CHAPTER 26

The Finishing Touches

You're in the home stretch. You have almost all the forms and reports required for the tasks you want to implement in your application, but you need some additional forms to make it easier to navigate to different tasks. To add a professional touch, you should design a custom ribbon for your main navigation forms, another custom ribbon for most data entry forms, and perhaps one for reports. You should take advantage of built-in tools to check the efficiency of your design, and you should make sure that none of your forms and reports allow Layout view if you're using a client database. Finally, you need to set the startup properties of your database to let Microsoft Access 2010 know how to get your application rolling, and you need to perform a final compile of your Microsoft Visual Basic code to achieve maximum performance.

Note

The ribbon examples in this chapter are based on the custom ribbons in Contacts.accdb, Housing.accdb, HousingSP.accdb, and BOSS.accdb on the companion CD included with this book. All screen images in this chapter were taken on a Windows 7 system with the Access color scheme set to Silver. Your results might look different if you are using a different operating system or a different theme.

Creating Custom Ribbons with XML

Throughout this book, you've seen how to use the ribbon commands on the four main tabs—Home, Create, External Data, and Database Tools—as well as on the many contextual tabs. You even might have noticed the custom ribbons we created in the Conrad Systems Contacts (Contacts.accdb), Housing Reservations (Housing.accdb), and Back Office Software System (BOSS.accdb) sample databases when you opened any of the forms and reports used by the application. In the following sections, you'll learn what steps are necessary to create a simple custom ribbon for a form. You'll see how to create Extensible Markup

Language (XML) for this ribbon, which displays existing groups from the four main ribbon tabs. You'll also create a new data entry form and assign your new ribbon to this form to test the new commands.

When your application is running, the user probably won't want or need some of the design features of Access 2010. However, you might want to provide some additional buttons on your form ribbon so that the user has direct access to commands such as Save Record and Refresh All. For example, open the Conrad Systems Contacts sample database (Contacts.accdb), open the frmContactsPlain form (which uses the built-in ribbon), and then open the frmContacts form. As you click each form window, Access changes the ribbon. You can see some useful differences between the two ribbons, as shown in Figure 26-1.

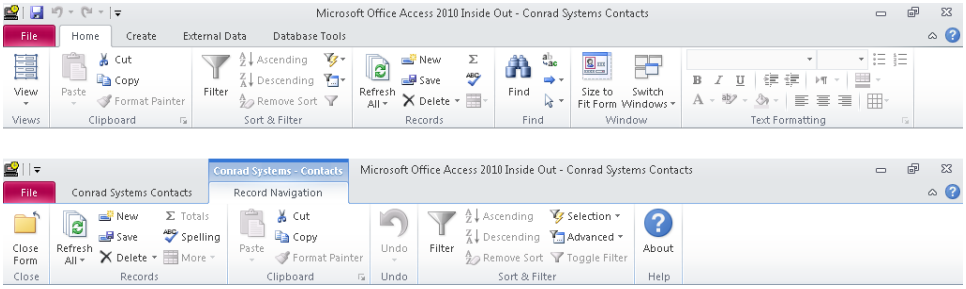


Figure 26-1 The standard ribbon (top) displays many commands and tabs your users won't need, compared to the custom form ribbon (bottom) from the Conrad Systems Contacts sample database.

Buttons, groups, and tabs the user won't need (such as the buttons in the Views and Windows groups) aren't available on the custom ribbon. Also, Access disables all the buttons on the Quick Access Toolbar for the frmContacts form, such as Undo, Save, and Quick Print. (None of the forms in the Conrad Systems Contacts database are designed to be printed.) However, the custom ribbon does have a Close Form button added at the left end of the Record Navigation tab, and we provided a custom Undo command because Undo is no longer available on the Quick Access Toolbar. In the Conrad Systems Contacts application, all forms (except frmContactsPlain and a few other example forms) have their Ribbon Name properties set to use the custom ribbon.

The same is true of the built-in tabs. For example, you don't want your users to be able to create new database objects using the buttons on the Create tab or to be able to use the tools available on the Database Tools tab. For most forms, you also don't want the user to be able to switch to PivotTable or PivotChart view using the View button. The following sections show you how to build a custom main ribbon and custom ribbons for forms and reports.

Creating a USysRibbons Table

When you open an Access 2010 database, Access looks for a local table called USysRibbons during the startup process to see whether it needs to load any custom ribbons. If Access does not find this table, it proceeds to load all built-in ribbons. You can load custom ribbons into your application by writing Visual Basic code to load the XML stored in a different table or defined within your code. We'll discuss how to do this later in this chapter.

Access does not create a local table called USysRibbons when you create a new blank database or use one of the database templates—you need to create this table yourself. For Access to use the USysRibbons table, the table must contain the two fields listed in Table 26-1. The RibbonName field is a unique name used to identify the name of the ribbon. The RibbonXML field contains the XML used to define the custom ribbon. The XML must be well formed for Access to interpret the code and apply it to the ribbon. Note that you can have additional fields in this table if you want (such as a field that documents what's in your custom ribbon), but Access looks for only the two fields listed in Table 26-1 when loading your ribbons.

Table 26-1 USysRibbons Table Fields

Field Name	Data Type
RibbonName	Text
RibbonXml	Memo

By default, Access does not display in the Navigation pane any local tables that start with the prefix *USys* because it considers these to be system tables. Depending upon what settings you have configured in the Navigation Options dialog box, you might not be able to see any system tables. For example, if you create and save a new table with the name USysRibbons, you might not see this new table in the Navigation pane.

In the Conrad Systems Contacts database, we have included this table to load the custom ribbons we use in the application. Open the Contacts.accdb database and click OK in the opening message box. Click the Navigation menu at the top of the Navigation pane, click Object Type under Navigate To Category, and then click Tables under Filter By Group to display a list of tables available in this database. If you scroll through the list of tables, you'll notice that you do not see a table called USysRibbons. To see this table in the Navigation pane, right-click the Navigation menu at the top of the Navigation pane and click Navigation Options. In the Display Options section in the Navigation Options dialog box, select the Show System Objects check box to display all system objects in the database, as shown in Figure 26-2.

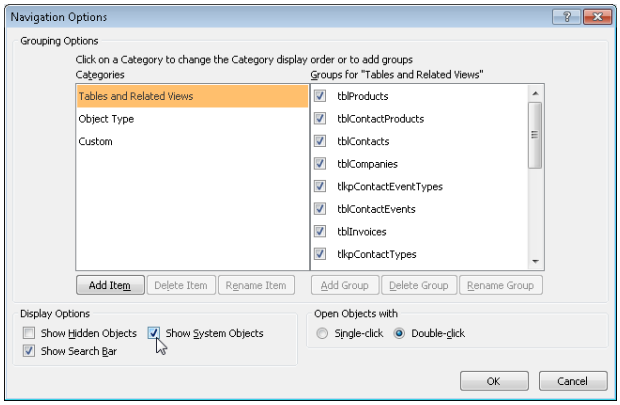


Figure 26-2 Select the Show System Objects check box to display the USysRibbons table.

Click OK to close the Navigation Options dialog box, and then review the list of tables in the Navigation pane. You'll notice that you can now see the USysRibbons table and six additional tables that start with the prefix *MSys*. Right-click the USysRibbons table in the Navigation pane and click Design View from the shortcut menu to see this table in Design view. In Figure 26-3, you can see the table has one additional field—ID—with a data type of AutoNumber. Remember that Access needs to have only the RibbonName and RibbonXml fields and will ignore any other fields. We added the ID field and used it as the primary key for the table to make sure our entries are unique.

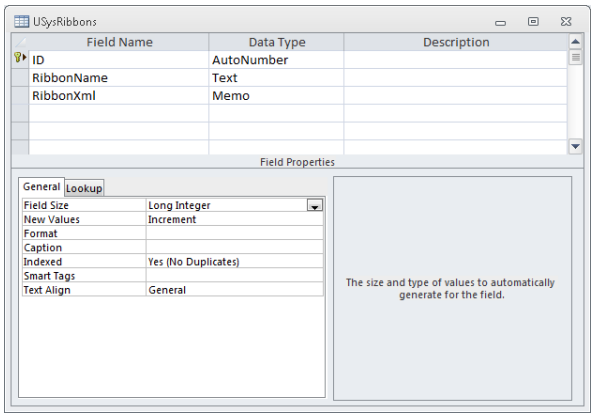


Figure 26-3 Access looks for a table called USysRibbons during startup to load custom ribbons.

CAUTION

Do not attempt to modify or delete the system tables with the *MSys* prefix. Access uses these tables internally to manage the various objects and other elements of your database.

Switch to Datasheet view by clicking the arrow in the Views group on the Design tab and clicking Datasheet View from the list of available views. You'll notice that there are three records in the USysRibbons table, as shown in Figure 26-4. Each of the records in this table denotes a specific custom ribbon. You can see the names of each of the ribbons in the RibbonName field—rbnForms, rbnPrintPreview, and rbnCSD. The rbnForms ribbon is used by most of the data entry forms in the Conrad Systems Contacts database, the rbnPrintPreview ribbon is used by the reports, and the rbnCSD ribbon displays for the frmMain, frmCodeLists, and frmReports forms.

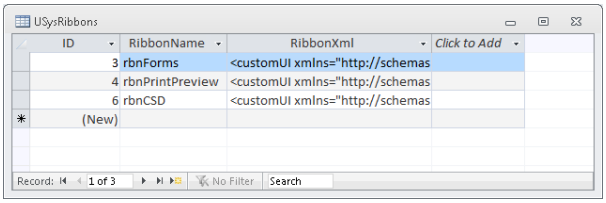


Figure 26-4 The Conrad Systems Contacts database includes three custom ribbons.

In the RibbonXml field, you can see the XML for each of these three custom ribbons. The well-formed XML in these fields, however, is not particularly easy to read in table Datasheet view. You can place your insertion point in the field and use the arrow keys to read the XML, you can press Shift+F2 to open the Zoom box, or you can expand the height of the individual rows. However, a much easier way to view and modify the XML is to create a form bound to this table. In the Conrad Systems Contacts database (and the Housing Reservations database as well), we created a form to add and edit the records in this table. Close the USysRibbons table, click the Navigation menu at the top of the Navigation pane, click Object Type under Navigate To Category, and then click Forms under Filter By Group to display a list of forms available in the this database. Find the form called zfrmChangeRibbonXML, and open it in Form view, as shown in Figure 26-5. If you use the record navigation buttons, you can see the XML for each of the three custom ribbons. Creating a form to work with your USysRibbons table is not a requirement, but you'll have an easier time viewing and modifying your XML by using a large text box on a form for the RibbonXml field.

Note

Now that you understand how to change your settings in the Navigation Options dialog box to view system objects like the USysRibbons table in the Navigation pane, we recommend that you change your settings back to *not* display system objects. To do this, right-click the Navigation menu at the top of the Navigation pane and click Navigation Options. In the Display Options section in the Navigation Options dialog box, clear the Show System Objects check box, and then click OK. You'll be using our zfrmChangeRibbonXML form in the remaining sections to work with the data in the USysRibbons table, so you don't need to see this table in the Navigation pane.

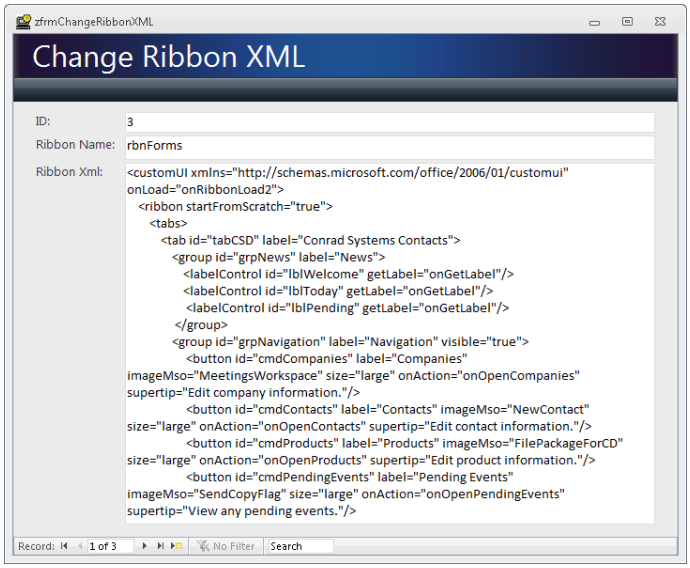


Figure 26-5 You'll have an easier time editing your XML for the USysRibbons table if you use a form.

Creating a Test Form

Most of the forms and reports in the Conrad Systems Contacts database already have a custom ribbon applied. In the section, we'll walk you through creating XML for a new form ribbon. Before we build the XML, let's first create a new data entry form based on the tblContacts table. We'll use this new form to test the XML without disturbing any of the existing database objects. First, close the zfrmChangeRibbonXML form if you still have it open. Next, click the Navigation menu at the top of the Navigation pane, click Object

Type under Navigate To Category, and then click Tables under Filter By Group. Finally, select the tblContacts table in the Navigation pane, and click the Form button in the Forms group on the Create tab. Access creates a new columnar form based on the table and opens it in Form view. Click the Save button on the Quick Access Toolbar, name the form **frmRibbonTest**, and then close the form.

Building the Ribbon XML

To create a custom ribbon for a form or report, you must first create the XML in a text editor such as Notepad, Notepad 2007, or in Visual Basic 2010 Express Edition. We used the Notepad text editor to create our XML. To create well-formed XML for ribbons, you need to use the Microsoft Office 2010 Reference: Office Fluent User Interface XML Schema Reference, which contains the schema information that Access 2010 needs to validate the ribbon customizations. You might also want to download the Office Fluent User Interface Control Identifiers, which contains a complete list of the ControlIDs of the built-in tabs, groups, buttons, and other commands. Each button, group, and tab is assigned a unique ControlID in the ribbon schema file. You can use these ControlIDs to place existing built-in ribbon elements onto your custom ribbons.

Note

If you use an XML editor such as Notepad 2007 or a tool such as Visual Basic 2010 Express Edition, you can use Microsoft IntelliSense to assist with constructing your XML. You can download the free XML Notepad 2007 editor from Microsoft at *http://www.microsoft.com/downloads/details.aspx?familyid=72D6AA49-787D-4118-BA5F-4F30FE913628&displaylang=en*.

You can download the Visual Basic 2010 Express Edition from Microsoft at *http://msdn.microsoft.com/vstudio/express/vb/*.

You can download the Office 2010 Reference: Office Fluent User Interface XML Schema from Microsoft at *http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=c2aa691a-8004-46ac-9852-102f1d5bcd18*.

You can download the Office 2010 Help Files: Office Fluent User Interface Control Identifiers from Microsoft at *http://www.microsoft.com/downloads/details.aspx?familyid=3F2FE784-610E-4BF1-8143-41E481993AC6&displaylang=en*.

Hiding Existing Ribbon Elements

Open Notepad (or an XML editor) to begin building your XML. We'll create a custom ribbon for our test form that includes two groups from the Home tab and hides all the built-in main tabs. As we proceed, you'll test each step to see the ribbon take shape. The XML for ribbons needs to start with the following line:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
```

The first line tells Access which schema file to use when building this specific ribbon. The next line should read as follows:

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

If you specify True, Access hides the four main tabs on the ribbon when your custom ribbon is loaded. Also, the Quick Access Toolbar shows no options except the arrow—you can select only the options to place the Quick Access Toolbar above or below the ribbon, or you can minimize the ribbon. If you want to have a more controlled interface and show only your custom ribbons to users, you should set this XML attribute to True. If you set startFromScratch to False, Access does not hide any of the main ribbon tabs. Any new tabs that you create appear to the right of the Database Tools tab.

After these first two lines, you can begin to build any tabs, groups, buttons, and other ribbon elements. For now, let's complete this simple XML example with some ending tags for ribbon and customUI. Your XML up to this point should look like the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="true">
  </ribbon>
</customUI>
```

Testing Your XML

As you build your XML, it's a good idea to test it along the way to ensure that everything is working properly. You'll have an easier time debugging any issues in your XML if you systematically test it after each major step. Highlight all the XML you've created so far and copy it to the Clipboard. Next, open the zfrmChangeRibbonXML form in the Conrad Systems Contacts database in Form view and navigate to a new record. In the Ribbon Name text box control, enter **rbnTest** for the name of this ribbon. Finally, use the Tab key to move to the Ribbon Xml text box control and paste in the XML content from the Clipboard. Your record in the form should look like Figure 26-6.

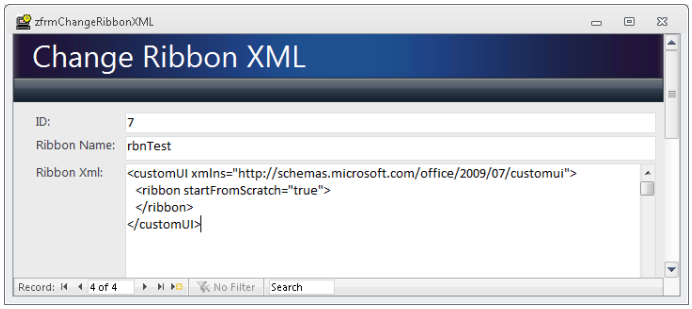


Figure 26-6 Create a new record in the USysRibbons table for your test ribbon by using the zfrmChangeRibbonXML form.

Close the zfrmChangeRibbonXML form to save your changes to the USysRibbons table. To display this ribbon for your test form to see how it looks, you need to assign the rbnTest ribbon to the Ribbon Name property of the form. Before you do this, however, you need to close the database and then reopen it. You'll remember we mentioned earlier that Access loads all the ribbons found in the USysRibbons table during the application startup process. Because you just added this new record to the table, Access has not loaded this ribbon into memory. If you opened the Property Sheet window for your test form at this point, you will not see rbnTest as an available option for the Ribbon Name property. (Note that you can type **rbnTest** in the property line, but you still won't see this ribbon displayed until you close and reopen the database.)

Close the database now and then reopen it to have Access load your new test ribbon into memory. After you reopen the database, open your test form—frmRibbonTest—in Design view. Click the Property Sheet button in the Tools group on the Design tab to open the property sheet for the form. On the Other or All tabs of the property sheet, click the arrow on the Ribbon Name property line, and then select rbnTest from the list of four ribbons. Click the Save button on the Quick Access Toolbar to save your changes and then switch to Form view to see the result.

When you switch views, you'll notice the entire ribbon disappears, as shown in Figure 26-7. Access also hides all the options on the Quick Access Toolbar except for the arrow to open the Customize Quick Access Toolbar menu. Click the File tab on the Backstage view and you'll notice that you still see all the options available. Because you specified StartFromScratch=True in your XML and did not specify any other custom tabs, Access presents a very limited user interface. Unless you want to provide features such as filtering and sorting directly on your forms, you clearly need to improve this custom ribbon beyond the bare essentials.

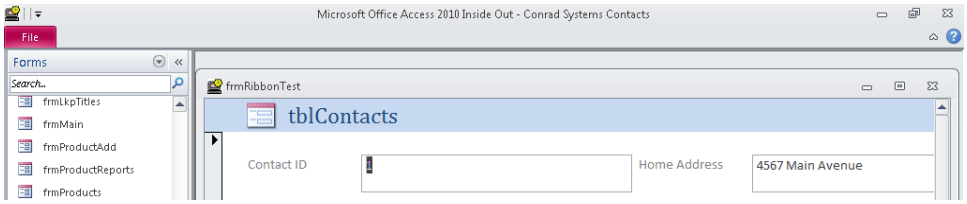
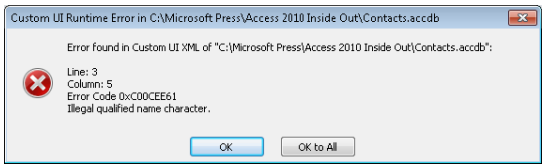


Figure 26-7 The simple XML you created earlier completely hides the ribbon.

INSIDE OUT Displaying Ribbon Errors

If you create XML for a ribbon that is not well formed, Access does not display your custom ribbon. In this situation, Access displays the built-in four main ribbon tabs because it cannot interpret and display the appropriate customization elements. Access is not forgiving in this case because even a single line of XML that is not well formed causes Access to revert to showing all the main tabs of the ribbon. You will have a difficult time debugging the XML and finding the cause of the problem because Access does not automatically display errors when it encounters errors in your XML.

Fortunately, Access includes an option that you can enable to display errors in these cases. Click the File tab on the Backstage view, click Options, click the Client Settings category, and then scroll down to the General area. If you select the Show Add-In User Interface Errors check box (cleared by default) and then click OK, Access displays a dialog box if it finds any problems in your XML. For example, if you select the Show Add-In User Interface Errors check box and add an extra < at the beginning of the third line of the XML you've been working with up to this point, Access displays the following error message when you open your test form (after you close and reopen the database):



You can see in the error message text that Access found a problem on line 3 of your XML. Selecting this option can help you debug your XML for ribbons.

Creating Tabs

Creating an interface with no ribbon showing, such as the one you just tested in the previous section, might work for some applications, but what if you want to provide your users with the same navigation and filter options for your forms that normally display on the Home tab? To add buttons and controls to your custom ribbon, you first need to create a tab to hold these controls. The Access ribbon schema uses a *tabs* tag to denote a new tab to display on the ribbon.

Open Notepad (or an XML editor), and return to the XML file you were creating earlier. To create a new tab, you first need to use a tabs tag (<tabs>) followed by a line of XML with the following syntax:

```
<tab id=UniqueTabName label=LabelCaption
```

The UniqueTabName must be a unique name for the current ribbon XML. The LabelCaption attribute is optional, but if you don't provide a caption for the tab, Access displays a small, empty tab header. At the end of any XML for the tab, you also need to provide an ending tab tag (</tab>) for each tab and an ending tabs tag (</tabs>) following all the individual tab tags. For this example, we'll create a new tab called tabTest with a caption called Navigation. Add the following XML between the two ribbon tags (<ribbon> and </ribbon>) that we created earlier to create this new tab:

```
    <tabs>
      <tab id="tabTest" label="Navigation">
      </tab>
    </tabs>
```

Your completed XML should now look like the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tabTest" label="Navigation">
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Now, let's test this markup with our test form and see how it looks. Highlight all the XML you've created so far and copy it to the Clipboard. Next, open the zfrmChangeRibbonXML form in the Conrad Systems Contacts database in Form view and navigate to the record that has rbnTest in the Ribbon Name text box. Tab to the Ribbon XML text box control, and paste the XML content from the Clipboard, overwriting the previous XML. Next, close the zfrmChangeRibbonXML form to save your changes to the USysRibbons table. Finally, close the database, reopen it, and then open the frmRibbonTest form in Form view. Access now displays the ribbon at full height with a new tab and caption of Navigation, as shown in Figure 26-8.

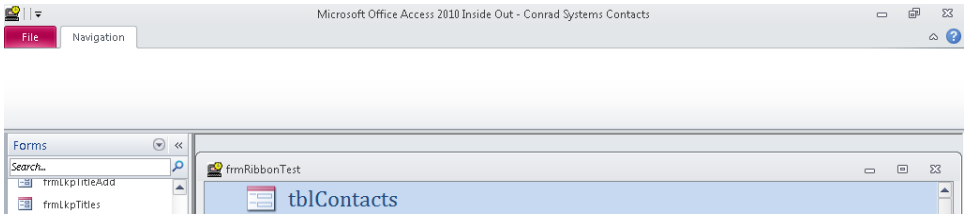


Figure 26-8 You can create custom tabs for your ribbons.

Adding Built-In Groups to Tabs

Now that you have a new tab created in your markup, you can begin the process of adding different controls to this tab. For this test form, it would help your users to navigate through the contact records if they could use some of the built-in buttons and commands found on the Home tab—specifically, the Records, Sort & Filter, and Find groups. You could write your own XML to create custom buttons that mimic the actions of each of the individual buttons in those three groups, but to make things easier, you can use the RibbonX architecture to copy buttons, commands, and options found in a built-in group onto one of your custom tabs. (We'll show you how to create individual custom buttons on tabs in "Using Ribbon Attributes," on page 1682.)

Open Notepad (or an XML editor), and return to the XML file you were creating earlier. To use an existing built-in group on a custom tab, use the following syntax:

```
<group idMso=ControlID>
```

You can add this tag anywhere within a tab definition (delimited with <tab> </tab> tags). You can use the idMso attribute to denote a built-in control ID—a built-in control defined in the RibbonX architecture. Every button, group, and tab on the built-in ribbon has an internal control ID that you can reference. In this case, you're going to show how to use the idMso attribute to refer to a specific group that you want to use on your tab. To help identify the names of these groups, you can use the Office 2010 User Interface Control Identifiers Excel spreadsheet, which contains a complete list of these internal control IDs. For this example, you want to add the Records group on the Home tab to your custom navigation tab to help your users add, save, and delete records. The control ID for the Records group is GroupRecords. At the end of any XML for a new group, you also need to provide an ending group tag. Add the following XML between the tab tags to create this new group:

```
<group idMso="GroupRecords">
</group>
```

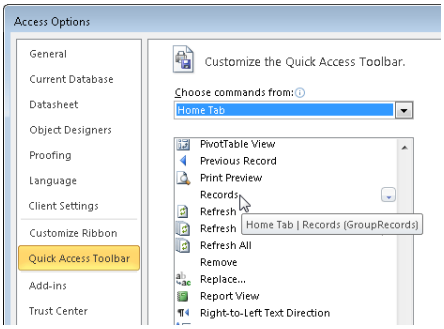
Your completed XML up to this point should now look like the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tabTest" label="Navigation">
        <group idMso="GroupRecords">
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

INSIDE OUT

Finding Control IDs for Built-In Controls

You can also find the control IDs for built-in controls listed in the Customize category in the Access Options dialog box. Click the File tab on the Backstage view, click Options, and then select either the Customize Ribbon or Quick Access Toolbar category. In the command list on the left, you can see the built-in Access commands. If you rest your mouse pointer on one of these commands, Access displays a tooltip that lists the internal control ID. To help you identify control groups, Access displays an icon with a down arrow next to any group names. The internal control ID for the Records group on the Home tab—GroupRecords—is within the parentheses on the tooltip, as shown here:



Now, let's test this XML with our test form to see how it looks. As you did previously, highlight all the XML you've created so far and copy it to the Clipboard. Next, open the zfrmChangeRibbonXML form in the Conrad Systems Contacts database in Form view and navigate to the record that has rbnTest in the Ribbon Name text box. Use Tab to move to the Ribbon XML text box control, and paste in the XML content from the Clipboard,

overwriting the existing XML. Next, close the zfrmChangeRibbonXML form to save your changes to the USysRibbons table. Finally, close the database, reopen it, and then open the frmRibbonTest form in Form view. Access now displays all the buttons and commands from the Records group on your custom Navigation tab, as shown in Figure 26-9. If you test some of the buttons, you'll see that they all work just as they do on the Home tab.

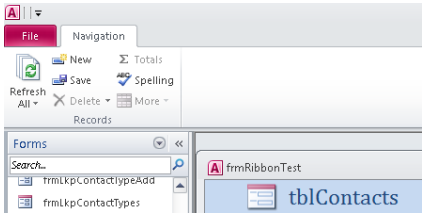


Figure 26-9 The built-in Records group now appears on your custom ribbon.

Now that you've added the Records group to your tab, let's finish the XML by adding the Sort & Filter and Find groups to your custom ribbon. Open Notepad (or an XML editor), and return to the XML file you were creating earlier. The control ID for the Sort & Filter group is GroupSortAndFilter, and the control ID for the Find group is GroupFindAccess. Add the following XML after the group end tag and before the end tab tag:

```
        <group idMso="GroupSortAndFilter">
        </group>
        <group idMso="GroupFindAccess">
        </group>
```

Your final XML should now look like the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tabTest" label="Navigation">
        <group idMso="GroupRecords">
        </group>
        <group idMso="GroupSortAndFilter">
        </group>
        <group idMso="GroupFindAccess">
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Now, test your completed XML on your test form. As you did previously, highlight all the XML you've created so far, and copy it to the Clipboard. Next, open the zfrmChangeRibbonXML form in the Conrad Systems Contacts database in Form view, and paste in the XML content from the Clipboard into the Ribbon XML text box on the rbnTest record, overwriting the existing XML. Close the zfrmChangeRibbonXML form to save your changes to the USysRibbons table. Close the database and reopen it to have Access load the new ribbon changes. Finally, open the frmRibbonTest form in Form view to see the results. Access now displays all the buttons and commands from the Records, Sort & Filter, and Find groups on your custom Navigation tab, as shown in Figure 26-10. With only a few lines of XML, you've created a custom ribbon that you can use in your application. You can assign this ribbon to any of the forms in your application. Also, because the XML is stored in a local table, you can easily import this table into another database and reuse the ribbon for those forms.

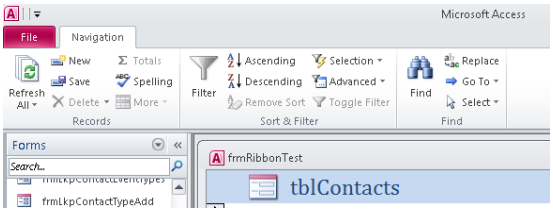


Figure 26-10 The custom ribbon you created now includes buttons and commands from three built-in groups.

Loading Ribbon XML

In the previous sections, you've learned the basic structure of ribbon XML and how to create a USysRibbons table to store the XML for each of your custom ribbons. You learned that Access searches for this table during startup and that if it finds this table (and correct fields within the table), Access loads these custom ribbons into memory. In the Conrad Systems Contacts (Contacts.accdb) sample database, we created three custom ribbons for the application in the USysRibbons table—rbnCSD, rbnForms, and rbnPrintPreview.

After you define custom ribbons in the USysRibbons table, you can specify that Access load a specific custom ribbon each time you open the database. To accomplish this, click the File tab on the Backstage view, click Options, and then click the Current Database category. In the Ribbon And Toolbar Options section, click the arrow on the Ribbon Name option, and then select your custom ribbon from the list of loaded ribbons, as shown in Figure 26-11. (Note that you might need to close and reopen the database to see any new ribbon you just created appear in the list.) Click OK to save your changes, and close the Access Options dialog box. The next time you open your database, Access applies that custom ribbon.

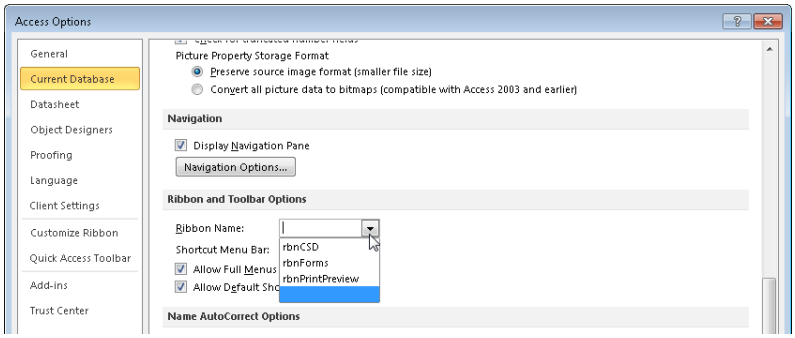


Figure 26-11 In the Current Database category in the Access Options dialog box, you can select a specific custom ribbon to load each time you open the database.

> **Note**
>
> **To prevent Access from automatically loading any custom ribbons during the startup procedure, press and hold the Shift key when you open the database.**

When you create a USysRibbons table, Access takes care of the work of loading your custom ribbons. You can also load custom ribbons into your application by using the LoadCustomUI method. When you dynamically load your ribbon customization using the LoadCustomUI method, you can store your XML in a table with a different name, in a different database, or in a Visual Basic module.

Syntax

```
Application.LoadCustomUI(CustomUIName, CustomUIXML)
```

Notes

CustomUIName is a string variable or literal containing the unique name of the custom ribbon to be associated with this XML, and *CustomUIXML* is a string variable or literal that contains the well-formed XML that defines your custom ribbon.

If you want to dynamically load custom ribbons, you need to call the LoadCustomUI method each time you open the database. In the HousingSP.accdb database, we use this method to load similar custom ribbons that you see in the Housing Reservations database (Housing.accdb). Close the Contacts.accdb file if you still have it open, and then open the HousingSP.accdb file. After you database opens, click OK on the opening message box. Next, click the Navigation menu at the top of the Navigation pane, click Object Type under

Navigate To Category, and then click Tables under Filter By Group to display a list of tables available in this database. If you open the zTblRibbons table in Datasheet view, you'll notice we have two of the same custom ribbons in this table—rbnForms and rbnProseWare—as we have in the USysRibbons table in the Housing.accdb sample database.

INSIDE OUT

Loading Ribbons into Access Data Projects

Because all tables for an Access project file (.adp) are stored in Microsoft SQL Server, you cannot define a local USysRibbons table to have Access automatically load your custom ribbons. Defining a USysRibbons table in SQL Server doesn't work because those tables don't become available until after Access has completed the initialization of the project file.

Close the zTblRibbons table, and let's examine the Visual Basic code we use to load these custom ribbons. Click the Navigation menu at the top of the Navigation pane, click Object Type under Navigate To Category, and then click Modules under Filter By Group to display a list of Visual Basic modules available in this database. Right-click modRibbonCallbacks in the Navigation pane, and click Design View from the shortcut menu to open the Visual Basic Editor. The first function in this module—LoadRibbons—and the Declarations above it use the LoadCustomUI method as follows:

```
' This serves as a cached copy of the RibbonUI.
' We can use this to then invalidate the Ribbon and refresh the controls
Public gobjRibbon As IRibbonUI

Public Function LoadRibbons() As Integer
' Code called by frmCopyright to verify custom Ribbon load
Dim db As DAO.Database, rst As DAO.Recordset
    ' Set an error trap
    On Error GoTo LoadRibbon_Err
    ' Do nothing if gobjRibbon is set
    If Not (gobjRibbon Is Nothing) Then
        ' Set OK return
        LoadRibbons = True
    Else
        ' Try to load - open recordset on Ribbons
        Set db = CurrentDb
        Set rst = db.OpenRecordset("ztblRibbons", dbOpenDynaset)
        Do Until rst.EOF
            Application.LoadCustomUI rst!RibbonName, rst!RibbonXML
            rst.MoveNext
        Loop
        ' Close out
        rst.Close
        Set rst = Nothing
```

```
        Set db = Nothing
        LoadRibbons = True
    End If
LoadRibbon_Exit:
    Exit Function
LoadRibbon_Err:
    ' Silently log error
    ErrorLog "LoadRibbons", Err, Error
    LoadRibbons = False
    Resume LoadRibbon_Exit
End Function
```

The first line of code in the LoadRibbons function after the Dim statement instructs Access to go to the LoadRibbon_Err line if any errors occur. The If statement checks to see whether Access has already loaded customization by verifying whether a cached copy of the RibbonUI has been set. If the main custom ribbon has been loaded, Is Nothing returns a value of False to indicate Access already has loaded the ribbon. If the custom ribbon isn't loaded, the code following the Else line executes. To load the custom ribbons, the code opens a Data Access Objects (DAO) recordset on the zTblRibbons table, loops through each record in the recordset, and calls the LoadCustomUI method once for each record. The code uses the RibbonName field from the recordset to pass the name to the LoadCustomUI method, and the RibbonXml field contains the well-formed XML that defines each of our custom ribbons. After Access loads each ribbon, the code closes the recordset and sets it to Nothing. The last line before the End If statement returns a value of True for the LoadRibbons function, indicating success. The last part of the code has an exit procedure and our error handling code to handle the case if Access encounters an error.

As you can see, you can use the LoadCustomUI method to load a custom ribbon from a different table, but you're not limited to storing the XML in a table. You could also store the XML directly within a code module and set it to a string variable.

Using Ribbon Attributes

The RibbonX architecture contains many controls and attributes you can use in your applications. To help you understand some of the elements you can create with RibbonX, we'll look at the custom ribbons we created in the Conrad Systems Contacts sample database. Close the HousingSP.accdb database, open the Contacts.accdb sample database, and click OK on the opening message box. Next, open the frmSplash form in Form view either by double-clicking the form in the Navigation pane or by right-clicking it and clicking Open on the shortcut menu. The frmSplash form displays for a few seconds and then opens the frmSignOn form, where you can sign into the database. Select Jeff's name from the User Name combo box control, and click Sign On to sign into the database. (Neither of the users has a password assigned.) Access opens the frmMain form and displays the custom main ribbon for this database, as shown in Figure 26-12. The main ribbon in this database,

rbnCSD, has a tab called Conrad Systems Contacts and four groups—News, Navigation, Exit, and About. The News group displays three labels, one of which displays the name of the current user signed into the database. (You'll learn how to dynamically change the ribbon later in this chapter.) The remaining three groups display custom buttons that allow you to navigate to the various parts of the application.

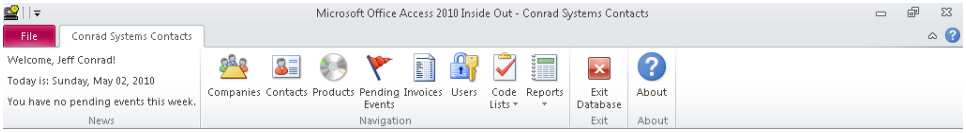


Figure 26-12 The main ribbon in the Conrad Systems Contacts database displays custom controls.

You can see in Figure 26-12 that all built-in ribbon elements are hidden. As you recall from earlier in this chapter, if you set the StartFromScratch attribute to True, Access hides all four built-in ribbon tabs and displays a limited Quick Access Toolbar.

When you build a custom ribbon definition, you can either define an entire custom ribbon or define XML that modifies one of the built-in ribbons. In your XML, you define custom controls. You assign *attributes* to your controls to define how they look and where they are positioned, and you define *callbacks* for your controls to define how they act. In Visual Basic terms, you can think of an attribute as a property of a control object and a callback as a method or event of a control object. When an attribute of a control is a callback, it is essentially the same as an event property of an Access control—you assign to a callback attribute the name of a procedure that will handle the event. You can also copy attributes of built-in controls by using an attribute name that is a Control ID, and you can reference a control in another custom ribbon using a Qualified ID.

Let's look at attributes you can assign to controls first. Table 26-2 lists the RibbonX attributes you can use in your XML customization.

Table 26-2 RibbonX Attributes

Attribute	Value or Type	Description
autoScale	True or False	Determines whether a group should scale its contents automatically.
centerVertically	True or False	Determines whether the content inside a group should be centered vertically.
description	String	Defines the text shown in menu controls when itemSize="large".
enabled	True or False	Returns the control's enabled state. Disabled controls (enabled = False) appear dimmed in the ribbon.

Attribute	Value or Type	Description
getDescription	Callback	Names the macro or procedure that can set the description attribute of a control.
getEnabled	Callback	Defines the macro or procedure that can set the enabled state of a control.
getImage	Callback	Names the macro or procedure that can set the image attribute of a control.
getLabel	Callback	Defines the macro or procedure that can set the label attribute of a control.
getPressed	Callback	Names the procedure that can respond to the current state of a toggle button or check box.
getSupertip	Callback	Names the procedure that can set the Enhanced ScreenTip of a control.
getTooltip	Callback	Names the procedure that can set the tooltip of a control.
getVisible	Callback	Names the procedure that can set the visibility state of a control.
id	String	Provides the Unique ID for a user-defined control.
idMso	Control ID	Provides the Control ID for a built-in ribbon element.
idQ	Qualified ID	Provides the qualified name of a control on another ribbon.
image	String	Defines the image displayed on the control.
imageMso	Control ID	Provides the icon of a built-in control.
insertAfterMso	Control ID	Positions a custom control after a built-in control.
insertAfterQ	Qualified ID	Positions a custom control after a control defined by another ribbon using a qualified name.
insertBeforeMso	Control ID	Positions a custom control before a built-in control.
insertBeforeQ	Qualified ID	Positions a custom control before a control defined by another ribbon using a qualified name.
label	String	Returns a control's label text.
onAction	Callback	Defines the macro or procedure called when a user clicks this control.
pressed	True or False	Returns the state of toggle button or check box control.
showInRibbon	False or 0	Note that this appears in the schema, but cannot be used in RibbonX.
showLabel	True or False	Specifies whether or not to show the label on a control. Specifies a callback which returns whether to show the label.
size	Normal or Large	Sets the image size of custom control.

Attribute	Value or Type	Description
supertip	String	Defines the text to display an enhanced ScreenTip when the user rests their mouse pointer on the control.
title	String	Provides the title for a menuSeparator.
tooltip	String	Defines the text to display as a tooltip.
visible	True or False	Returns the visible status of a control.

In addition to attributes you can use in your XML customization, you can create many types of controls using RibbonX. Table 26-3 lists the types of controls you can use in a custom ribbon definition and their associated attributes (properties) and callbacks (event properties).

Table 26-3 RibbonX Controls

Control Name	Attributes	Callbacks
button	description, enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, screentip, showImage, showLabel, size, supertip, tag, and visible	getDescription, getEnabled, getImage, getKeytip, getLabel, getScreentip, getShowImage, getShowLabel, getSize, getSupertip, getVisible, and onAction
buttonGroup	id, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, tag, and visible	getVisible
checkBox	description, enabled, id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, screentip, supertip, tag, and visible	getDescription, getEnabled, getKeytip, getLabel, getPressed, getScreentip, getSupertip, getVisible, and onAction
comboBox	enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, label, invalidateContentOnDrop, maxLength, screentip, showItemImage, showImage, showLabel, sizeString, supertip, tag, and visible	getEnabled, getImage, getItemCount, getItemID, getItemImage, getItemLabel, getItemScreentip, getItemSupertip, getKeytip, getLabel, getScreentip, getShowImage, getShowLabel, getSize, getSupertip, getText, getVisible, and onChange
dialogBoxLauncher	None Note that you must create a button inside a dialogBoxLauncher tag and then use the attributes of that button to specify attributes for the dialogBoxLauncher.	None Note that you must create a button inside a dailogBoxLauncher tag and then use the callbacks of that button to specify events for the dialogBoxLauncher.

Control Name	Attributes	Callbacks
dropDown	enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, screentip, showImage, showItemLabel, showLabel, supertip, tag, and visible	getEnabled, getImage, getItemCount, getItemID, getItemImage, getItemLabel, getItemScreentip, getItemSupertip, getKeytip, getLabel, getScreentip, getSelectedItemID, getSelectedItemIndex, getShowImage, getShowLabel, getSize, getSupertip, getText, getVisible, and onChange
dynamicMenu	description, enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, invalidateContentOnDrop, keytip, label, screentip, showImage, showLabel, size, supertip, tag, and visible	getDescription, getEnabled, getContent, getImage, getKeytip, getLabel, getScreentip, getShowImage, getShowLabel, getSize, getSupertip, and getVisible
editBox	enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, maxLength screentip, showImage, showLabel, sizeString, supertip, tag, and visible	getEnabled, getImage, getKeytip, getLabel, getScreentip, getShowImage, getShowLabel, getSupertip, getText, getVisible, and onChange
gallery	columns, description, enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, itemHeight, itemWidth, keytip, label, rows, screentip, showImage, showInRibbon, showItemImage, showItemLabel, showLabel, size, sizeString, supertip, tag, and visible	getDescription, getEnabled, getImage, getItemCount, getItemHeight, getItemID, getItemImage, getItemLabel, getItemScreentip, getItemSupertip, getItemWidth, getKeytip, getLabel, getScreentip, getSelectedItemID, getSelectedItemIndex, getShowImage, getShowLabel, getSize, getSupertip, getText, getVisible, and onAction
group	autoScale, centerVertically, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, screentip, supertip, tag, and visible	getImage, getKeytip, getLabel, getScreentip, getSupertip, and getVisible
labelControl	enabled, id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, label, screentip, showLabel, supertip, tag, and visible	getEnabled, getLabel, getScreentip, getShowLabel, getSupertip, and getVisible

Control Name	Attributes	Callbacks
menu	description, enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, itemSize, keytip, label, screentip, showImage, showLabel, size, supertip, tag, and visible	getDescription, getEnabled, getImage, getKeytip, getLabel, getScreentip, getShowImage, getShowLabel, getSize, getSupertip, and getVisible
menuSeparator	id, idQ, insertAfterMso, insertAfterQ, insertBeforeQ, tag, and title	getTitle
separator	id, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, tag, and visible	getVisible
splitButton	enabled, id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, screentip, showLabel, size, supertip, tag, and visible	getEnabled, getKeytip, getShowLabel, getSize, and getVisible
tab	id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, tag, and visible	getKeytip, getLabel, and getVisible
toggleButton	description, enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keytip, label, screentip, showImage, showLabel, size, supertip, tag, and visible	getDescription, getEnabled, getImage, getKeytip, getLabel, getPressed, getScreentip, getShowImage, getShowLabel, getSize, getSupertip, getVisible, and onAction

Now that you know the attributes and callbacks available in the RibbonX architecture, you can begin to study how we created the main ribbon in the Conrad Systems Contacts database. If you still have the frmMain form open, click Exit to close it, and then click Yes to confirm that you want to exit. Next, find the zfrmChangeRibbonXML form in the Navigation pane and open it in Form view. Finally, move to the third record in the table where the Ribbon Name text box control displays rbnCSD. Listed next is the XML customization for this ribbon in the Ribbon XML text box control. We've added line numbers to this code listing so that you can follow along with the line-by-line explanations in Table 26-4, which follows the listing.

```
1 <customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"
2  onLoad="onRibbonLoad1">
3  <ribbon startFromScratch="true">
4    <tabs>
5       <tab id="tabCSD" label="Conrad Systems Contacts">
6          <group id="grpNews" label="News">
7             <labelControl id="lblWelcome" getLabel="onGetLabel"/>
8             <labelControl id="lblToday" getLabel="onGetLabel"/>
```

```
9             <labelControl id="lblPending" getLabel="onGetLabel"/>
10          </group>
11          <group id="grpNavigation" label="Navigation" visible="true">
12              <button id="cmdCompanies" label="Companies"
13                imageMso="MeetingsWorkspace" size="large"
14                onAction="onOpenCompanies"
15                supertip="Edit company information."/>
16              <button id="cmdContacts" label="Contacts" imageMso="NewContact"
17                size="large" onAction="onOpenContacts"
18                supertip="Edit contact information."/>
19              <button id="cmdProducts" label="Products"
20                imageMso="FilePackageForCD" size="large"
21                onAction="onOpenProducts"
22                supertip="Edit product information."/>
23              <button id="cmdPendingEvents" label="Pending Events"
24                imageMso="SendCopyFlag" size="large"
25                onAction="onOpenPendingEvents"
26                supertip="View any pending events."/>
27               <button id="cmdInvoices" label="Invoices"
28                 imageMso="CustomTableOfContentsGallery" size="large"
29                 onAction="onOpenInvoices"
30                 supertip="Edit invoice information."/>
31               <button id="cmdUsers" label="Users"
32                 imageMso="FileDocumentEncrypt"
33                 size="large" onAction="onOpenUsers"
34                 supertip="Edit user information."/>
35           <splitButton id="sbCodeList" size="large">
36             <button id="cmdCodeLists" imageMso="NewTask"
37               onAction="onOpenCodeLists" label="Code Lists"/>
38             <menu id="sbMnuCodeLists">
39                <button id="cmdContactTypes" label="Contact Types"
40                   imageMso="NewTask" onAction="onOpenContactTypes"/>
41                <button id="cmdEventTypes" label="Event Types"
42                   imageMso="NewTask" onAction="onOpenEventTypes"/>
43                <button id="cmdProductCategories" label="Product Categories"
44                   imageMso="NewTask" onAction="onOpenProductCategories"/>
45                <button id="cmdPersonTitles" label="Person Titles"
46                   imageMso="NewTask" onAction="onOpenPersonTitles"/>
47                <button id="cmdPersonSuffixes" label="Person Suffixes"
48                   imageMso="NewTask" onAction="onOpenPersonSuffixes"/>
49             </menu>
50           </splitButton>
51           <splitButton id="sbReports" size="large">
52             <button id="cmdReports" imageMso="CreateReport"
53               onAction="onOpenReports" label="Reports"/>
54             <menu id="sbMnuReports" supertip="View reports.">
55               <button id="cmdCompanyReports" label="Company Reports"
56                 imageMso="CreateReport" onAction="onOpenCompanyReports"/>
57               <button id="cmdContactReports" label="Contact Reports"
58                 imageMso="CreateReport" onAction="onOpenContactReports"/>
59               <button id="cmdProductReports" label="Product Reports"
60                 imageMso="CreateReport" onAction="onOpenProductReports"/>
```

```
61              </menu>
62            </splitButton>
63          </group>
64          <group id="grpExit" label="Exit">
65            <button id="cmdExitDatabase" label="Exit Database"
66              imageMso="PrintPreviewClose" size="large"
67              onAction="onCloseDatabase" supertip="Exit the database."/>
68          </group>
69          <group id="grpAbout" label="About">
70            <button id="cmdHelpAbout" label="About" size="large"
71              imageMso="Help" onAction="onOpenFormEdit" tag="frmAbout"
72              supertip="View the About form."/>
73          </group>
74        </tab>
75      </tabs>
76    </ribbon>
77  </customUI>
```

Table 26-4 Explanation of the XML in Ribbon rbnCSD

Line(s)	Explanation
1	Tells Access which schema file to use when building this specific ribbon.
2	Specifies a procedure that processes the RibbonLoad event when Access first displays the ribbon. In this event, you can save a pointer to the ribbon to enable your code to dynamically update it. (We'll explain how to update the ribbon later in this chapter.)
3	Hides all built-in ribbon elements.
4	Specifies the beginning tag to create a new set of tabs.
5	Creates a new tab with a Control ID called tabCSD and displays Conrad Systems Contacts in the tab caption.
6	Creates a new group with a Control ID called grpNews and displays News as the group label.
7	Creates a new label control, lblWelcome, and specifies the onGetLabel procedure to respond to the getLabel event to dynamically update the text displayed in the label.
8	Creates a new label control, lblToday, that also calls onGetLabel.
9	Creates a new label control, lblPending, that also calls onGetLabel.
10	Ends the group tag for the grpNews group.
11	Creates a new group with a Control ID called grpNavigation and displays Navigation as the group label.
12–15	Creates a new button, cmdCompanies, with a label of Companies. Instead of specifying an image attribute, we used imageMSO to copy the image from the built-in control named FilePackageForCD. The button size is set to large, and the onAction attribute issues a callback to the opOpenProducts procedure. Finally, we designate text to display as a supertip.

Line(s)	Explanation
16–18	Creates a new button, cmdContacts, with a label of Contacts, an image copied from a built-in control, and a defined callback.
19–22	Creates a new button, cmdProducts, with a label of Products, an image copied from a built-in control, and a defined callback.
23–26	Creates a new button, cmdPendingEvents, with a label of Pending Events, an image copied from a built-in control, and a defined callback.
27–30	Creates a new button, cmdInvoices, with a label of Invoices, an image copied from a built-in control, and a defined callback.
31–34	Creates a new button, cmdUsers, with a label of Users, an image copied from a built-in control, and a defined callback.
35	Creates a new split button, sbCodeList, with a large size.
36–37	Creates a new button, cmdCodeLists, with a label of Code Lists, an image copied from a built-in control, and a defined callback. This button becomes the top half of the split button. If you click the top half of the button, Access calls the onAction procedure for this button.
38	Creates a new menu control, sbMnuCodeLists, for the bottom half of the split button.
39–40	Creates a new button, cmdContactTypes, with a label of Contact Types, an image copied from a built-in control, and a defined callback.
41–42	Creates a new button, cmdEventTypes, with a label of Event Types, an image copied from a built-in control, and a defined callback.
43–44	Creates a new button, cmdProductCategories, with a label of Product Categories, an image copied from a built-in control, and a defined callback.
45–46	Creates a new button, cmdPersonTitles, with a label of Person Titles, an image copied from a built-in control, and a defined callback.
47–48	Creates a new button, cmdPersonSuffixes, with a label of Person Suffixes, an image copied from a built-in control, and a defined callback.
49	Ends the menu tag for menu sbMnuCodeLists.
50	Ends the split button tag for sbCodeList.
51	Creates a new split button, sbReports, with a large size.
52–53	Creates a new button, cmdReports, with a label of Reports, an image copied from a built-in control, and a callback defined. This button becomes the top half of the split button. If you click the top half of the button, Access calls the onAction procedure for this button.
54	Creates a new menu control, sbMnuReports, for the bottom half of the split button with a supertip.
55–56	Creates a new button, cmdCompanyReports, with a label of Company Reports, an image copied from a built-in control, and a defined callback.
57–58	Creates a new button, cmdContactReports, with a label of Contact Reports, an image copied from a built-in control, and a defined callback.

Line(s)	Explanation
59–60	Creates a new button, cmdProductReports, with a label of Product Reports, an image copied from a built-in control, and a defined callback.
61	Defines the ending menu tag for menu sbMnuReports.
62	Defines the ending split button tag for sbReports.
63	Defines the ending group tag for the grpNavigation group.
64	Creates a new group with a Control ID called grpExit and displays Exit for the group label.
65–67	Creates a new button, cmdExitDatabase, with a label of Exit Database, an image copied from a built-in control, and a callback defined.
68	Ends the group tag for the grpExit group.
69	Creates a new group with a Control ID called grpAbout and displays About for the group label.
70–72	Creates a new button, cmdHelpAbout, with a label of About, an image copied from a built-in control, and a defined callback. A custom tag value is assigned to this control using the tag attribute.
73	Defines the ending group tag for the grpAbout group.
74	Defines the ending tag for the tabCSD tab.
75	Defines the ending tabs tag.
76	Defines the ending ribbon tag.
77	Defines the ending customUI tag.

Creating VBA Callbacks

As you reviewed the XML customization for the rbnCSD ribbon, you no doubt noticed that most of the buttons used the onAction callback. When you use onAction, you specify a saved macro object (you cannot use an embedded macro on a form or report in this case) or a Microsoft Visual Basic for Applications (VBA) procedure to respond to the event—namely, the user clicking the button. (You can think of onAction for a button in a ribbon as the same as On Click for a command button on an Access form.) In the Conrad Systems Contacts sample database (Contacts.accdb), we defined all the procedures to respond to callback events for our custom ribbons in the module modRibbonCallbacks. Let's look at some of these VBA procedures so that you can see how everything ties together. Close any objects you have open at the moment, right-click modRibbonCallbacks in the Navigation pane (you might need to adjust your Navigation pane display to see the modules), and click Design View on the shortcut menu to open this module in Design view.

Scroll down the procedures and functions in this module until you come to the onOpen-Companies procedure. This procedure responds to the On Click event of the very first button defined in our custom ribbon—the cmdCompanies button. The procedure is as follows:

```
Public Sub onOpenCompanies(control As IRibbonControl)
' User wants to open Companies form
    ' Make sure frmMain is there
    If IsFormLoaded("frmMain") Then
        ' Yes - execute the Companies procedure
        Form_frmMain.cmdCompanies_Click
    End If
End Sub
```

The idea is to duplicate what happens when the user clicks the Companies button in the main switchboard form. That form (frmMain) already has code to process the request, so why duplicate it? If you remember from Chapter 24, "Understanding Visual Basic Fundamentals," on the companion CD, you can make any procedure in a form's class module a method of the form by declaring it Public. We did exactly that with the cmdCompanies_Click procedure in the frmMain form so that this onAction procedure can call it and not duplicate the code to open companies. The procedure first verifies that frmMain form is open because a call to the public method will fail if the form is closed. If the frmMain form is open, the procedure calls the cmdExit_Click procedure as a method of the form. If you scroll through the other procedures in the modRibbonCallbacks modules, you'll find that most of them follow this same pattern—they run an existing command button Click event procedure on the frmMain form.

Dynamically Updating Ribbon Elements

When Access first opens a custom ribbon, it issues callbacks for all the controls to set their attributes. Note that, other than for the Ribbon Name defined in the Current Database category of Access Options, Access opens a custom ribbon when it is first referenced in the Ribbon Name property of a form or report that you open or that your application code opens. Executing the LoadCustomUI method of the Application object loads the ribbon definition into memory, but it does not actually open the ribbon.

After a ribbon is loaded, the ribbon is static, and Access does not update any elements. If you want to update the ribbon, such as to change the text in a label control to display the current user's name, you must explicitly tell Access to reinitialize the entire ribbon or a control on the ribbon. Fortunately, the RibbonX architecture allows you to save a copy of a pointer to the ribbon in a public variable so you that can update it at a later time. In RibbonX terms, you *invalidate* a control (or the entire ribbon) to force Access to reload the object and issue any attribute setting callbacks. At the beginning of the XML customization for rbnCSD, you'll recall seeing this line of code:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"
  onLoad="onRibbonLoad1">
```

The first procedure in the modRibbonCallbacks module is onRibbonLoad1. In the procedure called by onRibbonLoad1, you can save a pointer to the ribbon that is being opened. If you do not plan to ever update any of your ribbon elements, you do not need to add the onRibbonLoad1 attribute to the <customUI> element. Access calls the onLoad callback only once during the process of opening the custom ribbon for the first time. The following is the onRibbonLoad1 procedure that executes in the onLoad callback of the rbnCSD custom ribbon:

```
' This is the customUI onLoad event handler for main ribbon
Public Sub onRibbonLoad1(ribbon As IRibbonUI)
    ' Cache a copy of the Ribbon so we can refresh later at any time
    Set gobjRibbon1 = ribbon
End Sub
```

The IRibbonUI object is a parameter that you can use to save a pointer to the opened custom ribbon. The IRibbonUI class provides methods that you can use to invalidate a single control in your customization or the entire ribbon. Table 26-5 lists these methods.

Table 26-5 IRibbonUI Methods

Method	Description
Invalidate()	Access reinitializes all custom controls.
InvalidateControl(string controlID)	Access reinitializes one specific control.

After you signed into the Conrad Systems Contacts database, you'll recall seeing the user name displayed in the News group on our custom ribbon. To update the display of the three labels in the News group, we use the following onGetLabel procedure:

```
' This serves as a getLabel callback for the labels.
' We determine which Control ID was passed from the Ribbon
' and set the label appropriately.
Public Sub onGetLabel(control As IRibbonControl, ByRef label)
    Select Case control.Id
        Case "lblWelcome"
            ' Update welcome information
            label = GetWelcomeMessage()
        Case "lblToday"
            ' Update Time
            label = "Today is: " & FormatDateTime(Date, vbLongDate)
        Case "lblPending"
            ' Update Pending Events message
            label = GetPendingEventsNumber
    End Select
End Sub
```

We use a Select Case procedure to test the value of the label Control ID passed into the onGetLabel procedure. For the lblWelcome label, we update the display by calling the GetWelcomeMessage function to retrieve the name of the user currently signed into the database. For the lblToday label, we retrieve the current date from the Windows system date and format it to display as Long Date. For the lblPending label, we retrieve the number of pending events for the current user by calling the GetPendingEventsNumber function.

You might be wondering, when does Access know to update these labels? On the frmSignOn form, you have to select a user name and provide a password. In the Click event of the cmdSignOn command button, we have this code just before the form close code:

```
' Refresh the data in the Ribbon
gobjRibbon1.InvalidateControl "lblWelcome"
gobjRibbon1.InvalidateControl "lblPending"
```

A few lines above this code, Access saves the user name to a public string variable, gstrThisUser, which is referenced in the GetWelcomeMessage function and the GetPendingEventsNumber function. We invalidate the lblWelcome control in the cmdSignOn procedure, which causes Access to refresh this specific control and change the label's text. We also invalidate the lblPending control so that the correct number of pending events is displayed for the current user. After you've saved a pointer to the ribbon, you can invalidate the controls or the entire ribbon as often as you need.

INSIDE OUT

Adjusting Ribbon State

You can minimize the ribbon programmatically by using the following SendKeys command:

```
SendKeys "^{F1}", True
```

That command sends a Ctrl+F1 key combination. The key combination acts like a toggle for the current state of the ribbon. If the ribbon is currently expanded, Access minimizes the ribbon. If you use this key combination when the ribbon is currently minimized, Access expands the ribbon. You can also check the current height of the ribbon using the following line of VBA code:

```
Application.CommandBars.Item("Ribbon").Height
```

Loading Images into Custom Controls

The easiest way to display images on a custom button in your custom ribbon is to use the imageMso attribute. As you'll recall from earlier in this chapter, you can use the imageMso attribute to apply an existing icon from Access 2010 or from any Office 2010 system applications that support ribbons. All the custom command buttons on the ribbons in the Conrad Systems Contacts and Housing Reservations sample databases reuse icons and images from other built-in controls.

You can also load images from an attachment field stored in a table or use the LoadPicture method to load image files stored in a folder. In the Housing Reservations sample database (Housing.accdb), we store all the pictures for the employees in a table called USysRibbonImages. (Note that you can name your table whatever you like, but we chose to name our table with the *USys* prefix so that it displays in the Navigation pane only if you have enabled your Navigation Options to display system objects.) Open the Housing.accdb database now (and close Contacts.accdb if you have it open), and click OK on the opening message box. Next, open the frmSplash form in Form view. After a few seconds, the frmSplash form closes, and then frmSignOn opens. Select Jeff's name from the User Name combo box, and enter **password** in the Password text box. (All the passwords in this sample database are **password**.) Finally, click Sign On to sign into the database under Jeff's name. Access opens the frmMain form and displays a custom ribbon with Jeff's picture in the News group, as shown in Figure 26-13.

Figure 26-13 You can load images from Attachment fields onto custom controls in your ribbons.

Access is placing the image on a button in the News group, cmdPicture, that has no onAction attribute assigned to it. (We don't want anything to happen if you click the button, but we're using the button to display our custom image.) If you were to sign in as a different user, Access would update the picture because we invalidated the control in the Click event of the cmdSignOn button on the frmSignOn form. To set the picture, we use the getImage callback attribute of the custom cmdPicture control in our main ribbon. The getImage callback runs the GetButtonImage function in the modRibbonCallbacks module listed here:

```
Public Function GetButtonImage(control As IRibbonControl, ByRef image)
    ' This function displays a picture of the logged on user
    ' The command button in the Welcome group will update from
    ' a picture stored in an Attachment field in the table USysRibbonImages.
    ' We have to load a hidden form in order to grab the picture
```

```
    Dim frmRibbonImages As Form
    Static rsForm As DAO.Recordset2
    If frmRibbonImages Is Nothing Then
        ' Form is not opened, so open it
        DoCmd.OpenForm "zfrmUSysRibbonImages", WindowMode:=acHidden
        Set frmRibbonImages = Forms("zfrmUSysRibbonImages")
        Set rsForm = frmRibbonImages.Recordset
    End If
    ' Find the picture for the logged on user
    rsForm.FindFirst "UserID='" & gstrThisEmployee & "'"
    If rsForm.NoMatch Then
        ' User not found so set the image to nothing
        Set image = Nothing
    Else
        ' Found a match, so update the display on the Ribbon
        Set image = frmRibbonImages.RibbonImages.PictureDisp
    End If
    Set rsForm = Nothing
End Function
```

To load an image from an attachment field onto a custom ribbon, you have to assign an object that is in the correct format. The PictureDisp property of an Attachment control bound to a picture in an attachment field returns the correct object type. Using an open form bound to the table that contains the attachment field is a simple way to get what we need. (You could also write a COM object in C#, but that's far beyond the scope of this book.) We created a special form, zfrmUSysRibbonImages, especially for this purpose. The code opens this form in hidden mode so it never becomes visible on screen. Next, the code searches the records in the table for a match to the public variable that contains the name of the user signed into the database—gstrThisEmployee. If no match is found, Access sets the image to nothing. If a match is found, Access updates the image on the custom button with the picture stored in the attachment field.

Hiding Options on the Backstage View

You've previously learned that if you set the StartFromScratch attribute of your customization to True, Access hides some of the options available when your ribbon is open. You can selectively hide buttons and commands on the Backstage view by using the <backstage> tags and setting the visible attribute for the built-in tabs and controls to false. For example, if you want to hide everything on the Backstage view except the recent list of database files opened, use the following XML example in a custom ribbon that you load:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="false">
    <!-- Your custom Ribbon definition here -->
  </ribbon>
  <backstage>
    <tab idMso="TabInfo" visible="false"/>
```

```
    <button idMso="FileSave" visible="false"/>
    <button idMso="SaveObjectAs" visible="false"/>
    <button idMso="FileSaveAsCurrentFileFormat" visible="false"/>
    <button idMso="FileOpen" visible="false"/>
    <button idMso="FileCloseDatabase" visible="false"/>
    <tab idMso="TabRecent" visible="false"/>
    <tab idMso="TabNew" visible="false"/>
    <tab idMso="TabPrint" visible="false"/>
    <tab idMso="TabShare" visible="false"/>
    <tab idMso="TabHelp" visible="false"/>
    <button idMso="ApplicationOptionsDialog" visible="false"/>
    <button idMso="FileExit" visible="false"/>
  </backstage>
</customUI>
```

If you click the Backstage view using this customization, you'll see a list of recent database files opened for your users, as shown in Figure 26-14.

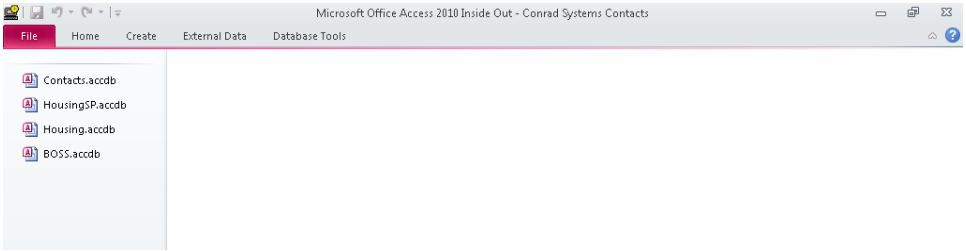


Figure 26-14 You now have limited options available when you click the Backstage view.

Adding Options to the Backstage View

In addition to hiding options on the Backstage view, you might want to create your own options on the Backstage view. A full discussion of all the elements you can add to the Backstage view goes far beyond the scope of this chapter, but we'll show you a sample of some of the customizing you can do to the Backstage view. In the Back Office Software System web database (BOSS.accdb), we created a custom ribbon that displays when you open the database. This custom ribbon—rbnMain—does not hide or add any elements to the default Backstage view. We also created a second, more advanced custom ribbon in this database—rbnMainBackstage—that displays our own Backstage customizations. Close any database you might have open, and then open the BOSS.accdb sample web database. Next, click the File tab on the Backstage view, click Options, and then click the Current Database category. In the Ribbon And Toolbar Options section, click the arrow on the Ribbon Name option, and then select your rbnMainBackstage from the list of loaded ribbons. Click OK to save your changes, and close the Access Options dialog box. Finally, close the database,

reopen the database so Access loads the different ribbon, and then click the File tab on the Backstage view. You'll see a list of recent database files opened for your users, the Exit button to close Access, and a custom tab called BOSS Information, as shown in Figure 26-15.

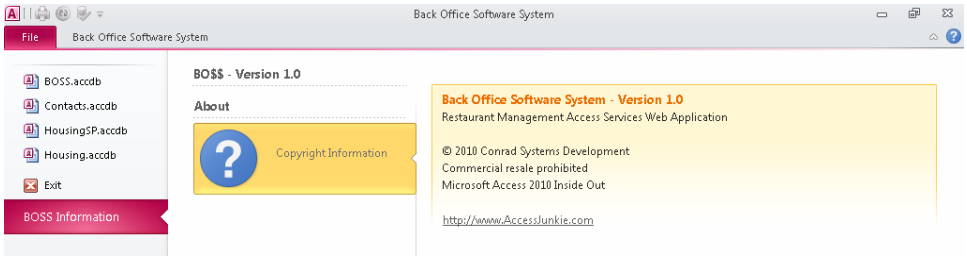


Figure 26-15 You can add your own customization to the Backstage view.

If you open the USysRibbons table in this database, navigate to the third record and examine the custom XML for the rbnMainBackstage ribbon, you'll see the following (note that we've omitted the first part of the ribbon customization for brevity):

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="false">
....(remaining XML customization here)....
  </ribbon>
  <backstage>
   <tab idMso="TabInfo" visible="false"/>
   <button idMso="FileSave" visible="false"/>
   <button idMso="SaveObjectAs" visible="false"/>
   <button idMso="FileSaveAsCurrentFileFormat" visible="false"/>
   <button idMso="FileOpen" visible="false"/>
   <button idMso="FileCloseDatabase" visible="false"/>
   <tab idMso="TabRecent" visible="false"/>
   <tab idMso="TabNew" visible="false"/>
   <tab idMso="TabPrint" visible="false"/>
   <tab idMso="TabShare" visible="false"/>
   <tab idMso="TabHelp" visible="false"/>
   <button idMso="ApplicationOptionsDialog" visible="false"/>
   <tab id="plcInfo" label="BOSS Information" title="Program Options">
    <firstColumn>
     <taskFormGroup id="frmbtnSlab1" label="BO$$ - Version 1.0">
      <category id="myTask" label="About" >
      <task id="tskTest" description="Copyright Information" imageMso="Help">
      <group id="s1" label="Back Office Software System - Version 1.0"
       style="warning">
      <topItems>
      <layoutContainer id="layOptions1" layoutChildren="vertical">
      <labelControl id="lblFrmOptions1" label="Restaurant Management Access
```

```
      Services Web Application"/>
     <labelControl id="lblFrmOptions2" label=" "/>
     <labelControl id="lblFrmOptions3" label="© 2010 Conrad Systems Development"/>
     <labelControl id="lblFrmOptions4" label="Commercial resale prohibited"/>
     <labelControl id="lblFrmOptions5" label="Microsoft Access 2010 Inside Out"/>
     <labelControl id="lblFrmOptions6" label=" "/>
     <hyperlink id="hlnkCSD" label=http://www.AccessJunkie.com
        target="http://www.AccessJunkie.com"/>
     </layoutContainer>
     </topItems>
     </group>
     </task>
    </category>
   </taskFormGroup>
  </firstColumn>
 </tab>
</backstage>
</customUI>
```

In this ribbon customization, we left the Exit option available on the Backstage view so users can exit the application. After we hide the existing options on the Backstage view, the remaining XML customization creates a new tab—plcInfo—to display information about the application using buttons, labels, and a hyperlink.

INSIDE OUT

Adding a Custom Button to the Backstage View

You can also add your own custom button to the Backstage view by adding XML such as the following example:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="false">
    <!-- Ribbon XML -->
  </ribbon>
  <backstage>
   <button id="btnBackstage" label="My Button" imageMso="Help"
     onAction="MyCustomAction"/>
  </backstage>
</customUI>
```

Let's take a look at attributes you can assign to Backstage view controls. Table 26-6 lists the Backstage view attributes you can use in your XML customization.

Table 26-6 Backstage View Attributes

Attribute	Value or Type	Description
align	topLeft, top, topRight, left, center, right, bottomLeft, bottom, or bottomRight	Determines where child controls will be aligned in a layoutContainer.
alignLabel	topLeft, top, topRight, left, center, right, bottomLeft, bottom, or bottomRight	Determines the alignment of a label relative to its control.
allowedTaskSizes	largeMediumSmall, largeMedium, large, mediumSmall, medium, or small	Determines the sizes that are allowed for a task control in a taskGroup or taskFormGroup control.
altText	String	Specifies text that appears when the mouse is hovered over an imageControl.
columnWidthPercent	Positive Integer	Determines the division of columns as a percentage of the width of an entire tab.
expand	horizontal, vertical, both, or neither	Determines how a control expands in a container.
firstColumnMaxWidth	Positive Integer	Determines the maximum width (in pixels) of the first column in a tab.
firstColumnMinWidth	Positive Integer	Determines the minimum width (in pixels) of the first column in a tab.
getAltText	Callback	Defines the macro or procedure that sets the altText for an imageControl.
getHelperText	Callback	Defines the macro or procedure that sets the helperText for a group.
getStyle	Callback	Defines the macro or procedure that sets the style for a group.
helperText	String	Specifies additional text to appear with a group.
isDefinitive	True or False	Determines whether an action should close the Backstage.
layoutChildren	horizontal or vertical	Determines the layout direction for child controls in a layoutContainer.

Attribute	Value or Type	Description
onHide	Callback	Defines the macro or procedure that runs when the Backstage is closed.
onShow	Callback	Defines the macro or procedure that runs when the Backstage is opened.
secondColumnMaxWidth	Positive Integer	Determines the maximum width in pixels of the second column in a tab.
secondColumnMinWidth	Positive Integer	Determines the minimum width in pixels of the second column in a tab.
style	normal, warning, or error	Determines the appearance for a group.
style	normal, borderless, or large	Determines the size and style for a button.

In addition to attributes you can use in your XML customization, you can create many types of Backstage view controls. Table 26-7 lists the types of controls you can use in a custom Backstage view definition and their associated attributes (properties) and callbacks (event properties).

Table 26-7 Backstage Controls

Attribute	Value or Type	Callbacks
backstage		onHide, onShow
bottomItems	None—defined in a Backstage group element	None—defined in a Backstage group element
button	enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, isDefinitive, keyTip, label, tag, visible	getEnabled, getImage, getKeytip, getLabel, getVisible, onAction
category	id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, label, tag, visible	getLabel, getVisible,
checkBox	description, enabled, expand, id, idQ, keyTip, label, screenTip, superTip, tag, visible	getDescription, getEnabled, getKeytip, getLabel, getPressed, getScreentip, getSupertip, getVisible, onAction
comboBox	alignLabel, enabled, expand, id, idQ, keyTip, label, sizeString, tag, visible	getEnabled, getItemCount, getItemID, getItemLabel, getKeytip, getLabel, getText, getVisible, and onChange

Attribute	Value or Type	Callbacks
dropDown	alignLabel, enabled, expand, id, idQ, keyTip, label, screentip, sizeString, supertip, tag, visible	getEnabled, getItemCount, getItemID, getItemLabel, getKeytip, getLabel, getScreentip, getSelectedItemIndex, getSupertip, getText, getVisible, and onAction
editBox	alignLabel, enabled, expand, id, idQ, keytip, label, maxLength, sizeString, tag, visible	getEnabled, getKeytip, getLabel, getText, getVisible
firstColumn	None—defined in a Backstage tab element	None—defined in a Backstage tab element
group	helperText, id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, label, showLabel, style, tag, visible	getHelperText, getLabel, getShowLabel, getStyle, getVisible
groupBox	expand, id, idQ, label, and tag	getLabel
hyperlink	alignLabel, enabled, expand, id, idQ, image, imageMso, keytip, label, screentip, supertip, tag, target, visible	getEnabled, getImage, getKeytip, getLabel, getScreentip, getSupertip, getTarget, getVisible, onAction
imageControl	altText, enabled, id, idQ, image, imageMso, tag, visible	getAltText, getEnabled, getImage, getVisible
labelControl	alignLabel, enabled, expand, id, idQ, label, noWrap, tag, visible	getEnabled, getLabel, getVisible
layoutContainer	align, expand, id, idQ, layoutChildren, visible	None
menu	enabled, id, idQ, image, imageMso, keytip, label, screentip, supertip, tag, visible	getEnabled, getImage, getKeytip, getLabel, getScreentip, getSupertip, getVisible,
menuGroup	id, idQ, itemSize, label, and tag	getLabel
primaryItem	None—defined in a Backstage group element	None—defined in a Backstage group element
radioGroup	alignLabel, enabled, expand, id, idQ, keytip, label, tag, visible	getEnabled, getItemCount, getItemID, getItemLabel, getKeytip, getLabel, getSelectedItemIndex, getVisible, onAction
secondColumn	None—defined in a Backstage tab element	None—defined in a Backstage tab element

Attribute	Value or Type	Callbacks
tab	columnWidthPercent, firstColumnMinWidth, firstColumnMaxWidth, secondColumnMinWidth, secondColumnMaxWidth, id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, enabled, label, visible, keyTip, title, tag	getEnabled, getKeytip, getLabel, getTitle, getVisible
task	description, enabled, id, idMso, idQ, image, imageMso, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, keyTip, label, tag, visible	getDescription, getEnabled, getImage, getKeytip, getLabel, getVisible
taskFormGroup	allowedTaskSizes, helperText, id, idMso, idQ, label, tag, and visible	getHelperText, getLabel, getShowLabel, getVisible
taskGroup	allowedTaskSizes, helperText, id, idMso, idQ, insertAfterMso, insertAfterQ, insertBeforeMso, insertBeforeQ, label, tag, and visible	getHelperText, getLabel, getShowLabel, getVisible
toggleButton	description, enabled, id, idQ, image, imageMso, keytip, label, tag, visible	getDescription, getEnabled, getImage, getKeytip, getPressed, getLabel, getVisible
topItems	None—defined in a Backstage group element	None—defined in a Backstage group element

Creating a Custom Quick Access Toolbar

If you take a close look at the Quick Access Toolbar in Figure 26-15, you'll notice that we also have our own custom Quick Access Toolbar. You've previously learned that if you set the StartFromScratch attribute of your customization to True, Access disables all the buttons on the Quick Access Toolbar. You can selectively add buttons and commands to your custom Quick Access Toolbar by using the <qat> and <documentControls> tags. Near the top of the rbnMain and rbnMainBackstage ribbons in the USysRibbons table, you can see the following XML that we're using to show a few of the built-in commands on the Quick Access Toolbar:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="false">
    <qat>
      <documentControls>
       <button idMso="PrintDialogAccess"/>
       <button idMso="Synchronize" />
       <button idMso="DatabaseServerCompatibilityCheckDatabase" />
      </documentControls>
     </qat>
....(remaining XML customization here)....
  </ribbon>
</customUI>
```

Setting Focus to a Tab

In the Conrad Systems Contacts and Housing Reservations sample databases, we want to display our main ribbon at all times. For the data entry forms, we want to still see the main tab when the custom ribbon for forms is open but put the focus on the Navigation tab as a visual cue for the users of the application. Instead, the Navigation tab opens to the right of the main tab but does not receive focus. Fortunately, RibbonX does provide a TabSetFormReportExtensibility element that you can use for these cases.

When you use the TabSetFormReportExtensibility element, Access places the content into the current tabSet, moves the focus to this tab, and places a caption above the tab. The tab caption matches the Caption property of the current form or report if this property is set. If no caption is set, Access uses the current name of the object. In the rbnForms custom ribbon in both Conrad Systems Contacts and Housing Reservations, we duplicated all the controls in the main ribbon—rbnCSD or rbnProseware—and then added the XML necessary to display the tab we wanted with groups and controls for form navigation. The specific customization for these follows this format:

```
  <contextualTabs>
    <tabSet idMso="TabSetFormReportExtensibility">
        <tab id="tabRecNav" label="Record Navigation">
....(remaining XML customization here)....
        </tab>
    </tabSet>
</contextualTabs>
```

INSIDE OUT

Use VBA to Change Ribbon Tab Focus

In Office 2007, the RibbonX architecture did not support a method to place the focus on a specific tab. In Office 2010, Microsoft now added a method that you can use to set focus to a ribbon tab. An example VBA call in Access to put focus on the main ribbon tab—rbnProseware—in the Housing.accdb would be the following:

```
gobjRibbon1.ActivateTab "tabProseware"
```

In essence, we created our own contextual tab that appears next to the main ribbon tab but receives the focus when the data entry forms open, as shown in Figure 26-16.

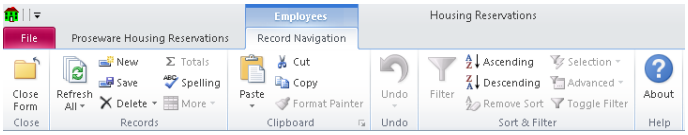


Figure 26-16 Use the TabSetFormReportExtensibility element to set focus to a specific tab.

Quite frankly, we could write an entire book about ribbon customization, but you should have enough information at this point to get started building your own custom ribbons. For more information, visit the Microsoft Developer Network website at *http://msdn.microsoft.com/*. In the remainder of this chapter, you'll learn additional techniques that you can use to customize your applications for your users.

INSIDE OUT

Use One Line of VBA to Hide All Ribbon Elements

If you'd like to hide all elements of the ribbon, including the Quick Access Toolbar and Backstage view, you can write one line of Visual Basic code. In the Load event of your startup form, enter the following code to hide everything:

```
DoCmd.ShowToolbar "Ribbon", acToolbarNo
```

Disabling Layout View

You might have noticed as you built new forms and reports in your Access 2010 client databases that Access sets the new Allow Layout View property to Yes by default. This is a handy feature while you're building forms and reports because it allows you to align, position, and resize controls while you view live data. When you're ready to put your application in production, however, you need to reset this new property to No for all your forms and reports so that the users see your forms and reports as you intended. You could open every form and report in Design view, change the property, and save the form or report. But why do it the hard way? In the Access Options dialog box, you can select an option that disables the ability to open forms and reports in Layout view. Click the File tab on the Backstage view, click Options, and then click the Current Database category. In the Application Options section, clear the Enable Layout View check box, as shown in Figure 26-17.

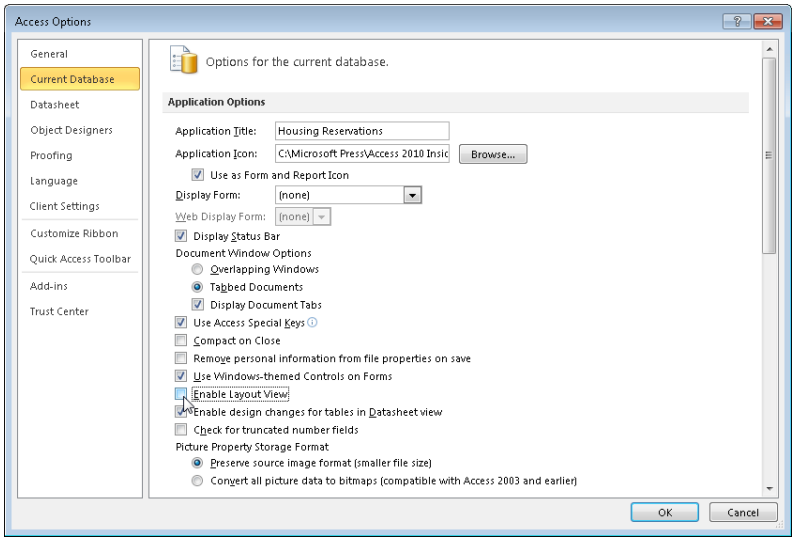


Figure 26-17 You can disable the ability to view objects in Layout view in the Access Options dialog box.

When you clear this option, Access does not show Layout View as an option in the Views group on the ribbon or on any shortcut menus.

Controlling How Your Application Starts and Runs

Especially if you're distributing your application for others to use, you probably want your application to automatically start when the user opens your database. You'll also want to create a main form, such as a navigation form, to help the user navigate to the various parts of your application. You should also set properties and write code to ensure that your user can cleanly exit your application.

Setting Startup Properties for Your Database

At this point, you know how to build all the pieces you need to fully implement your database application. But what if you want your application to start automatically when you open your database? One way is to create a macro named *Autoexec*—Access always runs this macro if it exists when you open the database (unless you hold down the Shift key when you open the database). In the Conrad Systems Contacts database, we first use an AutoExec macro to determine whether the database is being run in a trusted environment. You can also specify an opening form in the startup properties for the database. You can set these properties by clicking the File tab on the Backstage view, clicking Options, and then clicking the Current Database category, as shown in Figure 26-18.

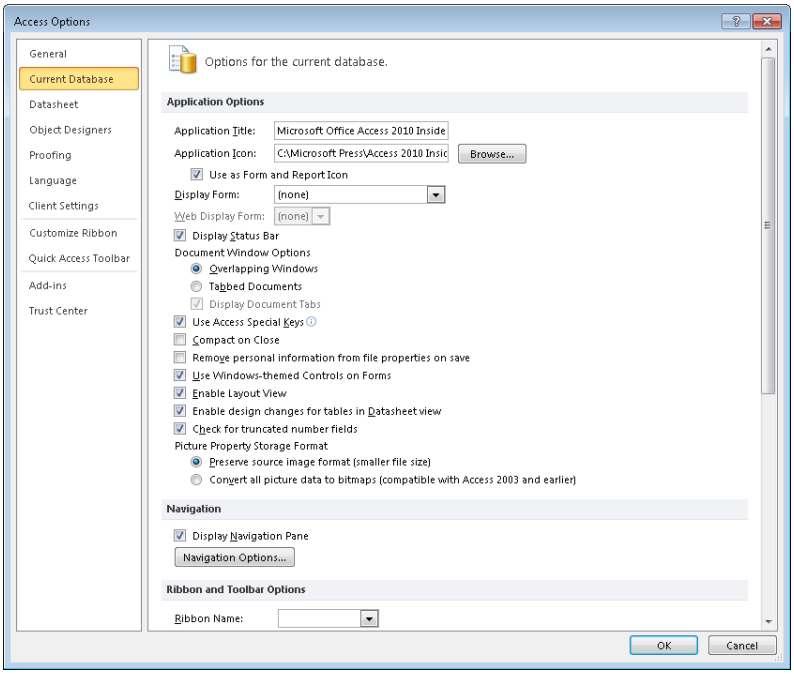


Figure 26-18 You can set startup properties for your database in the Current Database category of the Access Options dialog box.

You can specify which form opens your database by selecting a form from the Display Form list. You can also specify a custom title for the application, an icon for the application, and a custom ribbon to override the built-in ribbon. If you always open the database with its folder set to the current directory, you can simply enter the icon file name, as shown in Figure 26-18. If you're not sure which folder will be current when the application opens, you should enter a fully qualified file name location. Note that you can also ask Access to display the icon you specify as the form and report icon instead of the standard Access icons.

If you clear the Display Navigation Pane check box, Access hides the Navigation pane when your application starts. (As you'll learn in the next section, you can also write code that executes in your startup form to ensure that the Navigation pane is hidden.) You can also hide the status bar if you want by clearing the Display Status Bar check box. We like to use the SysCmd function to display information on the status bar, so we usually leave the Display Status Bar check box selected. We recommend that you always clear the Enable Design Changes For Tables In Datasheet View (For This Database) check box if you're using a client database. If you leave this check box selected, your users can make design changes to your tables displayed in Datasheet view, as well as any forms that open in Datasheet view. Note that this option is selected by default in a web database and is dimmed, so you cannot change the setting.

Finally, you can disable special keys—such as F11 to reveal the Navigation pane, Ctrl+G to open the Debug window, or Ctrl+Break to halt code execution—by clearing the Use Access Special Keys check box. As you can see, you have many powerful options for customizing how your application starts and how it operates.

Starting and Stopping Your Application

Although you can set startup properties asking Access to hide the Navigation pane, you might want to include code in the Load event of your startup form to make sure that it is hidden. All the sample databases provided with this book open the frmCopyright form as the startup form. Note that the AutoExec macro in these sample databases first checks to see whether the database is running in a trusted location. If the database is in a trusted location, the macro opens frmCopyright; otherwise, the macro opens the fdlgNotTrusted form followed by the frmCopyrightNotTrusted form. The copyright forms display information about the database. In the trusted version, code behind the form checks connections to linked tables. In both the Conrad Systems Contacts and Housing Reservations sample applications, the code behind the frmCopyright form tells you to open the frmSplash form to actually start the application.

When the frmSplash form opens, code in the Load event uses the following procedure to make sure the Navigation pane is hidden:

```
' Select the Navigation Pane
DoCmd.SelectObject acForm, "frmSplash", True
' .. and hide it
RunCommand acCmdWindowHide
```

The procedure hides the Navigation pane by selecting a known object in the Navigation pane to give the Navigation pane the focus and then executing the WindowHide command. The splash form waits for a timer to expire (the Timer event procedure) and then opens a form to sign on to the application. When you sign on successfully, the frmMain form finally opens.

The frmMain form in the Conrad Systems Contacts application has no Close button and no control menu button. The database also has an AutoKeys macro defined that intercepts any attempt to close a window using the Ctrl+F4 keys. (You'll learn about creating an AutoKeys macro in the next section.) Therefore, you must click the Exit button on the frmMain form to close the application. On the other hand, the frmMain form in the Housing Reservations application does allow you to press Ctrl+F4 or click the Close button to close the form and exit the application.

You should always write code to clean up any open forms, reset variables, and close any open recordsets when the user asks to exit your application. Because the user can't close the frmMain form in Conrad Systems Contacts application except by clicking Exit, you'll find such cleanup code in the command button's Click event. In the frmMain form in the Housing Reservations database, the cleanup code is in the form's Close event procedure. The code in both forms is similar, so here's the exit code in the Conrad Systems Contacts sample application:

```
Private Sub cmdExit_Click()
Dim intErr As Integer, frm As Form, intI As Integer
Dim strData As String, strDir As String
Dim lngOpen As Long, datBackup As Date
Dim strLowBkp As String, strBkp As String, intBkp As Integer
Dim db As DAO.Database, rst As DAO.Recordset
  If vbNo = MsgBox("Are you sure you want to exit?", _
    vbYesNo + vbQuestion + vbDefaultButton2, _
      gstrAppTitle) Then
      Exit Sub
  End If
  ' Trap any errors
  On Error Resume Next
  ' Make sure all forms are closed
  For intI = (Forms.Count - 1) To 0 Step -1
    Set frm = Forms(intI)
    ' Don't close myself!
    If frm.Name <> "frmMain" Then
      ' Use the form's "Cancel" routine
      frm.cmdCancel_Click
      DoEvents
    End If
    ' Note any error that occurred
    If Err <> 0 Then intErr = -1
  Next intI
  ' Log any error beyond here
  On Error GoTo frmMain_Error
  ' Skip backup check if there were errors
  If intErr = 0 Then
    Set db = CurrentDb
    ' Open ztblVersion to see if we need to do a backup
    Set rst = db.OpenRecordset("ztblVersion", dbOpenDynaset)
    rst.MoveFirst
    lngOpen = rst!OpenCount
    datBackup = rst!LastBackup
    rst.Close
    Set rst = Nothing
    ' If the user has opened 10 times
    ' or last backup was more than 2 weeks ago...
```

```
If (lngOpen Mod 10 = 0) Or ((Date - datBackup) > 14) Then
  ' Ask if they want to backup...
  If vbYes = MsgBox("CSD highly recommends backing up " & _
    "your data to avoid " & _
    "any accidental data loss.  Would you like to backup now?", _
    vbYesNo + vbQuestion, gstrAppTitle) Then
    ' Get the name of the data file
    strData = Mid(db.TableDefs("ztblVersion").Connect, 11)
    ' Get the name of its folder
    strDir = Left(strData, InStrRev(strData, "\"))
    ' See if the "BackupData" folder exists
    If Len(Dir(strDir & "BackupData", vbDirectory)) = 0 Then
      ' Nope, build it!
      MkDir strDir & "BackupData"
    End If
    ' Now find any existing backups - keep only three
    strBkp = Dir(strDir & "BackupData\CSDBkp*.accdb")
    Do While Len(strBkp) > 0
      intBkp = intBkp + 1
      If (strBkp < strLowBkp) Or (Len(strLowBkp) = 0) Then
        ' Save the name of the oldest backup found
        strLowBkp = strBkp
      End If
      ' Get the next file
      strBkp = Dir
    Loop
    ' If more than two backup files
    If intBkp > 2 Then
      ' Delete the oldest one
      Kill strDir & "BackupData\" & strLowBkp
    End If
    ' Now, setup new backup name based on today's date
    strBkp = strDir & "BackupData\CSDBkp" & _
      Format(Date, "yymmdd") & ".accdb"
    ' Make sure the target file doesn't exist
    If Len(Dir(strBkp)) > 0 Then Kill strBkp
    ' Create the backup file using Compact
    DBEngine.CompactDatabase strData, strBkp
    ' Now update the backup date
    db.Execute "UPDATE ztblVersion SET LastBackup = #" & _
      Date & "#", dbFailOnError
    MsgBox "Backup created successfully!", vbInformation, gstrAppTitle
  End If
  ' See if error log has 20 or more entries
  If db.TableDefs("ErrorLog").RecordCount > 20 Then
    ' Don't ask if they've said not to...
    If Not (DLookup("DontSendError", "tblUsers", _
      "UserName = '" & gstrThisUser & "'")) Then
      DoCmd.OpenForm "fdlgErrorSend", WindowMode:=acDialog
    Else
      db.Execute "DELETE * FROM ErrorLog", dbFailOnError
    End If
```

```
      End If
    End If
    Set db = Nothing
  End If
  ' Restore original keyboard behavior
  ' Disabled in this sample
'  Application.SetOption "Behavior Entering Field", gintEnterField
'  Application.SetOption "Move After Enter", gintMoveEnter
'    Application.SetOption "Arrow Key Behavior", gintArrowKey
  ' We're outta here!
frmMain_Exit:
  On Error GoTo 0
  DoCmd.Close acForm, Me.Name
  ' In a production application, would quit here
  DoCmd.SelectObject acForm, "frmMain", True
  Exit Sub
frmMain_Error:
  ErrorLog "frmMain", Err, Error
  Resume frmMain_Exit
End Sub
```

After confirming that the user really wants to exit, the code looks at every open form. All forms have a public cmdCancel_Click event procedure that this code can call to ask the form to clear any pending edits and close itself. The DoEvents statement gives that code a chance to complete before going on to the next form. Notice that the code skips the form named frmMain (the form where this code is running).

If there were no errors closing all the forms, then the code opens a table that contains a count of how many times this application has run and the date of the last backup. Every 10th time the application has run, or two weeks after the last backup, the code offers to create a backup of the application data. If the user confirms creating a backup, the code creates a Backup subfolder if it does not exist, deletes the oldest backup if there are three or more in the folder, and then backs up the data using the CompactDatabase method of the DBEngine.

Next, the code checks to see whether more than 20 errors have been logged by code running in the application. If so, it opens a dialog box that gives the user the option to e-mail the error log, print out the error log, skip printing the error log this time, or turn off the option to print the log. Because the error log option form opens in Dialog mode, this code waits until that form closes. Finally, the code closes this form and selects an object in the Navigation pane to reveal that pane. If this weren't a demonstration application, the code would use the Quit method of the Application object to close and exit Access.

This might seem like a lot of extra work, but taking care of details like this really gives your application a professional polish.

Creating an AutoKeys Macro

As noted earlier, the Conrad Systems Contacts sample application (Contacts.accdb) has an AutoKeys macro defined to intercept pressing Ctrl+F4. You can normally press this key combination to close any window that has the focus, but the application is designed so that you must close the frmMain form using the Exit button, not Ctrl+F4. You can create an AutoKeys macro to define most keystrokes that you want to intercept and handle in some other way. You can define something as simple as a StopMacro action to effectively disable the keystroke, create a series of macro actions that respond to the keystrokes, or use the RunCode action to call complex Visual Basic code. Figure 26-19 shows you the AutoKeys macro in the Conrad Systems Contacts database, open in Design view.

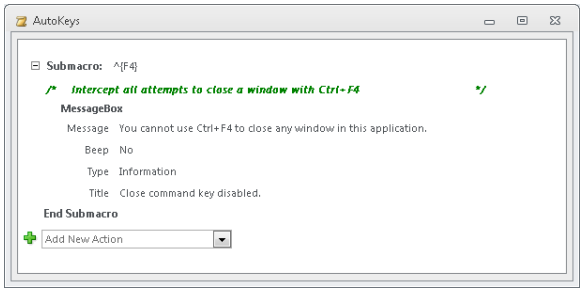


Figure 26-19 The design of this AutoKeys macro intercepts the Ctrl+F4 combination.

The critical part of a macro defined as an AutoKeys macro is the submacro name. When the submacro name is a special combination of characters that match a key name, the submacro executes whenever you press those keys. Table 26-8 shows you how to construct submacro names in an AutoKeys macro to respond to specific keys.

Table 26-8 AutoKeys Macro Key Codes

AutoKeys Submacro Name	Key Intercepted
^letter or ^number	Ctrl+[the named letter or number key]
{Fn}	The named function key (F1–F12)
^{Fn}	Ctrl+[the named function key]
+{Fn}	Shift+[the named function key]
{Insert}	Insert
^{Insert}	Ctrl+Insert
+{Insert}	Shift+Insert

AutoKeys Submacro Name	Key Intercepted
{Delete} or {Del}	Delete
^{Delete} or ^{Del}	Ctrl+Delete
+{Delete} or +{Del}	Shift+Delete

Keep in mind that you can also intercept any keystroke on a form in the KeyDown and KeyPress events when you want to trap a particular key on only one form or control.

Performing a Final Visual Basic Compile

The very last task you should perform before placing your application in production is to compile and save all your Visual Basic procedures. When you do this, Access stores a compiled version of the code in your database. Access uses the compiled code when it needs to execute a procedure you have written. If you don't do this, Access has to load and interpret your procedures the first time you reference them—each and every time you start your application. For example, if you have several procedures in a form module, the form will open more slowly the first time because Access has to also load and compile the code.

To compile and save all the Visual Basic procedures in your application, open any module—either a module object or a module associated with a form or report. Choose Compile *project-name* from the Debug menu, as shown in Figure 26-20. If your code compiles successfully, be sure to save the result by choosing File, Save or by clicking the Save button on the toolbar. (If you have errors in any of your code, the compiler halts at the first error it finds, displays the line of code, and displays an error message dialog box.) After successfully compiling and saving your Visual Basic project, close your database and compact it, as described in Chapter 5, "Modifying Your Table Design."

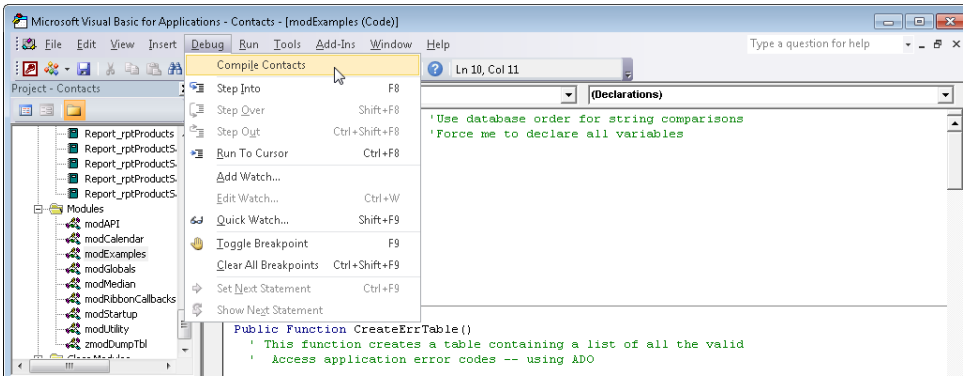


Figure 26-20 Choose the Debug, Compile project-name command to compile all the Visual Basic procedures in your database.

As you've seen in this book, you can quickly learn to build complex client and web applications. You can use the relational database management system (RDMS) in Access 2010 to store and manage your data locally, on a network, or on a server running Microsoft SharePoint, and you can access information in other popular database formats or in any server- or mainframe-hosted database that supports the Open Database Connectivity (ODBC) standard. You can get started with macros to become familiar with event-oriented programming in web and client databases. With a little practice, you'll soon find yourself writing Visual Basic event procedures like a pro. In Chapter 27, "Distributing Your Application," you'll learn how to set up your application so that you can distribute it to others.