RibbonX
Customizing the Office 2007 Ribbon

Robert Martin, Ken Puls, Teresa Hennig
RibbonX: Customizing the Office 2007 Ribbon

Chapter 7: comboBox and dropDown Controls

ISBN: 978-0-470-19111-8
Contents

Introduction xxix

Part I The Building Blocks for a Successful Customization 1

Chapter 1 An Introduction to the Office User Interface 3

What Is the Ribbon and Why Does It Exist? 3
Problems with the Old UI 4
Issues Solved with the New UI 6
Issues Created with the New UI 7
What Happened to the Toolbars from My Pre-2007 Files? 9
A Customization Example for Pre-2007 UIs 10
Ribbon Components 16
Tips for Navigating the Ribbon and Quick Access Toolbar (QAT) 17
Using Keyboard Shortcuts and Keytips 17
Using the Mouse Wheel 19
Minimizing and Maximizing the Ribbon 19
Adding Commands to the QAT 20
Assigning a Macro to a QAT Button 22
Changing the QAT Location 24
Preparing for Ribbon Customization 24
Showing the Developer Tab 24
Showing CustomUI Errors at Load Time 26
Reviewing Office 2007 Security 26
Has Customization of the Office UI Become Easier? 27
Conclusion 27

Chapter 2 Accessing the UI Customization Layer 29

Accessing the Excel and Word Ribbon Customization Layers 30
What's New in Excel and Word Files? 30
Creating a Ribbon Customization with Notepad 30
Contents

Creating the customUI File
Creating the File to Use the Customized UI
Attaching the XML to the File
Using the Microsoft Office 2007 Custom UI Editor to Modify Your UI
Installing Microsoft .NET Framework 2.0 for Windows XP Users
Installing the Microsoft Office 2007 Custom UI Editor
Using the CustomUI Editor to Customize the Ribbon
Storing Customization Templates in the CustomUI Editor
Some Notes About Using the CustomUI Editor
XML Notepad
Installing XML Notepad
Using XML Notepad
The Benefits of XML Notepad
The Drawbacks of XML Notepad
A Final Word on Excel and Word Customizations
Microsoft Access Customizations
Storing the CustomUI Information in Tables
Creating an Access UI Modification Using a Table
Access USysRibbons Caveat
Other Techniques for Access UI Customizations
Conclusion

Chapter 3 Understanding XML
What Is XML and Why Do You Need It?
Essential Background
  Tags
  Elements
  Attributes
    The id Attribute
    The label Attribute
  Tips for Laying Out XML Code
  Creating Comments in XML Code
The Core XML Framework
  The customUI Element
    Required Attributes of the customUI Element
    Optional Static and Dynamic Attributes with Callback Signatures
    Allowed Children Objects of the customUI Element
  The ribbon Element
    Required Attributes of the ribbon Element
    Optional Static Attributes
    Allowed Children Objects of the ribbon Element
    Graphical View of ribbon Attributes
  The tabs Element
    Required Attributes of the tabs Element
    Allowed Children Objects of the tabs Element
Writing Your Own Code 110
Naming Conventions 111
Data Types 112
Working with Events 114
Workbook Events 115
Worksheet Events 117
Form and Report Events in Access 119
Document-Level Events in Word 122
Application-Level Events 123
The Object Browser 125
Referencing Libraries 126
Early and Late Bindings Explained 128
Debugging Your Code 129
Debug.Print and Debug.Assert 130
Stop Statement 131
Immediate Window 132
Locals Window 134
Watches Window 135
Error Handling 137
On Error Resume Next 138
On Error GoTo 138
Working with Arrays 140
Determining the Boundaries of an Array 141
Resizing Arrays 142
Conclusion 143

Chapter 5  Callbacks: The Key to Adding Functionality to Your Custom UI 145

Callbacks: What They Are and Why You Need Them 145
Setting Up the File for Dynamic Callbacks 146
Capturing the IRibbonUI Object 147
  Adjusting the XML to Include onLoad 147
  Setting Up VBA Code to Handle the onLoad Event 147
Generating Your First Callback 148
  Writing Your Callback from Scratch 148
  Using the Office CustomUI Editor to Generate Callbacks 150
Understanding the Order of Events When a File Is Open 151
Can I Have Two Callbacks with the Same Name But Different Signatures? 152
Calling Procedures Located in Different Workbooks 153
Organizing Your Callbacks 155
  Individual Callback Handlers 155
  Using Global Callback Handlers 157
Handling Callbacks in Access 158
  Using VBA to Handle Callbacks 158
  Using Macros to Handle Callbacks 160
Invalidating UI Components 162
   What Invalidating Does and Why You Need It 162
   Invalidating the Entire Ribbon 163
   Invalidating Individual Controls 165
Conclusion 167

Chapter 6 RibbonX Basic Controls 169
The button Element 169
   Required Attributes of the button Element 170
   Optional Static and Dynamic Attributes with Callback Signatures 171
   Allowed Children Objects of the button Element 173
   Parent Objects of the button Element 173
   Graphical View of button Attributes 173
   Using Built-in button Controls 174
   A button Idiosyncrasy: The showLabel Attribute 175
   Creating Custom button Controls 176
      An Excel Example 176
      A Word Example 179
      An Access Example 181
The checkBox Element 183
   Required Attributes of the checkBox Element 184
   Optional Static and Dynamic Attributes with Callback Signatures 184
   Allowed Children Objects of the checkBox Element 186
   Parent Objects of the button Element 186
   Graphical View of checkBox Attributes 186
   Using Built-in checkBox Controls 187
   Creating Custom Controls 188
      An Excel Example 188
      A Word Example 192
      An Access Example 194
The editBox Element 196
   Required Attributes of the editBox Element 197
   Optional Static and Dynamic Attributes with Callback Signatures 197
   Allowed Children Objects of the editBox Element 199
   Parent Objects of the editBox Element 199
   Graphical View of editBox Attributes 200
   Using Built-in editBox Controls 200
   Creating Custom Controls 200
      An Excel Example 200
      A Word Example 203
      An Access Example 205
The toggleButton Element 209
   Required Attributes of the toggleButton Element 209
   Optional Static and Dynamic Attributes with Callback Signatures 210
Contents

Allowed Children Objects of the toggleButton Element 212
Parent Objects of the toggleButton Element 212
Graphical View of toggleButton Attributes 212
Using Built-in toggleButton Controls 213
Creating Custom Controls 214
   An Excel Example 214
   A Word Example 217
   An Access Example 220
Conclusion 223

Chapter 7 comboBox and dropDown Controls 225
The item Element 225
   Required Attributes of the item Element 226
   Optional Static and Dynamic Attributes with Callback Signatures 226
   Allowed Children Objects of the item Element 227
   Parent Objects of the item Element 227
   Graphical View of item Attributes 227
   Using Built-in Controls 228
   Creating Custom Controls 228
The comboBox Element 229
   Required Attributes of the comboBox Element 229
   Optional Static and Dynamic Attributes with Callback Signatures 229
   Allowed Children Objects of the comboBox Element 232
   Parent Objects of the comboBox Element 232
   Graphical View of comboBox Attributes 232
   Using Built-in Controls 232
   Creating Custom Controls 234
      An Excel Example 235
      A Word Example 237
      An Access Example 239
The dropDown Element 244
   Required Attributes of the dropDown Element 244
   Optional Static and Dynamic Attributes with Callback Signatures 244
   Allowed Children Objects of the dropDown Element 247
   Parent Objects of the dropDown Element 247
   Graphical View of dropDown Attributes 248
   Using Built-in Controls 248
   Creating Custom Controls 249
      An Excel Example 249
      A Word Example 254
      An Access Example 258
Conclusion 261
Chapter 10 Formatting Elements

The box Element
- Required Attributes of the box Element
- Optional Static and Dynamic Attributes with Callback Signatures
- Allowed Children Objects of the box Element
- Parent Objects of the box Element
- Graphical View of box Attributes
- Using Built-in box Elements
- Creating Custom box Elements
- Horizontal Alignment
- Vertical Alignment
- Nesting box Controls

The buttonGroup element
- Required Attributes of the buttonGroup element
- Optional Static and Dynamic Attributes with Callback Signatures
- Allowed Children Objects of the buttonGroup Element
- Parent Objects of the buttonGroup Element
- Graphical View of a buttonGroup
- Using Built-in buttonGroup Elements
- Creating Custom buttonGroup Elements

The labelControl Element
- Required Attributes
- Optional Static and Dynamic Attributes with Callback Signatures
- Allowed Children Objects of the labelControl Element
- Parent Objects of the labelControl Element
- Graphical View of a labelControl
- Using Built-in labelControl Elements
- Creating Custom labelControl Elements

The separator Element
- Required Attributes of the separator Element
- Optional Static and Dynamic Attributes with Callback Signatures
- Allowed Children Objects of the separator Element
- Parent Objects of the separator Element
- Graphical View of a Separator
- Using Built-in separator Elements
- Creating Custom separator Elements
Chapter 11 Using Controls and Attributes to Help Your Users

The menuSeparator Element
- Required Attributes of the menuSeparator Element 347
- Optional Static and Dynamic Attributes with Callback Signatures 348
- Allowed Children Objects of the menuSeparator Element 349
- Parent Objects of the menuSeparator Element 349
- Graphical View of the menuSeparator Element 349
- Using Built-in menuSeparator Elements 350
- Creating Custom menuSeparator Elements 350

Conclusion 352

Part II Advanced Concepts in Ribbon Customization

Chapter 12 Advanced VBA Techniques

Working with Collections
- Determining Whether an Item Belongs to a Collection 377
Class Modules
- Properties, Methods, and Events 378
- Working with Properties 379
- Working with Methods 380
- Working with Events 382
- Web Services and CustomUI 383
Using VBA Custom Properties
- Setting Up the Custom Properties 389
- Saving and Retrieving Values from the Registry 394

Conclusion 399

Chapter 13 Overriding Built-in Controls in the Ribbon

Starting the UI from Scratch
- Setting the startFromScratch Attribute 402
- Activating a Tab at Startup 404
- Disabling and Repurposing Commands 406
xxii  Contents

Disabling Commands, Application Options, and Exit 406
  Disabling Commands 406
  Disabling the Commands Associated with the Application Options and Exit Controls 407
  Repurposing a Command Associated with a Generic Control 408
  Affecting the Keyboard Shortcuts and Keytips 410
Conclusion 412

Chapter 14  Customizing the Office Menu and the QAT 413
Adding Items to the Office Menu 413
Adding Items to the QAT 418
  Customization Overview 418
    sharedControls versus documentControls 419
Adding Custom and Built-in Commands to the QAT 420
Adding Custom and Built-in Groups to the QAT 422
Repurposing QAT Controls 424
Table-Driven Approach for QAT Customization (Excel and Word) 428
Table-Driven Approach for QAT Customization (Access) 430
QAT Caveats 433
  Inability to Load Controls 433
  Inability to Load Custom Images to Controls 434
  Duplication of Controls on XML-Based and XML-Free Customizations 434
Conclusion 435

Chapter 15  Working with Contextual Controls 437
Making Your Items Contextual 437
  Tabs 438
  Groups 439
Working Through Nonvisibility Methods 441
  Enabling and Disabling Controls 441
Working with Contextual Tabs and tabSets 442
  Creating a Custom Contextual Tab in Access 442
    Renaming a tabSet 444
    Modifying Built-in Contextual Tabs 445
Working with Contextual Pop-up Menus 447
  Replacing Built-in Pop-up Menus in Their Entirety 448
    Adding Individual Items to Pop-up Menus 453
Multilingual UI 455
Conclusion 458

Chapter 16  Sharing and Deploying Ribbon Customizations 459
Excel Deployment Techniques 460
  Distributing Workbooks 460
  Using Templates 461
Creating and Deploying Add-ins 463
    Preparing a Workbook for Conversion to an Add-in 464
    Converting a Workbook to an Add-in Format 465
Installing an Add-in 465
Unloading and Removing Add-ins 467
Toggling the IsAddin Property 467
A Note on the PERSONAL.XLSB Workbook 468
Word Deployment Techniques 469
Distributing Documents 469
Using Templates 470
Configuring Template Directories 470
Creating Templates 471
Global Templates 472
Preparing a Document for Conversion to a Global Template 473
Converting a Template to a Global Template 474
Editing Global Templates 475
Removing Global Templates 476
A Note on the Normal.dotm Template 476
Sharing Ribbon Items Across Files(Word and Excel) 477
Creating a Shared Namespace 478
Sharing Tabs and Groups in Excel 479
Sharing Tabs and Groups in Word 485
Deploying Word and Excel Solutions Where Multiple
Versions of Office are in Use 491
Do Legacy CommandBar Customizations Still Work? 491
Method 1: Creating Separate Versions 492
Method 2: Calling a Previous Version from a New Add-in 493
Using a 2003 Excel Add-in as a Front-End Loader for a 2007 Add-in 494
Using a Word 2007 Global Template as a Front-End for a 2003 Template 500
Access Deployment Techniques 504
General Information Concerning Database Deployment 504
Preparing the Files for Multi-User Environments 504
Managing Access Startup Options 507
Leveraging the startFromScratch Attribute 507
Adjusting Access Options for Your Users 508
Creating an ACCDE File 510
Loading the customUI from an External Source 511
Deploying Solutions to Users with Full-Access Installations 514
Deploying Customizations with Full Versions of Access 514
Deploying Solutions to Users with the Access Runtime Version 518
Conclusion 519

Chapter 17 Security In Microsoft Office 523
Security Prior to Office 2007 524
Macro-Enabled and Macro-Free File Formats 524
The Trust Center 525
Trusted Publishers 526
## Contents

Trusted Locations
- Adding, Modifying, or Removing Trusted Locations  528
- Trusting Network Locations  529
- Disabling Trusted Locations  529

Add-ins
- Requiring Add-ins to Be Signed  530
- Disabling Notification for Unsigned Add-ins  530
- Disabling All Add-ins  531

ActiveX Settings  531

Macro Settings
- Setting Macro Options  532
- Trusting VBA Project Access  533

Message Bar  533

Privacy Options  534

Digital Certificates  534
- How Digital Certificates Work  534
- Acquiring a Digital Certificate  535
- Using SELFCERT.exe to Create a Digital Signature  536
- Adding a Digital Certificate to a Project  537
- Trusting a Digital Certificate on Another Machine  538
- Deleting a Digital Certificate from Your Machine  540

Conclusion  542

### Appendix A Tables of RibbonX Tags

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to Use This Appendix</td>
<td>545</td>
</tr>
<tr>
<td>Ribbon Container Elements</td>
<td>546</td>
</tr>
<tr>
<td>customUI Element</td>
<td>546</td>
</tr>
<tr>
<td>ribbon Element</td>
<td>548</td>
</tr>
<tr>
<td>contextualTabs Element</td>
<td>548</td>
</tr>
<tr>
<td>tabSet Element</td>
<td>548</td>
</tr>
<tr>
<td>qat Element</td>
<td>549</td>
</tr>
<tr>
<td>sharedControls Element</td>
<td>549</td>
</tr>
<tr>
<td>documentControls Element</td>
<td>549</td>
</tr>
<tr>
<td>officeMenu Element</td>
<td>549</td>
</tr>
<tr>
<td>tabs Element</td>
<td>550</td>
</tr>
<tr>
<td>tab Element</td>
<td>550</td>
</tr>
<tr>
<td>group Element</td>
<td>551</td>
</tr>
<tr>
<td>Ribbon Control Elements</td>
<td>552</td>
</tr>
<tr>
<td>box Element</td>
<td>553</td>
</tr>
<tr>
<td>button Element</td>
<td>553</td>
</tr>
<tr>
<td>buttonGroup Element</td>
<td>556</td>
</tr>
<tr>
<td>checkBox Element</td>
<td>557</td>
</tr>
<tr>
<td>comboBox Element</td>
<td>558</td>
</tr>
<tr>
<td>dialogBoxLauncher Element</td>
<td>562</td>
</tr>
<tr>
<td>dropDown Element</td>
<td>562</td>
</tr>
<tr>
<td>dynamicMenu Element</td>
<td>566</td>
</tr>
</tbody>
</table>
Appendix B Tables of Tab and Group idMso Names 587
  Common Tab idMso Identifiers 587
  Contextual Tab idMso Identifiers 588
    Contextual Tab idMso Identifiers for Excel 588
    Contextual Tab idMso Identifiers for Access 589
    Contextual Tab idMso Identifiers for Word 590
  Group idMso Identifiers 590
    Excel’s Group idMso Identifiers 590
    Access’s Group idMso Identifiers 595
    Word’s Group idMso Identifiers 600

Appendix C imageMso Reference Guide 607
  How to Get Your Own imageMso References 607
  Your Own Reference Tool 608

Appendix D Keytips and Accelerator keys 611
  Keytips and Accelerator Keys for Excel 611

Appendix E RibbonX Naming Conventions 615
  How Our Naming System Works 615
  Naming Samples 617

Appendix F Where to Find Help 621
  Websites with RibbonX Information 621
  Websites Maintained by the Authoring and Tech Edit Team 623
  Newsgroups 623
  Web Forums 624

Index 627
In the previous chapter, you learned about the button, checkBox, editBox, and toggleButton controls. This chapter explores two new controls: the comboBox and the dropDown. The comboBox and dropDown list are similar in a great many ways, including design, implementation, and appearance. They also have some important differences that are discussed in this chapter.

Before you can start exploring these two controls, you need to learn about the fundamental element that supports them: the item element. The chapter begins by exploring this critical piece.

Following the section on the item element, you’ll find both the comboBox and dropDown sections, which explore these two elements in great detail. As in the previous chapter, examples are included for each application, some of which display the creation of static versions of controls, while others create fully dynamic versions. Whether you’re working through the examples or just reading the chapter, you will appreciate seeing a fully functioning version. As you are preparing to work through the examples, we encourage you to download the companion files. The source code and files for this chapter can be found on the book’s web site at wiley.com/go/ribbonx.

The item Element

The item element is used to create static items that must be used within a gallery, dropDown, or comboBox. This particular element is not intended for use on its own, but rather must be an integral part of other controls, such as those mentioned above.
Unlike other sections of this book, where we provide full working examples of the element being discussed, we review only the XML construct and discuss the structure of the `item` element. This departure from the pattern reflects the fact that the `item` element is so tightly integrated with the `comboBox`, `dropDown`, and gallery controls that it cannot be separated from them. Because of this, you need to know a little bit about `item` before you can move on. The way these are used may seem rather complicated for now, but rest assured that the processes will become surprisingly clear when you examine the `comboBox` and `dropDown` RibbonX elements later in this chapter.

**Required Attributes of the item Element**

Each `item` requires a unique `id` attribute, as described in Table 7-1.

<table>
<thead>
<tr>
<th>ATTRIBUTE</th>
<th>WHEN TO USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Use this attribute to create your own item.</td>
</tr>
</tbody>
</table>

The `item` element has only one attribute, the `id`. As we just mentioned, the `item` control must be used in conjunction with other elements; therefore, it relies on the other elements for all other attributes.

**NOTE** Unlike other elements, there is no `idMso` or `idQ` attribute available for the `item` control.

**Optional Static and Dynamic Attributes with Callback Signatures**

Each `item` element can optionally make use of any or all of the attributes shown in Table 7-2.

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED VALUES</th>
<th>DEFAULT VALUE</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>imageMso</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>label</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>(none)</td>
</tr>
</tbody>
</table>
Table 7-2 (continued)

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED VALUES</th>
<th>DEFAULT VALUE</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>screentip</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>supertip</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Did you notice the lack of callback signatures in Table 7-2?

It is important to understand that the items in the XML code underlying Ribbon modifications are always static. This is actually a good thing, as it means that you can provide a static list of items to the control without having to write a single line of VBA.

Don’t misunderstand this to mean that you can’t create items for a control on-the-fly using VBA, as that is not the case. When you create an item for a control, the callback is actually associated with the Parent object (i.e., the comboBox, dropDown, or gallery control), not the actual item itself.

**NOTE** As you design your XML code, you are given a choice between using static item elements (specified in your XML), or dynamic elements (generated via the parent control’s callback signature.) Whichever route you choose is mutually exclusive of the other. In other words, if you specify static items for a control, you cannot also specify dynamic items for that control as well.

**Allowed Children Objects of the item Element**

The item element does not support child objects of any kind, so it cannot have any embedded controls.

**Parent Objects of the item Element**

An item may only be used in the following controls:

- comboBox
- dropDown
- gallery

**Graphical View of item Attributes**

Figure 7-1 shows a dropDown control on a Ribbon group. It houses three static item elements that are written into the XML code. The callouts annotate where the item’s static attributes are displayed.
Using Built-in Controls

You don’t have the opportunity to leverage built-in item elements because Microsoft does not expose them for our use, but the chances are fairly good that you wouldn’t want to use them anyway, so it’s not really a big loss. If you did want to have some of these items in a control, you could just include the entire parent element in your Ribbon.

Creating Custom Controls

We explore the setup for the parent controls shortly, so this section focuses on the underlying XML code needed to create the static item elements demonstrated in Figure 7-1.

As mentioned earlier, Figure 7-1 makes use of a dropDown control to house these items. The dropDown control should be constructed, like all controls, with opening and closing XML tags. The following code goes between those tags:

```xml
<item id="rxitemddColor1"
  imageMso="AppointmentColor1"
  label="Red"
  screentip="This is Red!"
  supertip="Trust me, it really is Red!"/>
<item id="rxitemddColor2"
  imageMso="AppointmentColor2"
  label="Blue"
  screentip="This is Blue!"
  supertip="Trust me, it really is Blue!"/>
<item id="rxitemddColor3"
  imageMso="AppointmentColor10"
  label="Yellow"
  screentip="This is Yellow!"
  supertip="Trust me, it really is Yellow!"/>
```

As you review Figure 7-1, you will see each of the elements in the live representation on the Ribbon.
The comboBox Element

The comboBox control displays data based on a designated record source, and it is a hybrid of the editBox that we covered in Chapter 6 and the dropDown control that we review next.

One of the great features of a comboBox control is that, in addition to being able to pick something from the list, the user can also type something into the text box. In searching and selecting from the list, it acts like a "hot lookup," enabling users to skip to the closest match. As the user keeps typing, the choices are narrowed down. At any time, users may accept one of the displayed values or they may keep typing to create a new entry.

The comboBox is best used in the following situations:

- The list is very long and you wish to give users the capability to quickly jump to the appropriate place by typing a few keys. (The fonts control is a good example of this.)
- You wish to present users with a pre-defined list, but also want them to be able to add items to the list.

As mentioned earlier in the chapter, you can populate the comboBox with both static lists and dynamically created lists.

Required Attributes of the comboBox Element

The comboBox control requires any one of the id attributes shown in Table 7-3.

Table 7-3: Required id Attributes of the comboBox Element

<table>
<thead>
<tr>
<th>ATTRIBUTE</th>
<th>WHEN TO USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>When creating your own comboBox</td>
</tr>
<tr>
<td>idMso</td>
<td>When using an existing Microsoft comboBox</td>
</tr>
<tr>
<td>idQ</td>
<td>When creating a comboBox shared between namespaces</td>
</tr>
</tbody>
</table>

Optional Static and Dynamic Attributes withCallback Signatures

In addition to the required id attribute, the comboBox control will optionally accept any one of the insert attributes listed in Table 7-4.
Table 7-4: Optional insert Attributes of the comboBox Element

<table>
<thead>
<tr>
<th>INSERT ATTRIBUTE</th>
<th>ALLOWED VALUES</th>
<th>DEFAULT VALUE</th>
<th>WHEN TO USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertAfterMso</td>
<td>Valid Mso Group</td>
<td>Insert at end of group</td>
<td>Insert after Microsoft control</td>
</tr>
<tr>
<td>insertBeforeMso</td>
<td>Valid Mso Group</td>
<td>Insert at end of group</td>
<td>Insert before Microsoft control</td>
</tr>
<tr>
<td>insertAfterQ</td>
<td>Valid Group idQ</td>
<td>Insert at end of group</td>
<td>Insert after shared namespace control</td>
</tr>
<tr>
<td>insertBeforeQ</td>
<td>Valid Group idQ</td>
<td>Insert at end of group</td>
<td>Insert before shared namespace control</td>
</tr>
</tbody>
</table>

Finally, the comboBox may also be configured to use any or all of the optional attributes or callbacks shown in Table 7-5.

Table 7-5: Optional Attributes and Callbacks of the comboBox Element

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED VALUES</th>
<th>DEFAULT VALUE</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>onChange</td>
<td>1 to 4096 characters</td>
<td>(none)</td>
<td>Sub OnChange (control As IRibbonControl, text As String)</td>
</tr>
<tr>
<td>enabled</td>
<td>getEnabled</td>
<td>true, false, 1, 0</td>
<td>true</td>
<td>Sub GetEnabled (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>image</td>
<td>getImage</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetImage (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>imageMso</td>
<td>getImage</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Same as above</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemCount</td>
<td>1 to 1024</td>
<td>(none)</td>
<td>Sub GetItemCount (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemID</td>
<td>1 to 1024</td>
<td>(none)</td>
<td>Sub getItemID (control As IRibbonControl, index As Integer, ByRef id)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemImage</td>
<td>Unique text string</td>
<td>(none)</td>
<td>Sub GetItemImage (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>STATIC ATTRIBUTE</td>
<td>DYNAMIC ATTRIBUTE</td>
<td>ALLOWED VALUES</td>
<td>DEFAULT VALUE</td>
<td>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
<td>----------------</td>
<td>--------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemLabel</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetItemLabel (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemScreenTip</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetItemScreenTip (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemSupertip</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetItemSupertip (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>keytip</td>
<td>getKeytip</td>
<td>1 to 3 characters</td>
<td>(none)</td>
<td>Sub GetKeytip (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>label</td>
<td>getLabel</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetLabel (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>maxLength</td>
<td>(none)</td>
<td>1 to 1024</td>
<td>1024</td>
<td>(none)</td>
</tr>
<tr>
<td>screentip</td>
<td>getScreentip</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetScreentip (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>showImage</td>
<td>getShowImage</td>
<td>true, false, 1, 0</td>
<td>true</td>
<td>Sub GetShowImage (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>showItemImage</td>
<td>(none)</td>
<td>true, false, 1, 0</td>
<td>true</td>
<td>(none)</td>
</tr>
<tr>
<td>showItemLabel</td>
<td>(none)</td>
<td>true, false, 1, 0</td>
<td>true</td>
<td>(none)</td>
</tr>
<tr>
<td>showLabel</td>
<td>getShowLabel</td>
<td>true, false</td>
<td>true</td>
<td>Sub GetShowLabel (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>sizeString</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>12*</td>
<td>(none)</td>
</tr>
<tr>
<td>supertip</td>
<td>getSupertip</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>Sub GetSupertip (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
</tbody>
</table>

Continued
Table 7-5 (continued)

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED VALUES</th>
<th>DEFAULT VALUE</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>(none)</td>
<td>getText</td>
<td>1 to 4096 characters</td>
<td>(none)</td>
<td>Sub GetText (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>visible</td>
<td>getVisible</td>
<td>true, false, 1, 0</td>
<td>true</td>
<td>Sub GetVisible (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
</tbody>
</table>

**NOTE** The default value for the sizeString attribute (if the attribute is not declared at all) is approximately 12, but this varies depending on the characters used and the system font.

**Allowed Children Objects of the comboBox Element**

The only child object that can be used with the comboBox element is the item element.

**Parent Objects of the comboBox Element**

The comboBox element may be nested within the following elements:

- box
- group

**Graphical View of comboBox Attributes**

Figure 7-2 gives a graphical representation of the visible attributes that can be set on the comboBox control.

Figure 7-2 shows all of the comboBox attributes except for the screentip and superstip. These two attributes only show when the dropDown list portion is not active; consequently, they could not be captured while showing the dropDown list.

**Using Built-in Controls**

Of all the controls in Excel and Word, probably the best known is the Fonts comboBox. If you are creating custom Ribbon tabs to group your most commonly used controls together, then you will certainly want to add this control. It is therefore the first one that we look at.
Chapter 7 • comboBox and dropDown Controls  233

Figure 7-2: Graphical view of the comboBox elements

The XML code for the example that you are about to build is almost completely application agnostic; it will work equally well in Excel and Word, and it only requires a very minor change to work in Access. We’ll go through examples for each application so that you will be comfortable and confident in all three.

NOTE The completed example files (comboBox-Fonts.xlsx, comboBox-Fonts.docx, and comboBox-Fonts.accdb) can be downloaded from the book’s website at www.wiley.com/go/ribbonx.

Because we are only using built-in controls, this example does not require VBA. We’ll begin with Excel, so start by creating and saving a new .xlsx file. Open the file in the CustomUI Editor, apply the RibbonBase template you created in Chapter 2, and insert the following code between the <tabs> and </tabs> tags:

```xml
<tab id="rxtabDemo"
  label="Demo"
  insertBeforeMso="TabHome">
  <group id="rxgrpDemo"
    label="Demo">
    <comboBox idMso="Font"/>
    <comboBox idMso="FontSize"/>
  </group>
</tab>
```

Remember to validate your code to catch any pesky typing errors. Once you have done this, save the file, and open it in Excel. Navigate to the Demo tab and you will see the UI shown in Figure 7-3.

Figure 7-3: The Font and FontSize comboBox controls on a new Excel tab
With regard to differences between Excel and Word, this code is 100% application agnostic. Try it for yourself — create a new .docx file and open it in the CustomUI Editor. Enter the same XML code and save the file. (Remember to validate the code if you retyped it all by hand!) Upon opening Word, you’ll see the Ribbon as it appears in Figure 7-4.

![Figure 7-4: The Font and FontSize comboBox controls on a new Word tab](image)

As we mentioned, there is a minor change required to get this code to work in Access. You merely need to revise the insertBeforeMso attribute, as TabHome is not a defined tab in Access. In other words, the Demo tab’s insertBeforeMso attribute for Access should read as follows:

```
insertBeforeMso="TabHomeAccess"
```

With this code in place, when you reopen the database, the UI will appear as shown in Figure 7-5.

![Figure 7-5: The Font and FontSize comboBox controls on a new Access tab](image)

“But wait,” you say, “that does not look exactly the same!” That’s true — the name of the font is not displayed because nothing is open that has a font that can be edited. If you open a table or a form, you’ll find that these controls are immediately enabled. Our example was intended to provide the added benefit of demonstrating when you might see the blank list, so you needn’t be alarmed that your code might have failed.

**Creating Custom Controls**

Demonstrating how to use Font and FontSize comboBoxes as was an easy example that you will reuse many times, but we can only get so far using Microsoft’s built-in comboBoxes. It’s time to look at some examples of building items that do things that Microsoft hasn’t allowed for.
Chapter 7 • comboBox and dropDown Controls

The Excel and Word examples in this section display how to employ static lists in the comboBox controls and incorporate the item controls that were covered in the first section of this chapter. With the Access example, we start to explore a dynamic comboBox control.

An Excel Example

For this example, we again assume that you have hidden the entire user interface. In addition, also imagine that in an attempt to make your application look less like Excel, you have hidden all the worksheet tabs. You still want users to be able to move between the three worksheets, however, so you need to provide some vehicle to accomplish this. In many ways, the comboBox can be an ideal control for this kind of navigation: It lists all the “pages” in your application, and allows users to type in a specific one that they may wish to jump to.

Naturally, you need a macro or two to make this work, so create a new macro-enabled Excel file (.xlsm). Save it, and then open it in the CustomUI Editor. Apply the RibbonBase template, and insert the following code between the <tabs> and </tabs> tags:

```xml
	<tab id="rxtabDemo"
    label="Navigation"
    insertBeforeMso="TabHome">  
		<group id="rxgrpNavigate"
        label="Navigate To">
			<comboBox id="rxcboSelectSheet"
                        label="Activate:
                        onChange="rxcboSelectSheet_Click">
				<item id="rxitemcboSelectSheet1"
                        label="Sheet1"/>
				<item id="rxitemcboSelectSheet2"
                        label="Sheet2"/>
				<item id="rxitemcboSelectSheet3"
                        label="Sheet3"/>
			</comboBox>
		</group>
	</tab>
```

Notice that the comboBox makes use of the onChange callback to take action when an item is selected. In addition, this comboBox holds three items: Sheet1, Sheet2, and Sheet3. These items are static and cannot be changed from within the file; nor will the user be able to add additional items. This is a perfect fit for the goal of ensuring that users are able to navigate only to these worksheets.

Before you close the CustomUI Editor, be sure to validate your code for typing errors, and then copy the onChange callback.

CROSS-REFERENCE For a refresher on working with callbacks and the CustomUI Editor, see Chapter 5.
When you are back in Excel, open the VBE and paste the callback into a new standard module. Now you need to edit the callback to make it react as you wish. You can figure out how to do that by thinking through the order of events:

1. The user will select an item from the Worksheet list.
2. The callback will be triggered.
3. The selected value of the comboBox (in this case, the worksheet name) will be passed to the routine.
4. The worksheet will be activated.

So far so good, but there is one more piece that may cause a glitch: We stated that users can only select one of the sheets specified in the code. And what happens if they type in a different value? To deal with these eventualities, you should edit the callback signature to read as follows:

```
'Callback for rxcboSelectSheet onChange
Sub rxcboSelectSheet_Click(control As IRibbonControl, text As String)
    On Error Resume Next
    Worksheets(text).Activate
    If Err.Number <> 0 Then
        MsgBox "Sorry, that worksheet does not exist!"
    End If
End Sub
```

The second line of code attempts to activate the worksheet that has been passed to the callback. Items chosen from the list will always be a valid name, but text typed in by the user may not match an item in the list. The On Error statement at the beginning of the routine deals with this by telling the code to continue to the next line even if an error is present.

At this point, you can check whether the Err property is zero. If the Err property does not equal zero, then an error must have occurred; therefore, the value the user typed is not valid. In addition, because you want the user to know that their input wasn’t acceptable, you include a message box. Of course, you can display whatever message that you’d like and even add other options, but message box customization is outside our focus, so let’s keep moving.

Now that your callback is set up correctly, click the Navigation tab and play with the comboBox. Try selecting items from the list, as well as typing in a value, as shown in Figure 7-6.

![Figure 7-6: Using a comboBox to change worksheets in Excel](image)
A Word Example

Recall that in Chapter 6 we built a customization that allowed the user to enter the width of the Styles pane in an editBox. Although this works nicely for giving users the capability to control the width, it isn’t very intuitive, as there is no indication of what type of values to put in the box. There are some techniques that provide this information to the user, but those are more complex, and we cover them in Chapter 11. For now, you can use the comboBox to provide the user with a pre-set list of options from which to choose.

To make things easy for the user, we keep the checkBox that we used to toggle the Styles pane on and off. In addition, rather than create everything from scratch, you can use the editBox example file created in Chapter 6. In that, we replaced the editBox with the comboBox control.

**Note** Instead of using the editBox example you created in the previous chapter, you can also download the complete editBox example, editBox-Style Inspector Width.docm, from the book’s website.

To get started, open the existing file in the CustomUI Editor and replace the editBox-specific XML with the following:

```
<comboBox id="rxcboStyleWidth"
    label="Inspector Width"
    sizeString="1234"
    onChange="rxcboStyleWidth_Click">
    <item id="rxitemcboStyleWidth1"
        label="1.00"/>
    <item id="rxitemcboStyleWidth2"
        label="2.00"/>
    <item id="rxitemcboStyleWidth3"
        label="3.00"/>
    <item id="rxitemcboStyleWidth4"
        label="4.00"/>
    <item id="rxitemcboStyleWidth5"
        label="5.00"/>
    <item id="rxitemcboStyleWidth6"
        label="6.00"/>
    <item id="rxitemcboStyleWidth7"
        label="7.00"/>
    <item id="rxitemcboStyleWidth8"
        label="8.00"/>
</comboBox>
```

As usual, validate the XML code before saving the file, and because you already have the checkBox callbacks programmed in the file, you only need to copy the callback signature for the rxcboStyleWidth_click event.

If you examine this XML, you will see that we have added eight options to the comboBox’s drop-down list portion: the numbers 1.00 through 8.00. Remember that while each item’s id must be unique, it is the item’s label that will be passed to the callback when the control is accessed.
In this example, a four-character `sizeString` is declared even though we only show three digits. This is because we want to allow a number with two decimal places and one leading digit. The decimal also counts as a character, so we must include it in our maximum size.

Upon opening Word, you will again want to jump into the VBE right away to paste your callback. Recall that earlier in the chapter we mentioned that the `comboBox` control is a hybrid between the `editBox` and `dropDown` controls. This works very much to your benefit in this case because the `editBox` callback already exists in the file.

To port the code from the `editBox` to the `comboBox`, you have two options:

1. Copy the code from the `rxtxtStyleWidth_getText` routine to the `rxComboBoxStyleWidth_Click` routine and delete the original `rxtxt` routine.
2. Replace the `rxtxtStyleWidth_getText` signature line with the `rxComboBoxStyleWidth_Click` line.

If you elect to rewrite the signature line, it is a good idea to get into the practice of making sure that you rewrite all of the parameters as well, not just the name of the callback. This will ensure that if you are updating from one type of callback to another it will still run.

If you paste a callback signature line into your module and then update another callback to an identical signature, be sure to delete or rename the original. Otherwise, you will receive an error that an “Ambiguous Name Has Been Detected.”

Once you have updated your procedure, save the file and select the View tab. Your updated Style Options will look like what is shown in Figure 7-7.

![Figure 7-7: Using a comboBox to change Word’s Style Inspector width](image)
Try using the comboBox to change the width of the Style Inspector pane. Notice the following:

- It only shows when the checkbox is checked.
- The width changes each time you select a new value.
- Typing a custom value also changes the Style Inspector width.
- Typing text will return a custom error message.

An Access Example

In this example, we again build on the Access database used in the last chapter. This time, we add a new group containing a comboBox that opens a form directly to a specific field.

Unlike previous examples that used static callbacks, this customization makes use of callbacks to dynamically populate the comboBox with a list of all the authors in the tblAuthors table. To do this, we repopulate the comboBox every time the form is closed. We recognize that the data in the comboBox and the table may become out of sync, but it could be impractical to maintain the XML code for each and every record added to the database, as frequent updates would have a significant performance impact.

To get started, open the Access example from the previous section or download the toggleButton-Form Tools.accdb file from the book's website. Copy the RibbonX information from the RibbonXML field of the USysRibbons table and paste it into a fresh instance of the CustomUI Editor. Between the </group> and </group id="rxgrpTools"> tags, enter the following XML for the comboBox control:

```xml
<group id="rxgrpSearch"
      label="Search Tools">
  <comboBox id="rxcboSearchAuthor"
            label="Review Author:
                   
                   onChange="rxcboSearchAuthor_Click"
                   getItemCount="rxcboSearchAuthor_getItemCount"
                   getItemCount="rxcboSearchAuthor_getItemLabel"/>
</group>
```

As you can immediately see, the comboBox specification looks much different than that shown in the Excel and Word examples. A host of callbacks are declared and no item controls are listed. Because the items will be provided dynamically by the callbacks at run-time, it isn’t necessary to declare any static items.

**NOTE** Although it has been mentioned before, it is worth repeating a very important point: Even if you did want to declare one or more static item controls that will show up in every comboBox control you use, it must be done through the callback if you elect to populate any controls dynamically. Use of dynamic (or static) callbacks is an “all or nothing” approach.
Another thing to notice about the preceding XML code is that there is no \</comboBox\> tag. Instead, the comboBox declaration ends with a /> instead of the / character. As discussed in Chapter 3, this is possible because no child elements are declared for the comboBox control.

Once you have the code validated and copied into the RibbonXML field of the USysRibbons table, generate and copy all of the rxcboSearchAuthor callbacks.

If you are keeping track, you will notice that we have declared four callbacks, but only three callouts are generated by the CustomUI Editor. Specifically, the rxcboSearchAuthor_getItemId callback signature is missing, which appears to be a bug in the CustomUI Editor program.

Fortunately, you can consult Table 7-5, earlier in this chapter, and obtain the signature for this callback. It is declared as follows:

Sub GetItemID(control As IRibbonControl, index As Integer, ByRef id)

NOTE If for some reason you don’t have this book with you when you’re attempting this in a project, you can also search for these callback signatures on the Internet. An article containing all of the callback signatures can be located on the MSDN site at the following URL: http://msdn2.microsoft.com/en-us/library/ad7222523.aspx

Most of the callback signatures have been generated for you, so copy and paste them into the code module that holds the rest of the code. Next, you need to manually type in the GetItemID callback signature.

Before we start writing the code, it’s time to figure out exactly how this works. Again, you do this by thinking through the steps in a logical manner. We want to do the following:

1. Create a list of all the authors currently in the database.
2. Submit that list to the comboBox.
3. Open the form when the user clicks on (or manually enters) an author’s name.
4. Let the user know when an author cannot be found (and close the form).
5. Start again at Step 1 when the form is closed (to ensure that the list is current if any authors were added).

As with all code, the methods by which you accomplish these things are limited only to your imagination and ability. What is detailed below can certainly be done differently, but it works well for the purposes of this example.

To begin, the code needs a place to store the list of author names, and a count of those names. To do this, you add two global variables to the top of your project, just underneath the Option lines. The first is a variant array (a dynamically sized space to hold the list of names), and the second is a Long data type, which can hold the count of the names in the array. They read as follows:

Public gaAuthorList As Variant
Public glngItemCount As Long
Now that you have a variable to hold the list of names, it makes sense to work on the routine that populates the list. Because we indicated that the list will be populated when the file is loaded and every time a form is closed, it makes sense to build the process into a routine, and to call the routine whenever we need to check the table and populate the list. The routine looks as follows:

```vba
Sub PopulateAuthorArray()
'Populate the global array of author names
Dim db As DAO.Database
Dim rst As DAO.Recordset
Dim lngLoopCount As Long

'Open the recordset for the authors table
Set db = CurrentDb()
Set rst = db.OpenRecordset("tblAuthors", dbOpenTable)

'Store list of all authors in an array
On Error Resume Next
lngLoopCount = 0
With rst
    If .EOF Then .MoveFirst
    glngItemCount = .RecordCount
    ReDim gaAuthorList(glngItemCount - 1)
    Do While Not (.EOF)
        gaAuthorList(lngLoopCount) = !AuthorName.Value
        .MoveNext
        lngLoopCount = lngLoopCount + 1
    Loop
End With

'Release all objects
rst.Close
db.Close
Set rst = Nothing
Set db = Nothing
End Sub
```

The routine loops through all the records in the tblAuthors table and adds them to the gaAuthorList array. In addition, it sets the glngItemCount variable to hold the total number of records that were in the table (and therefore the array.)

Next, we deal with the callbacks.

The getItemCount callback simply returns the number stored in the glngItemCount variable, telling the control how many records you have:

```vba
'Callback for rxcboSearchAuthor getItemCount
Sub rxcboSearchAuthor_getItemCount(control As IRibbonControl, _
    ByRef returnedVal)
    returnedVal = glngItemCount
End Sub
```
The `getItemID` callback is used to create a unique item ID for each of the item controls that are dynamically fed into the `comboBox`. The index variable is always unique (numbered from zero to the number of items you have, less one), so you can simply add the index to a string of text, as shown here:

```vba
'Callback for rxcboSearchAuthor getItemID
Sub rxcboSearchAuthor_getItemID(control As IRibbonControl, _
    index As Integer, ByRef ID)
    ID = "rxitemcboSearchAuthor" & index
End Sub
```

**TIP** You may wonder why we didn’t just set up this callback using `ID = index` to generate an ID for the item. We could have used that approach, but it would fail if we added another `comboBox` to the project and used the same logic because the IDs would no longer be unique (i.e., two controls could end up with an `id=0`). It is a far better practice to ensure that your ID is tagged with your control name. Therefore, in addition to generating a unique ID, you are now generating a coding habit that will never leave you debugging this issue.

The `getItemLabel` callback returns the author’s name from the global array that was established earlier. When the array is created, each element has an index, or place, in the array. You can use the following callback to return the element of the array corresponding to the index number:

```vba
'Callback for rxcboSearchAuthor getItemLabel
Sub rxcboSearchAuthor_getItemLabel(control As IRibbonControl, _
    index As Integer, ByRef returnedVal)
    returnedVal = gaAuthorList(index)
End Sub
```

**NOTE** If you are experienced with changing the base of an array and intend to apply that technique in these databases, you need to adjust these indexes to coincide with your arrays.

The `Click` event is the final callback that needs programming in our example. It uses the following code:

```vba
'Callback for rxcboSearchAuthor onChange
Sub rxcboSearchAuthor_Click(control As IRibbonControl, text As String)
    'Open the form at the requested Author
    Dim sAuthorName As String
    Dim rs As DAO.Recordset

    sAuthorName = "[AuthorName] = '" & text & "'"

    'Open the appropriate form
    OpenForm "AuthorForm", FormFields = sAuthorName
End Sub
```
This event opens the form and attempts to activate the record pertaining to the author who has either been selected from the comboBox or typed in manually. If the record can be found it is “bookmarked” and activated. If the record is not found, the user is informed (via our friendly message box) and the form is closed.

There is one final thing left to do in order for this example to work: You need to hook the PopulateAuthorArray to the appropriate routines. Without this hook, the array will never be filled and the comboBox will sit empty.

To create the hook, simply insert the following code snippet in the rxIRibbonUI_onLoad event (before or after the line that captures the RibbonUI), as well as in the frmAuthor’s Form_Deactivate event (just before the line that invalidates the RibbonUI):

```
'Populate the array of Author names
Call PopulateAuthorArray
```

Now that you have made all of these modifications, save, close, and reopen your database. You will have a new group on the Ribbon that holds the comboBox. Select an author’s name from the list and you will be taken to their record, as shown in Figure 7-8.

---

**Figure 7-8:** Using a comboBox to jump to a specific
Now try adding your name to the Author’s table. After all, you have done a lot of work just to get to this point! After adding your name, close the form and check the comboBox list. Voilà, your name in print!

The dropDown Element

Like the comboBox, the dropDown control presents the user with a pre-defined list of options from which to choose. In addition, it too can be populated either at design-time using XML to provide a static list, or dynamically at run-time via callbacks.

The biggest of differences between the comboBox and dropDown Controls lies in the ability of the comboBox to accept user-entered data; the dropDown control has no such facility, forcing the user to select an item from the pre-defined list and only from that list.

At first glance, you may ask yourself why anyone would want to use a dropDown over a comboBox. After all, wouldn’t you always want to make the controls more robust and accessible? The answer depends upon your implementation, of course, but some reasons you may want to use the dropDown control include the following:

- You do not want users to enter their own information.
- Your list is not long, so using the “auto complete” capability is not a concern.
- You are not interested in programming the callbacks to validate user-entered data.

Required Attributes of the dropDown Element

To create a dropDown control, you need to define one and only one of the id attributes shown in Table 7-6.

<table>
<thead>
<tr>
<th>ATTRIBUTE</th>
<th>WHEN TO USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>When creating your own dropDown</td>
</tr>
<tr>
<td>idMso</td>
<td>When using an existing Microsoft dropDown</td>
</tr>
<tr>
<td>idQ</td>
<td>When creating a dropDown shared between namespaces</td>
</tr>
</tbody>
</table>

Optional Static and Dynamic Attributes with Callback Signatures

The dropDown element optionally accepts any one insert attribute shown in Table 7-7.
**Table 7-7:** Optional **insert** Attributes for the **dropDown** Element

<table>
<thead>
<tr>
<th>INSERT ATTRIBUTE</th>
<th>ALLOWED VALUES</th>
<th>DEFAULT VALUE</th>
<th>WHEN TO USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertAfterMso</td>
<td>Valid Mso Group</td>
<td>Insert at end of group</td>
<td>Insert after Microsoft control</td>
</tr>
<tr>
<td>insertBeforeMso</td>
<td>Valid Mso Group</td>
<td>Insert at end of group</td>
<td>Insert before Microsoft control</td>
</tr>
<tr>
<td>insertAfterQ</td>
<td>Valid Group idQ</td>
<td>Insert at end of group</td>
<td>Insert after shared namespace control</td>
</tr>
<tr>
<td>insertBeforeQ</td>
<td>Valid Group idQ</td>
<td>Insert at end of group</td>
<td>Insert before shared namespace control</td>
</tr>
</tbody>
</table>

In addition to the **insert** attribute, you may also include any or all of the optional static attributes or dynamic equivalents shown in **Table 7-8**.

**Table 7-8:** Optional Attributes and Callbacks of the **dropDown** Element

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED DEFAULT VALUES</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>getEnabled</td>
<td>true, false, 1, 0</td>
<td>Sub GetEnabled (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>image</td>
<td>getImage</td>
<td>1 to 1024 characters</td>
<td>Sub GetImage (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>imageMso</td>
<td>getImage</td>
<td>1 to 1024 characters</td>
<td>Same as above</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemsCount</td>
<td>1 to 1024 (none)</td>
<td>Sub GetItemsCount (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemsID</td>
<td>Unique text string</td>
<td>Sub GetItemsID (control As IRibbonControl, index As Integer, ByRef id)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemsImage</td>
<td>1 to 1024 characters</td>
<td>Sub GetItemsImage (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
</tbody>
</table>

*Continued*
Table 7-8 (continued)

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED DEFAULT VALUES</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>getItemLabel</td>
<td>1 to 1024 characters</td>
<td>Sub getItemLabel (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemScreentip</td>
<td>1 to 1024 characters</td>
<td>Sub getItemScreentip (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getItemSupertip</td>
<td>1 to 1024 characters</td>
<td>Sub getItemSupertip (control As IRibbonControl, index As Integer, ByRef returnedVal)</td>
</tr>
<tr>
<td>keytip</td>
<td>getKeytip</td>
<td>1 to 3 characters</td>
<td>Sub getKeytip (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>label</td>
<td>getLabel</td>
<td>1 to 1024 characters</td>
<td>Sub getLabel (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>screentip</td>
<td>getScreentip</td>
<td>1 to 1024 characters</td>
<td>Sub getScreentip (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getSelected ItemID</td>
<td>Unique text string</td>
<td>Sub getSelectedItemID (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>(none)</td>
<td>getSelected ItemIndex</td>
<td>1 to 1024 characters</td>
<td>Sub getSelectedItemIndex (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>showImage</td>
<td>getShowImage</td>
<td>true, false, 1, 0</td>
<td>Sub getShowImage (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>showItem Image</td>
<td>(none)</td>
<td>true, false, 1, 0</td>
<td>(none)</td>
</tr>
<tr>
<td>showItem Label</td>
<td>(none)</td>
<td>true, false, 1, 0</td>
<td>(none)</td>
</tr>
</tbody>
</table>
### Chapter 7 = ComboBox and dropDown Controls 247

#### Table 7-8  (continued)

<table>
<thead>
<tr>
<th>STATIC ATTRIBUTE</th>
<th>DYNAMIC ATTRIBUTE</th>
<th>ALLOWED DEFAULT VALUES</th>
<th>VBA CALLBACK SIGNATURE FOR DYNAMIC ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>showLabel</td>
<td>getShowLabel</td>
<td>true, false, 1, 0</td>
<td>Sub GetShowLabel (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>sizeString</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
</tr>
<tr>
<td>supertip</td>
<td>getSupertip</td>
<td>1 to 1024 characters</td>
<td>Sub GetSupertip (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td>tag</td>
<td>(none)</td>
<td>1 to 1024 characters</td>
<td>(none)</td>
</tr>
<tr>
<td>visible</td>
<td>getVisible</td>
<td>true, false, 1, 0</td>
<td>Sub GetVisible (control As IRibbonControl, ByRef returnedVal)</td>
</tr>
<tr>
<td></td>
<td>onAction</td>
<td>1 to 1024 characters</td>
<td>Sub OnAction (control As IRibbonControl, selectedId As String, selectedIndex As Integer)</td>
</tr>
</tbody>
</table>

**NOTE** The default value for the `sizeString` attribute (if the attribute is not declared at all) is approximately 12, but this will vary depending on the characters used and the system font.

#### Allowed Children Objects of the dropDown Element

The only child object that can be used with the dropDown element is the item element.

#### Parent Objects of the dropDown Element

The dropDown element may be used within the following controls:

- box
- group
Graphical View of dropDown Attributes

Figure 7-9 shows a dropDown control on a Ribbon group, which houses two dynamic items. The captions annotate the properties that can be set on a dropDown control.

Using Built-in Controls

Because there are no built-in dropDown controls that span all the applications, this example again focuses on referencing one of Excel’s native dropDown controls: the BorderStyle control.

To make use of the control, create a new .xlsx file and open it in the CustomUI Editor. Apply the RibbonBase template and place the following code between the <tabs> and </tabs> tags:

```xml
<tab id="rxtabDemo"
     label="Demo"
     insertBeforeMso="TabHome">
    <group id="rxgrpDemo"
           label="Demo">
        <dropDown idMso="BorderStyle"/>
    </group>
</tab>
```

Validate your XML, save the file, and reopen it in Excel. You will now see the fully functional control available on the Demo tab, as shown in Figure 7-10.
Creating Custom Controls

In this section we again create custom tools, rather than reuse the commands and functionality inherent in the programs.

In addition to employing a static dropDown list in Excel, the previous comboBox example will also be revised so that it dynamically populates a dropDown control. The Word example will remain static, but we add another useful tool to the collection that you have been building. The Access example again creates a completely dynamic list in the dropDown control.

An Excel Example

This example is quite interesting because it uses two dropDown elements in tandem. Similar to the comboBox example in the previous section, the first control lists all the worksheets in the workbook. In this case, however, we employ the available callbacks to update the dropDown list as worksheets are added to or removed from the workbook. Pretty cool — and you can certainly see the value in the example and anticipate ample opportunities to incorporate this into your customizations.

NOTE By switching from a comboBox to a dropDown, we lose the capability to type in the worksheet we want to activate. It should be assumed that this process was a conscious choice on the part of the developer.

The second control allows us to toggle the visibility of the selected worksheet between Excel's three states: xlSheetVisible, xlSheetHidden, and xlSheetVeryHidden.

NOTE The xlSheetVeryHidden state of an Excel worksheet is not known to most users because it must be set from VBA. This is a state that enables a developer to completely hide a worksheet from the users, and it will not show in the menu that allows users to unhide sheets.

One of the best ways to highlight the differences between the two controls is to convert one to the other, so that is exactly how this example begins. If you have not completed the previous example, or are unsure how it compares to the examples in this book, download the comboBox-Select Sheet.xlsm file from the book's website.

Before you open the file in Excel, open it in the CustomUI Editor and replace everything from and including the <comboBox> to the end of and including </comboBox> with the following XML:

```xml
<dropDown id="rxddSelectSheet"
  label="Apply To:"
  visible="true"
  onAction="rxddSelectSheet_Click"
  getItemCount="rxitemddSelectSheet_getItemCount"
  getItemCount="rxitemddSelectSheet_getItemCount"
  getItemCount="rxitemddSelectSheet_getItemCount"
  getItemCountLabel="rxitemddSelectSheet_getItemLabel"/>
```
Notice a few things about the preceding code:

- The first dropDown declaration has no ending </dropDown> tag. This is because it contains all of its attributes within the code block and is requesting all of its child objects via callbacks; therefore, it does not require any static child objects declared in the XML, and can be closed by using /> at the end.

- The callback signature for the dropDown is different from the comboBox. Whereas the comboBox used an onChange callback, the dropDown uses an onAction callback when it is clicked.

- The second dropDown list does have a </dropDown> tag to close it. This is because it holds a static list of item objects declared directly in the XML code.

In addition, we also need to capture the RibbonUI object in order to update the lists dynamically. Adjust the CustomUI tag to include the onLoad element as shown below:

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

As always, you should validate your code, save it, and generate and copy the callback signatures. As with the comboBox element, be aware that the callback for the getItemID callback will not be generated by the CustomUI Editor. If you were doing this on your own, you’d once again need to look this up and type it in manually (or copy it from a functional example).

Open Excel again, but don’t be alarmed when you see the error message indicating that it cannot run the macro, shown in Figure 7-11.

![Figure 7-11: Error message indicating a missing callback](image)

This is to be expected, as we have declared an onLoad callback but have not yet provided the programming. To do that, in the VBE, open the code module that holds all the RibbonX event code and paste the callback signatures at the end.
Before we go any further, let’s deal with the onLoad callback. You want to add two variables to the top of the project. The first will hold the RibbonUI object, while the second will store the worksheet name that was selected. They are placed just under the Option Explicit line and should be declared as follows:

```vba
Public RibbonUI As IRibbonUI
Dim sSheetName As String
```

Next, we make sure that the RibbonUI object is captured at load time by setting the onLoad callback as follows:

```vba
'Callback for customUI.onLoad
Sub rxIRibbonUI_onLoad(ribbon As IRibbonUI)
    Set RibbonUI = ribbon
End Sub
```

Now it’s time to look at the rest of the callback signatures, starting with the callback that tells the control how many items exist. Fortunately, it doesn’t require much code to make this work, as the item count will always be equal to the number of worksheets in the workbook. You can use the following code:

```vba
'Callback for rxddSelectSheet getItemCount
Sub rxitemddSelectSheet_getItemCount(control As IRibbonControl, ByRef returnedVal)
    returnedVal = Worksheets.Count
End Sub
```

Next, we set up the callback for the getItemLabel, which returns the text of each item to the dropDown list:

```vba
'Callback for rxddSelectSheet getItemLabel
Sub rxitemddSelectSheet_getItemLabel(control As IRibbonControl, _
    index As Integer, ByRef returnedVal)
    returnedVal = Worksheets(index + 1).Name
End Sub
```

If you haven’t worked much with arrays, you might not notice a very big issue lurking in the middle of this routine. Pay careful attention to the fact that the name being returned for an item is the index + 1.

The reason for this shift is that by default VBA works in zero-based arrays (as previously mentioned, VBA starts counting at 0, not 1), but Excel’s default worksheet indexes and names work in one-based arrays (Excel starts counting at 1, not 0.)

To understand the ramifications, assume that we have a workbook set up to start with the default 3 worksheets. In the getItemCount procedure, we asked for a count of the Worksheets 1, 2, and 3. We received a total count of 3, as we would expect. What is interesting, however, is how this actually dimensions the array. Have a look at Table 7-9 to see how the array will manifest itself.
Table 7-9: Dimensions and Values of an Array

<table>
<thead>
<tr>
<th>ARRAY INDEX</th>
<th>ELEMENT VALUE</th>
<th>ACTUAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Worksheets(index + 1).Name</td>
<td>Sheet1</td>
</tr>
<tr>
<td>1</td>
<td>Worksheets(index + 1).Name</td>
<td>Sheet2</td>
</tr>
<tr>
<td>2</td>
<td>Worksheets(index + 1).Name</td>
<td>Sheet3</td>
</tr>
</tbody>
</table>

**NOTE** If this appears intimidating, don’t worry about it. It is confusing for most users who are not accustomed to working with arrays. As a rule of thumb, if the values you are trying to pass in to or retrieve from an array appear to be out of sync by one, just adjust your index, as shown in the previous code snippet.

Next, you will want to ensure that the callback is programmed to dynamically generate the unique ID for each of the drop-down items. The callback should read as follows:

```vba
'Callback for rxddSelectSheet getItemID
Sub rxitemddSelectSheet_getItemID(control As IRibbonControl, _
    index As Integer, ByRef id)
    id = "rxitemddSelectSheet" & index
End Sub
```

There is one more callback to set up: the rxddSelectSheet_Click callback. Because we started with the previous file, you have all the code for the rxcbSelectSheet_Click event that was triggered by selecting a comboBox item. However, you can’t just rename and reuse. Take a look at Table 7-10, noting the difference between the callback signatures.

Table 7-10: Difference between dropDown and comboBox Callback Signatures

<table>
<thead>
<tr>
<th>CONTROL</th>
<th>CALLBACK SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>comboBox</td>
<td>rxcboSelectSheet_Click (control As IRibbonControl, text As String)</td>
</tr>
<tr>
<td>dropDown</td>
<td>rxddSelectSheet_Click (control As IRibbonControl, id As String, _</td>
</tr>
<tr>
<td></td>
<td>index As Integer)</td>
</tr>
</tbody>
</table>

You can see that while the comboBox passes the text of the control to the callback, the dropDown is not quite so friendly. Instead, it passes the ID and the index number. Unfortunately, we’re not interested in that number at this time; we need to show the user the actual name of the sheet.

In order to work out the control’s name, we can leverage one of the less obvious features of Ribbon construction. We can actually call the getItemLabel callback to give us this information!
If you take a good look at the `getItemLabel` callback, you’ll see that it accepts three parameters. The key to making this work is the keyword that prefaces the variable in the last parameter:

```vba
Sub rxitemddSelectSheet_getItemLabel(control As IRibbonControl, 
    index As Integer, ByRef returnedVal)
```

In VBA, every parameter is passed to a procedure specifying that the variable is passed by either `ByVal` (which is the default, so the parameter is typically omitted, as it is implied) or `ByRef`. The difference is that when a variable is passed `ByVal`, a copy of the variable is used inside the receiving procedure. Anything that is done to it within the procedure is lost when the procedure goes out of scope (ends).

In contrast, when a variable is passed to a procedure `ByRef`, the actual variable is passed as a parameter. Anything that is done to the variable inside that procedure is passed back to the calling procedure when the called procedure ends. It’s this capability that enables us to make use of the `getItemLabel` callback.

Update your callbacks so that the `rxddSelectSheet_Click` routine reads as follows, and delete the `rxcboSelectSheet_Click` event:

```vba
'Callback for rxddSelectSheet onAction
Sub rxddSelectSheet_Click(control As IRibbonControl, id As String, 
    Index As Integer)

On Error Resume Next
Call rxitemddSelectSheet_getItemLabel(control, index, sSheetName)
If Err.Number <> 0 Then
    MsgBox "Sorry, that worksheet does not exist!"
    RibbonUI.InvalidateControl "rxddSelectSheet"
End If
End Sub
```

Make note of how we are calling the `getItemLabel` callback, and especially how we are passing the `sSheetName` variable to the `returnedVal` parameter. Because the actual `sSheetName` variable is passed, and not a copy, the changes made in that procedure will replicate back to the global variable, and the worksheet name will be ready when we need it.

Finally, we’ve completed the setup for the dynamic dropDown control, and we can focus on setting up the sole callback for the static dropDown. Update the callback for this control to read as follows:

```vba
'Callback for rxddSheetVisible onAction
Sub rxddSheetVisible_Click(control As IRibbonControl, id As String, 
    index As Integer)

    'Check that a worksheet has been selected
    On Error Resume Next
    sSheetName = Worksheets(sSheetName).Name
    If Err.Number <> 0 Then
        MsgBox "Sorry, but you need to select a valid sheet first!"
        Exit Sub
    End If
End Sub
```
'Change the sheet's visibility
Select Case id
    Case "rxitemddSheetVisible1"
        Worksheets(sSheetName).Visible = xlSheetVisible
    Case "rxitemddSheetVisible2"
        Worksheets(sSheetName).Visible = xlSheetHidden
    Case "rxitemddSheetVisible3"
        Worksheets(sSheetName).Visible = xlSheetVeryHidden
End Select

'Tell user if it is last visible sheet
If Err.Number <> 0 Then
    MsgBox "Sorry, this is the only visible sheet." & vbCrLf & _
        "You can't hide them all!"
End If
On Error GoTo 0
End Sub

This callback evaluates whether any value is contained in the sSheetName variable. If there is one, it then sets the visibility of the sheet as chosen by the user.

You’ll recall that we experienced an error when the file was originally loaded, so we won’t be able to try out our customization until we save and reload the workbook. Once you’ve done this, navigate to the Review tab and play with the new controls, shown in Figure 7-12.

![Figure 7-12: The sheet visibility dropDown menu set](image)

**A Word Example**

In the tradition of adding useful tools to the Word Ribbon groups, we're going to continue adding another feature to the example that we created in the section on the comboBox.

If you frequently use Word templates, you may find that you are offered a ton of styles that you don’t use, which results in a lot of unwanted clutter. Word has four distinct settings to filter the styles to make them easier to use. In this section, we’ll create a customization that provides the capability to quickly select these four settings.

To start, make a copy of the Word comboBox example from the previous section, or download the completed comboBox-Style Inspector Width.docm file from the book’s
website. Open the new file in the CustomUI Editor and insert the following XML between the </comboBox> and </group> tags:

```xml
<dropDown id="rxddStylesView"
    label="Show Styles:"
    onAction="rxddStylesView_Click">
    <item id="rxitemddStylesAll"
        label="All"/>
    <item id="rxitemddStylesInCurrent"
        label="InCurrent"/>
    <item id="rxitemddStylesInUse"
        label="InUse"/>
    <item id="rxitemddStylesRecommended"
        label="Recommended"/>
    <item id="rxitemddStyles(none)"
        label="None"/>
</dropDown>
```

This will enter a static dropDown under the existing controls. It holds five different items: four related to the Styles view options and a final option to hide the Styles task pane.

Validate the code, as you usually do, and copy the rxddStylesView_Click callback. Open Word, open the VBE, and paste your callback signature in the module that holds the rest of the existing callback signatures. The next step is to record a macro that will capture as much information about the process as possible.

**TIP** Never overlook the macro recorder, as it is a handy tool. Even if it can only record pieces of the process, the documentation provides a great place to start learning about the objects that you are trying to manipulate. In addition, you can save a lot of time and avoid typos by copying and pasting from the generated code.

We’re trying to capture a couple of different things here. The first is how to show and hide the Styles task pane at the side of the document. The second is how to select the desired filtered view.

Let’s walk through the process. To begin, start the macro recorder and press Alt+Ctrl+Shift+S to show the Styles task pane on the side of your screen. (You could also navigate to the Home tab and click the little arrow in the bottom, right corner of the Styles group, if you prefer.)

Next, in the bottom right corner of the Styles pane, you will see an Options link. Click the link and you will see the dialog shown in Figure 7-13.

In the drop-down list, select Recommended and then click OK. Return to the Style Pane Options and select each option in turn, saying OK to each one. Finally, close the Styles task pane.
Now it’s time to examine the code that was recorded. You should see something similar to the annotated version shown here:

Sub Macro1()
  'All styles
  ActiveDocument.FormattingShowFilter = wdShowFilterFormattingInUse
  'In current document Styles
  ActiveDocument.FormattingShowFilter = wdShowFilterStylesAll
  'In use Styles
  ActiveDocument.FormattingShowFilter = wdShowFilterStylesInUse
  'Recommended Styles
  ActiveDocument.FormattingShowFilter = wdShowFilterStylesAvailable
  'Turn off the Styles task pane
  CommandBars("Styles").Visible = False
End Sub

NOTE  We’ve annotated the code because the constants that Microsoft used are less than intuitive. One might think that the wdShowFilterStylesAll would apply the All setting, not the In Current Document setting, as it does. Rather than have you waste time figuring out which is which, we’ve provided the clarification so that you can stay focused on the exercise.

NOTE  There is no code recorded to launch the Styles pane, but there is code to close it. Interestingly enough, once you have opened the Styles pane in your current Word instance, you can show the Styles task pane by executing the command CommandBars("Styles").Visible = true. Until you have activated it at least once manually, however, the code will fail. This is a known bug in Microsoft Word, and it is hoped that it will be fixed in an upcoming service pack. There is a workaround for the issue, however, which is to replace CommandBars("Styles").Visible = true with Application .TaskPanes(wdTaskPaneFormatting).Visible = True.
The good news here is that you can easily adapt this code into the callback signature, as the commands are relatively straightforward. Placing a case statement in the callback will give us the following:

```
'Callback for rxddStylesView onAction
Sub rxddStylesView_Click(control As IRibbonControl, id As String, _
Index As Integer)

On Error Resume Next
Application.TaskPanes(wdTaskPaneFormatting).Visible = True

Select Case id
    Case "rxitemddStylesAll"
        ActiveDocument.FormattingShowFilter = _
            wdShowFilterFormattingInUse
    Case "rxitemddStylesInCurrent"
        ActiveDocument.FormattingShowFilter = _
            wdShowFilterStylesAll
    Case "rxitemddStylesInUse"
        ActiveDocument.FormattingShowFilter = _
            wdShowFilterStylesInUse
    Case "rxitemddStylesRecommended"
        ActiveDocument.FormattingShowFilter = _
            wdShowFilterStylesAvailable
    Case "rxitemddStylesNone"
        CommandBars("Styles").Visible = False
End Select
End Sub
```

**NOTE** The addition of the line continuations (a space followed by the underscore and a hard return) is not essential to making the code work, but it does make it easier to read.

Once you have this code fully integrated, save the file and browse to the View tab. Try using various selections on the Show Styles dropDown, as shown in Figure 7-14.

![Show Styles dropDown in use](image)
An Access Example

The final example in this section also builds on the previous file. This time, you replace the button that was created to launch the Authors form with a dynamic dropDown list that will launch any of the forms. This dropDown will update each time a new form is added to, or removed from, the database. As an added twist, we also add a callback to retrieve an image for each item in the dropDown list.

Make a copy of the Access example from the previous section or download the comboBox-Author Query.accdb file from the book’s website. Open it and copy the RibbonX information from the RibbonXML field of the USysRibbons table. Once you’ve done that, open the CustomUI Editor and paste the code in the blank pane. Locate the following code:

```
<button id="rxbtnFrmAuthors"
    imageMso="FileCreateDocumentWorkspace"
    size="large"
    label="Enter Authors"
    onAction="rxbtnFrmAuthors_click"/>
```

Replace it with this code:

```
<dropDown id="rxddSelectForm"
    label="Select a form"
    imageMso="CreateForm"
    onAction="rxddSelectForm_click"
    getItemCount="rxddSelectForm_getItemCount"
    getItemID="rxddSelectForm_getItemID"
    getItemImage="rxddSelectForm_getItemImage"
    getItemLabel="rxddSelectForm_getItemLabel"/>
```

Like the comboBox code used in the previous example, setting up a dynamic dropDown involves a great many callbacks.

As always, validate the XML code. Once it is error-free, copy all of the XML and replace the code that is stored in the RibbonX field of your USysRibbons table.

Next, use the CustomUI Editor to generate the callbacks, copy all of the rxddSelectForm signatures, and paste them to the code module that holds the rest of the callback code. Be sure to manually create the following callback that the CustomUI Editor does not generate:

```
'Callback for rxddSelectFrom getItemID
Sub rxddSelectForm_getItemID(control As IRibbonControl, _
    index As Integer, ByRef ID)
End Sub
```

Now that all of the callbacks are placed in the module, it’s time to start hooking them up to do the things that we want. By examining the XML code, you can see that the dropDown control will be placed in the group with an icon beside it. The rest of the content, of course, will be dynamic.
Chapter 7 - ComboBox and dropDown Controls 259

In order to populate the dropDown list dynamically, you need to return the total number of items that the list will contain, which is done using the getItemCount callback. We’re interested in knowing the total number of forms in the database, and that number can be retrieved with the following code:

```vbnet
'Callback for rxddSelectForm getItemCount
Sub rxddSelectForm_getItemCount(control As IRibbonControl, _
    ByRef returnedVal)

    returnedVal = CurrentProject.AllForms.Count
End Sub
```

You also know that each item in the dropDown list must have its own unique ID. As in the previous examples, this can easily be accomplished by setting up the callback to read as follows:

```vbnet
'Callback for rxddSelectFrom getItemID
Sub rxddSelectForm_getItemID(control As IRibbonControl, _
    index As Integer, ByRef ID)

    ID = "rxitemddSelectForm" & index
End Sub
```

Next, you need to determine how to get the label for each item. Because each form has an index number in the collection, the callback can be set up as follows:

```vbnet
'Callback for rxddSelectForm getItemLabel
Sub rxddSelectForm_getItemLabel(control As IRibbonControl, _
    index As Integer, ByRef returnedVal)

    returnedVal = CurrentProject.AllForms(index).Name
End Sub
```

One more callback needs to be set up in order to populate the items in the list box. The XML code specifies that we also want an image for each item in the list, so we need a callback to get the image. The completed callback signature will look like this:

```vbnet
'Callback for rxddSelectForm getItemImage
Sub rxddSelectForm_getItemImage(control As IRibbonControl, _
    index As Integer, ByRef returnedVal)

    Select Case CurrentProject.AllForms(index).Name
        Case Is = "frmAuthors"
            returnedVal = "FileCreateDocumentWorkspace"
        Case Else
            returnedVal = "HappyFace"
    End Select
End Sub
```

This callback looks up the name of the form by passing the index number to the AllForms collection. Once that name has been returned, it is evaluated by the case
statement. If the form name matches any of the cases, it is assigned the picture specified. The else portion of this code assigns Microsoft’s HappyFace image to any form that is not specified in the code.

**NOTE** Remember that the contents of the dropDown list will be populated dynamically. This is important, as a user may add a new form — so it would not likely be in the list. By using the else statement to assign a default image, you don't have to worry about constantly updating the code.

Now that all the callbacks are in place to populate the dropDown, we need to add the callback to handle the user’s selections. We’ll use the onAction callback, which is set up as shown here:

```
'Callback for rxddSelectForm onAction
Sub rxddSelectForm_click(control As IRibbonControl, ID As String, Index As Integer)
    Dim sFormName As String
    Call rxddSelectForm_getItemLabel(control, index, sFormName)
    DoCmd.OpenForm sFormName, acNormal
    RibbonUI.InvalidateControl ("rxddSelectForm")
End Sub
```

Note that this routine again uses the trick of retrieving the item’s label by leveraging the ability to set the sFormName variable by executing the getItemLabel callback. Once the selected form name has been retrieved, the form is opened.

The invalidation contained in this routine ensures that each time a form is opened, the dropDown is repopulated. Therefore, if a new form has been added, then it will be added to and appear in the dropDown list.

**NOTE** Because a form needs to be opened via the dropDown to repopulate it, there is a one-click delay in the updating of the list each time a form is added to or deleted from the project. Ideally, the invalidation would actually be housed in a class module to monitor the creation and deletion of a form. However, this is more complicated and would distract from our demonstration of incorporating and using the dropDown control. We’ve placed the invalidation in the onAction callback. We encourage you to investigate creating a class module to monitor the form creation and removal events at your leisure.

The final piece of programming work is to clean up remnant code from the button you replaced. All you need to do is locate the rxbtnFrmAuthors_click routine and delete it from the project.

Now that all the code is in place, close and reload the Access database, thereby saving your work and allowing the new Ribbon to be installed as the file opens.

When you return to your database, check the dropDown menu. Next, try adding a new form. Click on the menu to launch the Authors form and then check the dropDown
list again. You will see that your control now also houses the new form as well, as shown in Figure 7-15.

This demonstrates that one-click delay just mentioned. The drop-down list needs to be refreshed after it is repopulated.

![Figure 7-15: Dynamic dropDown list to select a form in Access](image)

**Conclusion**

As you’ve seen, the **comboBox** and **dropDown** controls are extremely similar in implementation and appearance. The examples presented in this chapter covered using static items defined in the XML code, as well as leveraging VBA to provide dynamically updated controls. There is no question, however, that one of the biggest deterrents to using either of these methods to set up an effective, fully dynamic solution is the copious amount of code that would have to be manually generated.

When considering these controls, the main questions facing you are which one should you use, and when? The answer depends completely on how much latitude you wish to give your users. If, on the one hand, you need to limit users to selecting from a standard list that never changes, then you’ll want to use a **dropDown** control. On the other hand, if you want to offer your users either of the following options, then you will want to use a **comboBox**:

- The ability to jump to an item in the list
- The ability to enter text that is not already defined in the list

Now that you have learned how to present users with different lists of items, it is time to learn about two controls that can add impressive richness to your applications. In Chapter 8, you will learn how to incorporate the picture and gallery controls into your customizations.