want to hide the solution from inexperienced users.
- The task is file-oriented, rather than task-related.
- The user normally performs the task directly on specific data, rather than as part of achieving a particular goal.

Of course, the question is one of how a targeted solution differs from the menu-and-toolbar solutions of the past. The keyword is targeted. The menu-and-toolbar solution often uses the shotgun approach to solving problems, which means that they aren't very targeted. You can create a temporary table using a menu-and-toolbar setup, but it requires multiple steps, and the results aren't guaranteed. Using the Ribbon, you can create a solution that's quite focused and provides consistent results.

Defining Alternatives for Common Tasks

The dialog-box launcher adds a significant capability to the Ribbon. It lets you create a workflow or targeted solution with added capability for those who need it. For example, you could provide common features for the novice and intermediate user, but add other features for the expert user as part of a dialog-box launcher. The repetitive calculation example in the "Performing Redundant Calculations" section of Chapter 7 provides an excellent view of how this alternative-feature strategy can work.

Of course, you won't want to miss the examples that Microsoft provides either. Go through your favorite application and see all of the ways in which Microsoft uses the dialog-box launcher to provide alternatives for common tasks. For example, the Home tab often uses dialog-box launchers to provide access to advanced formatting features that don't appear as part of the Ribbon. Even though you can set many of the common font characteristics on the Ribbon, you can't set a special effect such as shadow without launching the Font dialog box.

This approach differs from the toolbar and menu approach in that you don't have to see the options all of the time or perhaps at all. When working with an older application, novice users would see options they really didn't need to use on the menus and toolbars. In many cases, these other options simply stirred up confusion and didn't really serve the user in any useful way. In fact, the result was often lost data or an improperly configured application that support staff had to spend time fixing. Using a dialog-box launcher makes features available to those who need them, but completely hides the features from everyone else.

Developing Organizational Aids

Many applications provide organizational aids that aren't very organized — at least not very flexible. Using the Ribbon can help a developer overcome this problem by making the organizational aid part of a targeted or workflow solution. For example, the "Creating a Mail-Management Tab" section of Chapter 9 discusses the requirement of placing e-mail in the correct folder to ensure that it doesn't get lost. It sounds like something too simple to require an actual application, but many users now get so much e-mail that an organizational tool really does help.

Because this aid does appear on a separate tab, the user can choose to ignore it if desired. Unlike toolbar and menu solutions, the Ribbon solution

you create doesn't have to become obnoxious in order to perform the required task. You can choose to make it an aid, a form of assistance, rather than a required part of the user's regimen.

Performing User-Assisted Application Integration

Many of the examples in this book rely on other applications to perform a task. For example, when Word or Excel requires information about the user, the applications in this book use Outlook as a source of data. A user record in Outlook contains all the details about the user that Word or Excel can't provide. Of course, you could just as easily store the information in Active Directory or as part of an Access or SQL Server database. The point is that the information resides outside of Word or Excel. In this case, the user isn't a participant — the interaction occurs automatically when the user selects specific options on the Ribbon.

The part that isn't automatic is that the user must make the selection. A letter need not always include the user's e-mail address, but the user can choose to include the information whenever it's desired. The user doesn't even have to remember that this option is available or locate it in some buried menu — it's part of the application workflow so the user simply has to remember to click a button. Since the user doesn't know or care where the data comes from, there isn't any need to complicate the application with too many options.

However, you can ask the user to provide some level of application-assisted configuration as part of the application. For example, you might choose to include a simple "Update My Information" button that updates the information in Outlook from a source in Active Directory. You'd probably place this kind of option in a dialog box and make it accessible through a dialog-box launcher.

The user-assisted integration can go to any level desired. You could add buttons to obtain information from specific sources (such as a specific database), but a problem occurs if you rely too heavily on the user's memory or powers of observation. A Ribbon application should make the choices simple. Rather than asking the user to choose a database from File Server One, you should provide a clear integration option that reflects the user's view of the data. If the data source is a server in the Denver branch office (for example), the button should probably say "Get Data from Denver" instead of using a particular server name.

Working with Web Services

Many applications today work with Web services. In fact, it's obvious that they work with Web services because the interface contains a wealth of clues about the Web service. For example, when a developer builds an application that interacts with Amazon Web Services, the Web service often takes a prominent place in the menu-and-toolbar system.

Web services shouldn't be obvious, and the Ribbon makes it possible to make the Web service invisible to the user (or at least nearly so). The goal of the Ribbon is to concentrate on the user's work so the process of obtaining and using data is merely a step, one that doesn't rely on location. In the "Creating an Amazon.com Custom Application" section of Chapter 11, the example demonstrates that the Web service need not be seen in order to perform a useful task. Whatever tasks you can perform with a public Web service, you can also perform with a private Web service.

For many companies, Web services are an important strategy for cross-platform applications, so integrating them seamlessly into your application is important. As companies merge and spin off new companies, the number of platforms you need to support increases. It's not just the idea of working with Linux itself that becomes problematic, but also of integrating Macintoshes, mainframes, minicomputers, and all kinds of other platforms into the mix. Using a Web service lets an outside source access the data as if it were coming from that platform. However, unless you can also hide the data source from the user, the application becomes difficult to support. It's important to remember that a new data source isn't very useful unless those who use it can actually access it in a way that feels natural to them.

Working with Hybrid Applications

One of the controls explored in great detail in Chapter 12 is the Menu control. With it, you can re-create all or part of a menu system. Of course, it doesn't possess the same flexibility of the menu system found in earlier versions of Office, but it does let you re-create a generic version of the menu system. This use of the `Menu` control lets you create what is effectively a *hybrid application*. You can obtain most of the benefits of both the Ribbon and the menu-and-toolbar interface in a single application. For some organizations, the hybrid application will spell reduced support and training costs without any loss of functionality.

A hybrid application provides a variety of ways for a user to get work done. The user can rely on a workflow to rough out a particular task, such as writing a letter, creating a worksheet, or even designing a database. Once the rough outline of the document, database, or other data container is in place, the user can rely on task groups to perform specific configuration tasks. Finally, when the user finds that a particular Ribbon-specific approach doesn't provide the required polish, it's possible to rely on a toolbar to obtain the final bit of required functionality.

Whether a hybrid application serves your needs depends on the current skills of the users in your organization. Yes, you can let users perform tasks using a combination of workflows, targeted groups, and menus, but the user has to have the skills required to make this solution useful. You won't want to use this solution with novice users because they'll become confused quite quickly and make mistakes — the hybrid application is specifically for expert users with a lot of experience with the application they're using. In many cases, you'll find that moving the user to a pure workflow solution is a better alternative.

One mistake that developers make is equating a user's intelligence with skill at using a computer. A lawyer or doctor is definitely intelligent but frequently in need of significant handholding when working at the computer. These users are often not well suited to using a hybrid application because the output of the application is more important to them than learning to use the application. However, a scientist working in a lab is a very good candidate for a hybrid application because the scientist is often concerned about getting more out of the application to successfully complete a task. For this group of users, *how the application performs a task* is important; that's your cue to build a hybrid application rather than a more generic application form.

Considering the User Task Criteria

Many developers see their primary task as building an application. It has also become popular to view the application as a solution to a problem. In most cases, however, both views miss the point. The user might not have a problem to solve. After all, it's relatively easy to write a letter with pen and paper (albeit significantly slower). The Ribbon helps the developer accomplish a new task — viewing the task at hand from the user's perspective.

All the latest strategies for designing an application place a high value on communication. In fact, many strategies place the user (or a representative of the

user) directly on the development team. However, the developer often misses the point anyway. Using a workflow approach to building a Ribbon application forces the developer to understand what the user does. It answers a useful question: "What does the user do to accomplish a task?"

Using the Ribbon as a tool to help understand the user's workflow is a new concept for many developers. Suddenly the developer isn't looking at how to build an application, but rather, how to build a *process.* This new task will help developers learn about user needs considerably faster than most of the other programming design strategies today. That's because the developer will see the user in the user's environment perform tasks that may not solve any problem at all, but simply log a new entry in a form or add a name to a list.

Using Code More Than Once

If you ever tried to move code from one Office application to another in the past, you know that any attempt to do so will usually result in some kind of catastrophe. The old `CommandBar` approach to programming was too application-specific. You had to know the structure of the application in order to add new features. The features you added appeared directly in the menus and toolbars that the user relied on to accomplish a task. Consequently, even if you could move the code, you wouldn't find it useful because the structure of the other Office applications was different.

Several of the applications in this book borrow code from other applications. You'll find that some Excel applications borrow the concepts of a sender entry from a Word application. In fact, it's possible to move entire tabs from one Office application to another. Except for some minor differences in the way that the Office application approaches a particular programming problem, you might be able to move the tab intact and make few, if any, changes to your code.

With careful programming, a developer tasked to create a person or place lookup tab for one Office application will be able to move it with very little effort to another Office application. When you think about it, this new task, the ability to create generic tabs that you can move anywhere, is actually a reduction of tasks because you have to create the tab only once. Obviously, you can't use this approach with every programming task, but you'll find that it works often enough to make it worth your while to at least study the feasibility of making the code movable.

Reining In Support and Training Costs

You might not consider reducing support and training costs a new task because every organization on the planet is looking for a way to accomplish this goal. In fact, you might consider support or training as part of your job; some developers are finding themselves in the trenches doing just that. If you want to get out of the support-and-training field and back into programming (or if you want to avoid going to the support-and-training field), consider the Ribbon a good friend.

The examples in this book provide you with a considerable number of ideas for reducing application complexity, while letting the application do more for the user. The workflow and the targeted tasks are both strategies that you should employ to make life easier for the user (and ultimately make life easier for you as well).

It's important to understand that the Ribbon also provides an opportunity to exercise your creative talents. The large buttons with their icons help you communicate information to the user in ways that the older menu-and-toolbar interface would never allow. If you don't have a lot of artistic skill, you can still rely on the built-in graphics that Office provides. Consider using clip-art libraries if necessary — the Ribbon resizes your art to fit within the required space. The whole idea of drawing the user a picture takes on a new meaning with the Ribbon because you can do just that.

You don't have to rely exclusively on the Ribbon to communicate with the user. When necessary, you also have forms and menus that you can add to a Ribbon application. The Ribbon provides a means of flowing information, of communicating through graphics and a good choice of controls, but you still have alternatives when you need them. The Ribbon lets you accomplish considerably more toward creating a great user interface than anything you could do in the past, so it really is a good technology despite the need to rebuild your applications from scratch.

Chapter 16

Ten RibbonX Resources

In This Chapter

- Getting the official news from MSDN
- Using the Microsoft blogs
- Locating other useful news sources
- Getting answers through the Microsoft forums
- Getting answers in other forums
- Obtaining RibbonX tools from PSchmid.net
- Installing the RibbonCustomizer
- Locating blogs with Technorati
- Understanding OpenXML better by visiting OpenXMLDeveloper.org
- Getting an *MSDN* or other print-magazine subscription

It's always helpful to know where you can go for additional information and helpful tools and enhancements when creating your RibbonX applications. This book already contains a number of useful resources in other chapters. For example, you'll discover the Custom UI Editor in the "Developing with the Office 2007 Custom UI Editor" section of Chapter 3. Likewise, the XML Notepad utility appears in the "Creating the XML File" section of Chapter 8. This chapter provides more of the same kind of information. The sections that follow contain ten useful resources you can use to make your RibbonX programming experience better.

Of course, this is *my* list of ten helpful resources; I might have missed your favorite. Because I'm always looking for something better, please be sure to write me about your favorite resource at `JMueller@mwt.net`. I can't guarantee that I'll use the information you provide, but I do guarantee I'll at least check it out. It's amazing to see how many sources people provide me over time — many of which prove indispensable at some point.

Starting with the Microsoft Developer Network

The Microsoft Developer Network (MSDN) has always provided the baseline material for all Microsoft development products. Actually, you'll find a whole warehouse of information there — more than any one human being can probably read in a lifetime. Consequently, you need to sift the information carefully or you'll quickly become lost in the MSDN labyrinth. The main MSDN site for working with the Ribbon is the Office Fluent Ribbon Developer Portal at `http://msdn2.microsoft.com/en-us/office/aa905530.aspx`. The links on this site provide you with news, resources, and access to other information such as samples. You'll also want to check out these other locations on MSDN:

- Office 2007 Technical Articles:

  `http://msdn2.microsoft.com/en-us/library/bb187362.aspx`
- Office 2007 Concepts:

  `http://msdn2.microsoft.com/en-us/library/aa432025.aspx`
- Office 2007 Programs:

  `http://msdn2.microsoft.com/en-us/office/aa905359.aspx`
- Office 2007 Tools and Technologies:

  `http://msdn2.microsoft.com/en-us/office/aa905362.aspx`
- Office 2007 How Do I . . . :

  `http://msdn2.microsoft.com/en-us/library/aa432026.aspx`
- User Experience:

  `http://msdn2.microsoft.com/en-us/architecture/aa699447.aspx`
- Word 2007 Technical Articles:

  `http://msdn2.microsoft.com/en-us/library/bb291002.aspx`
- Excel 2007 Technical Articles:

  `http://msdn2.microsoft.com/en-us/library/bb244233.aspx`
- Access 2007 Technical Articles:

  `http://msdn2.microsoft.com/en-us/library/bb190726.aspx`
- RibbonX for Outlook:

  `http://msdn2.microsoft.com/en-us/library/bb177007.aspx`
- Office User Interface Licensing:

  `https://msdn2.microsoft.com/en-us/aa973809.aspx`

Getting Tips from the Microsoft Blogs

Microsoft wants you to know how to work with RibbonX. In the past, you'd find much of the information you need on the MSDN Web site at `http://msdn2.microsoft.com`. The MSDN Web site is still a very good place to go, but many Microsoft developers complained that it was a bit too formal (the articles are pretty difficult to understand in some cases), and there wasn't any opportunity to interact with the authors. The Microsoft blogs (`http://blogs.msdn.com`) provide a friendlier environment to obtain information where you can actually talk to the author. Here are some of the blogs you'll definitely want to visit when working with RibbonX:

- Jensen Harris: An Office User Interface Blog (`http://blogs.msdn.com/jensenh/`)
- Kathleen's Weblog (`http://blogs.msdn.com/kathleen/`)
- Brian Jones: Open XML Formats (`http://blogs.msdn.com/brian_jones/`)
- Joe Friend: Word (`http://blogs.msdn.com/joe_friend/`)
- David Gainer: Excel (`http://blogs.msdn.com/excel/`)
- Erik Rucker: Access (`http://blogs.msdn.com/access/`)
- Will Kennedy: Outlook (`http://blogs.msdn.com/willkennedy/`)
- The PowerPoint & OfficeArt Team Blog (`http://blogs.msdn.com/powerpoint/`)

This is just the tip of the iceberg. Microsoft has more blogs than you can imagine (or possibly even Bill knows about). If you really want to get the full scoop on what you'll find on the Microsoft blogs, try this Google search: `http://www.google.com/search?q=RibbonX site:blogs.msdn.com`.

Finding Other News Sources for RibbonX

You won't find a tremendous number of news sources just yet for RibbonX, but you can be sure that people are working on them. Make sure you look at the usual sources that you rely on for other information such as DevSource (`http://www.devsource.com`) and DevX (`http://www.devx.com`). A great source for a less technical treatment of RibbonX topics is WindowsITPro (`http://www.windowsitpro.com/`). Unfortunately, you might not find what you need at the major Web sites until the editors obtain more articles from authors.

Fortunately, you can also find a wealth of information from other alternatives such as Feeds4All (`http://www.feeds4all.nl/Item.aspx`) and Softpedia

(http://news.softpedia.com/). In many cases, you'll find good references to stories on additional Web sites that might not show up using other methods. In other cases, the news site will have articles of its own, such as this article on graphics file usage on Softpedia:

```
http://news.softpedia.com/news/Image-Format-in-the-Office-
            2007-RibbonX-41340.shtml
```

Interacting with Others Through the Microsoft Forums

Microsoft has a significant number of forums that you can use to interact with other developers. You can ask questions and normally receive responses quite quickly. Although a majority of the responses you receive are from your peers, you'll also get occasional input from Microsoft Most Valuable Players (MVPs) — non-Microsoft developers who have proven their worth in the past. Sometimes the Office development staff will lend a hand in answering questions as well. The main Microsoft forum URL is

http://forums.microsoft.com/MSDN/default.aspx

Of course, you'll want to drill down into specific forums. Here are a few ideas to try when looking for RibbonX-specific information:

- Visual Studio: http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=16
- VBA: http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=74
- Non-Visual Studio Tools for Office (VSTO) questions: http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=8
- Community Chat (discuss your latest ideas): http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=273
- Site Feedback (when you need to know where to ask a question): http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=14

If you don't see a forum that interests you, or you need to use a language other than English, check out the full range of Microsoft forums at http://forums.microsoft.com/. Microsoft probably has a forum to meet your specific need.

Obtaining Answers from Other Sources

Microsoft doesn't have a lock on assistance with RibbonX problems. You'll find a wealth of developer-related forums online where you can ask questions about RibbonX. In many cases, you're probably already using this source and they simply haven't added a RibbonX section yet. If you have a list server or other forum source that you already use, encourage the owner to add RibbonX support.

One of the places you can look for help on RibbonX topics today is Lockergnome (`http://help.lockergnome.com/`). You'll find a number of questions on Lockergnome already, with a lot of helpful answers. As with many other forums, you'll find that some professionals take time to review the questions on Lockergnome and provide great responses. In some cases, they'll even provide you with some sample source code you can modify to meet your specific needs.

Don't neglect the Office product specific Web sites. For example, you'll find excellent help with Outlook questions on the All about Microsoft Office Outlook site at `http://office-outlook.com/outlook-forum/`. This site (`http://office-outlook.com/`) also provides you with an interesting array of add-ins, tools, and news specifically designed for Outlook.

You can also explore sites that don't have any affiliations with anyone, such as Midtown Computer Systems Enterprise (`http://www.mcse.ms/`). This site provides access to a number of useful usenet groups that can help you with RibbonX questions. For example, you can find a complete list of Office-specific usenet groups at `http://www.mcse.ms/forumdisplay.php?f=34`.

Getting Tools, Examples, and Products from PSchmid.net

You'll find the RibbonX Portal hosted by PSchmid.net at `http://pschmid.net/office2007/ribbonx/index.php`. This is a mandatory place to visit for anyone who is truly serious about working with RibbonX. You'll find a wealth of information and resources, including these:

- Getting Started guide
- RibbonX reference

- Tutorials
- Examples
- Downloads
- RibbonX forum
- RibbonCustomizer

Working with the RibbonCustomizer

The RibbonCustomizer is an interesting utility that you can install with Office products that support add-ins. (Of course, this means you can't use it with products such as Access.) The RibbonCustomizer is a professional tool that helps restore much of the customization capability provided with earlier versions of Office. The direct download for this product is at `http://pschmid.net/office2007/ribboncustomizer/index.php`. The RibbonCustomizer appears on the View tab, as shown in Figure 16-1.

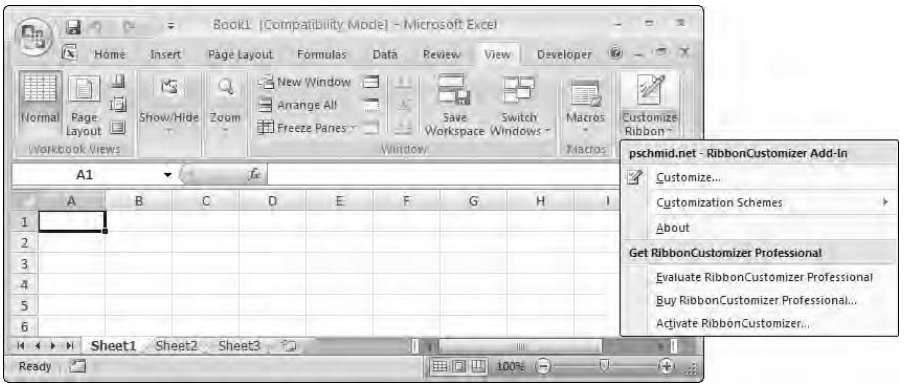


Figure 16-1: The RibbonCustomizer appears on the View tab in the Customize group.

If you want to try out the product, you can download either the 30-day trial version or the free starter edition. Of course, the free starter edition (`http://pschmid.net/office2007/ribboncustomizer/starter.php`) lacks a considerable number of the more interesting product features. For example, you can't reorder the tabs in your application to meet specific needs. Even so, the free starter edition can make your Office experience significantly better.

You can use RibbonCustomizer to change the appearance of the Ribbon in any application that supports it. For example, you can move groups from one

tab to another using the simple interface shown in Figure 16-2. All you need to do to display this dialog box is choose Customize from the list of options on the Customize Ribbon menu shown in Figure 16-1.

Notice that the options in Figure 16-2 also let you add new tabs or remove existing tabs. It's possible to create custom arrangements in workflows as needed. You can also choose to ignore Microsoft features you find cumbersome or move items from the Add-Ins tab to a standard tab. The product also lets you create new groups. This feature is especially important because it lets you move controls from the Add-Ins tab to specific groups.

It's possible that you'll want several Ribbon arrangements to perform different tasks. RibbonCustomizer lets you save multiple arrangements as schemes. Simply choose the scheme you want to use for a particular task. Overall, RibbonCustomizer can help you overcome a significant number of problems in moving your current applications to the Ribbon. Unfortunately, it doesn't solve them all.

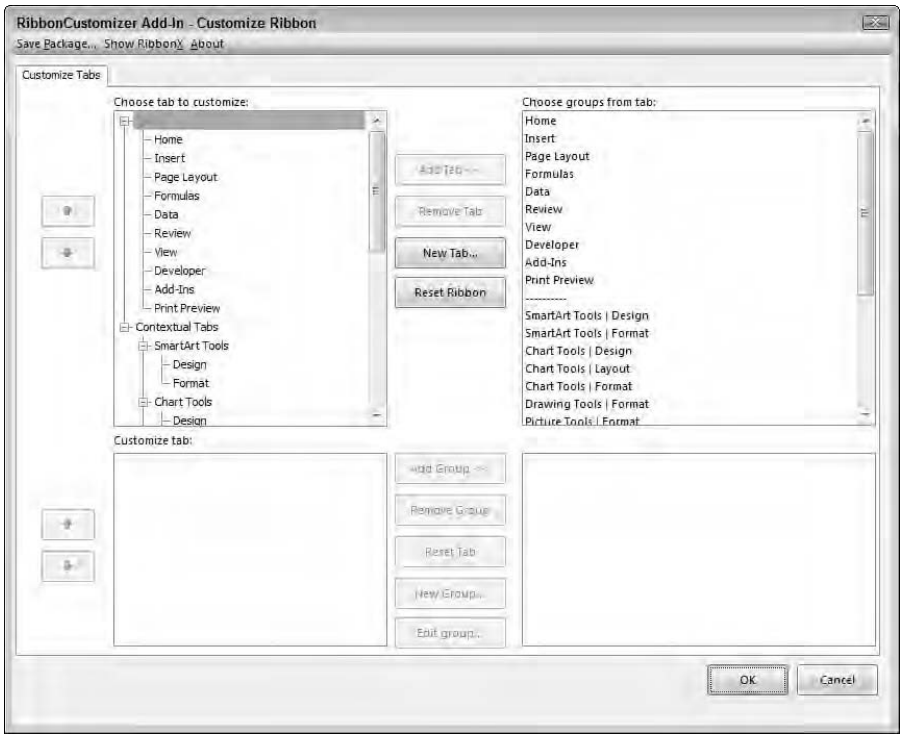


Figure 16-2: Move items around on the Ribbon using this simple interface.

Installing this product means your users can move things around again — adding to the support problems you already have with the Ribbon.

Using Blogs to Your Advantage with Technorati

Most people are familiar with search engines such as Google that can help you locate Web sites with information on a particular topic. A lot of good information on RibbonX doesn't appear on standard Web sites, and you might not find it if you don't look in the right place. One of the better places to look for such information is blogs; one of the better blog search engines is Technorati. You can normally locate a blog on any given topic using this search engine. Use the `http://technorati.com/tag/RibbonX` URL to find blogs on RibbonX topics. You should also try Office-specific tags, such as Excel or Word. Simply type the tag into the Search field on the Technorati Web site.

Using OpenXMLDeveloper.org

Open XML is the basis for the documents you create in Office. Consequently, you'll find more than a little discussion about the topic on one of the main Web sites for Open XML information, OpenXMLDeveloper.org (`http://openxmldeveloper.org/`). In most cases, you'll find the information you want in the various forums that this Web site supports. One of the more important forums for RibbonX developers is Open Packaging Convention (`http://openxmldeveloper.org/forums/12/ShowForum.aspx`), which describes how Office packages the document and template content.

You can find a forum for each of the major markup languages used by Office at

`http://openxmldeveloper.org/forums/default.aspx?GroupID=9`

Here are the specific markup languages covered by this site:

- WordprocessingML: `http://openxmldeveloper.org/forums/13/ShowForum.aspx`
- SpreadsheetML: `http://openxmldeveloper.org/forums/14/ShowForum.aspx`

- PresentationML: `http://openxmldeveloper.org/forums/15/ShowForum.aspx`
- DrawingML: `http://openxmldeveloper.org/forums/31/ShowForum.aspx`

Using MSDN and Other Print Magazines

The *MSDN* magazine has always included a broad range of technical articles on Microsoft product topics that appear well in advance of the actual product. Obviously, Microsoft feeds this magazine with the information even as its developers rush to complete the application. You'll find that *MSDN* contains high-quality articles with a lot of detail. It isn't always readable by mere mortals, but it does provide significant technical depth. If you find that *MSDN* magazine is a little too unapproachable, look at other print magazines on the market today.

The reason print magazines are nice is that you can take them anywhere, review them anytime, and depend on them to provide good graphics. Some people might think that print is dead, but there are a lot of reasons it continues to prosper, not the least of which is convenience and longevity. I'm always wondering where I last saw a particular Web site, but my magazines are always neatly filed away until I need them.

Index

• Numerics •

32-bit images, 70
2007 Office System: XML Schema Reference, 54

• A •

Access (Microsoft) applications
  Add-Ins tab, hiding, 221
  Column (Field) Selection field, filling with data, 230–231
  described, 201–202
  filtered result, 226
  groups, placing in correct order, 221–223
  Home tab, 27
  limitations, 202
  menu bars, 321
  new version, 29
  Ribbon macros, defining, 208
  Ribbon presentation, 226–228
  sample database, obtaining, 218–220
  standard user table, 212–214
  system table, `USysRibbons`, 209–211
  Table Selection field, filling with data, 228–230
  tabs, adding, 207–208
  temporary table, creating, 223–226
  user filtering selections, processing, 231–233
  VBA add-ins, 337
  VBA conversion, 335–337
  Visual Studio add-ins, 337–338
  XML editor, 202–203, 206–207
  XML file, using directly, 214–218
  XML schema, 204–205
ActiveX controls, 150
Add Reference dialog box, 157
Add-in Express Web site, 354
add-ins (DLLs)
  Access unavailability, 202
  context menus, 346–347, 354
  end users, making available for, 103–104
  Outlook applications, 236
  removing, 104–106
  as Ribbon storage solution, 336
  RibbonX and Visual Studio, 92–93
  RibbonX elements, 49
  VBA unavailability, 73
  Visual Studio conversion, 351–352
Add-Ins tab
  drawbacks of using, 11, 12
  hiding, 221
  limitations of, 330–331
Addintools software, 320
Agile development strategy, 367
alpha color, transparencies, 70
alternative use of controls
  described, 39, 75, 107
  sample code, 112
Amazon Web Services (AWS), 287
Amazon.com
  AWS developer tag, 289–290
  AWS Ribbon interface, 293–298
  AWS tasks, 292–293
  browser, queries working in, 290–292
  dialog-box launcher, 298–300
  query, making, 300–306
  services, 288–289
Animations tab, 30
annuity dialog box, 179
application, reason to move to Ribbon, 179
approval requirements, 139–142
Architects Lab, 317
assembly language DLLs, 361
author, contacting, 375
AutoExec macro, 217
AWS (Amazon Web Services), 287

• B •

bitmaps, icons versus, 69
BMP file type, 42
BOM (Byte Order Mark), 217
box control, 57
browser, queries working in, 290–292
business applications. *See* Excel (Microsoft) applications
business logic, 328
businesses, calculations specific to. *See* nonstandard equations tab
button control, 57
<button> element, 26
buttonGroup control, 57
buttons
  Access, creating with temporary table, 223–226
  as communication aid, 373
  content callbacks, 63
  described, 42
  Envelopes, 143
  Excel, images, 192
  graphics, 315–316
  multiple Outlook class issues, 250
  PowerPoint, 265–266
  Sort Criterion group, 296
  XML code for image, 44
Byte Order Mark (BOM), 217

• C •

C#
  add-ins, 336
  DLLs, 361
  RibbonX, 98–99
  unsafe code, 352
CalcType variable, 167
calculations
  date, 159–160
  nonstandard equations tab, performing, 174–175
calculations, redundant
  code, creating, 179–180
  data identification requirements, 186–187
  described, 176–177
  dialog boxes, designing, 178–179
  global and local variables, 185–186
  linkages to existing data, 180–182
  loop, 183–186
  problem solution, defining, 177–178
  procedure, choosing, 182–183
calendar method, 160
CartAdd interface, 293–294
CC requirements, 139–142
cell range, worksheet linkage, 194–195
changes in control data, detecting, 172
checkBox control
  callback control, 62
  described, 57
ChildNodes collection, 306
class boundaries, Visual Studio conversions, 359
classes, XML and Outlook, 247–248
Classic Menu for Office 2007, 320
code, 102–103
Column (Field) Selection field, filling with data, 230–231
ColumnGetText() method, 231
COM (Component Object Model), 74
COM+ application, Web services, 307
comboBox control
  content callbacks, 62, 63, 64
  described, 57
command line interface, Office Migration Planning Manager, 18
CommandBar
  common code module, 343
  conversion issues, 328
CommandBars object
  context menus, 36
  Visual Studio, 347
Community Chat forum, 378
Component Object Model (COM), 74
context menus
  add-ins, 346–347, 354
  described, 36
  Excel, relying on, 322–323
  Excel entry, defining, 322–323
Contextual Tabsets
  described, 33, 37–38
  feature hiding, 43

Control IDs, Outlook, 238
controls. *See also* groups
  attributes, 59–61
  described, 12
  graphics tools, 68–69
  hiding unneeded, 87
  IDs, checking listing of, 84
controls, RibbonX
  choosing, 41–42
  elements, defining, 55
  overview, 56–58
  scripts (VBA), 77–78
  Visual Studio, 110
conversion
  parsing numeric input, 174
  rate values, 175
conversion, VBA
  Access conversion, 335–337
  common code, 342–343
  conversion, issues behind, 328–329
  conversion strategy, 329–330
  described, 327
  existing menus and toolbars, 330–331
  forms, 330
  Office 2007 user, 341–342
  Office XP/2003 user, 339–341
  Outlook conversion, 337–339
  Quick Access Toolbar (QAT), 331–333
  RibbonX changes, 333–334
  toolbars and menus, designing rather than creating, 331
  Word, Excel, and PowerPoint conversion, 334–335
conversion, Visual Studio
  considering, 348–350
  customizing, 360–361
  described, 345–346
  existing add-ins from RibbonX, 346–347
  feasibility, 356
  form-based toolbars, 353
  functions, preserving old, 355
  menus and toolbars, 358
  re-creating projects, 350–352
  reviewing add-ins, 352
  for specific applications, 359–360
  styles, 350
  task pane resources, 353–354
  VBA walkthrough, 357–358
  VB.NET alternative, 356–357
  workflows and task-based solutions, 358–359
copy, e-mail, 245–246
cost calculations, 198–199
Create tab, 29
cross-platform applications, Web services, 370
current state, Toggle Button, 82
`CurrentDb.CreateQueryDef()` method, 233
custom namespace, 222
custom presentation tab, 263–267
Custom Task Panes (CTPs), 74
Custom UI Editor
  Access unavailability, 202
  callback subs, automatically generating, 76–77
  content callbacks, 64
  graphics, 75–76
  RibbonX elements, 64–66

• D •

data, external access
  constant, supporting, 278–280
  control, detecting changes in, 172
  entered, obtaining, 172–174
  identification requirements, 186–187
  redundant calculations, linking to existing, 180–182
data entry forms
  content, 192–194
  controls, 188
  cost, calculating, 198–199
  creating, 187
  date, validating, 159–160
  Design Mode, fields, 191
  employee selections, defining, 195–198
  flexible, providing, 170
  grid lines, 190–191
  protection options, 189
  Ribbon code, adding, 192
  from template, 189–190
  worksheet linkage, 194–195
data management, Outlook, 236

database
  exporting table to new, 212
  sample, obtaining, 218–220
  saving file in new format, 218
  SQL Server, creating, 20–21
dates
  Letter/Memo tab, 133–136
  validating, 159–160
DDBs (Device-Dependent Bitmaps), 70
description attribute, 59
design, RibbonX
  Contextual Tabsets, 37–38
  controls, choosing, 41–42
  described, 31–32, 39
  elements goals, 32–34
  existing Office features, 36
  features, hiding, 43
  MiniToolbar, 38–39
  names, effective use of, 40
  number of items on tab, 40–41
  Office Menu, 36–37
  tooltips, 34–36
  user hints, 43
  user's perspective on groups, 41
Design Mode toggle button, 191
Design tab, 30
desktop space, Ribbon's use of, 15
Developer tab, 266
developers, Ribbon, 11
Device-Dependent Bitmaps (DDBs), 70
Device-Independent Bitmaps (DIBs), 70
dialog boxes
  need, assessing, 33
  redundant calculations, 178–179
dialog-box launcher
  alternatives for common tasks, 368
  Amazon.com, 298–300
  described, 33
  implementing, 178
  tooltip, 36
  visual element, adding to group, 55
display, Word forms, 152–154
DLLs
  Access unavailability, 202
  context menus, 346–347, 354
  end users, making available for, 103–104
  Outlook applications, 236
  removing, 104–106
  as Ribbon storage solution, 336
  RibbonX and Visual Studio, 92–93
  RibbonX elements, 49
  VBA unavailability, 73
  Visual Studio conversion, 351–352
.DOC extension, 17
.DOCM file extension, 17
document
  caution against modifying while open, 74
  compatibility, 23
  document in older Word version, 85
  dynamic content, 93–94
  features, 124
  protecting, 149
  RibbonX and Visual Studio, 93
  type, changing, 126–127
  variable, 126–127
  Word forms, creating, 154–156
Document Object Model (DOM) document object, 217
document protection features, 28
.DOCX file extension, 17
DOT file
  choosing to use only, 313
  conversion issues, 328
DOTX file
  conversion issues, 328
  when not to use, 313
draft, saving e-mail as, 246–247
Drawing ML Web site, 383
dropDown control
  content callbacks, 62–63
  described, 57
dynamic document content, 93–94
Dynamic Help window, 204
dynamicMenu control
  content callback, 61
  described, 57

• E •

eBay Web service
  application, 287
  Software Development Kit, 285
editBox control
  content callbacks, 63–64
  described, 58
EFFECT equation, 169–170

elements, RibbonX
  bitmaps versus icons, 69
  control attributes, 59–61
  control callbacks, 61–64
  control graphics tools, 68–69
  controls, choosing, 26
  controls, coding, 28
  controls, defining, 27, 55
  controls overview, 56–58
  described, 23–24, 49
  groups, 26, 52–55
  objects versus, 59
  Office icons, obtaining list of, 67–68
  Office 2007 Custom UI Editor, 64–66
  tab, creating, 50–52
  tabs, 24–25
  32-bit images, 70
  transparency, 69–70
e-mail handling
  adding to task, 258–259
  caller's class, detecting, 239–241
  closing task, 259–260
  copy, creating, 245–246
  default project, 237–238
  defining task, 253–258
  draft, saving as, 246–247
  filing interface, designing, 241–243
  folder list, obtaining, 243–245
  multiple class issues, 247–251
  reading versus responding, 237
  supplemental information, 258–259
  task-creation interface, 251–253
  text, turning into task entry, 251–253
Employee Name field, expense report, 192–194
employee selections, data entry forms, 195–198
`enabled` attribute, 59
ending, PowerPoint, 280–283
envelopes
  button size, 143
  custom output, 143–145
  groups, 142–143
equation, in use, 167
equations, nonstandard
  calculation, performing, 174–175
  described, 165–166
  entered data, obtaining, 172–174
  equation type, choosing, 167–168
  multiple Ribbon elements, 168–172
  starting element, 166
error messages, XML Notepad, 204–205
Excel (Microsoft) applications
  button to access form, 88–89
  combining VBA and Visual Studio, 165
  context menus, relying on, 322–323
  described, 163
  Design Mode toggle button, 191
  equations, choosing, 167–168
  Home tab, 27
  new version, 28–29
  protection, 189
  sample RibbonX tab, 50–52
  task panes, 323
  template, saving document as, 189–190
  variety of, 164
  VBA conversion, 164, 334–335
  Visual Studio, 164
Excel (Microsoft) data entry forms
  content, 192–194
  controls, 188
  cost, calculating, 198–199
  creating, 187
  date, validating, 159–160
  Design Mode, fields, 191
  employee selections, defining, 195–198
  flexible, providing, 170
  grid lines, 190–191
  protection options, 189
  Ribbon code, adding, 192
  from template, 189–190
  worksheet linkage, 194–195
Excel (Microsoft) nonstandard equations tab
  calculation, performing, 174–175
  described, 165–166
  entered data, obtaining, 172–174
  equation type, choosing, 167–168
  multiple Ribbon elements, 168–172
  starting element, 166
Excel (Microsoft) redundant calculations
  code, creating, 179–180
  data identification requirements, 186–187
  described, 176–177
  dialog boxes, designing, 178–179
  global and local variables, 185–186

Excel (Microsoft) redundant calculations *(continued)*
linkages to existing data, 180–182
loop, 183–186
problem solution, defining, 177–178
procedure, choosing, 182–183
expense report
cost calculations, 198–199
disappearing and reappearing fields, 194
Employee Name field, 192–194
employee selections, defining, 195–198
exporting table to new database, 212
eXtensible Markup Language (XML)
button image code, 44
file, using directly in Access, 214–218
group, creating (`<group>` element), 25
multiple Outlook class issues, 247–248
new Ribbon element, creating, 25
problem solution, defining, 177
schema in Access applications, 204–205
SQL Server version, 19
table, loading as part of, 208
eXtensible Markup Language (XML) Notepad 2007
correct entries, ensuring, 205
downloading, 203
error messages, 204–205
schema, removing, 205
External Data tab, 29

• F •

`FetchAmazonData()` method, 301
File Email code, 245–246
file-format converters, 341–342
files, 101–102
filing interface, Outlook, 241–243
folder list, e-mail, 243–245
`For Each` loop, 306
foreign language forums, 378
formatting, effective rate calculation, 176
form-based toolbars, 353
forms
conversion issues, 328
Excel button to access, 88–89
protecting numbers, 199
substituting for menus and toolbars, 313–316, 318
VBA conversion, 328, 330
Visual Studio conversion, 351
forms, data entry
content, 192–194
controls, 188
cost, calculating, 198–199
creating, 187
date, validating, 159–160
Design Mode, fields, 191
employee selections, defining, 195–198
flexible, providing, 170
grid lines, 190–191
protection options, 189
Ribbon code, adding, 192
from template, 189–190
worksheet linkage, 194–195
forms, Word
Content Controls versus Legacy Controls, 150
creating, 147–149
selecting, 150
Formulas tab
described, 28–29
goals, 32
functions
underlining normal versus alternative, 113–114
Visual Studio conversion, preserving old, 355
FV equation, 169

• G •

gallery
Business forms, 147
content, defining, 151
content callbacks, 62–63
form, choosing, 151
items, providing, 152–153
`gallery` control, 58
General number format, 176
`getContent` attribute, 61
`getDescription` attribute, 61
`getEnabled` attribute, 61

getImage attribute, 61
getImageMso attribute, 62
getItemCount attribute, 62
getItemCount() method, 154
getItemHeight attribute, 62
getItemID attribute, 62
getItemID() method, 154
getItemImage attribute, 62
getItemLabel attribute, 62
getItemLabel() method, 154
getItemScreenTip attribute, 62
getItemSuperTip attribute, 62
getItemWidth attribute, 62
getKeytip attribute, 62
getLabel attribute, 62
getPressed attribute, 62
getPressed callback, 80
getScreentip attribute, 63
getSelectedItemIndex attribute, 63
getSelectItemID attribute, 63
getSelLbl() method, 296–297
getShowImage attribute, 63
getShowLabel attribute, 63
getSize attribute, 63
getSupertip attribute, 63
getTemplatePath() method, 152
getText attribute, 63, 181–182
getTitle attribute, 63
getVisible attribute, 63
GIF file type, 42
global and local variables, redundant calculations, 185–186
Google Maps, 286
graphics
  Custom UI Editor, 75–76
  RibbonX and Visual Studio, 107–109
  toolbar buttons, 315–316
  VBA, 74
  Word, 123
graphics cards, 70
greeting, adding to letter, 138–139
grid lines, removing from forms, 190–191
groups
  automating envelopes, 142–143
  described, 12
  envelopes, 142–143
  line in middle, 54
  multiple Outlook class issues, 250
  organizing by usage, 33
  as part of tabs, 24
  placing in correct order, 221–223
  user's perspective on, 41
  XML, creating (<group> element), 25
groups, RibbonX
  elements, 52–55
  scripts (VBA), 77–78
  Visual Studio, 110

• H •

handling e-mail
  adding to task, 258–259
  caller's class, detecting, 239–241
  closing task, 259–260
  copy, creating, 245–246
  default project, 237–238
  defining task, 253–258
  draft, saving as, 246–247
  filing interface, designing, 241–243
  folder list, obtaining, 243–245
  multiple class issues, 247–251
  reading versus responding, 237
  supplemental information, 258–259
  task-creation interface, 251–253
  text, turning into task entry, 251–253
heading, worksheet, 186
help, keytip
  appearance, 51
  attribute (keyTip), 60
  group, 53
hiding
  expense report fields, 194
  features, 15–16
  Ribbon, 15
  RibbonX features, 43
  unneeded controls, 87
  Web services, 286
hierarchy, XML elements, 24
HLP (help) files, rewriting, 219–220
Home tab
  described, 27
  group and control, adding, 111

Home tab *(continued)*
Styles gallery, 325
toggle buttons, 207–208

• I •

icons
Office, obtaining list of, 67–68
toolbar buttons, 315–316
ID
gallery, 154
obtaining for existing tab, group, or control, 78–79
Outlook control, 238
`id` attribute, 59
`idMso` attribute, 59
`idQ` attribute, 60, 222–223
`image` attribute, 60
`imageMso` attribute, 60
images
button, providing, 26
button file types, 42
Custom UI Editor, 75–76
Excel buttons, 192
group, 53
linking to button, 26
modifying at runtime, 125
RibbonX and Visual Studio, 107–109
32-bit, 70
toolbar buttons, 315–316
VBA, 74
Word, 123
industry-specific calculations. *See* nonstandard equations tab
inexperienced users
alternatives, 368
targeted solutions, 367
initial slide
creating, 268–271
importance of, 267–268
injection, VBA, 74
`InputBox`, developer tag, 299
Insert tab, 27
`insertAfterMso` attribute, 60
`insertAfterQ` attribute, 60, 222–223
`insertBeforeMso` attribute, 60
`insertBeforeQ` attribute, 60
`Int32.Parse()` method, 174
intelligence, computer skills versus, 371
interface, effective, 31
intermediate users, 367
Internet connection, Web services, 286
`InvalidateControl` method, 128
`ItemClicked()` method, 154
`itemSize` attribute, 60

• J •

JPG file type, 42

• K •

keytip
appearance, 51
attribute (`keyTip`), 60
group, 53

• L •

`label` attribute, 60
`labelControl` control, 58
labels, modifying at runtime, 125
Las Vegas effect, 43
Letter/Memo tab
CC, routing, and approval requirements, 139–142
custom envelope output, 143–145
dates, 133–136
described, 123–124
document type, changing, 126–127
greeting recipient and adding signature, 138–139
labels and images, modifying, 125–126
recipient, adding, 129–133
restoring document type after Ribbon loads, 128
sender, adding, 136–138
style, setting, 124–125
workflow, 366
`loadImage` attribute, 64
loan dialog box, 179

Lockergnome Web site, 379
lookup tabs, sharing code between applications, 372
loop, redundant calculation, 183–186

• M •

macros
  file extensions, 17
  Ribbon, defining for Access applications, 208
  saving, 72–73
magazines, print, 383
mail-management tab, Outlook
  caller's class, detecting, 239–241
  copy, creating, 245–246
  default project, 237–238
  draft, saving as, 246–247
  filing interface, designing, 241–243
  folder list, obtaining, 243–245
  reading versus responding, 237
managed code, advantages of, 94–95
markup languages, online resources for, 382–383
MDB file extension, 218
memos (Letter/Memo tab)
  CC, routing, and approval requirements, 139–142
  custom envelope output, 143–145
  dates, 133–136
  described, 123–124
  document type, changing, 126–127
  greeting recipient and adding signature, 138–139
  labels and images, modifying, 125–126
  recipient, adding, 129–133
  restoring document type after Ribbon loads, 128
  sender, adding, 136–138
  style, setting, 124–125
  workflow, 366
`Menu` control
  described, 58
  forms, 318–319
  hybrid applications, 370–371
  when to use, 320
menu conversion issues, 328
menus
  forms, substituting, 313–316, 318
  inadequacy of, 13–14
  targeted solutions versus, 367
  third-party tools, mimicking with, 317
  VBA conversion, 330–331
  Visual Studio conversion, 358
`menuSeparator` control, 58
Microsoft
  decision to change interface, 13
  magazines, 383
  online forums, 377–378
Microsoft Access applications
  Add-Ins tab, hiding, 221
  Column (Field) Selection field, filling with data, 230–231
  described, 201–202
  filtered result, 226
  groups, placing in correct order, 221–223
  Home tab, 27
  limitations, 202
  menu bars, 321
  new version, 29
  Ribbon macros, defining, 208
  Ribbon presentation, 226–228
  sample database, obtaining, 218–220
  standard user table, 212–214
  system table, `USysRibbons`, 209–211
  Table Selection field, filling with data, 228–230
  tabs, adding, 207–208
  temporary table, creating, 223–226
  user filtering selections, processing, 231–233
  VBA add-ins, 337
  VBA conversion, 335–337
  Visual Studio add-ins, 337–338
  XML editor, 202–203, 206–207
  XML file, using directly, 214–218
  XML schema, 204–205
Microsoft Developer Network (MSDN)
  MSDN and other print magazines, 383
  Web site, 377
Microsoft Excel applications
  button to access form, 88–89
  combining VBA and Visual Studio, 165
  context menus, relying on, 322–323

Microsoft Excel applications *(continued)*
described, 163
Design Mode toggle button, 191
equations, choosing, 167–168
Home tab, 27
new version, 28–29
protection, 189
sample RibbonX tab, 50–52
task panes, 323
template, saving document as, 189–190
variety of, 164
VBA conversion, 164, 334–335
Visual Studio, 164
Microsoft Excel data entry forms
content, 192–194
controls, 188
cost, calculating, 198–199
creating, 187
date, validating, 159–160
Design Mode, fields, 191
employee selections, defining, 195–198
flexible, providing, 170
grid lines, 190–191
protection options, 189
Ribbon code, adding, 192
from template, 189–190
worksheet linkage, 194–195
Microsoft Excel nonstandard equations tab
calculation, performing, 174–175
described, 165–166
entered data, obtaining, 172–174
equation type, choosing, 167–168
multiple Ribbon elements, 168–172
starting element, 166
Microsoft Excel redundant calculations
code, creating, 179–180
data identification requirements, 186–187
described, 176–177
dialog boxes, designing, 178–179
global and local variables, 185–186
linkages to existing data, 180–182
loop, 183–186
problem solution, defining, 177–178
procedure, choosing, 182–183
Microsoft Office
customization, rate of, 72
existing version, hanging onto, 12–13
icons, obtaining list of, 67–68
interface changes, 323
new features, using, 320
Quick Access Toolbar, 324
reusing code, 372
Ribbon support, 13–16
RibbonCustomizer utility, 380–381
tab, creating, 25
VBA applications, converting to VB.NET, 355–356
Web resources, 379
workflow solutions, designing, 366–367
Microsoft Office Compatibility Pack, 16–17
Microsoft Office e-mail handling
adding to task, 258–259
caller's class, detecting, 239–241
closing task, 259–260
copy, creating, 245–246
default project, 237–238
defining task, 253–258
draft, saving as, 246–247
filing interface, designing, 241–243
folder list, obtaining, 243–245
multiple class issues, 247–251
reading versus responding, 237
supplemental information, 258–259
task-creation interface, 251–253
text, turning into task entry, 251–253
Microsoft Office Fluent Ribbon Developer Portal, 376
Microsoft Office mail-management tab
caller's class, detecting, 239–241
copy, creating, 245–246
default project, 237–238
draft, saving as, 246–247
filing interface, designing, 241–243
folder list, obtaining, 243–245
reading versus responding, 237
Microsoft Office Migration Planning Manager
command line interface, 18
described, 17
scanning tools, 18–19
Microsoft Office 2003 updates, 16, 339–341
Microsoft Office 2007
Access, 29
common Ribbon elements, 27–28
document, saving, 85
Excel, 28–29

Outlook, 29–30
PowerPoint, 30
VBA conversion, 341–342
Word, 28
Microsoft Office 2007 Custom UI Editor
Access unavailability, 202
callback subs, automatically generating, 76–77
content callbacks, 64
graphics, 75–76
RibbonX elements, 64–66
Microsoft Office XP
updates, 16
VBA conversion, 339–341
Microsoft Outlook applications. *See also* e-mail handling; mail-management tab, Outlook
accessing add-ins, 236
communications problems, avoiding, 239
data management, 236
messages, viewing at certain time interval, 131
new version, 29–30
options, user selecting, 369
user data, obtaining, 274–275
VBA, lack of support for, 235
VBA conversion, 337–339
Visual Studio add-ins, 236
Microsoft PowerPoint applications
built-in property values, 272–273
constant data, supporting, 278–280
creating initial slide, 268–271
custom presentation tab, 263–267
custom properties, saving, 271–272
custom property values, 273–274
described, 261–262
ending, providing presentation, 280–283
Home tab, 27
initial slide, importance of, 267–268
new version, 30
optional slides, 276–278
template, saving and using, 283–284
template, usefulness of, 262–263
user data, obtaining from Outlook, 274–275
VBA conversion, 334–335
Microsoft Windows 95, 350
Microsoft Windows Vista
help (HLP) files, rewriting, 219–220
Visual Studio, installing, 96–97
Visual Studio appearance, 350
Microsoft Windows XP, 350
Microsoft Word applications. *See also* Letter/Memo tab
described, 121–122
envelopes, 142–145
Home tab, 27
labels, 145
new version, 28
Quick Style Set, modifying and storing in template, 325
Styles group user hint, 43
Table Tools Contextual Tabset, 37–38
VBA, 122–123
VBA conversion, 334–335
Visual Studio, 123
Microsoft Word forms
Content Controls versus Legacy Controls, 150
creating, 147–149
date, including, 159–161
flow of information, 146–147
new document, creating, 154–156
physical presentation, 151
selecting, 150
tab, 146
template information for display, 152–154
Templates variable and interaction, 151–152
user information, adding, 156–159
Microsoft Word Letter/Memo tab
CC, routing, and approval requirements, 139–142
custom envelope output, 143–145
dates, 133–136
described, 123–124
document type, changing, 126–127
greeting recipient and adding signature, 138–139
labels and images, modifying, 125–126
recipient, adding, 129–133
restoring document type after Ribbon loads, 128
sender, adding, 136–138

Microsoft Word Letter/Memo tab *(continued)*
  style, setting, 124–125
  workflow, 366
Midtown Computer Systems Enterprise Web site, 379
MiniToolbar, 38–39
mockups, importance of, 357
Most Valuable Players (MVPs), Microsoft, 378
MSDN (Microsoft Developer Network)
  MSDN and other print magazines, 383
  Web site, 377

• N •

names
  controls, 148
  RibbonX, effective use of, 40
namespaces
  defining, 25
  requirement for, 23
native code modules, 361
.NET framework, assemblies, 349
.NET Programmability Support option, installing, 95–98
news sources, 377–378
non-English forums, 378
nonstandard equations tab
  calculation, performing, 174–175
  described, 165–166
  entered data, obtaining, 172–174
  equation type, choosing, 167–168
  multiple Ribbon elements, 168–172
  starting element, 166
Non-Visual Studio Tools for Office (VSTO), 378
novice users
  alternatives, 368
  targeted solutions, 367
numbers
  converting (`Int32.Parse()` method), 174
  forms, 199
  VBA formats, 176

• O •

objects
  Component Object Model (COM), 74
  Document Object Model (DOM) document object, 217
  RibbonX elements versus, 59
Office (Microsoft)
  customization, rate of, 72
  existing version, hanging onto, 12–13
  icons, obtaining list of, 67–68
  interface changes, 323
  new features, using, 320
  Quick Access Toolbar, 324
  reusing code, 372
  Ribbon support, 13–16
  RibbonCustomizer utility, 380–381
  tab, creating, 25
  VBA applications, converting to VB.NET, 355–356
  Web resources, 379
  workflow solutions, designing, 366–367
Office (Microsoft) e-mail handling
  adding to task, 258–259
  caller's class, detecting, 239–241
  closing task, 259–260
  copy, creating, 245–246
  default project, 237–238
  defining task, 253–258
  draft, saving as, 246–247
  filing interface, designing, 241–243
  folder list, obtaining, 243–245
  multiple class issues, 247–251
  reading versus responding, 237
  supplemental information, 258–259
  task-creation interface, 251–253
  text, turning into task entry, 251–253
Office (Microsoft) mail-management tab
  caller's class, detecting, 239–241
  copy, creating, 245–246
  default project, 237–238
  draft, saving as, 246–247
  filing interface, designing, 241–243
  folder list, obtaining, 243–245
  reading versus responding, 237

Office Compatibility Pack (Microsoft), 16–17
Office Fluent Ribbon Developer Portal (Microsoft), 376
Office menu
  adding items to, 83–84
  controls, repurposing, 85–86
  modifying with Visual Studio, 114–116
  tasks, caution against adding to, 37
Office Migration Planning Manager (Microsoft)
  command line interface, 18
  described, 17
  scanning tools, 18–19
Office 2003 (Microsoft) updates, 16, 339–341
Office 2007 (Microsoft)
  Access, 29
  common Ribbon elements, 27–28
  document, saving, 85
  Excel, 28–29
  Outlook, 29–30
  PowerPoint, 30
  VBA conversion, 341–342
  Word, 28
Office 2007 (Microsoft) Custom UI Editor
  Access unavailability, 202
  callback subs, automatically generating, 76–77
  content callbacks, 64
  graphics, 75–76
  RibbonX elements, 64–66
Office XP (Microsoft)
  updates, 16
  VBA conversion, 339–341
`onAction`
  attribute, 26, 64
  callback, 80
  element, 26–27
`onChange` attribute, 64
`onLoad` attribute, 64
Open Packaging Convention, 382
OpenXMLDeveloper.org, 382–383
operation, 292
Orcas, 348
order, button, 42
organizational aids, Ribbon X, 368–369
OSQL utility, 21–22
Outlook (Microsoft) applications. *See also* e-mail handling; mail-management tab, Outlook
  accessing add-ins, 236
  communications problems, avoiding, 239
  data management, 236
  messages, viewing at certain time interval, 131
  new version, 29–30
  options, user selecting, 369
  user data, obtaining, 274–275
  VBA, lack of support for, 235
  VBA conversion, 337–339
  Visual Studio add-ins, 236

• *p* •

Page Layout tab, 28
Paint, creating icons with, 68
Paint.NET graphics editor, 69
parallelism, 340–341
Percentage number format, 176
Personal Folders, Outlook, 243
pictures
  Custom UI Editor, 75–76
  RibbonX and Visual Studio, 107–109
  toolbar buttons, 315–316
  VBA, 74
  Word, 123
`Platform Invoke (PInvoke)` methods, 352
PMT equation, 169
PNG file type
  buttons, 42, 69
  templates, 154
PowerPoint (Microsoft) applications
  built-in property values, 272–273
  constant data, supporting, 278–280
  creating initial slide, 268–271
  custom presentation tab, 263–267
  custom properties, saving, 271–272

PowerPoint (Microsoft) applications *(continued)*
custom property values, 273–274
described, 261–262
ending, providing presentation, 280–283
Home tab, 27
initial slide, importance of, 267–268
new version, 30
optional slides, 276–278
template, saving and using, 283–284
template, uses for, 262–263
user data, obtaining from Outlook, 274–275
VBA conversion, 334–335
Presentation ML Web site, 383
printing labels to document, 145–146
procedure, redundant calculations, 182–183
project, defining, 100–101
properties, custom, 271–272
property values (PowerPoint)
built-in, 272–273
custom, 273–274
protection
Excel forms, 189
numbers into forms, 199
Word documents, 149
PSchmid.net tools, examples, and products, 379–380
public versus private Web services, 287–288
public-use equations. *See* nonstandard equations tab
pushbutton
`onAction` attribute, 26
size, 26

• Q •

Quick Access Toolbar (QAT)
design rules, 35
Office, 324
VBA conversion, 331–333
Quick Style Set, modifying and storing in template, 325

• R •

rate values, converting, 175
reading e-mail, 237
recipient
greeting in letter, 138–139
Letter/Memo tab, adding, 129–133
redundant calculations
code, creating, 179–180
data identification requirements, 186–187
described, 176–177
dialog boxes, designing, 178–179
global and local variables, 185–186
linkages to existing data, 180–182
loop, 183–186
problem solution, defining, 177–178
procedure, choosing, 182–183
References tab, 28
Registry, removing DLL entries, 106
report, expense
cost calculations, 198–199
disappearing and reappearing fields, 194
Employee Name field, 192–194
employee selections, defining, 195–198
REpresentational State Transfer (REST), 302
repurposing controls
described, 39, 75, 107
sample code, 112
Resource Hacker, 316
resources
SmartTag, 354
task panes, 353–354
VBA, converting to VB.NET, 356–357
resources, RibbonX
answers from other sources, 379
Microsoft blogs, 377
Microsoft Developer Network, 376
Microsoft Forums, 378
MSDN and other print magazines, 383
news sources, 377–378
OpenXMLDeveloper.org, 382–383
PSchmid.net tools, examples, and products, 379–380
RibbonCustomizer, 380–382
Technorati blogs, 382

responding to e-mail, 237
REST (REpresentational State Transfer), 302
Restrict Formatting and Editing task pane, 149
return address, omitting from envelope, 145
reverse order, XML file entries, 222
Review tab, 28
Ribbon
  benefits of using, 12–13, 373
  creating from scratch with Visual Studio, 116–117
  custom templates, 12
  history of, 11–12
  macros, defining in Access applications, 208
  Office Compatibility Pack, 16–17
  Office Migration Planning Manager, 17–23
  Office support, 13–16
  training users to use, 311–313
  XML information, 23
RibbonCustomizer utility, 312, 380–381
`ribbon.Invalidate()` call, 172
RibbonX
  alternatives for common tasks, 368
  hybrid applications, 370–371
  namespaces, 23
  organizational aids, 368–369
  reusing code, 372
  support and training costs, 373
  user needs, targeting specific, 367
  user task criteria, 371–372
  user-assisted application integration, 369
  VBA conversion, 333–334
  Web services, 370
  workflow solution, 366–367
RibbonX and Visual Studio
  add-ins, 92–93
  code, adding, 102–103
  described, 91–92
  documents, 93
  dynamic document content, 93–94
  end users, creating package for, 103–104
  files, adding, 101–102
  graphics, handling, 107–109
  loading Ribbon, performing tasks while, 109–110
  managed code, advantages of, 94–95
  modifying Office menu, 114–116
  .NET Programmability Support option, installing, 95–98
  new tabs, groups, and controls, 110
  order of entries, 44
  problem layout, 45–46
  project, defining, 100–101
  removing add-in, 104–106
  repurposing tabs, groups, and controls, 111–114
  Ribbon, creating from scratch, 116–117
  secure environment, creating, 94
  tasks to code, 106–107
  templates, 93
  VBA versus, 99–100
  VB.NET and C#, 98–99
RibbonX design
  Contextual Tabsets, 37–38
  controls, choosing, 41–42
  described, 31–32, 39
  elements goals, 32–34
  existing Office features, 36
  features, hiding, 43
  MiniToolbar, 38–39
  names, effective use of, 40
  number of items on tab, 40–41
  Office Menu, 36–37
  tooltips, 34–36
  user hints, 43
  user's perspective on groups, 41
RibbonX elements
  bitmaps versus icons, 69
  control attributes, 59–61
  control callbacks, 61–64
  control graphics tools, 68–69
  controls, choosing, 26
  controls, coding, 28
  controls, defining, 27, 55
  controls overview, 56–58
  described, 23–24, 49
  groups, 26, 52–55
  objects versus, 59
  Office icons, obtaining list of, 67–68

RibbonX elements *(continued)*
Office 2007 Custom UI Editor, 64–66
tab, 24–25, 50–52
32-bit images, 70
transparency, 69–70
RibbonX groups
elements, 52–55
scripts (VBA), 77–78
Visual Studio, 110
RibbonX resources
answers from other sources, 379
Microsoft blogs, 377
Microsoft Developer Network, 376
Microsoft Forums, 378
MSDN and other print magazines, 383
news sources, 377–378
OpenXMLDeveloper.org, 382–383
PSchmid.net tools, examples, and products, 379–380
RibbonCustomizer, 380–382
Technorati blogs, 382
RibbonX scripts (VBA)
basic tab, 75–76
callback subs, automatically generating, 76–77
creating Ribbon from scratch, 87–88
custom user interface, 79–81
described, 71
developers, 72–73
forms, adding, 88–90
groups and controls, 77–78
identifier, obtaining for existing tab, group, or control, 78–79
limitations, 73–75
Office menu, adding items to, 83–84
Office menu controls, repurposing, 85–86
tasks, performing when Ribbon loads, 86–87
user input, reacting to, 81–82
routing requirements, 139–142
row size, optimal, 192
RTF (Rich Text Format), 255

• S •

saving
database file in new format, 218
e-mail as draft, 246–247
Excel document as template, 189–190
form as template, 149
macros, 72–73
Office 2007 document, 85
PowerPoint custom properties, 271–272
PowerPoint template, 283–284
scanning tools, Office Migration Planning Manager, 18–19
schema, 205
Schneider, Robert (*SQL Server 2005 Express Edition For Dummies*), 19
`screenTip`
attribute, 60
group, 53, 55
scripts, RibbonX (VBA)
basic tab, 75–76
callback subs, automatically generating, 76–77
creating Ribbon from scratch, 87–88
custom user interface, 79–81
described, 71
developers, 72–73
forms, adding, 88–90
groups and controls, 77–78
identifier, obtaining for existing tab, group, or control, 78–79
limitations, 73–75
Office menu, adding items to, 83–84
Office menu controls, repurposing, 85–86
tasks, performing when Ribbon loads, 86–87
user input, reacting to, 81–82
`Search Criteria` group, 295
`SearchIndexes`, 292–293
secure environment, creating, 94
sender, adding to Letter/Memo tab, 136–138
`SetRateText()` method, 182
`SetSelectedTerm()` method, 182
`showImage` attribute, 60
`showItemImage` attribute, 60
`showItemLabel` attribute, 60
`showLabel` attribute, 61
signature, adding to letter, 138–139
Simple Object Access Protocol (SOAP), 302–305
Site Feedback forum, 378
size
bitmap button images, 69
button, 42, 56, 143
pushbutton, 26

`size` attribute, 61
`sizeString` attribute, 61
slide
  initial, creating, 268–271
  initial, importance of, 267–268
  optional, 276–278
  Slideshow tab, 30
Slideshow tab, 30
SmartTag
  conversion issues, 346–347
  resources, 354
Softpedia graphics file usage article, 378
Software Development Kit, eBay Web service, 285
Solution Explorer
  multiple Outlook class issues, 248–249
  Visual Studio conversion, 349
Sort Criterion group, 296
Split Button
  controls, 42
  RibbonX element (`splitButton`), 58
Spreadsheet ML Web site, 382
SQL Server
  database, creating, 20–21
  downloading, 19
  Web services, 307
SQL Server 2005 Express Edition For Dummies (Schneider), 19
SQL Server 2005 For Dummies (Watt), 19
SqlXml 3.0 Service Pack 3 (SP3) add-in, 19
Start group, 166
`StartFromScratch` mode, QAT, 34
styles
  Letter/Memo tab, 124–125
  Visual Studio conversion, 350
Styles gallery, Home tab, 325
Styles group, Word, 43
`superTip` attribute, 61
`supertip` group, 53, 55
super-tooltip, 35
system table, `USysRibbons`, 209–211

• T •

tab
  Access applications, adding, 207–208
  creating with group and control, using Custom UI Editor, 65–66
  described, 12
  groups as part of, 24
  number of items on, 40–41
  RibbonX and Visual Studio, 110
  RibbonX elements, creating, 50–52
table
  exporting to new database, 212
  system, loading Ribbon in, 209–211
  XML, loading as part of, 208
Table Selection field, filling with data, 228–230
`tag` attribute, 61
`TakeFocusOnClick` property, 315
targeted solutions, menus and toolbars versus, 367
task entry, turning e-mail text into, 251–253
task pane
  Excel, 323
  resources, 353–354
  Visual Studio, 347
tasks
  Office menu, caution against adding to, 37
  performing when Ribbon loads, 86–87
  RibbonX and Visual Studio, performing while loading Ribbon, 109–110
  solutions based on, 358–359
Technorati blogs, 382
telephone number, adding to task, 259
template
  caution against modifying while open, 74
  custom, 12
  data entry forms, 189–190
  Excel, saving document as, 189–190
  Excel forms, 189
  features, 124
  form, saving as, 149
  PNG file type, 154
  PowerPoint, saving and using, 283–284
  PowerPoint applications, 262–263
  RibbonX and Visual Studio, 93
  Visual Studio, 99
  Word forms information for display, 152–154
  Word forms variable and interaction, 151–152
`TempVars()` function, 229
text
  e-mail, defining task, 253–258
  e-mail, turning into task entry, 251–253
third-party licensing, RibbonX, 13

third-party tools
  Add-in Express Web site, 354
  menus and toolbars, mimicking, 317
32-bit images, 70
`ThisAddIn` class, 152
time estimate, conversion, 338
time interval, viewing Outlook messages, 131
`title` attribute, 61
Toggle Button
  callback control, 62
  code sample, 80
  control (`toggleButton`), 58
  current state, showing, 82
  described, 42
  Excel equations, choosing, 167–168
  `getPressed` callback, 80
  Home tab, 207–208
  `onAction` callback, 80
toolbar buttons. *See* buttons
Toolbar Toggle add-in, 317
toolbars
  conversion issues, 328, 334–335
  forms, substituting, 313–316, 318
  in QAT, 332
  inadequacy of, 13–14
  targeted solutions versus, 367
  third-party tools, mimicking with, 317
  VBA conversion, 330–331
  Visual Studio conversion, 358
tooltips, RibbonX design, 34–36
transition, easing with RibbonCustomizer, 312
transparency, RibbonX elements, 69–70
tutorials
  task panes, 354
  XML, 23
2007 Office System: XML Schema Reference, 54

• U •

underlining
  normal versus alternative function, 113–114
  Toggle Button, 82
United States `SearchIndexes`, 292–293
updates
  Office XP or Office 2003, 16
  Ribbon interface, 15
URL, Amazon Web Services, 290
user customization, Quick Access Toolbar (QAT), 35
user data, obtaining from Outlook, 274–275
user interface
  conversion issues, 328
  Ribbons, adding from table, 213–214
  RibbonX, 73
  RibbonX scripts (VBA), 79–81
user selections for incoming mail
  closing task, 259–260
  defining task, 253–258
  multiple class issues, 247–251
  supplemental information, 258–259
  task-creation interface, 251–253
users
  information, adding to Word forms, 156–159
  RibbonX scripts (VBA), reacting to input, 81–82
  specific needs, targeting, 367
  training to use Ribbon, 311–313
  transition, easing with RibbonCustomizer, 312
  workflow solution, 366–367
users, meeting needs of. *See* RibbonX design
`USysRibbons`, 209–211

• V •

validating data, 159–160
variables, global and local, 185–186
VBA
  Access add-ins, 337
  Excel applications, 164–165
  graphics, 74
  lack of support for, 235
  number formats, 176
  RibbonX and Visual Studio versus, 99–100
  32-bit images, 70
  VB.NET, converting to, 355–356
  Visual Studio conversion, 357–358
  Word applications, 122–123
VBA conversion
  Access conversion, 335–337
  common code, 342–343
  conversion, issues behind, 328–329
  conversion strategy, 329–330

described, 327
existing menus and toolbars, 330–331
forms, 330
Office 2007 user, 341–342
Office XP/2003 user, 339–341
Outlook conversion, 337–339
Quick Access Toolbar (QAT), 331–333
RibbonX changes, 333–334
toolbars and menus, designing rather than creating, 331
Word, Excel, and PowerPoint conversion, 334–335
VBA forum, 378
VBA scripts, RibbonX
basic tab, 75–76
callback subs, automatically generating, 76–77
creating Ribbon from scratch, 87–88
custom user interface, 79–81
described, 71
developers, 72–73
forms, adding, 88–90
groups and controls, 77–78
identifier, obtaining for existing tab, group, or control, 78–79
limitations, 73–75
Office menu, adding items to, 83–84
Office menu controls, repurposing, 85–86
tasks, performing when Ribbon loads, 86–87
user input, reacting to, 81–82
VB.NET
as Ribbon storage solution, 336
RibbonX and Visual Studio, 98–99
VBA solutions, converting, 355–356
versions, older
Access, 29, 335–337
documents, saving as, 85
Office Compatibility Pack, 16–17
`visible` attribute, 61
Visual Studio
Access add-ins, 337–338
Excel applications, 164–165
installing, 96–97
Outlook applications, 236
32-bit images, 70
Visual Studio and RibbonX
add-ins, 92–93
code, adding, 102–103
described, 91–92
documents, 93
dynamic document content, 93–94
end users, creating package for, 103–104
files, adding, 101–102
graphics, handling, 107–109
loading Ribbon, performing tasks while, 109–110
managed code, advantages of, 94–95
modifying Office menu, 114–116
.NET Programmability Support option, installing, 95–98
new tabs, groups, and controls, 110
order of entries, 44
problem layout, 45–46
project, defining, 100–101
removing add-in, 104–106
repurposing tabs, groups, and controls, 111–114
Ribbon, creating from scratch, 116–117
secure environment, creating, 94
tasks to code, 106–107
templates, 93
VBA versus, 99–100
VB.NET and C#, 98–99
Visual Studio conversion
considering, 348–350
customizing, 360–361
described, 345–346
existing add-ins from RibbonX, 346–347
feasibility, 356
form-based toolbars, 353
functions, preserving old, 355
menus and toolbars, 358
re-creating projects, 350–352
reviewing add-ins, 352
for specific applications, 359–360
styles, 350
task pane resources, 353–354
VBA walkthrough, 357–358
VB.NET alternative, 356–357
workflows and task-based solutions, 358–359
Visual Studio Conversion Wizard, 348–349
Visual Studio forum, 378
Visual Studio Tools for Office 2005 Second Edition, 97–98
VSTO (Non-Visual Studio Tools for Office), 37

• W •

warning
  error messages, XML Notepad, 204–205
  when saving document as older Word version, 85
Watt, Andrew (*SQL Server 2005 For Dummies*), 19
Web services. *See also* Amazon.com
  described, 285
  hiding, 286
  private, 307
  public versus private, 287–288
  Ribbon X, 370
Windows 95 (Microsoft), 350
Windows Vista (Microsoft)
  help (HLP) files, rewriting, 219–220
  Visual Studio, installing, 96–97
  Visual Studio appearance, 350
Windows XP (Microsoft), 350
Word (Microsoft) applications. *See also* Letter/Memo tab
  described, 121–122
  envelopes, 142–145
  Home tab, 27
  labels, 145
  new version, 28
  Quick Style Set, modifying and storing in template, 325
  Styles group user hint, 43
  Table Tools Contextual Tabset, 37–38
  VBA, 122–123
  VBA conversion, 334–335
  Visual Studio, 123
Word (Microsoft) forms
  Content Controls versus Legacy Controls, 150
  creating, 147–149
  date, including, 159–161
  flow of information, 146–147
  new document, creating, 154–156
  physical presentation, 151
  selecting, 150
  tab, 146
  template information for display, 152–154
  Templates variable and interaction, 151–152
  user information, adding, 156–159
Word (Microsoft) Letter/Memo tab
  CC, routing, and approval requirements, 139–142
  custom envelope output, 143–145
  dates, 133–136
  described, 123–124
  document type, changing, 126–127
  greeting recipient and adding signature, 138–139
  labels and images, modifying, 125–126
  recipient, adding, 129–133
  restoring document type after Ribbon loads, 128
  sender, adding, 136–138
  style, setting, 124–125
  workflow, 366
WordprocessingML Web site, 382
workflow solutions
  designing, 366–367
  Visual Studio conversion, 358–359
worksheet
  cell range linkage, 194–195
  external data access, 29
  heading, 186

• X •

XML (eXtensible Markup Language)
  button image code, 44
  file, using directly in Access, 214–218
  group, creating (<group> element), 25
  multiple Outlook class issues, 247–248
  new Ribbon element, creating, 25
  problem solution, defining, 177
  schema in Access applications, 204–205
  SQL Server version, 19
  table, loading as part of, 208
XML (eXtensible Markup Language) Notepad 2007
  correct entries, ensuring, 205
  downloading, 203
  error messages, 204–205
  schema, removing, 205