

Windows Ribbon Introductory Tutorial

Adding a Windows Ribbon to a Win32 Application

**Legal Notice**

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This tutorial is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

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.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, Windows Server, Windows Vista, Active Directory, ActiveSync, ActiveX, Direct3D, DirectDraw, DirectInput, DirectMusic, DirectPlay, DirectShow, DirectSound, DirectX, Expression, FrontPage, HighMAT, Internet Explorer, JScript, Microsoft Press, MSN, Outlook, PowerPoint, SideShow, Silverlight, Visual Basic, Visual C++, Visual InterDev, Visual J++, Visual Studio, WebTV, Windows Media, Win32, Win32s, and Zune are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

***Estimated Time to Complete this Tutorial: 75 Minutes Total (broken into 3 separate exercises)***

# Learning Objectives

While completing this tutorial, you will learn how to:

* Configure a Visual Studio project to make use of the Ribbon
* Integrate a Ribbon with a Win32 window (HWND)
* Add controls such as buttons, checkboxes, tabs, and chunks
* Connect the application business code with Ribbon controls
* Fine tune the layout of controls and specify how the Ribbon resizes

# Overview

This tutorial is intended for C++ developers who are developing desktop applications and want to take advantage of the new Windows Ribbon framework. The tutorial steps you through how to add an empty Ribbon to a small application, add various controls with icons, labels, and other resources to the Ribbon, and then connect the controls to the existing command infrastructure of the application. You will also learn how the API maintains separation between control organization and event handling. Finally, the tutorial will demonstrate how to specify layouts and resizing behavior to show how the Ribbon adapts and performs at different sizes. When you are finished, you will have performed all the steps necessary to add and customize, a basic Ribbon in an application.

The tutorial will involve real-time compiling of code and markup copied from this document. In the event that a copying error (or any other problem) prevents the application from compiling, you can find fully completed samples in the tutorial package with the final source code for each exercise. These samples can be used to unblock you from compiling errors, or can be used as a starting point for the next exercise if desired.

# Prerequisites

* **Microsoft Visual Studio 2005 (or newer)**The tutorial makes several references to configuring and using Visual Studio. While it is not necessary to use this environment to author a Ribbon, the tutorial assumes that Visual Studio 2005 (or newer) is installed on your system. *Note: While Visual Studio Express editions can be used to create Ribbons, this version lacks certain capabilities that are necessary to complete this specific tutorial.*
* **Windows 7 SDK (v7.0, RC Build)**This tutorial was first released as a hands-on lab at the October 2008 Professional Developers Conference (PDC2008). It has since been updated to work with the RC release of the Windows 7 SDK. ***Please note****: We recommend that the SDK be installed after Visual Studio is installed. This will help to prevent potential SDK version conflicts when compiling. If you encounter problems when compiling your code, the steps in the “Important Note” found on page 5 may be helpful.*

# Anatomy of the Windows Ribbon

This tutorial uses common Windows Ribbon terminology to describe various parts of the UI. The diagram below should help describe what these parts are and where they can be found:



# Audience Considerations

Before working on this tutorial, it would be helpful (though not required) to have:

* Some familiarity with Win32 development and the C++ language
* A basic understanding of COM programming and concepts
* Some familiarity with Visual Studio

# For More Information

For information and documentation Windows Ribbon, its API, and the Microsoft Office Fluent user interface, the following links may be useful:

* Windows Ribbon Documentation on MSDN  
  <http://msdn.microsoft.com/en-us/library/dd371191.aspx>
* Windows Ribbon Guidelines on MSDN  
  <http://msdn.microsoft.com/en-us/library/cc872782.aspx>
* The Microsoft Office Fluent user interface overview  
  <http://office.microsoft.com/en-us/products/HA101679411033.aspx>
* MSDN Forum on Windows Ribbon development  
  <http://social.msdn.microsoft.com/Forums/en-US/windowsribbondevelopment/threads/>

# Exercise 1 Creating an empty Ribbon and adding it to an application.

***Estimated Time: 25 minutes***

In this exercise you will start by creating a Visual Studio project from scratch. Then, you will write the necessary code to compile and initialize a Ribbon in an application.

## Background Information The Ribbon API consists of 2 parts:

* XML Markup, used to define the Ribbon structure and organization of controls
* C++ COM interfaces, used to initialize and handle events

The markup must be compiled into a binary format in order to be used during execution. The compilation of the Ribbon is done through the use of a new compiler tool named uicc.exe. During this exercise we will show you how to configure Visual Studio to compile the markup automatically.

All files referred to in the steps below can be found in the tutorial package under the folder “Exercise1\RibbonApp\”.

*\*\*Note: Throughout this tutorial, code and/or markup that already exists in source files will be colored grey (*example*). This will provide a sense of context when you are instructed to add new code to these files.*

↙ Create a new project in Visual Studio and generate a basic application.

In this task you will setup the bases required to start adding a Ribbon to an application.

1. Launch Visual Studio.
2. Create a new project by selecting File 🡪 New 🡪 Project…
3. Under “Visual C++” select “Win32”. Then select “Win32 Project”.
4. Enter “RibbonApp” as the name of the project.
5. Select “OK”, then on the next screen select “Finish”.
6. Compile your new application, and launch it (hotkey F5). A window with a simple menu will open.  
   *Note: When building the application, if a prompt appears stating “This project is out of date”, just ignore it and continue the building process.*

↙ Create a basic Ribbon markup file.

You will now add a small amount of Ribbon XML markup to your project. The markup will define commands to be exposed in the Ribbon, and will also define how controls will be arranged when exposing these commands.

1. Create a new file named “ribbonmarkup.xml” and add it to the project.  
   In the Solution Explorer, right-click the project name 🡪 Add 🡪 New Item. Under “Visual C++” select “Web”. Then select “XML File (.xml)”, and in the Name field call it “ribbonmarkup.xml”.
2. Copy+Paste the markup below into the newly created ribbonmarkup.xml file, overwriting any previous text that was auto-generated when the .xml file was created. This will define a minimum Ribbon with one Tab:

<Application xmlns="http://schemas.microsoft.com/windows/2009/Ribbon">

<Application.Commands>

<Command Name="TabHome" Symbol="cmdTabHome" Id="30000" />

</Application.Commands>

<Application.Views>

<Ribbon>

<Ribbon.Tabs>

<Tab CommandName="TabHome">

</Tab>

</Ribbon.Tabs>

</Ribbon>

</Application.Views>

</Application>

Note: Exercise 2 is dedicated to understanding the Ribbon XML format in greater detail. However, feel free to examine the syntax now and see how the Ribbon, Tabs, and Commands are being defined.

↙ Configure Visual Studio to automatically compile the Ribbon markup.

1. In Visual Studio, right click on the new XML file and select “Properties”. Under “Custom Build Step”, enter the following into the Command Line field:  
     
   uicc.exe ribbonmarkup.xml ribbonmarkup.bml /res:ribbonres.rc   
     
   uicc.exe is a new compiler tool dedicated to the Ribbon used to generate resources consumable at runtime. "ribbonmarkup.bml” is the output: a binary representation of “ribbonmarkup.xml”.
2. On the same property page, enter the following into the Outputs field:  
     
   ribbonmarkup.bml;ribbonres.rc  
     
   Close the Properties page by clicking “OK”. The generated file ribbonres.rc defines a resource that you’ll use to link the generated binary markup to your application module. The binary markup file (.bml) will be generated when you build the application. Feel free to look at ribbonres.rc (in order to do so you first need to build your project so the file can be generated by uicc.exe).
3. Finally, you should include the binary markup in your project. To do so, edit the code of RibbonApp.rc (right-click it in the Solution Explorer, select “view code”) and add the following line right before where the icons are defined (before the comment block called “Icon”):  
     
   #include "ribbonres.rc"

**Important Note**: If you are unable to compile your application at any time during this exercise, despite having previously installed the Windows 7 RC SDK, it’s possible that your system is not configured to use the SDK correctly. To check and/or fix this issue, do the following:

1. Click the Start button 🡪 All Programs 🡪 Microsoft Windows SDK v7.0 🡪 Visual Studio Registration 🡪 Windows SDK Configuration Tool.   
   If a dialog appears with the title “Do you want to allow the following program to make changes to this computer?”, click “Yes”.
2. Look at the combo box in the dialog that appears and make sure that v7.0 of the Windows SDK is the current target for Visual Studio. If it is set to something else (such as v6.0), then change it to v7.0 and click “Make Current”. Even if you find that it is already set to v7.0, click “Make Current” anyway—this can be particularly important if you are running VS2005.

↙ Create a host for the Ribbon.

A host is used to receive different notifications from the Ribbon. It is required for the Ribbon to be initialized.

1. Configure your project to use ATL.  
   In the Solution Explorer, right-click the project name 🡪 Properties. Expand “Configuration Properties”, and select the “General” page. On the property page, set the “Use of ATL” field to either of the following values, depending on your preference:
   * Static Link to ATL  
     -or-
   * Dynamic Link to ATL
2. Add a new file to the project named “Ribbon.cpp”.  
   In the Solution Explorer, right-click the project name 🡪 Add 🡪 New Item. Under “Visual C++” select “Code”. Select “C++ File (.cpp)”, and in the Name field call it “Ribbon.cpp”.
3. Edit the new “Ribbon.cpp” file and include the following headers:  
     
   Precompiled header for the project:  
   #include "stdafx.h"  
     
   ATL/COM header files:  
   #include <atlbase.h>  
   #include <atlcom.h>  
   #include <initguid.h>  
     
   And finally, the Ribbon header file which declares all the necessary Ribbon interfaces:  
   #include <uiribbon.h>
4. In Ribbon.cpp, implement the interface IUIApplication defined in uiribbon.h. In this exercise we are not actually interested in implementing any of the notifications, so you can return E\_NOTIMPL for all 3 functions.  
     
   The following code can be placed into Ribbon.cpp to implement IUIApplication:

class CApplication

: public CComObjectRootEx<CComMultiThreadModel>

, public IUIApplication

{

public:

BEGIN\_COM\_MAP(CApplication)

COM\_INTERFACE\_ENTRY(IUIApplication)

END\_COM\_MAP()

STDMETHOD(OnViewChanged)(UINT32 nViewID, \_\_in UI\_VIEWTYPE typeID, \_\_in IUnknown\* pView, UI\_VIEWVERB verb, INT32 uReasonCode)

{

return E\_NOTIMPL;

}

STDMETHOD(OnCreateUICommand)(UINT32 nCmdID, \_\_in UI\_COMMANDTYPE typeID, \_\_deref\_out IUICommandHandler\*\* ppCommandHandler)

{

return E\_NOTIMPL;

}

STDMETHOD(OnDestroyUICommand)(UINT32 commandId, \_\_in UI\_COMMANDTYPE typeID, \_\_in\_opt IUICommandHandler\* pCommandHandler)

{

return E\_NOTIMPL;

}

};

↙ Create and Initialize the Ribbon.

1. Now that you’ve implemented CApplication, declare a global g\_pFramework of type IUIFramework\* near the top of the Ribbon.cpp file (put it just before the declaration of the CApplication class, after the include lines). This pointer will be used to instantiate the Ribbon platform.   
     
   IUIFramework\* g\_pFramework = NULL;  
     
   Then, at the very bottom of Ribbon.cpp (after the CApplication definition), define a Ribbon initialization function:  
     
   HRESULT InitRibbon(HWND hWindowFrame)  
   {  
     
   hWindowFrame will be a handle on the application main window.  
   Steps 2-5 below should be implemented, in order, inside the newly created InitRibbon function.
2. Use CoCreateInstance to instantiate a class CLSID\_UIRibbonFramework. You’ll get an IUIFramework pointer back. IUIFramework is the main interface exposed by the Ribbon platform.  
     
   HRESULT hr = ::CoCreateInstance(CLSID\_UIRibbonFramework, NULL, CLSCTX\_INPROC\_SERVER, IID\_PPV\_ARGS(&g\_pFramework));  
   if(FAILED(hr))  
   {  
    return hr;  
   }
3. Instantiate a CApplication object.  
     
   CComObject<CApplication> \*pApplication = NULL;  
   hr = CComObject<CApplication>::CreateInstance(&pApplication);  
   if(FAILED(hr))  
   {  
    return hr;  
   }
4. Call Initialize on the returned IUIFramework pointer and specify both the host HWND and the newly created CApplication object.  
     
   hr = g\_pFramework->Initialize(hWindowFrame, pApplication);  
   if(FAILED(hr))  
   {  
    return hr;  
   }
5. Now let’s load the markup. On the IUIFramework pointer, call LoadUI.  
     
   g\_pFramework->LoadUI(GetModuleHandle(NULL), L"APPLICATION\_RIBBON");  
   if(FAILED(hr))  
   {  
    return hr;  
   }
6. Close the function. It is now complete.  
     
    return S\_OK;  
   }

↙ Integrate with the application.

We now need to call InitRibbon from the pre-generated application code.

1. Initialize COM in the application. Open RibbonApp.cpp and right after the include lines, add:  
     
   #include <atlbase.h>  
   CComModule \_Module;  
   extern HRESULT InitRibbon(HWND hWindowFrame);
2. Inside the generated function \_tWinMain, add at the beginning of the function:  
     
   CoInitialize(NULL);  
     
   Add at the end, before the “return” statement:  
     
   CoUninitialize();
3. Inside the function InitInstance, right before ShowWindow is called, call InitRibbon:  
     
   HRESULT hr = InitRibbon(hWnd);  
   if (FAILED(hr))  
   {  
    return FALSE;  
   }
4. Compile your application and launch it. It should now have an empty Ribbon.

You’ll notice that the only change performed in the application code at this point is the call to InitRibbon. The rest of the code added is standard COM initialization.

↙ Complete the integration.

1. Inside Ribbon.cpp, add a DestroyRibbon() function. Paste the following definition at the end of the file:

void DestroyRibbon()

{

if (g\_pFramework)

{

g\_pFramework->Destroy();

g\_pFramework->Release();

g\_pFramework = NULL;

}

}

1. Inside RibbonApp.cpp, declare a DestroyRibbon() function near the beginning of the file:  
     
   #include <atlbase.h>  
   CComModule \_Module;  
   extern HRESULT InitRibbon(HWND host);  
   extern void DestroyRibbon();
2. Inside RibbonApp.cpp, in the WndProc function, call DestroyRibbon() during the switch statement’s WM\_DESTROY processing. After you’re finished, the code should look like this:

case WM\_DESTROY:

DestroyRibbon();

PostQuitMessage(0);

break;

1. If you launch the application, you’ll notice that the Ribbon flickers during resizing. You can fix this by adding the window style WS\_CLIPCHILDREN to the CreateWindow call inside the function InitInstance.  
     
   hWnd = CreateWindow(szWindowClass, szTitle,   
    WS\_OVERLAPPEDWINDOW | WS\_CLIPCHILDREN,  
    CW\_USEDEFAULT, 0, CW\_USEDEFAULT, 0, NULL, NULL,  
    hInstance, NULL);

*Note: The flickering was due to the host window drawing over the ribbon. WS\_CLIPCHILDREN tells the host window to exclude the area occupied by child windows when drawing occurs within the parent window.*

# Exercise 2 Adding simple controls to an existing Ribbon.

***Estimated Time: 40 minutes***

In this exercise you will start with the completed project from Exercise 1 and write the code to add controls to the Ribbon. You will also learn how to get notifications from the ribbon controls and how to get and set properties on those controls.

## Background Information All files referred to in the steps below can be found in the tutorial package under the folder “Exercise2\RibbonApp\”.

↙ Add a button in a Tab.

In this task you will add a Button control to the ribbon.

1. Continue from the project you used for Exercise 1. Or, launch Visual Studio from Exercise1/RibbonApp.sln (found in the tutorial package).
2. In RibbonMarkup.xml, inside <Application.Commands> add the following markup to define commands for the new controls.

<Command Name="GroupMain" Symbol="cmdGroupMain" Id="30001"/>  
<Command Name="MyButton" Symbol="cmdMyButton" Id="30002"/>

1. Let’s now add a new Button control inside a new Group. A Ribbon contains Tabs, and inside those tabs are groups. These groups contain the controls you see in the Ribbon that can be interacted with. Inside the <Tab CommandName="TabHome"> tag add the following markup:

<Group CommandName="GroupMain">

<Button CommandName="MyButton"/>

</Group>

1. Build and run the application. You will notice an empty button control inside a new group if you hover the mouse over the group. (Please note that the Button is not visible unless it is hovered over with the mouse pointer because its background color is same as that of the group.)

↙ Specify Label, Tooltip and Icon of the button.

All resources used by the Ribbon are defined in a header file. First you’ll learn how to generate this header file. Then, you’ll specify all the different attributes (Label, Tooltip, Icons) for the different commands.

1. Setup project properties to generate header file by uicc.exe. For this do the following:
   1. Right click on RibbonMarkup.xml 🡪 Properties 🡪 Custom build step
   2. In “Command Line” add the following before /res:ribbonres.rc in the current command line:  
      /header:ribbonres.h  
        
      New full command line becomes:   
      uicc.exe ribbonmarkup.xml ribbonmarkup.bml /header:ribbonres.h /res:ribbonres.rc  
        
      In “Outputs” add the following at the end:  
      ;ribbonres.h  
        
      The new outputs should now read as:   
      ribbonmarkup.bml;ribbonres.rc;ribbonres.h  
        
      Click “OK” to close the Properties dialog.
2. Right click RibbonApp.rc 🡪 View Code and add the following include line above the include line for ribbonres.rc (recall that these include lines will be lower in the file, above the “Icon” comment block):  
     
   #include "ribbonres.h"  
   #include "ribbonres.rc"
3. Copy Button\_Image.bmp from the BITMAPS folder (included with this tutorial) to the project directory; this is the same directory where RibbonApp.rc exists. Typically, the destination would be:   
   C:\Users\your\_username\Documents\Visual Studio ####\Projects\RibbonApp\RibbonApp
4. In the Visual Studio Solution Explorer, add Button\_Image.bmp to the project by right clicking RibbonApp project 🡪 Add 🡪 Existing Item 🡪 Button\_Image.bmp
5. In this step we are adding resource properties to the Command definition of cmdMyButton. We are adding a Label, ToolTip Title, ToolTip Description and an Image to be used by the control that references this command. In ribbonmarkup.xml, change the command definition for cmdMyButton under <Application.Commands> to the following:

<Command Name="MyButton" Symbol="cmdMyButton" Id="30002" LabelTitle="My Button">

<Command.TooltipTitle>My Button</Command.TooltipTitle>

<Command.TooltipDescription>Click on My Button</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="Button\_Image.bmp"/>

</Command.LargeImages>

</Command>

1. Rebuild (in the menu Build 🡪 Rebuild solution) and run the project. Notice that a label and image appear for the ribbon button in the group.
2. Add Labels for some other controls (Tab, Group etc) as well in the project by modifying their command definition as follows: (Notice the different syntax allowed to specify XAML properties)  
     
   Replace:  
   <Command Name="TabHome" Symbol="cmdTabHome" Id="30000" />  
     
   With:

<Command Name="TabHome" Symbol="cmdTabHome" Id="30000" LabelTitle="Home" />

Then, replace:  
<Command Name="GroupMain" Symbol="cmdGroupMain" Id="30001" />  
  
With:

<Command Name="GroupMain" Symbol="cmdGroupMain" Id="30001" LabelTitle="Main" />

1. Build and run the project, and see the new Label added for the Ribbon’s “Home” Tab and “Main” Group.

↙ Display a message box when the button is clicked.

A Command Handler (IUICommandHandler) is an interface that the application implements in order to handle notifications received when a user interacts with a command’s controls. It exposes the Execute and UpdateProperty methods. Execute is called by the Ribbon framework to notify the application of an event from the control such as when a button is clicked. UpdateProperty is called by the framework when it needs a property (such as if the command is enabled/disabled) of the control from the application.

1. Add the following include line after the existing include lines in Ribbon.cpp:  
     
   #include <uiribbon.h>  
   #include "ribbonres.h"  
     
   ribbonres.h is generated by uicc.exe and contains the commands that are defined in markup, making them accessible from code.
2. In Ribbon.cpp, change CApplication to implement IUICommandHandler for all the controls in the project.   
   First derive CApplication from the IUICommandHandler interface by adding the following code as the last interface that CApplication derives from:

class CApplication

: public CComObjectRootEx<CComMultiThreadModel>

, public IUIApplication

, public IUICommandHandler

1. Then add the following COM\_INTERFACE\_MAP entry to the existing COM\_MAP:

BEGIN\_COM\_MAP(CApplication)

COM\_INTERFACE\_ENTRY(IUIApplication)

COM\_INTERFACE\_ENTRY(IUICommandHandler)

END\_COM\_MAP()

1. Add the following code inside the CApplication class to implement the 2 methods of the IUICommandHandler interface:  
     
   STDMETHODIMP Execute(UINT nCmdID,

UI\_EXECUTIONVERB verb,

\_\_in\_opt const PROPERTYKEY\* key,

\_\_in\_opt const PROPVARIANT\* ppropvarValue,

\_\_in\_opt IUISimplePropertySet\* pCommandExecutionProperties)

{

HRESULT hr = S\_OK;

switch (verb)

{

case UI\_EXECUTIONVERB\_EXECUTE:

if (nCmdID == cmdMyButton)

{

MessageBox(NULL, L"Clicked on My Button!",

L"My Button Execute", MB\_OK);

}

break;

}

return hr;

}

STDMETHODIMP UpdateProperty(UINT nCmdID,

\_\_in REFPROPERTYKEY key,

\_\_in\_opt const PROPVARIANT\* ppropvarCurrentValue,

\_\_out PROPVARIANT\* ppropvarNewValue)

{

return E\_NOTIMPL;

}

1. Add the following if statement to CApplication::OnCreateUICommand, before the return line:

STDMETHOD(OnCreateUICommand)(UINT32 nCmdID, \_\_in UI\_COMMAND\_TYPE typeID, \_\_deref\_out IUICommandHandler\*\* ppCommandHandler)

{

if (nCmdID == cmdMyButton)

{

return QueryInterface(IID\_PPV\_ARGS(ppCommandHandler));

}

return E\_NOTIMPL;

}

1. Build and Run the application and click on the button to see the MessageBox popup.

↙ Add a checkbox control.

Follow the steps to create a CheckBox and an associated a command handler for CheckBox commands.

1. Add the following Commands Definition in RibbonMarkup.xml inside the <Application.Commands> tag:

<Command Name="MyChoice" Symbol="cmdMyChoice" Id="30003" LabelTitle="My Choice">

<Command.TooltipTitle>My Choice</Command.TooltipTitle>

<Command.TooltipDescription>Select My Choice!</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="Button\_Image.bmp"/>

</Command.LargeImages>

</Command>

Add a CheckBox inside the Group by adding the following markup:

<Group CommandName="GroupMain">  
 <CheckBox CommandName="MyChoice"/>

<Button CommandName="MyButton"/>

</Group>

1. Associate a command handler for the new checkbox. In the OnCreateUICommand method of Ribbon.cpp, modify the if statement to associate the command handler to CheckBox control as well:

if (nCmdID == cmdMyButton || nCmdID == cmdMyChoice )

{

return QueryInterface(IID\_PPV\_ARGS(ppCommandHandler));  
}

1. In the Ribbon.cpp Execute method add the following if statement (just above the break; line) to get notified when CheckBox is clicked:

case UI\_EXECUTIONVERB\_EXECUTE:

if (nCmdID == cmdMyButton)

{

MessageBox(NULL, L"Clicked on My Button!", L"My Button Execute", MB\_OK);

}

else if (nCmdID == cmdMyChoice)

{

MessageBox(NULL, L"Toggled My Choice!", L"My Choice Execute", MB\_OK);

}

break;

1. Build and run the project and observe the new CheckBox. Click on the CheckBox and observe the CheckBox toggle its state. Also, notice the MessageBox added in the code popup.

↙ Swap the checkbox for a toggle button.

In this part we will change the CheckBox control in the ribbon to a ToggleButton control. Both the CheckBox and ToggleButton controls are of the same command type, and the application code handles them the same way. So when changing a CheckBox control to a ToggleButton control, no C++ code change is needed.

1. In RibbonMarkup.xml, replace the CheckBox control with a ToggleButton control. This can be done by simply renaming the tag to “ToggleButton”.  
     
   Replace:  
   <CheckBox CommandName="MyChoice"/>  
     
   With:  
   <ToggleButton CommandName="MyChoice"/>
2. Build and run the project to observe that the checkbox has changed to a toggle button. You’ll notice that clicking on the toggle button invokes the same message box as before.

↙ Specify a SizeDefinition template for the Group.

In this step a SizeDefinition template will be specified for the Group using its SizeDefinition attribute. Templates provide physical layout information for the controls inside a Group. The Windows Ribbon includes several predefined templates, and it is possible to define your own as well. Since there are now two buttons in the group, the standard predefined “TwoButtons” template can be used.

1. Add a “TwoButtons” SizeDefinition template to the GroupMain. The Group definition becomes:  
     
   <Group CommandName="GroupMain" SizeDefinition="TwoButtons">
2. Build and run the project, and observe that by using the ‘TwoButtons’ template, large buttons laid side-by-side are used in the group.

↙ Now we’ll make use of the ToggleButton to disable/enable the Button control.

1. Add a private member \_fEnabled to the class CApplication by adding the following code just before the closing brace of the class:

private:

BOOL \_fEnabled;

1. At the top of Ribbon.cpp, above the include line (#include "ribbonres.h") add the following include line:  
     
     
   #include <uiribbon.h>  
   #include <UIRibbonPropertyHelpers.h>  
   #include "ribbonres.h"
2. Link to propsys.lib in the linker properties. Right click on RibbonApp project 🡪 properties 🡪 Linker 🡪 Input. In the “Additional Dependencies” field, add the file **propsys.lib**. Click OK.
3. In Ribbon.cpp, inside the Execute method, replace the MessageBox for ToggleButton click with the following code:  
     
   Replace:  
   MessageBox(NULL, L"Toggled My Choice!", L"My Choice Execute", MB\_OK);  
     
   With the code:

PROPVARIANT var, varNew;

hr = g\_pFramework->GetUICommandProperty(cmdMyChoice, UI\_PKEY\_BooleanValue, &var);

if (FAILED(hr))

{

return hr;

}

hr = PropVariantToBoolean(var, &\_fEnabled);

if (FAILED(hr))

{

return hr;

}

\_fEnabled = !\_fEnabled;

hr = UIInitPropertyFromBoolean(UI\_PKEY\_Enabled, \_fEnabled, &varNew);

if (FAILED(hr))

{

return hr;

}

hr = g\_pFramework->SetUICommandProperty(cmdMyButton, UI\_PKEY\_Enabled, varNew);

if (FAILED(hr))

{

return hr;

}

1. Build and run the project. Click on the ToggleButton and observe that the “My Button” command becomes disabled and enabled.

↙ Update the Label property of the ToggleButton control at runtime.

1. In Ribbon.cpp, add the following code as the last lines of code inside the else if (nCmdID == cmdMyChoice) of the Execute method:

hr = g\_pFramework->InvalidateUICommand(cmdMyChoice, UI\_INVALIDATIONS\_PROPERTY, &UI\_PKEY\_Label);

if (FAILED(hr))

{

return hr;

}

1. Replace the implementation of the UpdateProperty method with code that will cause the MyChoice command to update its label.  
     
   In the UpdateProperty function, replace:  
   return E\_NOTIMPL;  
     
   With:  
   UNREFERENCED\_PARAMETER(ppropvarCurrentValue);

HRESULT hr = E\_FAIL;

if (key == UI\_PKEY\_Label)

{

// Update the Label of ToggleButton control

if (nCmdID == cmdMyChoice)

{

if (\_fEnabled)

{

hr = UIInitPropertyFromString(UI\_PKEY\_Label,

L"Disable Button", ppropvarNewValue);

}

else

{

hr = UIInitPropertyFromString(UI\_PKEY\_Label,

L"Enable Button", ppropvarNewValue);

}

}

}

return hr;

1. Build and run the project. Click on the toggle button and observe that the label of the button changes each time it is clicked. This shows that command resources can be defined and changed either in markup, or in the code at run time (or both).

# Exercise 3 Adding controls and groups to an existing Ribbon.

***Estimated Time: 10 minutes***

In this exercise you will start with the completed project from Exercise 2 and revise the ribbon markup to add more controls. You will learn how to better organize the controls using Groups.

## Background Information All files referred to in the steps below can be found in the tutorial package under the folder “Exercise3\RibbonApp\”.

↙ Add additional controls and groups in a Tab.

In this task you will add more controls and chunks to the ribbon.

1. Continue from the project you used for Exercise 2. Or Launch Visual Studio from Exercise2/RibbonApp.sln.
2. In RibbonMarkup.xml, inside <Application.Commands> add the following markup to define commands for new groups.

<Command Name="GroupDatabase" Symbol="cmdGroupDatabase" Id="30004">

<Command.LabelTitle>Database</Command.LabelTitle>

</Command>

<Command Name="GroupClipboard" Symbol="cmdGroupClipboard" Id="30005">

<Command.LabelTitle>Clipboard</Command.LabelTitle>

</Command>

1. Add button icons to the project resources. First, copy the following bitmaps from the BITMAPS folder to your project directory. Typically, the destination would be:   
   C:\Users\your\_username\Documents\Visual Studio ####\Projects\RibbonApp\RibbonApp  
     
   Then add those bitmaps to the project by right clicking RibbonApp project 🡪Add 🡪 existing Item 🡪(select each of the bitmaps below):  
     
   AddTableL.bmp  
   AddTableS.bmp  
   Copy.bmp  
   Cut.bmp  
   DeleteTableL.bmp  
   DeleteTableS.bmp  
   Paste.bmp  
   PrintRelationshipsL.bmp  
   PrintRelationshipsS.bmp
2. Inside <Application.Commands> add more commands for new Buttons:

<Command Name="AddTable" Symbol="cmdAddTable" Id="30006"

LabelTitle="Add Table">

<Command.TooltipTitle>Add Table</Command.TooltipTitle>

<Command.TooltipDescription>Add Table</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="AddTableL.bmp"/>

</Command.LargeImages>

</Command>

<Command Name="DeleteTable" Symbol="cmdDeleteTable" Id="30007" LabelTitle="Delete Table">

<Command.TooltipTitle>Delete Table</Command.TooltipTitle>

<Command.TooltipDescription>Delete Table</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="DeleteTableL.bmp"/>

</Command.LargeImages>

</Command>

<Command Name="PrintRelationships" Symbol="cmdPrintRelationships" Id="30008" LabelTitle="Print Relationships">

<Command.TooltipTitle>Print Relationships</Command.TooltipTitle>

<Command.TooltipDescription>Print Relationships</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="PrintRelationshipsL.bmp"/>

</Command.LargeImages>

</Command>

<Command Name="Paste" Symbol="cmdPaste" Id="30009" LabelTitle="Paste">

<Command.TooltipTitle>Paste</Command.TooltipTitle>  
 <Command.TooltipDescription>Paste</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="Paste.bmp"/>

</Command.LargeImages>

</Command>

<Command Name="Cut" Symbol="cmdCut" Id="30010" LabelTitle="Cut">

<Command.TooltipTitle>Cut</Command.TooltipTitle>

<Command.TooltipDescription>Cut</Command.TooltipDescription>

<Command.SmallImages>

<Image Source="Cut.bmp"/>

</Command.SmallImages>

</Command>

<Command Name="Copy" Symbol="cmdCopy" Id="30011" LabelTitle="Copy">

<Command.TooltipTitle>Copy</Command.TooltipTitle>

<Command.TooltipDescription>Copy</Command.TooltipDescription>

<Command.SmallImages>

<Image Source="Copy.bmp"/>

</Command.SmallImages>

</Command>

1. Inside <Tab CommandName="TabHome"> tag, add the following markup to add Groups and Buttons after the existing Group with the name GroupMain:

<Tab CommandName="TabHome">

<Group CommandName="GroupMain" Template="TwoButtons">

<ToggleButton CommandName="MyChoice"/>

<Button CommandName="MyButton"/>

</Group>

<Group CommandName="GroupDatabase" SizeDefinition="ThreeButtons">

<Button CommandName="AddTable"/>

<Button CommandName="DeleteTable"/>

<Button CommandName="PrintRelationships"/>

</Group>

<Group CommandName="GroupClipboard" SizeDefinition ="BigButtonsAndSmallButtonsOrInputs">

<ControlGroup>

<Button CommandName="Paste"/>

</ControlGroup>

<ControlGroup>

<Button CommandName="Cut"/>

<Button CommandName="Copy"/>

</ControlGroup>

</Group>

</Tab>

1. Build and run the application. Notice there are now three chunks in the Ribbon with each group containing three buttons arranged differently depending on the SizeDefinition template declared in the markup.
2. Now drag the right border of the application window to the left. Once the window’s right border moves beyond the right-most control, a pager control appears to indicate some controls are out of the window’s view. If you continue to resize the window small enough, the Ribbon will eventually disappear maximizing the room provided to the application’s working area.

↙ Specify adaptive resizing rules.

Now we’ll specify resizing rules for the Ribbon so that an adaptive group layout can be applied instead of a fixed layout.

1. Add scaling rules to TabHome. Locate <Tab CommandName="TabHome"> and add the following markup right after that line:

<Tab CommandName="TabHome">

<Tab.ScalingPolicy>

<ScalingPolicy>

<ScalingPolicy.IdealSizes>

<Scale Group="GroupMain" Size="Large"/>

<Scale Group ="GroupDatabase" Size="Large"/>

<Scale Group ="GroupClipboard" Size="Large"/>

</ScalingPolicy.IdealSizes>

<Scale Group ="GroupClipboard" Size="Medium"/>

<Scale Group ="GroupClipboard" Size="Popup"/>

<Scale Group ="GroupDatabase" Size="Medium"/>

</ScalingPolicy>

</Tab.ScalingPolicy>

1. Now rebuild the app, run it, and shrink the size window gradually. Notice the following behaviors:
   * The Clipboard group will shrink its size if there is not enough room for displaying its controls. The labels of small buttons disappear first.
   * Continue reducing the window size. The Clipboard group becomes a dropdown button. When you click the button, all controls display inside a popup.
   * Keep reducing the window size. The Database group rearranges its controls. All controls show as small buttons and lay themselves out vertically.
   * As previously mentioned, if you continue to resize the Ribbon smaller it will eventually disappear in an attempt to give as much space to the application’s workspace as possible.
2. You may discover a few issues while resizing the window.
   * When the Clipboard group shows as a dropdown button, the button doesn’t have icon. This is because an image needs to be specified for the group itself.
   * In the Database group, the icons used for the small buttons look slightly distorted. This is caused by the Ribbon trying to scale down the large-sized bitmap. To fix this, a small-sized icon (16x16) needs to be provided.
3. We will now add an image for the Clipboard group, and small images for the commands in the Database group. Replace the GroupClipboard, AddTable, DeleteTable and PrintRelationships commands under <Application.Commands> with the following:

<Command Name="GroupClipboard" Symbol="cmdGroupClipboard" Id="30005">

<Command.LabelTitle>Clipboard</Command.LabelTitle>

<Command.LargeImages>

<Image Source="Paste.bmp"/>

</Command.LargeImages>

</Command>

<Command Name="AddTable" Symbol="cmdAddTable" Id="30006"

LabelTitle="Add Table">

<Command.TooltipTitle>Add Table</Command.TooltipTitle>

<Command.TooltipDescription>Add Table</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="AddTableL.bmp"/>

</Command.LargeImages>

<Command.SmallImages>

<Image Source="AddTableS.bmp"/>

</Command.SmallImages>

</Command>

<Command Name="DeleteTable" Symbol="cmdDeleteTable" Id="30007" LabelTitle="Delete Table">

<Command.TooltipTitle>Delete Table</Command.TooltipTitle>

<Command.TooltipDescription>Delete Table</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="DeleteTableL.bmp"/>

</Command.LargeImages>

<Command.SmallImages>

<Image Source="DeleteTableS.bmp"/>

</Command.SmallImages>

</Command>

<Command Name="PrintRelationships" Symbol="cmdPrintRelationships" Id="30008" LabelTitle="Print Relationships">

<Command.TooltipTitle>Print Relationships</Command.TooltipTitle>

<Command.TooltipDescription>Print Relationships</Command.TooltipDescription>

<Command.LargeImages>

<Image Source="PrintRelationshipsL.bmp"/>

</Command.LargeImages>

<Command.SmallImages>

<Image Source="PrintRelationshipsS.bmp"/>

</Command.SmallImages>

</Command>

1. Build and run the project. Notice that the previous issues have been fixed and the small group icon that was missing before is now present. Once again, this was accomplished using markup alone and didn’t require any changes to C++ code.

# Final Thoughts

You have successfully completed this Windows Ribbon introductory tutorial! You have learned about the platform’s markup/code separated architecture, and you’ve seen how many traditional UI creation tasks such as exact positioning of controls are taken care of for you, saving significant development time.

This tutorial has only shown basic features of the Windows Ribbon, with many more interesting parts of the platform yet to be explored. We hope you found these exercises helpful and interesting.

Happy coding!  
- The Windows Ribbon team