# The Visual Studio Tools for Office ‘v3’ Community Technical Preview - Tutorial

Andrew Whitechapel  
Microsoft Corporation

March 21, 2006

Applies to:

    Microsoft® Office System 2007 Professional (Beta 1 Technical Refresh)  
    Microsoft® Visual Studio 2005  
    Microsoft® Visual Studio 2005 Tools for the Microsoft Office System

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release.  This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document.  Information in this document, including URL and other Internet Web site references, is subject to change without notice.  The entire risk of the use or the results from the use of this document remains with the user.  Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious.  No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.  Complying with all applicable copyright laws is the responsibility of the user.  Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document.  Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

 2006 Microsoft Corporation.  All rights reserved.

Microsoft, MS-DOS, Visual Basic, Visual C#, Visual Studio, Visual Studio Tools for Office, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

Summary: This paper walks you through creating a range of solutions that explore the functionality offered in the first Community Technical Preview of the Microsoft Visual Studio Tools for Office ‘v3’ (“Orcas”). The CTP is designed to work against the Beta 1 Technical Refresh of Microsoft Office 2007 applications, and is based on Visual Studio 2005.

#### Introduction

The primary goal of this CTP of the next version of Visual Studio Tools for Office is to give developers an early glimpse of some of the key new features and feature directions in the area of Office programmability, using the default Visual Studio Tools for Office toolset.

For detailed requirements and installation instructions please see the documentation that accompanies the CTP download.

One of the aims of the Visual Studio Tools for Office ‘v3’ release is that the functionality that was available for Visual Studio Tools for Office 2005 solutions will still be available for ‘v3’ solutions. With the exception of add-ins for Outlook, all the Visual Studio Tools for Office 2005 features are document-centric. That is, you build a customization based on a specific document or template. The intention is that all of these features will still work, both with existing Visual Studio Tools for Office 2005 solutions and with new Visual Studio Tools for Office ‘v3’ solutions.

This first CTP focuses on the new features, and not on the existing features. This tutorial also focuses on the new features. The new features in this first CTP are all application-centric. Therefore, this paper has walkthrough exercises on the following features:

* Creating an application-level add-in for any Office application.
* Adding custom task panes to an add-in.
* Adding ribbon customization to an add-in.
* Synchronizing ribbon and task pane.
* Adding multiple ribbons to an add-in.
* Adding custom form regions to an add-in.

Note that as well as focusing on application-centric features, this first CTP also focuses on runtime support and offers very little design-time support. Later CTPs will have far more comprehensive design-time support, to provide the seamless RAD experience that developers expect from Visual Studio Tools for Office. As you work through these exercises, bear in mind that the detailed implementation is subject to change as the product is increasingly fine-tuned for release.

Also, these exercises are deliberately designed to be very simple, and to illustrate how to get started as quickly as possible with the new features without the distraction of the additional checks and balances you would normally include in production code. For the same reason, no attempt is made to simulate realistic business scenarios. Each exercise deliberately uses a different Office host application, to demonstrate that the steps are largely similar regardless of which host you choose to target. (The exception to this is the exercise on custom form regions, which only apply to Outlook.)

You can create an add-in that has one or more custom task panes, and/or one or more ribbons (depending on the host application), and/or uses custom form regions (Outlook only). The following exercises are designed to be self-contained, so each one focuses on only a subset of the possible features you could implement.

#### Exercise 1: Creating an Add-in

The first CTP has project (design-time) and runtime support for creating Visual Studio Tools for Office add-ins for all 10 Office applications that support COM add-ins. In this exercise, you will create a simple add-in, and use the strong-typing provided by Visual Studio Tools for Office to make simple use of the host application’s object model.

1. Start Visual Studio 2005 (with the Visual Studio Tools for Office ‘v3’ CTP installed).
2. From the File menu, point to New, and then click Project. This displays the New Project dialog box. Expand out the tree of Project types to find the programming language you want to work in (C# or VB). Then select the Office node.

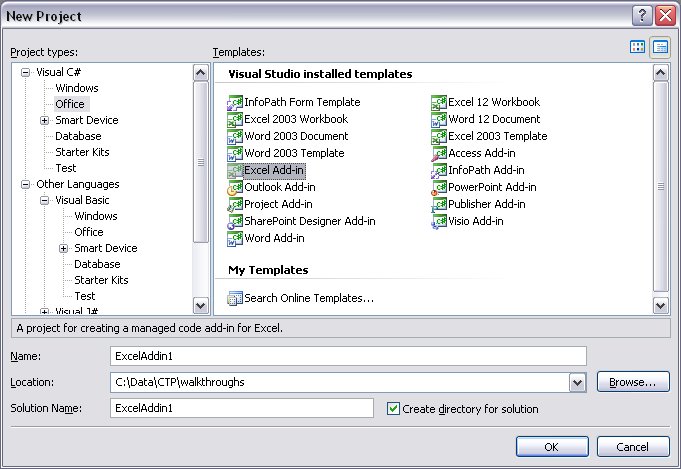


Figure The New Project dialog showing Visual Studio Tools for Office templates

1. You will see all the Visual Studio Tools for Office project types listed in the Templates pane: both document-level templates and application-level add-in templates. Select the add-in template for the Office application you want to target. In this walkthrough, we’ll select Excel, but the steps will be the same for any other application.
2. Specify a suitable name and location for your project, and click OK. In this example, we’ll accept the default name of ExcelAddin1.
3. This will generate all the baseline code necessary for your add-in. At this point, only the project code has been generated: Visual Studio has not made any registry entries, nor any security entries.
4. You can now Press Ctrl-F5 to test the add-in. This will build the add-in, register it, set up security for it, and run the target host application you chose – in this example, Excel. Every time you rebuild the add-in, the registry entries and security will be re-deployed on the local machine. This is standard behavior for Visual Studio Tools for Office solutions – to simplify deployment and testing on the development machine.
5. Recall that you have not yet made any changes to the project code, so although this add-in will load successfully, it won’t do anything.
6. With Excel running, you can now check that your add-in is getting loaded correctly. In Excel, go to the File menu, and click Excel Options. In the Excel Options dialog box, select Add-Ins from the navigator bar on the left. This will display a list of installed add-ins, including the one you’ve just created. Notice that Excel reports the full path to your add-in, not just mscoree.dll (as is the case with traditional “shared” add-ins).

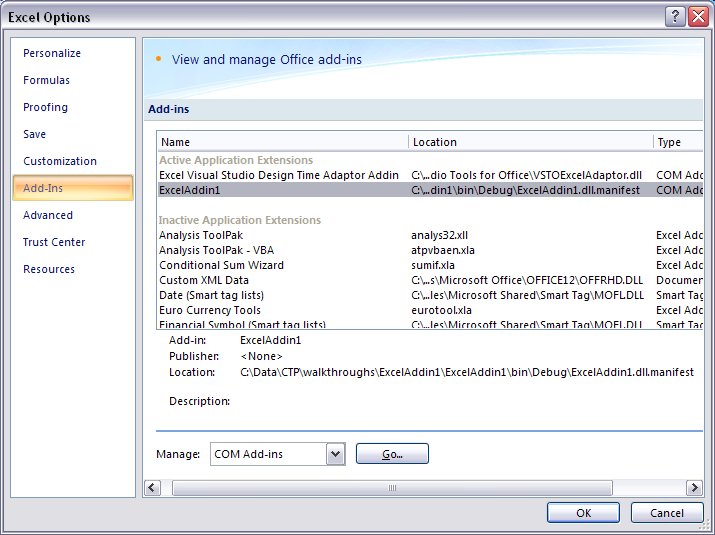


Figure The Application Options dialog showing installed add-ins

1. At the bottom of this dialog box, you will see a dropdown list, labeled “Manage”. If you select the COM Add-ins entry in this list, and click Go, this will take you to the traditional COM Add-ins dialog.

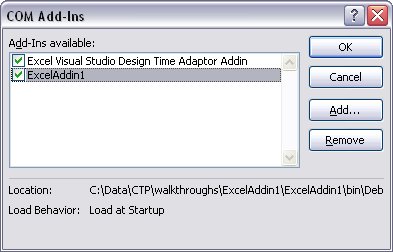


Figure The traditional COM Add-ins dialog

1. When you’ve confirmed that your add-in is indeed being loaded by Excel, you can close the application.
2. Back in Visual Studio, examine the project. In the Project Explorer window, you can see that the project wizard created two projects: a project for the add-in itself, and a setup project. This is consistent with the way traditional “shared” add-in projects are created, but note that this is one area which is likely to change significantly before final release.
3. The project wizard has also added all the necessary references to the .NET, Office and Visual Studio Tools for Office primary interop assemblies (PIAs), which supports the strong-typing that Visual Studio Tools for Office add-ins use.
4. Finally, notice that the Solution Explorer lists only one source file – ThisApplication.cs (C#) or ThisApplication.vb (VB). This provides the standard Visual Studio Tools for Office Startup and Shutdown methods. The purpose of these should be clear: you can put any initialization code you want into the Startup method, and any termination cleanup code you want into the Shutdown method.

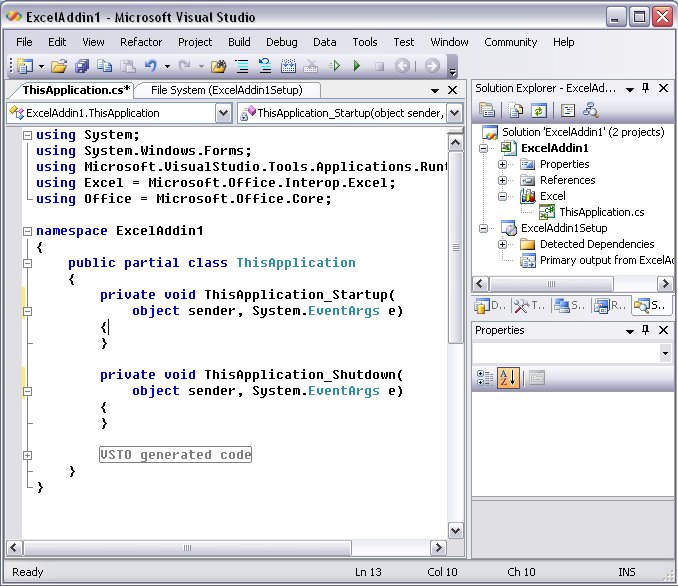


Figure Baseline code generated for an Excel Visual Studio Tools for Office add-in (C#)

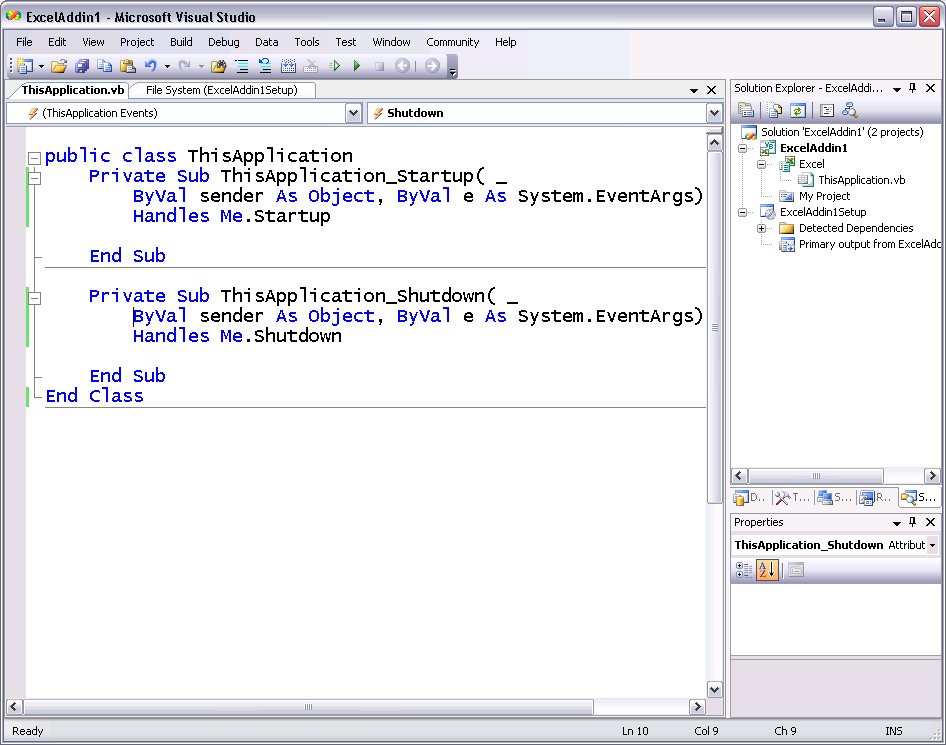


Figure Baseline code generated for an Excel Visual Studio Tools for Office add-in (VB)

1. Let’s make one simple change to call into Excel’s object model. Write one line of code to get the host application’s Name property and display it in a message box. Put this line of code in the ThisApplication\_Startup method:

C#

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

MessageBox.Show(this.Name);

}

VB

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

MessageBox.Show(Me.Name)

End Sub

1. As you type this. or Me., you will see that the Visual Studio autocomplete feature provides a dropdown list of members for this object. You can scroll through the list and select the member you want, or you can simply type it in.

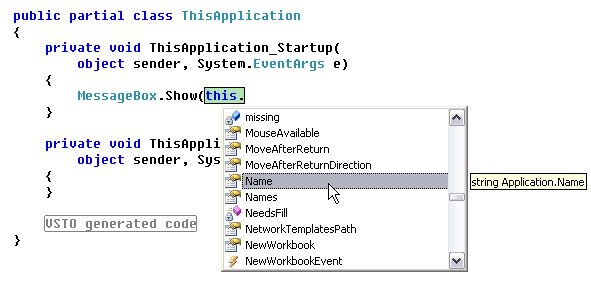


Figure Standard Visual Studio autocomplete for strongly-typed objects (C#)

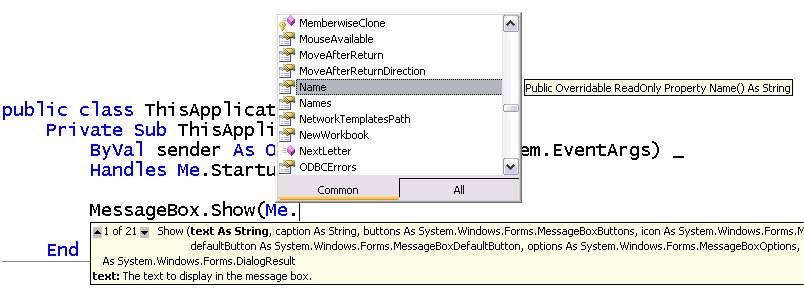


Figure Standard Visual Studio autocomplete for strongly-typed objects (VB)

1. Press Ctrl-F5 to test-run the solution again. This time, when the add-in is loaded (just as Excel is starting up), it will display the message box.

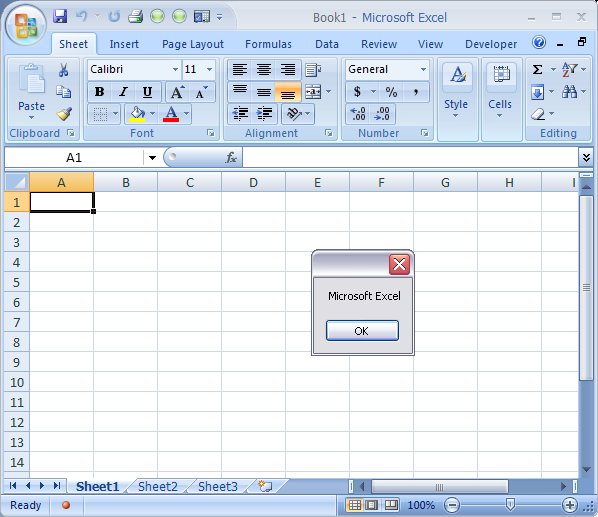


Figure A very simple Visual Studio Tools for Office add-in using the host's object model

1. This exercise has shown how easy it is to build a strongly-typed, robust Visual Studio Tools for Office add-in in Visual Studio. The project wizard generates all the “plumbing” code that is required, leaving you to focus on the business-specific custom code that you want to add. This exercise targeted Excel as the host application, but you can use the same steps to build Visual Studio Tools for Office add-ins for any other Office application. What is less obvious is that the Visual Studio Tools for Office runtime also implements the infrastructure necessary to ensure that your add-in runs in a separate application domain, and undergoes strict security checks.

#### Exercise 2: Adding Custom Task Panes

In this exercise, you will create an add-in with a custom task pane. Custom task panes are supported in Access, Excel, InfoPath, Outlook, PowerPoint and Word. This exercise uses PowerPoint as the target host.

Note that an add-in can implement one or more custom task panes by implementing the ICustomTaskPaneConsumer interface. However, Visual Studio Tools for Office provides a default implementation of this interface, so you can (if you wish) use this default implementation to dramatically streamline your development experience. The following exercise assumes you want to have this RAD experience instead of implementing the interface directly.

1. Start Visual Studio 2005 (with the Visual Studio Tools for Office ‘v3’ CTP installed). From the File menu, point to New, and then click Project. This displays the New Project dialog box. Expand out the tree of Project types to find the programming language you want to work in (C# or VB). Then select the Office node.
2. In the Templates panel, select the add-in template for the Office application you want to target. In this walkthrough, we’ll select PowerPoint, but the steps will be the same for any Office application that supports custom task panes.
3. Specify a suitable name and location for your project, and click OK. In this example, we’ll accept the default name of PowerPointAddin1.
4. This will generate all the baseline code necessary for your add-in.
5. In Solution Explorer, right-click the project, point to Add, then click User Control. This displays the Add New Item dialog box. You can type in any suitable name you want for this UserControl. In this exercise, accept the default “UserControl1” and click Add.

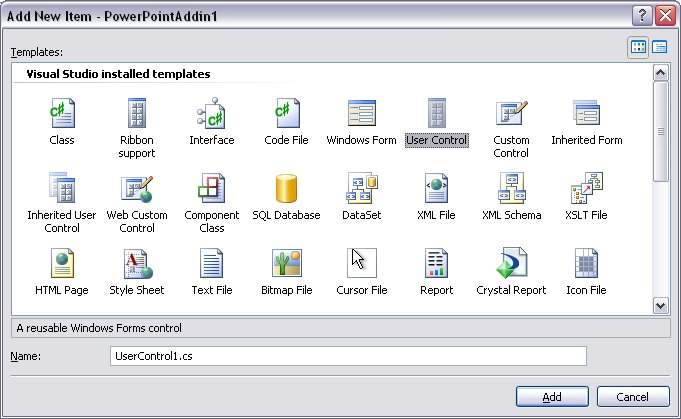


Figure Adding a new UserControl for a custom task pane

1. You could put any controls you like into your UserControl, including any managed Windows Forms controls and any ActiveX controls (these would be wrapped as managed controls). For this exercise, simply drag and drop a MonthCalendar control onto the designer, and size the control appropriately.

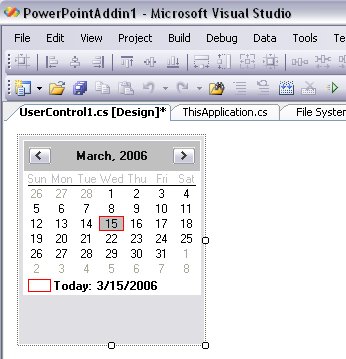


Figure The UserControl designer

1. Still in the UserControl designer, double-click the MonthCalendar control. This will generate the default event handler for this control – in this case, a DateChanged event handler.
2. The intent here is to respond to the user clicking the task pane by performing some simple operation that uses the host application’s object model. In reality, you could implement any kind of processing here, and you don’t need to talk to the host application if it is not appropriate. For the purposes of this exercise, implement the DateChanged event handler to insert the selected date in long format into the current presentation. To achieve this, first open the Application.cs or Application.vb file, and add a couple of statements above the add-in’s namespace declaration, to use the Office and PowerPoint namespaces.

C#

using Office = Microsoft.Office.Core;

using PowerPoint = Microsoft.Office.Interop.PowerPoint;

VB

Imports Office = Microsoft.Office.Core

Imports PowerPoint = Microsoft.Office.Interop.PowerPoint

1. Then, in the DateChanged event handler, set up a try..catch. In the try block, add code to add a textbox shape to the current presentation. At the same time, set the font properties for the text. As you type in code, Visual Studio will help you to type correct code, by providing autocomplete and intellisense.

C#

private void monthCalendar1\_DateChanged(object sender, DateRangeEventArgs e)

{

try

{

PowerPoint.Slide slide =   
 Globals.ThisApplication.ActivePresentation.Slides[1];

PowerPoint.Shape textbox = slide.Shapes.AddTextbox(

Office.MsoTextOrientation.msoTextOrientationHorizontal,

50, 100, 600, 50);

textbox.TextFrame.TextRange.Text = e.Start.ToLongDateString();

textbox.TextFrame.TextRange.Font.Size = 48;

textbox.TextFrame.TextRange.Font.Color.RGB =   
 Color.DarkViolet.ToArgb();

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

VB

Private Sub MonthCalendar1\_DateChanged( \_

ByVal sender As System.Object, \_

ByVal e As System.Windows.Forms.DateRangeEventArgs) \_

Handles MonthCalendar1.DateChanged

Try

Dim slide As PowerPoint.Slide

Dim textbox As PowerPoint.Shape

slide = Globals.ThisApplication.ActivePresentation.Slides(1)

textbox = slide.Shapes.AddTextbox( \_

Office.MsoTextOrientation.msoTextOrientationHorizontal, \_

50, 100, 600, 50)

TextBox.TextFrame.TextRange.Text = e.Start.ToLongDateString()

TextBox.TextFrame.TextRange.Font.Size = 48

TextBox.TextFrame.TextRange.Font.Color.RGB = Color.DarkViolet.ToArgb()

Catch ex As Exception

MessageBox.Show(ex.ToString())

End Try

End Sub

1. Finally, add code to the ThisApplication.cs or ThisApplication.vb file to create a custom task pane. At the top of this file, add a statement to use the Microsoft.Office.Tools namespace – this is where the CustomTaskPane class is defined.

C#

using Microsoft.Office.Tools;

VB

Imports Microsoft.Office.Tools

1. Then, in ThisApplication\_Startup, create a new custom task pane. You can specify any string you like for the task pane caption. At the same time, make sure the task pane is visible. As with all Visual Studio Tools for Office projects, Visual Studio Tools for Office supports a very RAD experience, reducing the amount of code you need to type in manually. In this CTP, as you can see from the code snippet below, you can create a custom task pane with one line of code, specifying the UserControl and the caption you want to use.

C#

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

CustomTaskPane ctp =

this.CustomTaskPanes.Add(new UserControl1(), "My Caption");

ctp.Visible = true;

}

VB

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

Dim ctp As CustomTaskPane

ctp = Me.CustomTaskPanes.Add(New UserControl1(), "My Caption")

ctp.Visible = True

End Sub

1. You can now Press Ctrl-F5 to test the add-in. This will build and register the add-in, and run the target host application you chose – in this example, PowerPoint. When the user changes the date in the custom task pane, this date is inserted as text into the presentation.

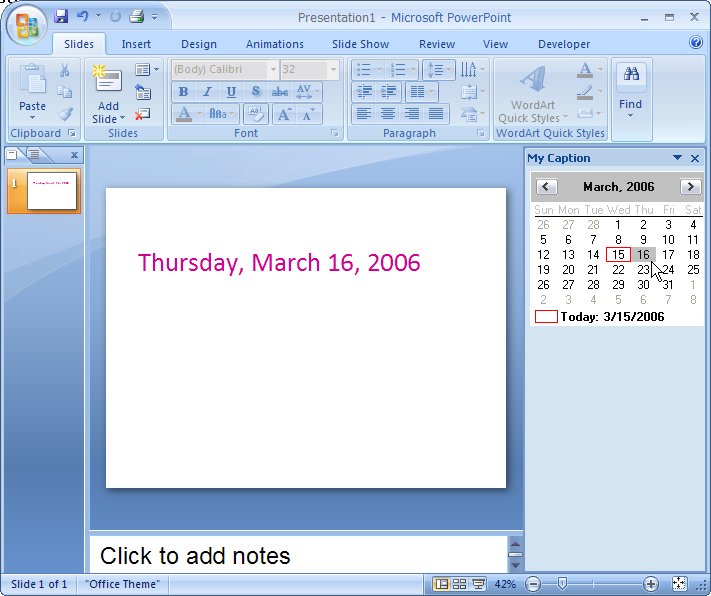


Figure A Visual Studio Tools for Office add-in with a custom task pane in PowerPoint

1. In this exercise, you created a very simple Visual Studio Tools for Office add-in with a custom task pane, and implemented the task pane user interface to make use of the host application’s object model. As with add-ins generally, the purpose of Visual Studio Tools for Office is to provide rich design-time tools to allow you to build solutions quickly and intuitively, and a robust runtime infrastructure. Visual Studio Tools for Office add-ins (and document-level customizations) are each loaded into separate application domains. The Visual Studio Tools for Office runtime ensures that your custom task pane also runs as part of your add-in in an application domain separate from any other add-in or managed customization.

#### Exercise 3: Adding Ribbon Customization

In this exercise, you will create an add-in with a ribbon customization. The new ribbon is supported in Access, Excel, Outlook, PowerPoint and Word. This exercise uses Word as the target host.

Note that an add-in can implement ribbon customization by implementing the IRibbonExtensibility interface. However, Visual Studio Tools for Office provides a default implementation of this interface, so you can use this default implementation to dramatically streamline your development experience. The following exercise assumes you want to have this RAD experience instead of implementing the interface directly.

1. Start Visual Studio 2005 (with the Visual Studio Tools for Office ‘v3’ CTP installed). From the File menu, point to New, and then click Project. This displays the New Project dialog box. Expand out the tree of Project types to find the programming language you want to work in (C# or VB). Then select the Office node.
2. In the Templates panel, select the add-in template for the Office application you want to target. In this walkthrough, we’ll select Word, but the steps will be the same for any Office application that supports ribbon customization.
3. Specify a suitable name and location for your project, and click OK. In this example, we’ll accept the default name of WordAddin1. This will generate all the baseline code necessary for your add-in.
4. In Solution Explorer, right-click on the project, point to Add, then click New Item. This displays the Add New Item dialog box. Select the Ribbon item. You can type in any suitable name you want for this ribbon. In this exercise, accept the default “Ribbon1” and click Add.

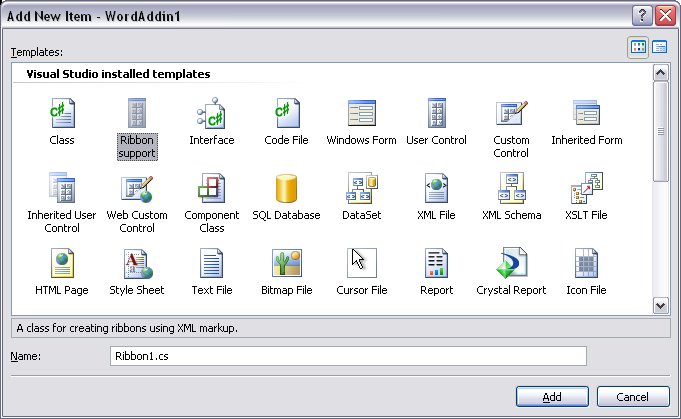


Figure Adding a new Ribbon to a Visual Studio Tools for Office add-in

1. This generates a new class, called Ribbon1 in the file Ribbon1.cs (or Ribbon1.vb), and an XML file called Ribbon1.xml. The Ribbon1.xml file contains markup for a very simple ribbon customization. This is provided as a starting point for your own ribbon markup. This markup specifies one custom tab, labeled “.NET Add-in Tab”, one custom group, labeled “Hello world Group”, and one custom togglebutton, labeled “Hello World”.

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

<ribbon startFromScratch="false">

<tabs>

<tab id="VSTO.Tab" label=".NET Add-in Tab" visible="1">

<group id="VSTO.Group" label="Hello World Group" visible="1">

<toggleButton id="toggleButton" label="Hello World"

tooltip="Hello World Tooltip" onAction="OnToggleButton" />

</group>

</tab>

</tabs>

</ribbon>

</customUI>

1. The Ribbon1 class is defined in Ribbon1.cs (or Ribbon1.vb). The interesting features of this class are described here. This class implements the IRibbonExtensibility interface defined by Office specifically for customizing the ribbon. This is implemented in a class that is separate from the main ThisApplication class for two reasons. First, it is good practice to factor out discrete functionality into separate classes. Second, the class that implements IRibbonExtensibility must be made visible to COM, and the ThisApplication class cannot be made COM-visible because it is derived from classes where this would make no sense, and which are explicitly *not* COM-visible.

C#

[ComVisible(true)]

public class Ribbon1 : Office.IRibbonExtensibility

{

VB

<ComVisible(True)> \_

Public Class Ribbon1

Implements Office.IRibbonExtensibility

1. The Ribbon1 class declares a private field for the underlying IRibbonUI interface.

C#

private Office.IRibbonUI ribbon;

VB

Private ribbon As Office.IRibbonUI

1. The Ribbon1 constructor hooks up the ServiceRequest event on the Visual Studio Tools for Office Application object for this application. This is a standard mechanism that is actually also used by custom task panes, custom form regions, and potentially other new Office programmability interfaces. Here’s how this works: Office loads the add-in, and queries the add-in to see if it implements any of the programmability interfaces. In this add-in we do implement the IRibbonExtensibility interface, so we check to see if this is the interface that Office is querying for, and if so, we set the Service property on the incoming event object to this current object. This tells Office that this is the object that implements IRibbonExtensibility, so that Office can later call back on this object when it needs to.

C#

public Ribbon1()

{

Globals.ThisApplication.ServiceRequest += new EventHandler<Microsoft.Office.Tools.ServiceRequestEventArgs>(ThisApplication\_ServiceRequest);

}

void ThisApplication\_ServiceRequest(object sender, Microsoft.Office.Tools.ServiceRequestEventArgs e)

{

Guid dispatchGuid = new Guid("{00020400-0000-0000-C000-000000000046}");

if (e.Guid == typeof(Office.IRibbonExtensibility).GUID || e.Guid == dispatchGuid)

{

Debug.Assert(e.Service == null, "something else is already providing this service");

if (e.Service == null)

e.Service = this as Office.IRibbonExtensibility;

}

}

VB

Public Sub New()

AddHandler Globals.ThisApplication.ServiceRequest, AddressOf Me.ThisApplication\_ServiceRequest

End Sub

Sub ThisApplication\_ServiceRequest(ByVal sender As Object, ByVal e As Microsoft.Office.Tools.ServiceRequestEventArgs)

Dim dispatchGuid As Guid = New Guid("{00020400-0000-0000-C000-000000000046}")

If (e.Guid = GetType(Office.IRibbonExtensibility).GUID Or e.Guid = dispatchGuid) Then

Debug.Assert(e.Service Is Nothing, "something else is already providing this service")

If (e.Service Is Nothing) Then

e.Service = CType(Me, Office.IRibbonExtensibility)

End If

End If

End Sub

1. The GetCustomUI method is the only method defined on the IRibbonExtensibility interface. Its purpose is to return the XML string for the ribbon customization markup back to Office when Office calls it. By default, in a Visual Studio Tools for Office implementation, this XML string is an embedded resource. Office does not impose any constraints on this, so you could if you wanted to, get this XML string from anywhere that is appropriate to your scenario – it doesn’t need to be an embedded resource. You could even generate the XML dynamically if that makes sense in your solution. However, embedding it as a resource is likely to be a common approach.

C#

public string GetCustomUI(string ribbonID)

{

return GetResourceText("Ribbon1.xml");

}

VB

Public Function GetCustomUI(ByVal ribbonID As String) As String Implements Office.IRibbonExtensibility.GetCustomUI

Return GetResourceText("Ribbon1.xml")

End Function

1. The OnLoad and OnToggleButton methods are not defined on the IRibbonExtensibility interface. However, IRibbonExtensibility is a dispatch interface, and dispatch interfaces can have any number of additional methods that are discoverable at runtime (through late binding). Therefore, OnLoad and OnToggleButton are not predefined – your ribbon customization might not need such methods at all. Nonetheless, whatever additional methods you provide must satisfy two conditions:
   1. They must match the reference in the XML string.
   2. Each one must match a specific signature, depending on how you declare that it will be used.
2. If you refer back to the ribbon XML above, you will see that the starter code generated by the Add Ribbon Item wizard declares the OnLoad to be called when the entire custom ribbon markup is loaded. For Office to call this method successfully, it must be a public method with a void return value, and it must take an IRibbonUI parameter.

C#

public void OnLoad(Office.IRibbonUI ribbonUI)

{

this.ribbon = ribbonUI;

}

VB

Public Sub OnLoad(ByVal ribbonUI As Office.IRibbonUI)

Me.ribbon = ribbonUI

End Sub

1. Similarly, the markup declares that the OnTogglebutton should be called when the user clicks the togglebutton. The callback for a togglebutton must be a public method with a void return, and must take two parameters: an IRibbonControl reference and a bool reference. In this example, the method is implemented to display a message box with one of two strings, depending on the current state of the togglebutton.

C#

public void OnToggleButton(Office.IRibbonControl control, bool isPressed)

{

if (isPressed == true)

MessageBox.Show("Hello World!");

else

MessageBox.Show("Released!");

}

VB

Public Sub OnToggleButton(ByVal control As Office.IRibbonControl, ByVal isPressed As Boolean)

If (isPressed) Then

MessageBox.Show("Hello World!")

Else

MessageBox.Show("Released!")

End If

End Sub

1. At the bottom of the class is the definition of a helper method, called GetResourceText. All this method does is parse the embedded resources in the current assembly to extract a text resource specified by name. In this case, this is used to extract the custom markup string. This is a custom method generated by the Add Ribbon Item wizard. It is not defined in any interface, and you could remove it if you wanted to get your XML string from somewhere other than your assembly resources.

C#

public static string GetResourceText(string resourceName)

{

Assembly asm = Assembly.GetExecutingAssembly();

string[] resources = asm.GetManifestResourceNames();

foreach (string resource in resources)

{

if (resource.EndsWith(resourceName))

{

System.IO.StreamReader resourceReader =

new System.IO.StreamReader(asm.GetManifestResourceStream(resource));

if (resourceReader != null)

{

return resourceReader.ReadToEnd();

}

}

}

return null;

}

}

VB

Public Shared Function GetResourceText(ByVal resourceName As String) As String

Dim asm As Assembly = Assembly.GetExecutingAssembly()

Dim resources As String() = asm.GetManifestResourceNames()

For Each resource As String In resources

If resource.EndsWith(resourceName) Then

Dim resourceReader As System.IO.StreamReader

resourceReader = New System.IO.StreamReader(asm.GetManifestResourceStream(resource))

If Not resourceReader Is Nothing Then

Return resourceReader.ReadToEnd()

End If

End If

Next

Return Nothing

End Function

1. One of the benefits of a Visual Studio Tools for Office add-in is that all this ribbon markup and hookup code is generated for you. All you have to do is instantiate the Ribbon1 class. To do this, add code to the Application class to use this new ribbon. In the ThisApplication.cs or ThisApplication.vb, add a field declaration for a Ribbon1 reference, and instantiate this in the ThisApplication\_Startup method.

C#

public partial class ThisApplication

{

private Ribbon1 ribbon;

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

ribbon = new Ribbon1();

}

VB

public class ThisApplication

Private ribbon As Ribbon1

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

ribbon = New Ribbon1()

End Sub

1. Press Ctrl-F5 to build and execute the add-in in the host application. When Word starts, you will see your custom tab, labeled “.NET Add-in Tab”. Select this tab to see the custom group. Hover over the custom button to see the tooltip. Then, click the button to invoke the message box. As this is a togglebutton, when you click it once, the message is “Hello World”. When you click it again, the message is “Released”.

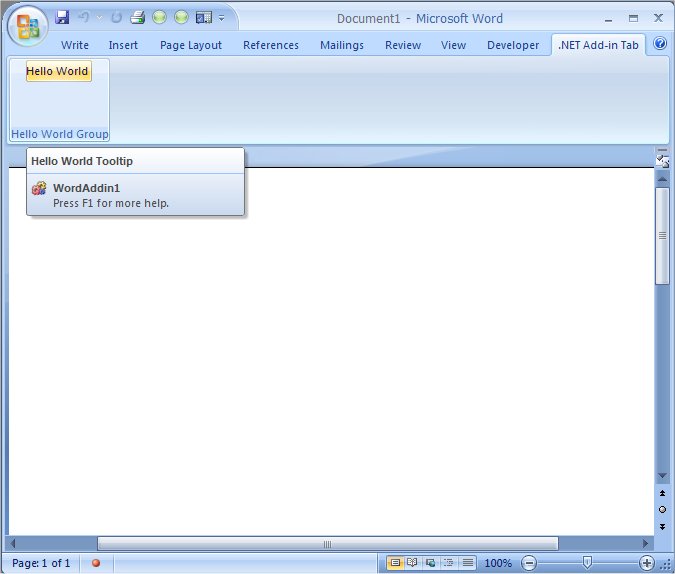


Figure A Visual Studio Tools for Office add-in for Word, with a custom ribbon

1. Close Word and return to Visual Studio. Creating a basic custom ribbon with Visual Studio Tools for Office is clearly trivial, as it provides all the code you need to get started. Typically, of course, you would want to specify additional controls and more sophisticated behavior. Let’s do this now.
2. The first thing to do is to add a custom icon to the button. Visual Studio 2005 shipped with a zip file of standard icons and bitmaps, and we can use one of these in this project. In Windows Explorer, navigate to the location where you installed Visual Studio. Typically, this is %ProgramFiles%Microsoft Visual Studio 8. Then go to the Common7 sub-directory. There you will see a zip file called VS2005ImageLibrary. Unzip this to some suitable location.
3. Back in Visual Studio, in Solution Explorer, right-click the project node and select Properties. In the Properties window, select the Resources tab. This will open the resources designer. From the menu at the top, drop down the Add Resource list and select Add Existing File.

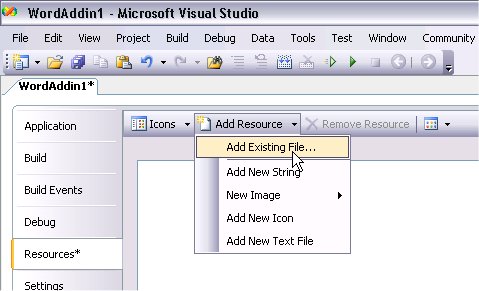


Figure Adding an existing resource file to the project

1. Then, navigate to the place where you unzipped the image library, and drill down to .\icons\WinXP. Find a suitable icon. For the purposes of this exercise, select the blankcd.ico file, and click Open.

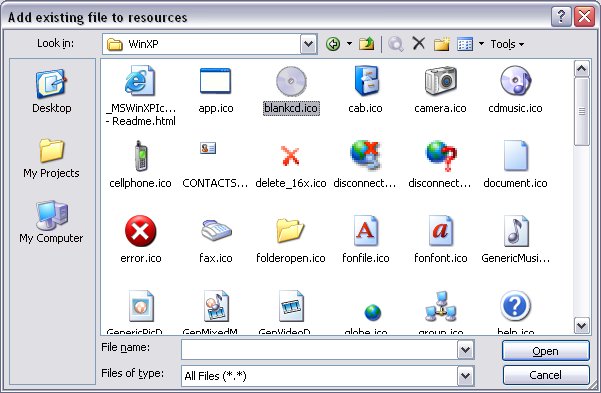


Figure Add an icon resource

1. This adds the icon to your resources. Next, expand out the Resources node in Solution Explorer and select the blankcd.ico node. In the properties grid, set the Build Action for this to be Embedded Resource.
2. Add a new helper method to the Ribbon1 class to extract bitmap resources from the assembly. Although this add-in only has one icon resource, you can easily make this method generic enough to cope with other image formats.

C#

public static System.Drawing.Bitmap GetResourceBitmap(string resourceName)

{

Assembly asm = Assembly.GetExecutingAssembly();

string[] resources = asm.GetManifestResourceNames();

System.Drawing.Bitmap image = null;

foreach (string resource in resources)

{

if (resource.EndsWith(resourceName))

{

System.IO.Stream stream =   
 asm.GetManifestResourceStream(resource);

if (stream == null)

{

break;

}

string extension =   
 System.IO.Path.GetExtension(resourceName).ToLower();

switch (extension)

{

case ".ico":

image = new System.Drawing.Icon(stream).ToBitmap();

break;

case ".jpg":

case ".bmp":

default:

image = new System.Drawing.Bitmap(stream);

image.MakeTransparent();

break;

}

stream.Close();

break;

}

}

return image;

}

VB

Public Shared Function GetResourceBitmap( \_

ByVal resourceName As String) As System.Drawing.Bitmap

Dim asm As Assembly

Dim resources As String()

Dim image As System.Drawing.Bitmap

asm = Assembly.GetExecutingAssembly()

resources = asm.GetManifestResourceNames()

image = Nothing

For Each resource As String In resources

If (resource.EndsWith(resourceName)) Then

Dim stream As System.IO.Stream

stream = asm.GetManifestResourceStream(resource)

If (stream Is Nothing) Then

Exit For

End If

Dim extension As String

extension = System.IO.Path.GetExtension(resourceName).ToLower()

Select Case extension

Case ".ico"

image = New System.Drawing.Icon(stream).ToBitmap()

Case ".jpg"

Case ".bmp"

Case Else

image = New System.Drawing.Bitmap(stream)

image.MakeTransparent()

End Select

stream.Close()

Exit For

End If

Next

Return image

End Function

1. Extracting the bitmap is only part of the problem. One of the interesting aspects of building managed solutions based on Office is that you sometimes have to deal with COM types directly. The ribbon is a case in point. The traditional Office CommandBarButton class uses IPictureDisp objects for its images. The ribbon also uses IPictureDisp for the same purpose. The problem is how to get a COM IPictureDisp object from a managed Image or Bitmap object.
2. There are a couple of alternative here. One approach is to write a cutom class to implement IPictureDisp directly, but the simplest approach is to use the AxHost class defined in System.Windows.Forms. This class is used by the AxImp tool to wrap ActiveX controls and expose them as Windows Forms controls. Crucially, for our purposes, it offers a method called GetIPictureDispFromPicture. This method takes in an Image and converts it to an IPictureDisp. So, we'll use AxHost to convert our icon resource into an IPictureDisp so that we can assign it to the button.
3. IPictureDisp is an interface defined in the stdole type library and interop assembly. You'll find the stdole interop assembly is already added as a reference in the project.
4. The next issue is that the GetIPictureDispFromPicture method is a protected static method in the AxHost class. Being static is not a problem, but being protected means that we can only access it from a class derived from AxHost. So, what we have to do is write a class that derives from AxHost – we can then write a custom method that internally calls GetIPictureDispFromPicture. You could declare this class in a separate file, but for convenience, just declare it in the Ribbon1.cs (or Ribbon1.vb) file. Be sure to intialize the base class (that is, AxHost) with null – this is where you'd specify the ActiveX control if you were using AxHost to wrap an ActiveX control.

C#

internal class ImageConverter : System.Windows.Forms.AxHost

{

private ImageConverter() : base(null)

{

}

public static stdole.IPictureDisp Convert(System.Drawing.Image image)

{

return (stdole.IPictureDisp)

AxHost.GetIPictureDispFromPicture(image);

}

}

VB

Friend Class ImageConverter

Inherits System.Windows.Forms.AxHost

Sub New()

MyBase.New(Nothing)

End Sub

Public Shared Function Convert(ByVal image As System.Drawing.Image) As stdole.IPictureDisp

Return AxHost.GetIPictureDispFromPicture(image)

End Function

End Class

Of course, the inconvenience of this code, converting from managed Images to IPictureDisps, is one of the things that later Visual Studio Tools for Office CTP releases are planned to address.

1. Now that you have set up the image-converter class, you can go ahead and write the callback method that Office will use to get this image. This method must take an IRibbonControl parameter and return an IPictureDisp.

C#

public stdole.IPictureDisp GetImage(Office.IRibbonControl control)

{

stdole.IPictureDisp pictureDisp = null;

switch (control.Id)

{

case "toggleButton":

pictureDisp =   
 ImageConverter.Convert(GetResourceBitmap("blankcd.ico"));

break;

}

return pictureDisp;

}

VB

Public Function GetImage(ByVal control As Office.IRibbonControl) As stdole.IPictureDisp

Dim pictureDisp As stdole.IPictureDisp

pictureDisp = Nothing

Select Case control.Id

Case "toggleButton"

pictureDisp = ImageConverter.Convert(GetResourceBitmap("blankcd.ico"))

End Select

Return pictureDisp

End Function

1. To match this, update the custom XML to specify this callback. At the same time, make the image large.

<toggleButton id="toggleButton" label="Hello World"

tooltip="Hello World Tooltip" onAction="OnToggleButton"

size="large" getImage="GetImage"/>

1. Press Ctrl-F5 to build the project and execute Word. You should see your new button image.

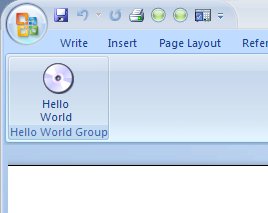


Figure A custom ribbon button with an image

1. In this exercise, you created a simple Visual Studio Tools for Office add-in with a custom ribbon. The first version used the starter code generated by the Visual Studio Tools for Office ribbon wizard. The second version built on this to add a custom image.

#### Exercise 4: Synchronizing Ribbon and Task Pane

One of the guiding principles of the new Office 2007 user interface is that the user should always be in control. Developers are encouraged to build solutions which include features that allow the user to determine what visual artifacts are displayed. One specific example is that the user should be able to make any custom task pane visible or invisible whenever they want. This is particularly important if you consider that any add-in can create any number of custom task panes. If all of them were visible all at the same time, there would be very little application real estate left.

In this exercise, you will create an add-in with a custom ribbon and custom task pane. The ribbon will include a togglebutton that toggles the visibility of the task pane.

1. Start Visual Studio 2005 (with the Visual Studio Tools for Office ‘v3’ CTP installed). From the File menu, point to New, and then click Project. This displays the New Project dialog box. Expand out the tree of Project types to find the programming language you want to work in (C# or VB). Then select the Office node.
2. In the Templates pane, select the add-in template for the Office application you want to target. In this walkthrough, we’ll select Access.
3. Specify a suitable name and location for your project, and click OK. In this example, we’ll accept the default name of AccessAddin1. This will generate all the baseline code necessary for your add-in.
4. In Solution Explorer, right-click the project, point to Add, then click New Item. This displays the Add New Item dialog box. Select the Ribbon item. You can type in any suitable name you want for this ribbon. In this exercise, accept the default “Ribbon1” and click Add.
5. This generates a new class, called Ribbon1 in the file Ribbon1.cs (or Ribbon1.vb), and an XML file called Ribbon1.xml.
6. Declare a Ribbon1 field in the ThisApplication class, and instantiate it in ThisApplication\_Startup.

C#

public partial class ThisApplication

{

private Ribbon1 ribbon;

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

ribbon = new Ribbon1();

}

VB

public class ThisApplication

Private ribbon As Ribbon1

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

ribbon = New Ribbon1()

End Sub

1. At this point, it is worth testing the add-in before making any changes. So, Press Ctrl-F5 to build the add-in and run Access.

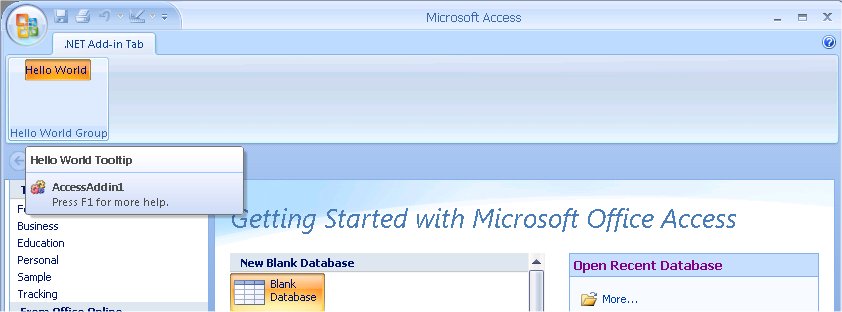


Figure A custom ribbon in Access

1. Close Access and return to Visual Studio. Add a simple custom task pane. You can either create a UserControl in the designer, or create one programmatically. In an earlier exercise, you saw how to use the designer to create a UserControl for a task pane. In this exercise, we’ll create one programmatically. First, in the Application.cs (or Application.vb), add a statement to use the Microsoft.Office.Tools namespace.

C#

using Microsoft.Office.Tools;

VB

Imports Microsoft.Office.Tools

1. Next, declare a CustomTaskPane reference as an internal field in the Application class. We want this to be internal or public because later on you will add code to access this field from the ribbon class.

C#

public partial class ThisApplication

{

private Ribbon1 ribbon;

internal CustomTaskPane ctp;

VB

public class ThisApplication

Private ribbon As Ribbon1

Public ctp As CustomTaskPane

Private WithEvents b As Button

1. Add code to ThisApplication\_Startup to programmatically create a UserControl and use it to create a custom task pane. The UserControl will have a Button on it, and you can create a Click event handler for this that simply displays a message box. Note that Visual Studio Tools for Office 2005 allows you to add managed controls (including standard Windows Forms controls such as Button controls) directly to the document-level ActionsPane. In the first CTP release, the application-level task pane is different in that you must add any controls to a UserControl first, and then add that UserControl to the task pane.

C#

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

ribbon = new Ribbon1();

// Create user control.

UserControl uc = new UserControl();

Button b = new Button();

b.Text = "MyButton";

b.Click += new EventHandler(b\_Click);

uc.Controls.Add(b);

// Create the task pane.

ctp = this.CustomTaskPanes.Add(uc, "My TaskPane");

ctp.Visible = true;

}

void b\_Click(object sender, EventArgs e)

{

MessageBox.Show("Hello");

}

VB

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

Dim uc As UserControl

ribbon = New Ribbon1()

uc = New UserControl()

b = New Button()

b.Text = "MyButton"

uc.Controls.Add(b)

ctp = Me.CustomTaskPanes.Add(uc, "My TaskPane")

ctp.Visible = True

End Sub

Private Sub b\_Click(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) Handles b.Click

MessageBox.Show("Hello")

End Sub

1. Again, Press Ctrl-F5 to build and test this. Both the custom ribbon and custom task pane should be visible and enabled in Access.

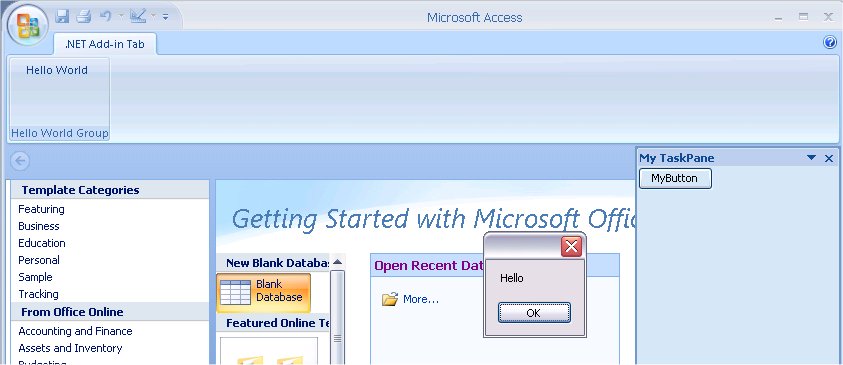


Figure Custom ribbon and task pane in Access

1. The remaining piece of the puzzle is to synchronize the ribbon button and the task pane. In the Ribbon1 class, find the OnToggleButton callback, and replace the implementation with code to toggle the visibility of the task pane.

C#

public void OnToggleButton(Office.IRibbonControl control, bool isPressed)

{

Globals.ThisApplication.ctp.Visible = !isPressed;

}

VB

Public Sub OnToggleButton( \_

ByVal control As Office.IRibbonControl, \_

ByVal isPressed As Boolean)

Globals.ThisApplication.ctp.Visible = Not isPressed

End Sub

1. It would probably also be reasonable to update the markup for the togglebutton to more accurately represent its purpose.

<toggleButton id="toggleButton"

label="Task Pane"

tooltip="Toggles the visibility of the custom task pane"

onAction="OnToggleButton" />

1. Press Ctrl-F5 to build and test again. This time, when you click the button in the ribbon, it will toggle the visibility of the task pane.
2. In this exercise, you created an add-in for Access, with a custom ribbon and custom task pane. You implemented a button in the ribbon to toggle the visibility of the task pane, in accordance with user interface guidelines. Clearly, in a more complex scenario, you might have multiple task panes, and you would then need to keep track of which one(s) you want to toggle at any given time. You have flexibility to make these decisions based on the context of the application itself.

#### Exercise 5: Multiple Ribbons

Because Outlook supports multiple ribbons, it would be unacceptable for all the markup for these ribbons to be loaded at application startup, as is the case with the other Office applications. Instead, Office supports delayed loading of ribbon markup specifically for Outlook inspectors. Whenever the first instance of an Inspector loads for a given RibbonID, Office calls GetCustomUI, passing the RibbonID (string) in the call. The table below lists the RibbonIDs and the corresponding Outlook MessageClasses:

|  |  |
| --- | --- |
| *RibbonID* | *MessageClass* |
| Microsoft.Outlook.Mail.Read | IPM.Note.\* |
| Microsoft.Outlook.Mail.Compose | IPM.Note.\* |
| Microsoft.Outlook.MeetingRequest.Read | IPM.Schedule.Meeting.Request or IPM.Schedule.Meeting.Canceled |
| Microsoft.Outlook.MeetingRequest.Send | IPM.Schedule.Meeting.Request |
| Microsoft.Outlook.Appointment | IPM.Appointment.\* |
| Microsoft.Outlook.Contact | IPM.Contact.\* |
| Microsoft.Outlook.Journal | IPM.Activity.\* |
| Microsoft.Outlook.Task | IPM.Task.\* and IPM.TaskRequest.\* |
| Microsoft.Outlook.DistributionList | IPM.DistList.\* |
| Microsoft.Outlook.Report | IPM.Report.\* |
| Microsoft.Outlook.Resend | IPM.Resend.\* |
| Microsoft.Outlook.Response.Read | IPM.Schedule.Meeting.Resp.\* |
| Microsoft.Outlook.Response.Compose | IPM.Schedule.Meeting.Resp.\* |
| Microsoft.Outlook.Response.CounterPropose | IPM.Schedule.Meeting.Resp.\* |
| Microsoft.Outlook.RSS | IPM.Post.Rss |
| Microsoft.Outlook.Post.Read | IPM.Post.\* |
| Microsoft.Outlook.Post.Compose | IPM.Post.\* |
| Microsoft.Outlook.Sharing.Read | IPM.Sharing.\* |
| Microsoft.Outlook.Sharing.Compose | IPM.Sharing.\* |

In this exercise, you will build an Outlook add-in with two custom ribbons. These ribbons will be used in different inspector windows, specifically, the Mail.Compose inspector and the Appointment inspector. The following steps specify fully how to create this project. If you need detailed descriptions of the starter code generated by the Add Ribbon Item wizard, you can refer to the instructions in the previous exercise.

1. Start Visual Studio 2005 (with the Visual Studio Tools for Office ‘v3’ CTP installed). From the File menu, point to New, and then click Project. This displays the New Project dialog box. Expand out the tree of Project types to find the programming language you want to work in (C# or VB). Then select the Office node.
2. In the Templates panel, select Outlook Add-in.
3. Specify a suitable name and location for your project, and click OK. In this example, we’ll accept the default name of OutlookAddin1. This will generate all the baseline code necessary for your add-in.
4. In Solution Explorer, right-click the project, point to select Add, and then click New Item. This displays the Add New Item dialog box. Select the Ribbon item. You can type in any suitable name you want for this ribbon. In this exercise, accept the default “Ribbon1” and click Add. This generates a new class, called Ribbon1 in the file Ribbon1.cs (or Ribbon1.vb), and an XML file called Ribbon1.xml.
5. Rename the XML file to RibbonMailCompose.xml, and modify the markup to make the labels and IDs more relevant to the Mail.Compose inspector

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

<ribbon startFromScratch="false">

<tabs>

<tab id="TabMailCompose" label="Mail Compose Tab" visible="1">

<group id="GroupMailCompose" label="Mail Compose Group" visible="1">

<toggleButton id="ButtonMailCompose" label="Mail Compose Button" tooltip="Mail Compose Tooltip"

size="large" getImage="GetImage" onAction="OnToggleButton" />

</group>

</tab>

</tabs>

</ribbon>

</customUI>

1. Add a second XML file – don’t add a new Ribbon item because you don’t need another class, just another XML file. To do this, right-click the project in Solution Explorer, point to Add, then click New Item. This displays the Add New Item dialog box. Select the XML File item. Name the file RibbonAppointment.xml.
2. Edit the RibbonAppointment.xml file to provide markup relevant for the Appointment inspector.

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

<ribbon startFromScratch="false">

<tabs>

<tab id="TabAppointment" label="Appointment Tab" visible="1">

<group id="GroupAppointment" label="Appointment Group" visible="1">

<toggleButton id="ButtonAppointment" label="Appointment Button" tooltip="Appointment Tooltip"

size="large" getImage="GetImage" onAction="OnToggleButton" />

</group>

</tab>

</tabs>

</ribbon>

</customUI>

1. Select RibbonAppointment.xml in Solution Explorer and set the Build Action property to Embedded Resource. Note that this step was done for you for the first ribbon XML file that was generated by the Add Ribbon Item wizard.
2. Update the GetCustomUI method to test the value of the ribbon ID string parameter passed in, and return the XML string resource that corresponds to this value.

C#

public string GetCustomUI(string ribbonId)

{

string xmlMarkup = String.Empty;

switch (ribbonId)

{

case "Microsoft.Outlook.Mail.Compose":

xmlMarkup = GetResourceText("RibbonMailCompose.xml");

break;

case "Microsoft.Outlook.Appointment":

xmlMarkup = GetResourceText("RibbonAppointment.xml");

break;

}

return xmlMarkup;

}

VB

Public Function GetCustomUI(ByVal ribbonID As String) \_

As String Implements Office.IRibbonExtensibility.GetCustomUI

Dim xmlMarkup As String

xmlMarkup = String.Empty

Select Case ribbonID

Case "Microsoft.Outlook.Mail.Compose"

xmlMarkup = GetResourceText("RibbonMailCompose.xml")

Case "Microsoft.Outlook.Appointment"

xmlMarkup = GetResourceText("RibbonAppointment.xml")

End Select

Return xmlMarkup

End Function

1. Declare a Ribbon1 field in the ThisApplication class, and instantiate it in the ThisApplication\_Startup method.

C#

public partial class ThisApplication

{

private Ribbon1 ribbon;

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

ribbon = new Ribbon1();

}

VB

Public Class ThisApplication

Private ribbon As Ribbon1

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

ribbon = New Ribbon1()

End Sub

1. Press Ctrl-F5 to test the add-in. In Outlook, create a new Mail item. This will display the Mail.Compose inspector, with the custom ribbon.

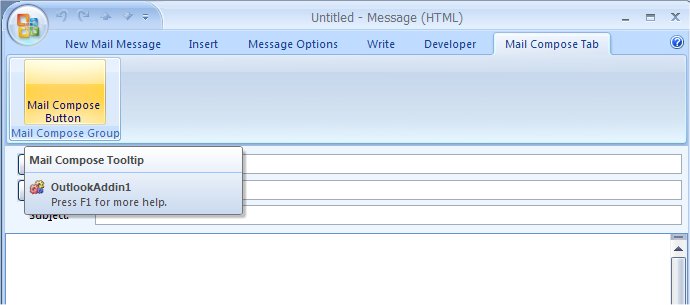


Figure Custom ribbon in the Outlook Mail.Compose inspector

1. Stay in Outlook, and create a new Appointment item to see the custom Appointment inspector ribbon.

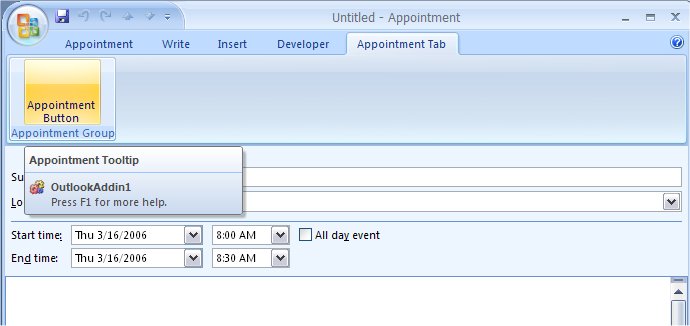


Figure Custom ribbon in the Outlook Appointment inspector

1. Close Outlook and return to Visual Studio.

As a minor enhancement, it would be useful to see how to set up your custom ribbon where one or more of the controls are either visible or enabled dependent on some runtime context. You can change the visible or enabled status of a control at runtime as a result of some user action, or some context in the application, or perhaps conditional upon the user’s role, and so on.

We’ll continue with the OutlookAddin1 project, add a second button to the Appointment ribbon, and set it up so that this button is only enabled if the appointment is on a Friday.

1. Add a button to the markup, specifying a callback for the getEnabled attribute.

<button id="FridayButton" label="Friday Button" tooltip="Friday Tooltip" size="large" onAction="OnFridayButtonClick" getEnabled="GetEnabled" />

1. The plan is to implement the GetEnabled callback method to test if the appointment is on a Friday. To do this, the ribbon code needs to work with the Outlook object model. So, first, add a statement at the top of the Ribbon1.cs (or Ribbon1.vb) file to declare the Outlook PIA namespace.

C#

using Outlook = Microsoft.Office.Interop.Outlook;

VB

Imports Outlook = Microsoft.Office.Interop.Outlook

1. Now implement the GetEnabled callback method. If the appointment is on a Friday, enable the button, otherwise disable it. When it calls your GetEnabled method, Office will pass in a reference to the specific control that is being modified. This control exposes a Context property, which in this case will be the Outlook appointment inspector. From the inspector, you can get to the currrent appointment item.

C#

public bool GetEnabled(Office.IRibbonControl control)

{

Outlook.Inspector inspector = (Outlook.Inspector)control.Context;

Outlook.AppointmentItem appointment = (Outlook.AppointmentItem)inspector.CurrentItem;

if (appointment.Start.DayOfWeek == DayOfWeek.Friday)

{

return true;

}

return false;

}

VB

Public Function GetEnabled(ByVal control As Office.IRibbonControl) As Boolean

Dim inspector As Outlook.Inspector

Dim appointment As Outlook.AppointmentItem

inspector = control.Context

appointment = inspector.CurrentItem

If (appointment.Start.DayOfWeek = DayOfWeek.Friday) Then

Return True

End If

Return False

End Function

1. Press Ctrl-F5 to build the add-in and execute Outlook. If you create a new appointment on a Friday, you should see the new button. Clearly, this is a very trivial condition (and for consistency, it would probably also make sense to add code to check when the user changes the initial Start property on the AppointmentItem from a Friday to another day). However simple this is, the exercise does serve to illustrate how to make a ribbon control conditionally enabled at runtime.

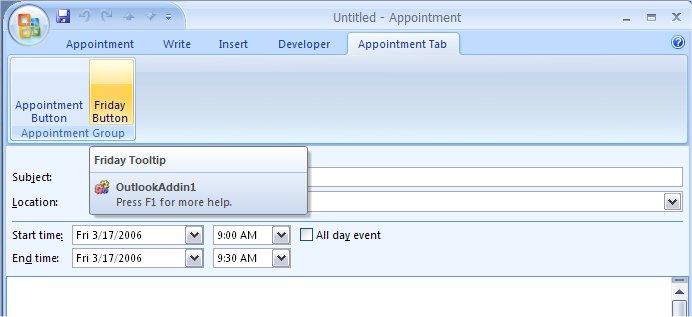


Figure Conditionally-enabled button in a custom ribbon

1. In this exercise, you created a simple Outlook add-in with multiple custom ribbons, and implemented conditional enabling of a ribbon button, dependent on some context in the host application.

#### Exercise 6: Adding Custom Form Regions

Outlook 2007 introduces a new UI extensibility mechanism named Form Regions. Solution developers can customize built-in Outlook forms, by adding custom regions to an existing Outlook form, by replacing the first tab page of the built-in form, or by completely replacing all of the Outlook form.

Unlike legacy custom forms, new form regions are installed locally and registered through the Windows registry. Form regions can be installed stand-alone or can be driven by a COM add-in (including Visual Studio Tools for Office add-ins).

There are three parts to a custom form region that is implemented in an add-in:

* An OFS (Outlook Form Storage) file that contains the layout of the form and its controls, produced as output by the Outlook forms designer.
* An XML file that contains manifest information about the form region, that tells Outlook how to load the form region and provides display names and localization information for the form region.
* Add-in code that implements the behavior of the controls, the response to user interaction, etc.

In this exercise, you will create a custom form region in Outlook to produce the OFS file. Then, use this OFS as a resource in an add-in, and write code to connect the two together. Note that this CTP release of Visual Studio Tools for Office does not provide a design-time experience for custom form regions, so this is one example where you need to enter much of the code manually. The basic steps are:

* Create a baseline add-in in Visual Studio.
* Create the OFS in Outlook.
* Add the OFS as a resource in the add-in.
* Implement the FormRegionStartup interface in your add-in.
* Write code to connect the form region controls with the add-in.
* Create and register an XML manifest for the form region.

1. Start Visual Studio 2005 (with the Visual Studio Tools for Office ‘v3’ CTP installed). From the File menu, point to New, and then click Project. This displays the New Project dialog box. Expand out the tree of Project types to find the programming language you want to work in (C# or VB). Then select the Office node.
2. In the Templates panel, select Outlook Add-in.
3. Specify a suitable name and location for your project, and click OK. In this example, we’ll accept the default name of OutlookAddin2. This will generate all the baseline code necessary for your add-in.
4. Now create the OFS file in Outlook. First, make sure that you have the Developer tab available in Outlook. To do this, start Outlook and open any inspector so that you can see the ribbon. (To open an inspector, open any mail, calendar, contact, task item etc). From the ribbon, go to the File menu, and select Editor Options. Select the Personalize item in the navigation bar, and check the Show Developer tab in the Ribbon option.

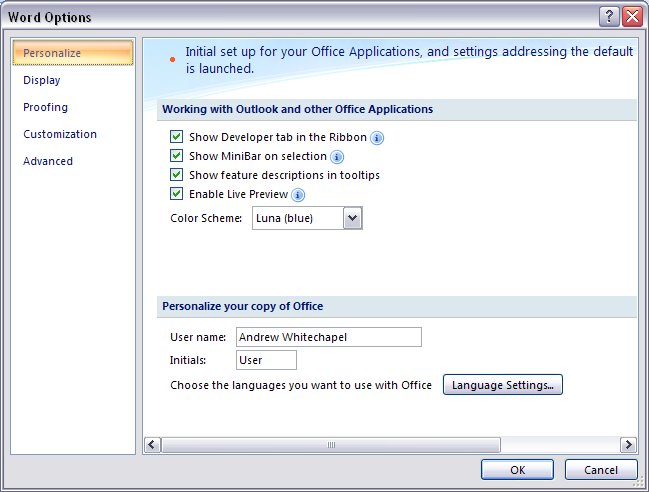


Figure Enabling the Developer tab in the ribbon

1. To create the OFS file, select the Tools menu in Outlook, then select Forms, then Design a Form. This displays the Design Form dialog box, from which you can select the type of form you want to design. In this exercise, select Task and click Open.
2. This displays the standard forms designer for the Task form. To design a form region, click the Form Region control on the ribbon, and then click New Form Region.

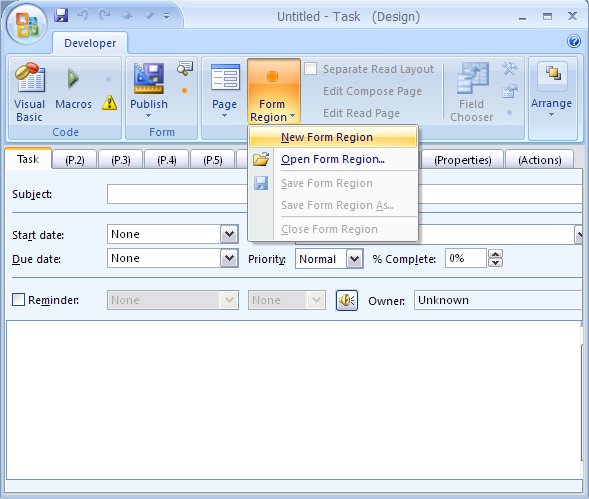


Figure Opening the form region designer

1. When the forms designer opens, click the Control Toolbox button to display the control toolbox, so that you can add controls to the form region. You can also add field controls by dragging fields from the field chooser to the form. For the purposes of this exercise, we’ll just add one of the standard fields and one standard control.
2. For the standard field, drag and drop the To-Do Title field onto the form. Also, drag and drop a command button control onto the form. By default, these controls will have the IDs OlkTextBox1 and CommandButton1. Realistically, you would change these, but for now just leave them as the defaults.

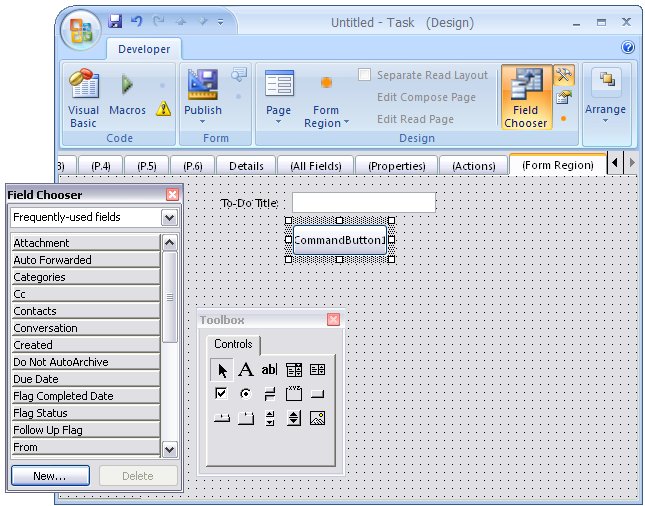


Figure A custom form region in the designer

1. When you have finished editing the form region, save it to an OFS file. To do this, click the Form Region drop-down box, and select Save Form Region As. Specify a suitable name and location. For this exercise, use the name TaskFormRegion.ofs and save it to the project folder of your add-in. Close Outlook and return to Visual Studio.
2. In your add-in project, add the OFS file as a resource. To do this, in Solution Explorer, right-click on the project node and click Properties. In the Properties window, select the Resources tab. This opens the resources designer. Drop down the Add Resource list and click Add Existing File.

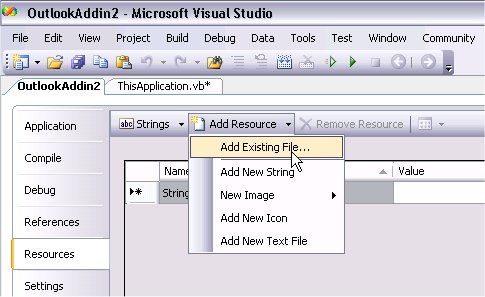


Figure Adding a file to the add-in resources

1. Navigate to the TaskFormRegion.ofs and click Open. This adds the OFS file to your add-in resources.

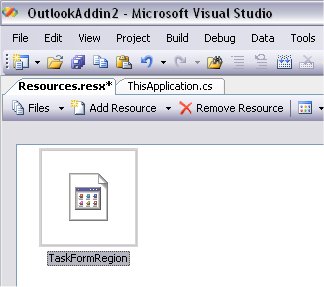


Figure An OFS file listed as an embedded resource

1. Note that you don’t have to embed the OFS file as a resource if you don’t want to. When Office calls your implementation of the GetFormRegionStorage, you have the option to return the path to and external OFS file, if you prefer. However, embedding it as a resource is likely to be a common scenario.
2. Now to implement the FormRegionStartup interface. The model here is similar to the other new extensibility interfaces (IRibbonExtensibility and ICustomTaskPaneConsumer) in Office 2007. For this CTP release, however, there is no additional Visual Studio Tools for Office support for FormRegionStartup – this support is planned for a later CTP release of Visual Studio Tools for Office ‘v3’.
3. First, add a reference to the Microsoft.Vbe.Interop.Forms interop assembly – this is listed on the .NET tab of the Add Reference dialog. This provides the definitions of the UserForm that represents the Outlook form.
4. Create a new class, calling it FormRegionHookup. At the top of the file, declare namespace using statement for System.Runtime.InteropServices (needed for the ComVisible attribute), Microsoft.Office.Interop.Outlook (the Outlook PIA), Microsoft.Vbe.Interop.Forms (for the definition of the UserForm class) and System.Windows.Forms (for the System.Windows.Forms.Application class).

C#

using System;

using System.Runtime.InteropServices;

using Outlook = Microsoft.Office.Interop.Outlook;

using Microsoft.Vbe.Interop.Forms;

using System.Windows.Forms;

VB

Imports System

Imports System.Runtime.InteropServices

Imports Outlook = Microsoft.Office.Interop.Outlook

Imports Microsoft.Vbe.Interop.Forms

Imports System.Windows.Forms

1. Make this class ComVisible, and give it a suitable GUID and ProgId. Also specify an autodual class interface. These attributes are all required in order for Outlook to connect correctly to this class. Specify that this class implements FormRegionStartup.

C#

[ComVisible(true),

Guid("88F7BFBE-7666-4a0c-BCFD-2740E6625E04"),

ProgId("OutlookAddin2.FormRegionHookup"),

ClassInterface(ClassInterfaceType.AutoDual)]

public class FormRegionHookup : Outlook.FormRegionStartup

{

VB

<ComVisible(True), \_

Guid("88F7BFBE-7666-4a0c-BCFD-2740E6625E04"), \_

ProgId("OutlookAddin2.FormRegionHookup"), \_

ClassInterface(ClassInterfaceType.AutoDual)> \_

Public Class FormRegionHookup

Implements Outlook.FormRegionStartup

1. Before looking at how to implement the interface, declare some fields for use with the form and the controls on the form.

C#

private object Item;

private Outlook.FormRegion FormRegion;

private UserForm UserForm;

private Outlook.OlkTextBox OlkTextBox1;

private Outlook.OlkCommandButton CommandButton1;

VB

Private Item As Object

Private FormRegion As Outlook.FormRegion

Private userForm As UserForm

Private OlkTextBox1 As Outlook.OlkTextBox

Private WithEvents CommandButton1 As Outlook.OlkCommandButton

1. The FormRegionStartup interface has two methods, GetFormRegionStorage and BeforeFormRegionShow. Outlook calls GetFormRegionStorage to retrieve the OFS. So, implement this method to extract the OFS from the add-in’s resources and return it as a byte array.

C#

public object GetFormRegionStorage(string FormRegionName, object Item, int LCID,

Outlook.OlFormRegionMode FormRegionMode, ref Outlook.OlFormRegionSize FormRegionSize)

{

Application.DoEvents();

switch (FormRegionName)

{

case "TaskFormRegion":

byte[] ofsBytes = Properties.Resources.TaskFormRegion;

return ofsBytes;

default:

return null;

}

}

VB

Public Function GetFormRegionStorage( \_

ByVal FormRegionName As String, \_

ByVal Item As Object, \_

ByVal LCID As Integer, \_

ByVal FormRegionMode As Outlook.OlFormRegionMode, \_

ByRef FormRegionSize As Outlook.OlFormRegionSize) As Object \_

Implements Outlook.FormRegionStartup.GetFormRegionStorage

Application.DoEvents()

Select Case FormRegionName

Case "TaskFormRegion"

Dim ofsBytes As Byte()

ofsBytes = My.Resources.TaskFormRegion

Return ofsBytes

Case Else

Return Nothing

End Select

End Function

1. Outlook calls BeforeFormRegionShow just before it shows the custom form region. You can think of this as an opportunity to initialize your code behind the form region. So, implement this to map the controls on the form to fields in your class. To do this, you index into the Item collection of controls on the form, specifying the ID of the control you want as the indexer.

C#

public void BeforeFormRegionShow(object Item, Outlook.FormRegion FormRegion)

{

this.Item = Item;

this.FormRegion = FormRegion;

this.UserForm = FormRegion.Form as UserForm;

try

{

OlkTextBox1 = UserForm.Controls.Item("OlkTextBox1") as Outlook.OlkTextBox;

CommandButton1 = UserForm.Controls.Item("CommandButton1") as Outlook.OlkCommandButton;

CommandButton1.Click += new Outlook.OlkCommandButtonEvents\_ClickEventHandler(CommandButton1\_Click);

}

catch (Exception ex)

{

MessageBox.Show("An error occured during InitalizeControls:\n" + ex.ToString());

}

}

VB

Public Sub BeforeFormRegionShow( \_

ByVal Item As Object, ByVal FormRegion As Outlook.FormRegion) \_

Implements Outlook.FormRegionStartup.BeforeFormRegionShow

Me.Item = Item

Me.FormRegion = FormRegion

Me.userForm = FormRegion.Form

Try

OlkTextBox1 = userForm.Controls.Item("OlkTextBox1")

CommandButton1 = userForm.Controls.Item("CommandButton1")

Catch ex As Exception

MessageBox.Show("An error occured during InitalizeControls:\n" + ex.ToString())

End Try

End Sub

1. Implement the commandbutton Click handler to display a simple message box with the contents of the textbox control. Obviously, this is not a realistic scenario, but it serves to get you started with custom form regions.

C#

void CommandButton1\_Click()

{

MessageBox.Show(OlkTextBox1.Text);

}

VB

Private Sub CommandButton1\_Click() Handles CommandButton1.Click

MessageBox.Show(OlkTextBox1.Text)

End Sub

1. Now hookup this new class in the ThisApplication class of the add-in. Declare a field in the class of type FormRegionHookup.

C#

public partial class ThisApplication

{

private FormRegionHookup formRegionHelper;

VB

public class ThisApplication

Private formRegionHelper As FormRegionHookup

1. Initialize the FormRegionHookup field in the ThisApplication\_Startup method. Also hook up the ServiceRequest event – this is one of the pieces of work that is done for you by the default Visual Studio Tools for Office implementation for ribbons and task panes, but not for form regions.

C#

private void ThisApplication\_Startup(object sender, System.EventArgs e)

{

formRegionHelper = new FormRegionHookup();

this.ServiceRequest += new EventHandler<Microsoft.Office.Tools.ServiceRequestEventArgs>(ThisApplication\_ServiceRequest);

}

VB

Private Sub ThisApplication\_Startup( \_

ByVal sender As Object, ByVal e As System.EventArgs) \_

Handles Me.Startup

formRegionHelper = New FormRegionHookup()

End Sub

1. In the event handler for the ServiceRequest, set the FormRegionHookup field into the Service property of the event argument.

C#

void ThisApplication\_ServiceRequest(object sender, Microsoft.Office.Tools.ServiceRequestEventArgs e)

{

if (e.Guid == typeof(Outlook.FormRegionStartup).GUID)

{

e.Service = formRegionHelper;

}

}

VB

Private Sub ThisApplication\_ServiceRequest( \_

ByVal sender As Object, \_

ByVal e As Microsoft.Office.Tools.ServiceRequestEventArgs) \_

Handles Me.ServiceRequest

If (e.Guid = GetType(Outlook.FormRegionStartup).GUID) Then

e.Service = formRegionHelper

End If

End Sub

1. Next, add an XML file for the form region manifest– this is required by Outlook to determine how to load the form region. For this exercise, the critical elements are the name, title, formRegionType and addin. Outlook will pass the name to your add-in when it wants the OFS information. The title is used as the display name of the form region – this will show up in the ribbon. For this exercise, use the name “Contoso”. The formRegionType specifies whether this form region appears on a separate tab, on an adjoining region or replaces the standard form. The addin element specifies the ProgID of the add-in associated with this form region.

<?xml version="1.0" encoding="utf-8"?>

<FormRegion xmlns="http://schemas.microsoft.com/office/outlook/12/formregion.xsd">

<name>TaskFormRegion</name>

<title>Contoso</title>

<formRegionType>separate</formRegionType>

<showCompose>true</showCompose>

<showRead>true</showRead>

<showPreview>false</showPreview>

<hidden>true</hidden>

<addin>OutlookAddin2</addin>

<version>1.0</version>

</FormRegion>

1. In the properties grid for the XML manifest, set the Copy to Output property value to Copy always. This will ensure that a fresh copy of the manifest is copied to the add-in target output location on every build.
2. Next, create a .REG file to register the custom form region manifest. Visual Studio Tools for Office add-in projects register the add-in by default on the developer machine, and provide a setup project so that you can deploy the registration when you deploy the add-in. In this CTP release, this registration information does not include the custom form region information. The .REG file shown below is a complete registration for this add-in, including both the entries that Visual Studio Tools for Office generates for you and the additional entries for the form region manifest. Note: you must change the path to the path to the add-in on your machine.

Windows Registry Editor Version 5.00

[HKEY\_CURRENT\_USER\Software\Microsoft\Office\Outlook\Addins\OutlookAddin2]

"CommandLineSafe"=dword:00000001

"Description"="OutlookAddin2 -- an addin created with VSTO technology"

"FriendlyName"="OutlookAddin2"

"LoadBehavior"=dword:00000003

"ManifestLocation"="C:\\Temp\\OutlookAddin2\\OutlookAddin2\\bin\\Debug\\"

"ManifestName"="OutlookAddin2.dll.manifest"

[HKEY\_CURRENT\_USER\Software\Microsoft\Office\Outlook\FormRegions\IPM.Task]

"TaskFormRegion"="C:\\Temp\\OutlookAddin2\\OutlookAddin2\\bin\\Debug\\TaskFormRegion.xml"

1. Press Ctrl-F5 to run Outlook. Select the Task option in the navigation bar, and click New in the menu. This will open the Task inspector. Select the Contoso item on the ribbon and Outlook will load your custom form region.

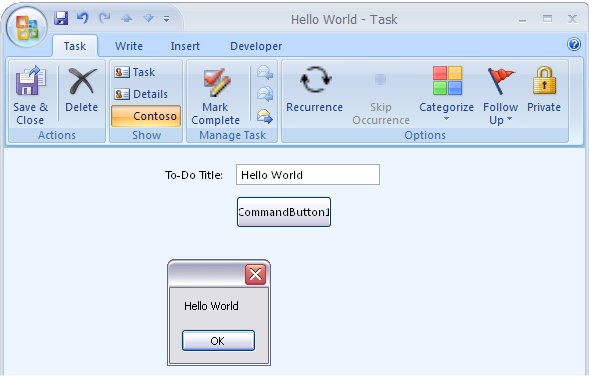


Figure A custom form region for the Task inspector

1. In this exercise, you created an add-in with a custom form region for Outlook. Unlike custom task panes and custom ribbons, this CTP release of Visual Studio Tools for Office ‘v3’ does not include any additional support for custom form regions, but this support is planned to be introduced in later CTP releases.

#### Conclusion

Office 2007 introduces a number of new features, and Visual Studio Tools for Office will provide design-time and runtime support for all these features. Office is converging on the add-in model for new programmability features, and the easiest and most robust way to create add-ins is with Visual Studio Tools for Office.

These simple exercises have shown how you can create an add-in that has one or more custom task panes, and/or one or more ribbons (depending on the host application), and/or uses custom form regions (Outlook only).

The first CTP focuses on runtime support to provide isolation, robustness and flexible deployment – most of which is largely invisible. The full RAD design-time support that developers expect from Visual Studio will be incrementally introduced in later CTP releases.