Windows Ribbon for WinForms

(written by Arik Poznanski, updated by harborsiem)

Part 0 – Table of Contents

First, I want to announce that the Windows Ribbon for WinForms library is no longer beta.

Click to the new Link for the Windows Ribbon for WinForms.

Since now the library covers every feature available by the windows ribbon. I think it would be useful to provide a table of contents for the posts I've written during the development process. Each post serves as documentation on how to use one or two ribbon features.

Part 1 – Introduction, Background on the windows ribbon.

So, here is the plan:
I'll write a series of posts regarding using the Windows Ribbon with .NET WinForms.

The goal:
Having .Net wrappers for using Windows Ribbon in WinForms applications and a set of examples of how to use each and every ribbon feature. **Windows Ribbon** is also called "**UIRibbon**" or as Microsoft development title "**Scenic Ribbon**".

Development methodology
I'll develop a library that will hold all the relevant classes, interfaces and native functions.

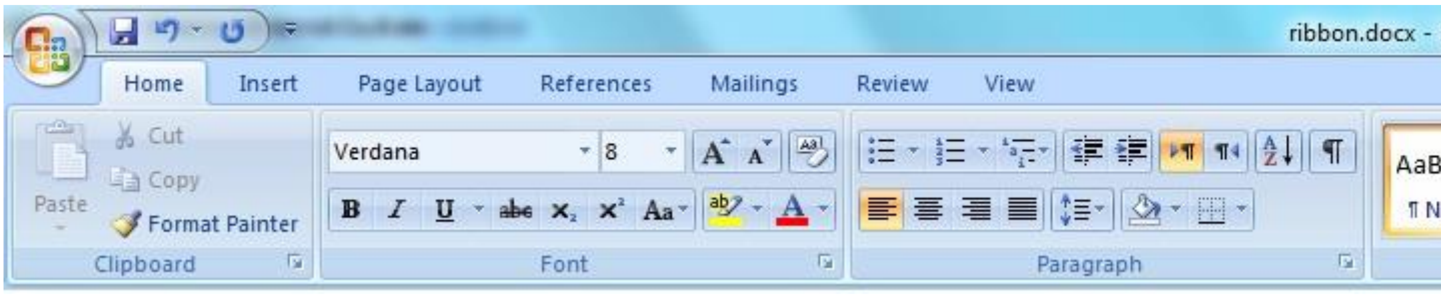
Along with the library I will develop a sample WinForms application (or more) that will show how to correctly use the library.

The most updated version of the code will be available on Github.

Background
So, before we start to code, let's give some background on the subject.

Definition (from MSDN): "*The Windows Ribbon (Ribbon) framework is a rich command presentation system that provides a modern alternative to the layered menus, toolbars, and task panes of traditional Windows applications.*"

In Office 2007 Microsoft introduced a new way to organize UI in an application. The problem with the old UI was that it wasn't flexible enough to represent the majority of office features. (People actually kept asking for features that already existed but were hidden deep in the menus…)



[Microsoft Word 2007 Ribbon]

After the Office 2007 Ribbon turned out to be a success, many UI companies provided their own implementation for a ribbon control to be used in your application.
When Microsoft realized they were onto something good they decided to supply the Ribbon feature to the general public.
They have released 3 (!) Ribbons:

- MFC version, available in Visual Studio 2008 SP1, to be used by native MFC developer
- WPF version, available in WPF toolkit for .Net 3.5 SP1 or integrated in .NET 4.5, to be used by managed WPF developers

- Windows Ribbon, available in Windows 7 and later Windows versions as a COM object [and in a future Vista update], to be used by native win32 developers.

So, what about managed WinForms users?
They should use the third, COM based version. So, the purpose of these posts is to give you a working example of using the Windows Ribbon within a WinForms application.

Why use Windows Ribbon?
Note that the question I want to answer is not "Why use A ribbon control?" but "Why use THIS ribbon control?"
Why not use other ribbon controls? There are dozens of Third-party controls; some of them are free.

The main reason why to use the Windows Ribbon Framework: It's developed by Microsoft. This means:

- Since it's the original one, it contains ALL the features, as opposed to other free/commercial ribbon controls which always have those "not implemented" sections.
- It has COMPLETE support and integration with windows 7 UI & accessibility features. Just think about touch screen support or high DPI screens compatibility. Later versions of Windows are welcome.

Part 2 – Basic Ribbon Wrapper

Today we will start looking at some code, but before we begin you might want to know where to find it.

The most updated version of "Windows Ribbon for WinForms" code will be at https://github.com/harborsiem/WindowsRibbon

It will include the latest version of the project code and samples of how to use its different features.

Let's get started.

Creating .NET wrappers
The first thing needed for using Windows Ribbon from .NET is converting the C++ / COM definitions to C# ones.

The files relevant to Ribbon are: UIRibbon.idl, UIRibbonKeydef.h and UIRibbonPropertyHelpers.h. Note that UIRibbon.h is not interesting since it is auto-generated from UIRibbon.idl by the MIDL compiler. All these files are installed with Windows 7 SDK or a later Windows SDK.

I won't discuss the details of the conversion since it's a very mechanical process, just change every C++ type and convert it to its corresponding .NET equivalent. If you are interested in these details, there are endless sources of information on .NET interoperability.

Following the Windows API Code Pack convention, the file UIRibbon.idl was converted to 4 different files:

- RibbonProperties.cs – containing Ribbon properties definition
- RibbonCOMGuids.cs – containing all Ribbon related GUIDs
- RibbonCOMInterfaces.cs – containing Ribbon interfaces definitions
- RibbonCOMClasses.cs – containing Ribbon classes definitions

These files are my conversions of the COM interfaces and types used by Windows Ribbon Framework. These files may change as the project continues since surely, I had some conversions error that will be discovered only when I'll try to use a certain feature.
The most updated version will be the one on Github.

How does the Windows Ribbon Framework work?
The full details are provided in MSDN, which I recommend reading. Here I'll only give a short overview so we are all on the same page.

To initialize a ribbon in your application you need to do the following:

1. Design your ribbon appearance using XAML-like markup.
2. Compile your XAML-like markup using Microsoft Ribbon Markup Compile, provided with Windows 7 SDK.
3. Have the binary output of the markup compiler stored as a (unmanaged) resource of your application.
4. On application load, *CoCreateInstance* the *UIRibbonFramework* class which implements IUIFramework interface
5. Call framework.Initialize and pass it a reference to your implementation of the IUIApplication interface along with the HWND of your application window.
6. The IUIApplication interface supply a callback for the ribbon framework to call when it needs a command handler, for handling commands (represented as buttons, combos and the other usual controls).
7. Call framework.LoadUI which loads the pre-compiled resource and shows the actual ribbon.

What have I done, up until now?
In order to facilitate the use of the ribbon within .NET applications we will create a class that will be used as a façade for the Windows Ribbon Framework.
Ribbon class is able to support applications with one or more ribbons on different forms.
This class, named *Ribbon*, will be in charge of initialization and communication with the Windows Ribbon Framework.

The Ribbon class will provide an implementation of *IUIApplication* and handle all the COM details. The Ribbon class is derived from the System.Windows.Forms.Control class and handle the steps 4 to 7 written before. The Ribbon must be placed to a Windows Form directly with Top docking.

The idea is to have your application provide only the minimal required details for the ribbon to work.

The Ribbon class currently exposes 2 methods:

- InitFramework – receives the resource name that will be used to load the ribbon configuration.
- DestroyFramework – cleanup code, free Windows Ribbon Framework resources.

These methods are called by the Ribbon class when the handle is created and destroyed, respectively.

Summary
This post turned out to be quite long, so I think I'll just stop here for now.
I've put the library code on Github along with a sample application that uses the code to create a simple WinForms application with Ribbon support.

On my next post I'll provide simple details on how to build your first WinForms Ribbon application using what we've developed so far.

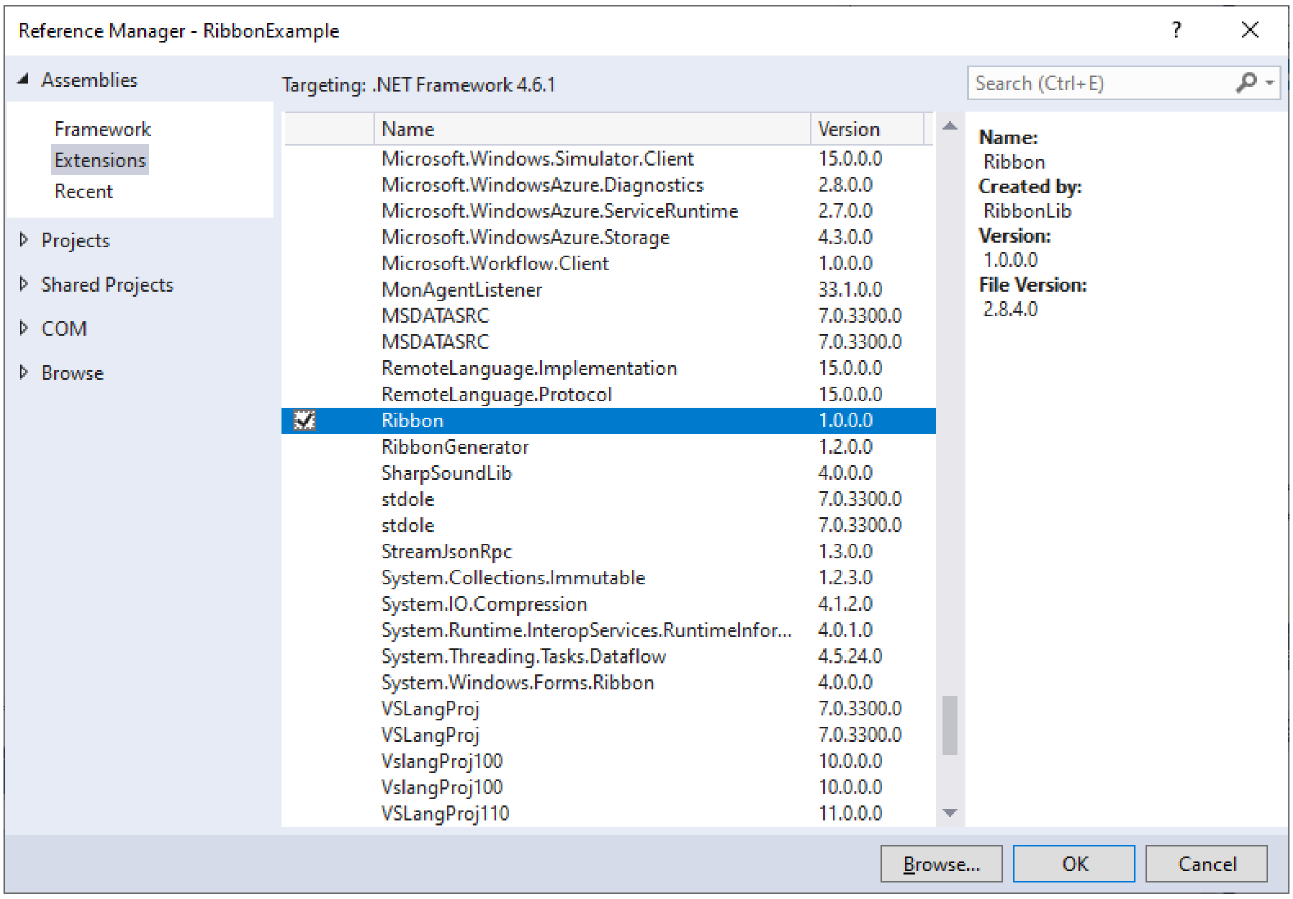
Part 3 – First WinForms Ribbon Application

So, let's see how to use the Ribbon class to add ribbon support to an empty WinForms application.

Step 0 – Download Windows Ribbon for WinForms
As I've mentioned in previous posts, you can download the code for my ribbon wrapper library along with numerous samples from https://github.com/harborsiem/WindowsRibbon .
In this post I'll review how to create your first, ribbon-enabled WinForms application, the result is the sample application "01-AddingRibbonSupport", included on the site. You should also download the latest MSI setups from the Releases page. Unpack the msi.zip and install all *.msi files by double clicking. Maybe you get a warning during installation. This is because the msi files are not signed, but safe.

Step 1 – Reference Windows Ribbon for WinForms
Create a new C# WinForms application and add a reference to our developed library, *Ribbon.dll*

(Example based on Visual Studio 2017 or 2019:) In Solution Explorer References: choose -> Add Reference -> Assemblies -> Extensions -> Ribbon.dll


Reference Manager - RibbonExample

Assemblies: Framework, Extensions, Recent; Projects; Shared Projects; COM; Browse

Targeting: .NET Framework 4.6.1

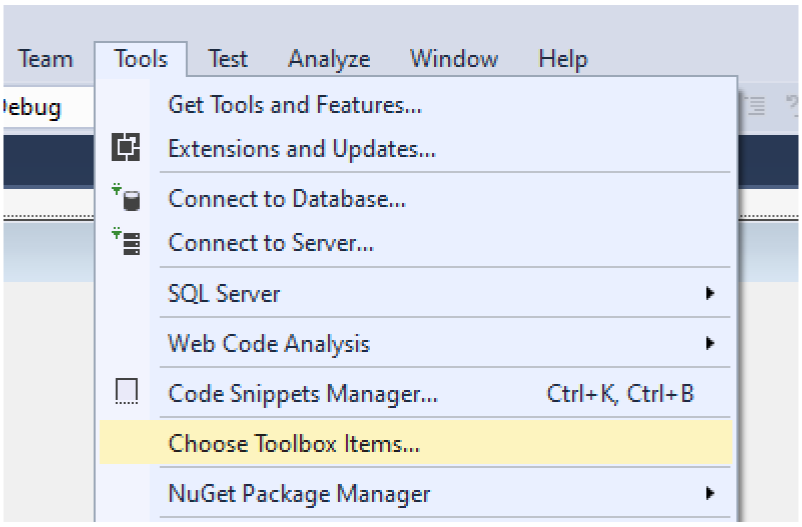
Search (Ctrl+E)

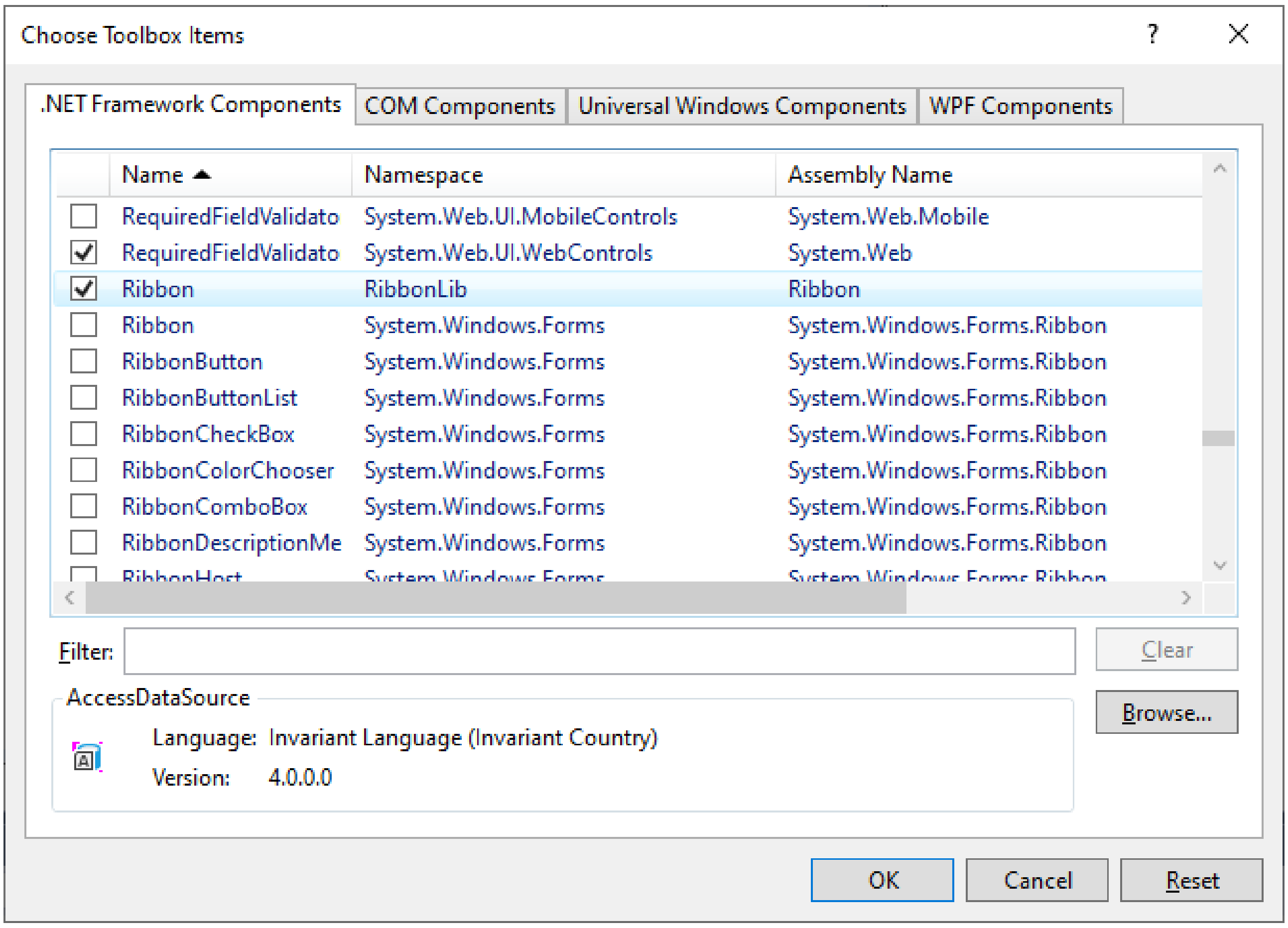
	Name	Version
	Microsoft.Windows.Simulator.Client	15.0.0.0
	Microsoft.WindowsAzure.Diagnostics	2.8.0.0
	Microsoft.WindowsAzure.ServiceRuntime	2.7.0.0
	Microsoft.WindowsAzure.Storage	4.3.0.0
	Microsoft.Workflow.Client	1.0.0.0
	MonAgentListener	33.1.0.0
	MSDATASRC	7.0.3300.0
	MSDATASRC	7.0.3300.0
	RemoteLanguage.Implementation	15.0.0.0
	RemoteLanguage.Protocol	15.0.0.0
✓	Ribbon	1.0.0.0
	RibbonGenerator	1.2.0.0
	SharpSoundLib	4.0.0.0
	stdole	7.0.3300.0
	stdole	7.0.3300.0
	StreamJsonRpc	1.3.0.0
	System.Collections.Immutable	1.2.3.0
	System.IO.Compression	4.1.2.0
	System.Runtime.InteropServices.RuntimeInfor...	4.0.1.0
	System.Threading.Tasks.Dataflow	4.5.24.0
	System.Windows.Forms.Ribbon	4.0.0.0
	VSLangProj	7.0.3300.0
	VSLangProj	7.0.3300.0
	VslangProj100	10.0.0.0
	VslangProj100	10.0.0.0
	VSLangProj110	11.0.0.0

Name: Ribbon
Created by: RibbonLib
Version: 1.0.0.0
File Version: 2.8.4.0

Browse... OK Cancel

Visual Studio Menu: Tool -> Choose Toolbox Items: Select Ribbon with Namespace RibbonLib




Choose Toolbox Items

.NET Framework Components | COM Components | Universal Windows Components | WPF Components

	Name	Namespace	Assembly Name
☐	RequiredFieldValidato	System.Web.UI.MobileControls	System.Web.Mobile
✓	RequiredFieldValidato	System.Web.UI.WebControls	System.Web
✓	Ribbon	RibbonLib	Ribbon
☐	Ribbon	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonButton	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonButtonList	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonCheckBox	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonColorChooser	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonComboBox	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonDescriptionMe	System.Windows.Forms	System.Windows.Forms.Ribbon
☐	RibbonHost	System.Windows.Forms	System.Windows.Forms.Ribbon

Filter:

AccessDataSource

Language: Invariant Language (Invariant Country)

Version: 4.0.0.0

Also, add to your main form (form1.cs) the following lines:

```
using RibbonLib;
using RibbonLib.Interop;

public partial class Form1 : Form
{
    private Ribbon _ribbon = new Ribbon();
    …
```

Instead defining "`private Ribbon _ribbon` …" you should place a Ribbon with the WinForms Designer with Dock.Top to the Form.

Step 2 – Add ribbon markup XML file

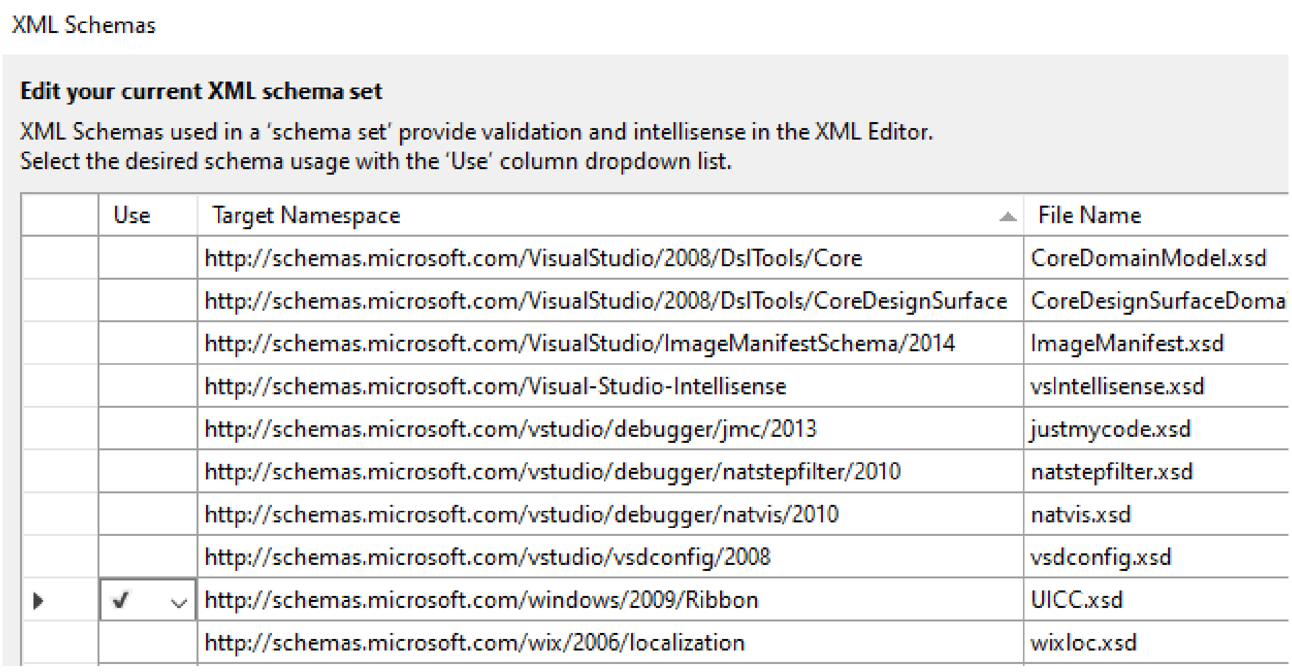
Add an empty file named RibbonMarkup.xml to the Visual Studio project with "Properties -> Build Action" = Content.

Set the file text to the following:

```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns='http://schemas.microsoft.com/windows/2009/Ribbon'>
  <Application.Commands>
  </Application.Commands>
 <Application.Views>
    <Ribbon>
    </Ribbon>
  </Application.Views>
</Application>
```

Right click on the editor where RibbonMarkup.xml is opened and click properties, now set the *Schemas* property to: *C:\Program Files\Microsoft SDKs\Windows\v7.0\Bin\UICC.xsd* or a path of a later SDK version.

Alternative you can do it by Visual Studio Menu: XML -> Schemas...: Select http://schemas.microsoft.com/windows/2009/Ribbon with Filename UICC.xsd (UICC.xsd is in the Windows SDK bin folder). Maybe you have to add this schema first in the dialog "XML Schemas".


XML Schemas

Edit your current XML schema set

XML Schemas used in a 'schema set' provide validation and intellisense in the XML Editor.
Select the desired schema usage with the 'Use' column dropdown list.

	Use	Target Namespace	File Name
		http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core	CoreDomainModel.xsd
		http://schemas.microsoft.com/VisualStudio/2008/DslTools/CoreDesignSurface	CoreDesignSurfaceDoma
		http://schemas.microsoft.com/VisualStudio/ImageManifestSchema/2014	ImageManifest.xsd
		http://schemas.microsoft.com/Visual-Studio-Intellisense	vsIntellisense.xsd
		http://schemas.microsoft.com/vstudio/debugger/jmc/2013	justmycode.xsd
		http://schemas.microsoft.com/vstudio/debugger/natstepfilter/2010	natstepfilter.xsd
		http://schemas.microsoft.com/vstudio/debugger/natvis/2010	natvis.xsd
		http://schemas.microsoft.com/vstudio/vsdconfig/2008	vsdconfig.xsd
▶	✓	http://schemas.microsoft.com/windows/2009/Ribbon	UICC.xsd
		http://schemas.microsoft.com/wix/2006/localization	wixloc.xsd

This will [visual] assist you in the future while editing the ribbon markup file (try Ctrl+Space on the xml editor to see the ribbon markup syntax).

Note: this is the moment where some of you will discover they didn't install Windows 7 SDK or a later Windows SDK, so go ahead and fix it, I'll wait. You should also install the C++ Tools in Visual Studio, because we need the Linker Link.exe.

Step 3 – Compile markup XML file

Open a Console window at your markup XML file and type "rgc RibbonMarkup.xml" without "".

This Command calls the Markup compiler UICC.exe, the ResourceCompiler Rc.exe and the Linker Link.exe to build a Resource dll named RibbonMarkup.ribbon and a file RibbonItems.Designer.cs.

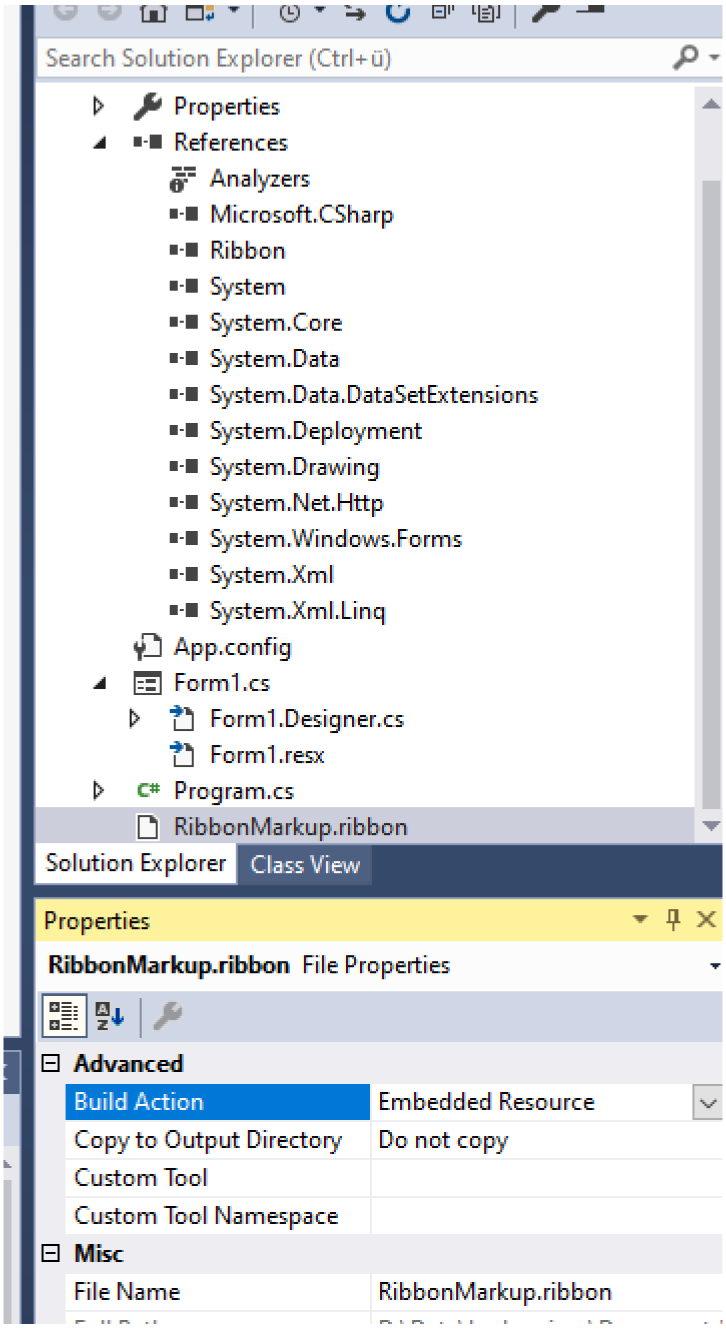
Another way is to use the program RibbonPreview, which you can find in the Windows StartMenu. Now you can select your RibbonMarkup.xml and build the RibbonMarkup.ribbon and a file named RibbonItems.Designer.cs.

Explanation:
The first program UICC.exe compiles the ribbon markup file to a binary compressed format, along with a small RC file that describes it.

The second program Rc.exe creates a native win32 resource file to be attached to the project native resources.
The third program Link.exe creates a resource dll from the native win32 resource file.

Step 4 – Add RibbonMarkup.ribbon to the project

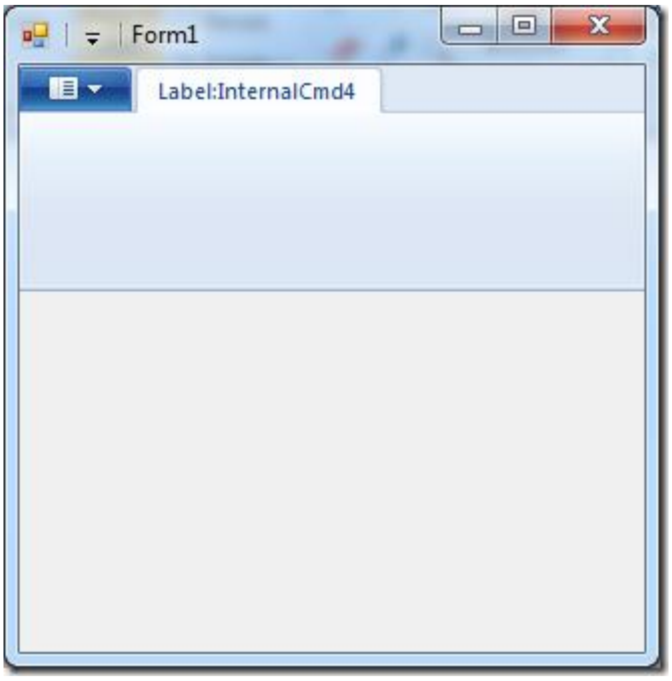
Add the files RibbonMarkup.ribbon and RibbonItems.Designer.cs to your project. The RibbonMarkup.ribbon should have the Build Property “Embedded Resource” and RibbonItems.Designer.cs have the Build Property “Compile”.



Build Action	Embedded Resource
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	RibbonMarkup.ribbon

Mark in Windows Form Designer to the Ribbon. Now you have to set in the Properties Page of the Ribbon Control the Property ResourceName to [default namespace of the project].RibbonMarkup.ribbon .

Step 5 – Enjoy your first Ribbon WinForms application



Note: If you run your application and don't see the ribbon, try to enlarge the window size. The windows ribbon has a feature that if the window size is too small, it doesn't show. Unfortunately, Visual Studio default form size is too small.

Part 4 – Application Menu with Buttons

Before we start to use ribbon features, we must learn the basics of ribbon markup.

Commands and Views
A command is an action that is identified by a number, it can be opening the save-as dialog, printing the current document, closing the application, etc. everything you can do in a function call.

A view is a graphical representation of [usually several] commands. It defines the type of controls used to activate the commands and their size, order and layout on screen.

So using commands and views is actually just another instance of the MVC design pattern, which allows us to separate business logic from presentation logic.

Now we will write a new WinForms application with ribbon that uses the application menu with simple buttons. We start this sample with an empty WinForms project that already includes ribbon support (see previous post for details). On the next sections I'll explain:

- Commands part of the ribbon markup
- Views part of the ribbon markup
- code-behind, responding to ribbon events

As always, the entire code is available at github.com/harborsiem/WindowsRibbon

General markup review

Just a reminder, our basic ribbon markup looks like this:

```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns='http://schemas.microsoft.com/windows/2009/Ribbon'>
  <Application.Commands>
  </Application.Commands>
 <Application.Views>
    <Ribbon>
    </Ribbon>
  </Application.Views>
</Application>
```

Defining Commands in Ribbon Markup

Following is a definition of some commands in ribbon markup:

```
<Application.Commands>
  <Command Name="cmdButtonNew"
           Id="1001"
           LabelTitle="&amp;New"
           LabelDescription="New Description"
           TooltipTitle="New"
           TooltipDescription="Create a new image.">
    <Command.LargeImages>
      <Image>Res/New32.bmp</Image>
    </Command.LargeImages>
    <Command.SmallImages>
      <Image>Res/New16.bmp</Image>
    </Command.SmallImages>
  </Command>

  <Command Name="cmdButtonOpen"
           Id="1002"
           LabelTitle="Open"
           LabelDescription="Open Description"
           TooltipTitle="Open"
           TooltipDescription="Open an existing image.">
    <Command.LargeImages>
      <Image>Res/Open32.bmp</Image>
    </Command.LargeImages>
    <Command.SmallImages>
      <Image>Res/Open16.bmp</Image>
    </Command.SmallImages>
  </Command>

  <Command Name="cmdButtonSave"
           Id="1003"
           LabelTitle="Save"
           LabelDescription="Save Description"
           TooltipTitle="Save"
           TooltipDescription="Save the current image.">
    <Command.LargeImages>
      <Image>Res/Save32.bmp</Image>
    </Command.LargeImages>
    <Command.SmallImages>
      <Image>Res/Save16.bmp</Image>
    </Command.SmallImages>
  </Command>

  <Command Name="cmdButtonExit"
           Id="1004"
           LabelTitle="Exit"
           LabelDescription="Exit Description"
           TooltipTitle="Exit"
           TooltipDescription="Exit application.">
    <Command.LargeImages>
      <Image>Res/Exit32.bmp</Image>
    </Command.LargeImages>
    <Command.SmallImages>
      <Image>Res/Exit16.bmp</Image>
    </Command.SmallImages>
  </Command>
</Application.Commands>
```

Explanation: here we define 4 different commands. Each command has properties assigned either by xml attributes or child elements. We use the following (full list is available at "Commands and Resources" on MSDN):

- Name – this name is used later in the views section to reference to this command
- Id – this is the ID of the command. We get it in code when a command event occurs.
- LabelTitle – the label title of the command
- LabelDescription – the label description of the command
- TooltipTitle – the tooltip title of the command
- TooltipDescription – the tooltip description of the command
- LargeImages – large image filename for the command, usually 32×32 pixels
- SmallImages – small image filename for the command, usually 16×16 pixels

Setting shortcuts to menu items
Setting a key shortcut for a menu item is done by adding "&amp;" in LabelTitle before the letter you want as a shortcut (similar to shortcuts in the "old" menu system), see the LabelTitle of "New" command for example.

Some comments about image resources in Ribbon markup
The filename defined in the markup (like in LargeImages and SmallImages element), should be a valid (relative or full) path to a filename, otherwise the resource compiler (rc.exe) will output a compilation error: "error RC2135: file not found: <filename>".

The image file format should be BMP with 32 BPP ARGB pixel format. Many image editing programs, like Microsoft Paint do not preserve the highest order 8-bit alpha channel when saving, thus creating only 24-bit images, the result is that the image will not appear at all.
Update (18.11.2009): convert2bmp is a tool that enables you to convert your images to the required format.

Under both images elements you can put several image files in different sizes, the ribbon framework will choose the best size according to the current DPI setting. For us, normal users, setting two images for 32×32 and 16×16 should be enough. For more information, see "Specifying Ribbon Image Resources" on MSDN.

Defining Views in Ribbon Markup

Following is a definition of the Application.Views part of our ribbon markup:

```
<Application.Views>
  <Ribbon>
    <Ribbon.ApplicationMenu>
      <ApplicationMenu>
        <MenuGroup>
          <Button CommandName='cmdButtonNew' />
          <Button CommandName='cmdButtonOpen' />
          <Button CommandName='cmdButtonSave' />
        </MenuGroup>
        <MenuGroup>
          <Button CommandName='cmdButtonExit' />
        </MenuGroup>
      </ApplicationMenu>
    </Ribbon.ApplicationMenu>
  </Ribbon>
</Application.Views>
```

Explanation: here we define an application menu that contains two menu groups and 4 buttons. The button CommandName attribute points to the command that this button should trigger upon click.

Handling Ribbon Events

Here we will see how to handle the event of clicking of one of our menu buttons.

The following code reside in our RibbonItems.Designer.cs code file:

```
private static class Cmd
{
     public const uint cmdApplicationMenu = 1000;
     public const uint cmdButtonNew = 1001;
     public const uint cmdButtonOpen = 1002;
     public const uint cmdButtonSave = 1003;
     public const uint cmdButtonExit = 1004;
}
public RibbonApplicationMenu ApplicationMenu { get; private set; }
public RibbonButton ButtonNew { get; private set; }
public RibbonButton ButtonOpen { get; private set; }
public RibbonButton ButtonSave { get; private set; }
public RibbonButton ButtonExit { get; private set; }
```

This is just a generated helper class, to make the code more readable. Every command ID gets a readable symbol.

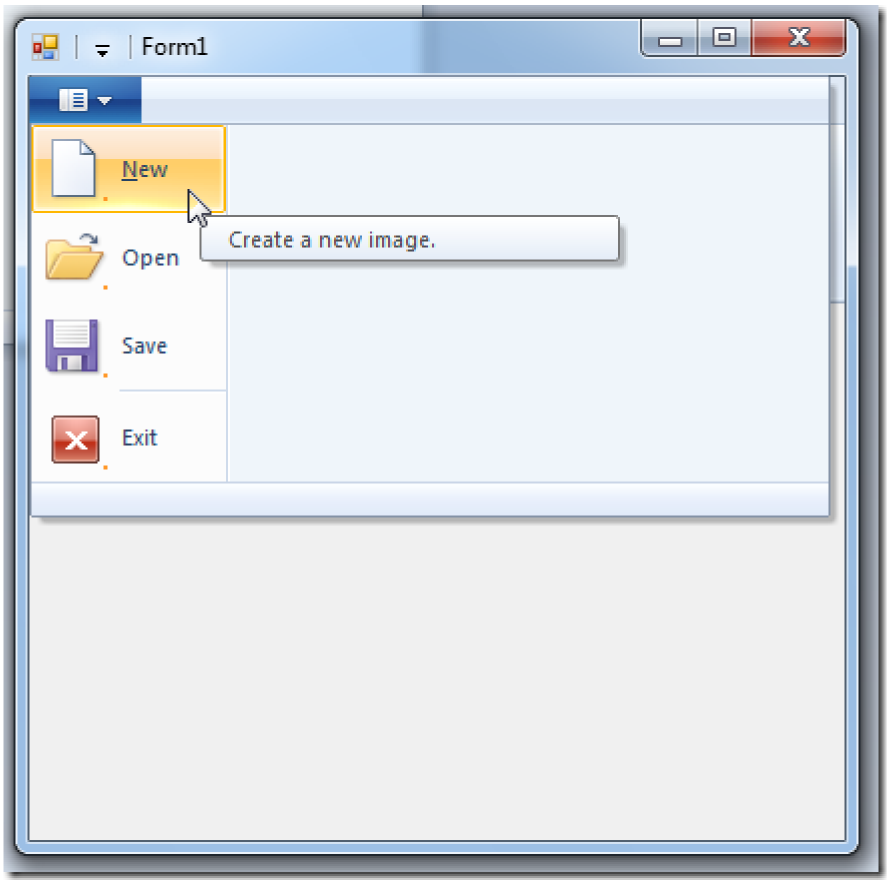
Update (18.11.2009): Handling ribbon events is now as simple as normal .NET events. Implementing *IUICommandHandler* by the user is no longer required.

```
private RibbonItems _ribbonItems;
private RibbonButton _buttonNew;
public Form1()
{
    InitializeComponent();
  _ribbonItems = new RibbonItems(_ribbon);
    _buttonNew = _ribbonItems.ButtonNew);
  _buttonNew.ExecuteEvent += new EventHandler<ExecuteEventArgs>
(_buttonNew_ExecuteEvent);
}
void _buttonNew_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    MessageBox.Show("new button pressed");
}
```

Naturally we added to the beginning of the file:

```
using RibbonLib;
using RibbonLib.Controls;
using RibbonLib.Controls.Events;
using RibbonLib.Interop;
```

So, there you have it, a WinForms application with a Ribbon Application Menu.

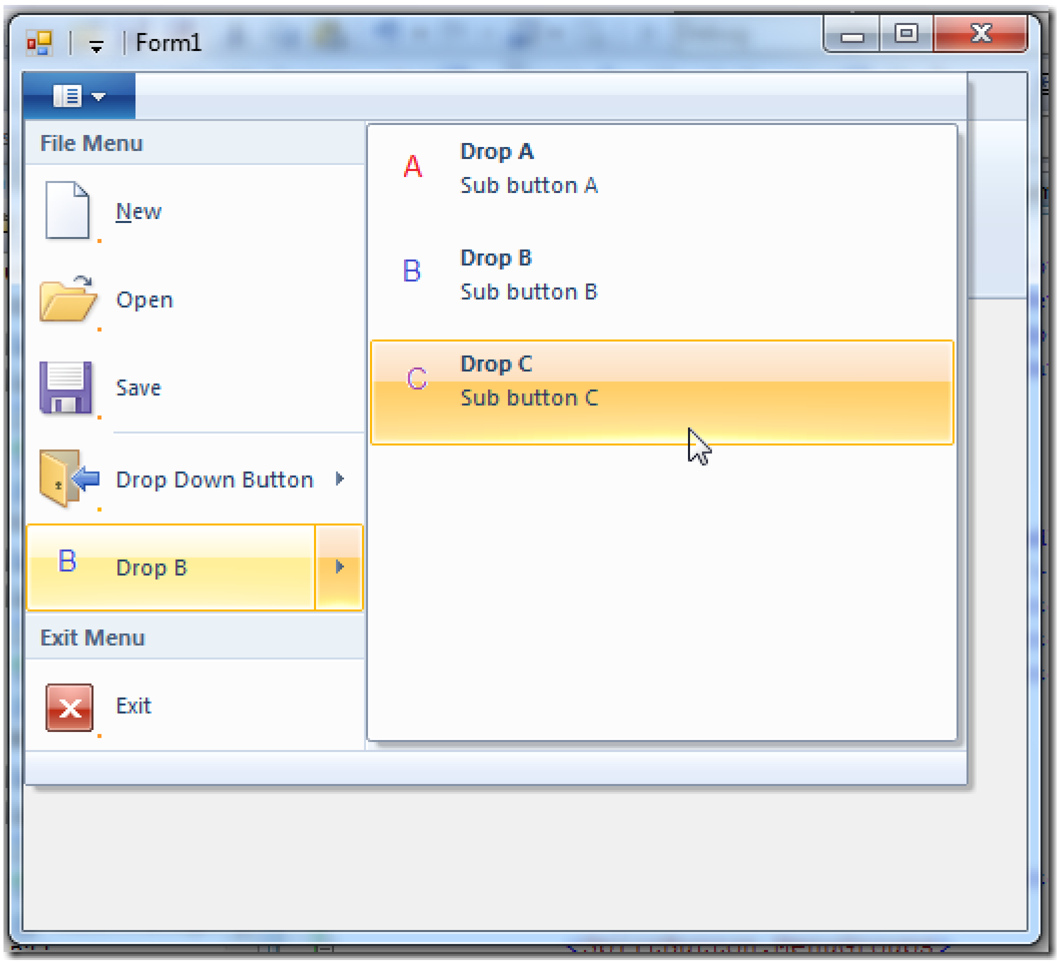


Part 5 – Application Menu with SplitButton and DropDownButton

Today I'll show you how to use the following ribbon features inside a WinForms application:

- Menu Group
- Split Button in an Application Menu
- Drop Down Button in an Application Menu

The result of this post looks like this:



SplitButton vs DropDownButton
What is actually the difference between those two?

DropDownButton is NOT a button, meaning clicking it does nothing. Hovering over it will open a list of buttons.
On the other hand, SplitButton is itself a button, which you can respond to. Hovering over it will also open a list of buttons.

The common use for a DropDownButton is when you want to expose a set of items which doesn't have an obvious default option. Think of "Rotate" feature in Paint. you have Rotate90, Rotate180 and Rotate270 but none of them is an obvious default.

The common use for a SplitButton is when you want to expose a set of items which has an obvious default option. Think of "Save As" button, where there is a default save format.

Using SplitButton and DropDownButton in Ribbon Application Menu
The commands markup is the same as before, just define some command to be listed later in the Application.Views markup. For example:

```
<Command Name="cmdButtonDropA"
    Id="1008"
    LabelTitle="Drop A"
    LabelDescription="Sub button A"
    TooltipTitle="Drop A">
  <Command.LargeImages>
    <Image>Res/DropA32.bmp</Image>
  </Command.LargeImages>
</Command>
<Command Name="cmdButtonDropB"
    Id="1009"
    LabelTitle="Drop B"
    LabelDescription="Sub button B"
    TooltipTitle="Drop B">
  <Command.LargeImages>
    <Image>Res/DropB32.bmp</Image>
  </Command.LargeImages>
</Command>
```

The relevant views markup is defined as follows:

```
<DropDownButton CommandName='cmdDropDownButton'>
  <MenuGroup Class='MajorItems'>
    <Button CommandName='cmdButtonDropA' />
    <Button CommandName='cmdButtonDropB' />
    <Button CommandName='cmdButtonDropC' />
  </MenuGroup>
</DropDownButton>
<SplitButton>
  <SplitButton.ButtonItem>
    <Button CommandName='cmdButtonDropB' />
  </SplitButton.ButtonItem>
  <SplitButton.MenuGroups>
    <MenuGroup Class='MajorItems'>
      <Button CommandName='cmdButtonDropA' />
      <Button CommandName='cmdButtonDropB' />
      <Button CommandName='cmdButtonDropC' />
    </MenuGroup>
  </SplitButton.MenuGroups>
</SplitButton>
```

The code behind section is also the same as in the previous post. Just register to the *ExecuteEvent* event of the corresponding button.

MenuGroup
A menu group is a collection of menu items inside the application menu. the most useful feature it provides is giving a title to a group of items, like “File Menu” in the last image.

If you just want a simple separator between menu items you use a MenuGroup that is not attached to any command.

```
<MenuGroup>
  …
</MenuGroup>
```

As always, the most updated version of “Windows Ribbon for WinForms” along with sample applications can be found here.

Part 6 – Tabs, Groups and HelpButton

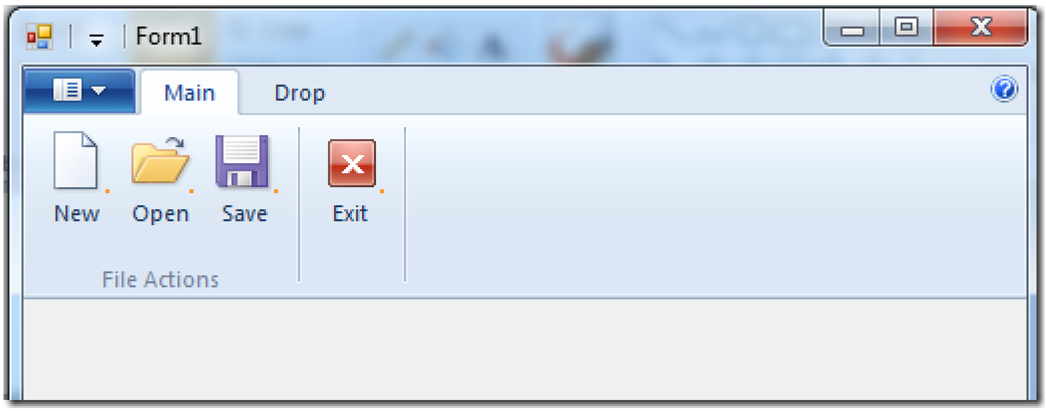
First, you should know that I changed my ribbon library such that Ribbon class is no longer a singleton.
The reason is that I wanted to make an application with two forms, each form has its own ribbon, so you needed two ribbon objects. So, this had to change.

Hint: If you have more than one Ribbon Control for different Forms in the project, then you should name the RibbonMarkup.xml to RibbonMarkup1.xml, RibbonMarkup2.xml, ... RibbonMarkup9.xml because we got different classes of RibbonItems(x).Designer.cs .

Second, in this post I’ll review some more common ribbon features, namely:

- Tabs
- Groups
- Help Button

The result of this post is yet another sample application, named 04-TabGroupHelp. You can find it on the project page and it looks like this:



Buttons, Buttons, Buttons
So what are tabs? Just containers for other controls. In this sample will use only **buttons**. I’ll elaborate on the other control types on future posts.

Every tab can contain several groups, which are just a logical division of the controls in the tab, in this post – **buttons**…

What is the help button? just another **button**. For some reason it got its own special place and predefined icon (look on the right side of the image).

Using Tabs and Groups

The commands markup is always the same, just a list of items that attach a mnemonic used by the programmer with an ID used by the ribbon framework. And some string and bitmap resources.

As always, the interesting part lies in the Application.Views markup:

```
<Application.Views>
  <Ribbon>
    <Ribbon.Tabs>
      <Tab CommandName="cmdTabMain">
        <!- scary part ->
        <Tab.ScalingPolicy>
          <ScalingPolicy>
            <ScalingPolicy.IdealSizes>
              <Scale Group="cmdGroupFileActions" Size="Large" />
              <Scale Group="cmdGroupExit" Size="Large" />
            </ScalingPolicy.IdealSizes>
            <Scale Group="cmdGroupFileActions" Size="Medium" />
          </ScalingPolicy>
        </Tab.ScalingPolicy>
        <!- useful part ->
        <Group CommandName="cmdGroupFileActions" SizeDefinition="ThreeButtons">
          <Button CommandName="cmdButtonNew" />
          <Button CommandName="cmdButtonOpen" />
          <Button CommandName="cmdButtonSave" />
        </Group>
        <Group CommandName="cmdGroupExit" SizeDefinition="OneButton">
          <Button CommandName="cmdButtonExit" />
        </Group>
      </Tab>
      <Tab CommandName ="cmdTabDrop">
        <Group CommandName="cmdGroupDrop" SizeDefinition="ThreeButtons">
          <Button CommandName="cmdButtonDropA" />
          <Button CommandName="cmdButtonDropB" />
          <Button CommandName="cmdButtonDropC" />
        </Group>
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

What I've defined in this markup is two tabs. The first tab has two groups and the second tab has one group.
I've marked to important parts of the markup in the first tab. Unfortunately, due to the ribbon schema definition the scary part must come before the useful part.

The useful part:
just a simple definition of the groups in the tab, and the controls in the group. one interesting thing to note is the *SizeDefinition* attribute on the *Group* tag. This is a definition of the layout for the controls in the group. You can define your own layouts or use a predefined list which usually is quite enough. see the full list (with convenient images) at "Customizing a Ribbon Through Size Definitions and Scaling Policies" on MSDN.

The scary part:
In order to understand this part you should know that one of the features of the ribbon framework is the ability to re-layout your ribbon controls according to the amount of space it has. It pretty much handle this automatically but it does require you to define hints on how you want your layout to scale when the application form gets smaller and smaller. So, in the scary part we first define the ideal size of each group. The size can be one of four values: Large, Medium, Small and Popup. Popup means that the whole group was shrunk to a single icon that upon clicking popups the original group.

After defining the ideal size for the group, you can define the order of scaling down, meaning which group should scale down first. In this way you can make your application most important controls more visible then less important ones.

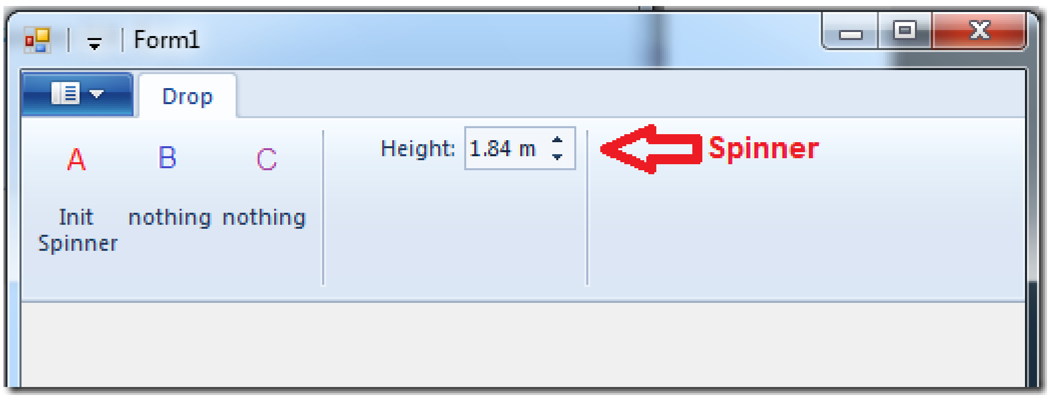
The code-behind for acting on these buttons is the same as in the previous posts. Just register to the ExecuteEvent event of the correct button.

Using Help Button
To use the help button just add the following to the view markup:

```
<Application.Views>
  <Ribbon>
    <Ribbon.HelpButton>
      <HelpButton CommandName="cmdHelp" />
    </Ribbon.HelpButton>
    <Ribbon.Tabs>
      …
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

Part 7 – Spinner

Spinner Control
A Spinner control is a control that represents a decimal value (like double, only with higher resolution). The control is composed of an edit box and two buttons, for increment and decrement. Let’s try the thing with the 1000 words:



Spinner Properties
Every ribbon control has properties that defines the way it looks and behaves. Here is a quick review of spinner properties, divided into logical groups:

Spinner Value Related Properties (internal details)

- Decimal Value – The actual decimal value of the spinner.
  Property Identifier: UI_PKEY_DecimalValue
- Increment – The size of the step when pressing on increment / decrement buttons.
  Property Identifier: UI_PKEY_Increment
- Max Value – Maximum value that can be set using the spinner control.
  Property Identifier: UI_PKEY_MaxValue
- Min Value – Minimum value that can be set using the spinner control.
  Property Identifier: UI_PKEY_MinValue

Note: When using decimal values with the Ribbon Framework you must use a PropVariant implementation that support decimal values. More on this at: Setting Decimal value on PropVariant.

Spinner Appearance Related Properties

- Decimal Places – The number of digits to show after the point.
  Property Identifier: UI_PKEY_DecimalPlaces
- Format String – The units of the value. In the previous image, it is “m”, for meters.
  Property Identifier: UI_PKEY_FormatString
- Representative String – A string that represents the common value for the spinner. This is used to calculate the width of the spinner, so you should set here the longest string you forecast. Note that it doesn’t have to be an actual value, it can be also: “XXXXXXXX”.
  Property Identifier: UI_PKEY_RepresentativeString

Common Appearance Properties

- Keytip – The keyboard accelerator for this command in the ribbon framework (view it by pressing ALT in a ribbon application).
  Property Identifier: UI_PKEY_Keytip
- Label – The label for this command. Usually appears next to the attached control.
  Property Identifier: UI_PKEY_Label
- Tooltip Title – The title of the tooltip for this command.
  Property Identifier: UI_PKEY_TooltipTitle
- Tooltip Description – The description of the tooltip for this command.
  Property Identifier: UI_PKEY_TooltipDescription
- Enabled – Flag that indicates whether this control is enabled or not.
  Property Identifier: UI_PKEY_Enabled

Image Properties

- Large Image – The large image for this command.
  Property Identifier: UI_PKEY_LargeImage
- Small Image – The small image for this command.
  Property Identifier: UI_PKEY_SmallImage

- Large High Contrast Image – The large high contrast image for this command. Property Identifier: UI_PKEY_LargeHighContrastImage
- Small High Contrast Image – The small high contrast image for this command. Property Identifier: UI_PKEY_SmallHighContrastImage

Using Spinner – Ribbon Markup

As always, a command should be defined:

```
<Command Name="cmdSpinner"
         Id="1018"
         LabelTitle="My Spinner">
</Command>
```

The views section is also simple:

```
<Application.Views>
  <Ribbon>
    <Ribbon.Tabs>
      <Tab>
        <Group>
          <Spinner CommandName="cmdSpinner" />
        </Group>
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

Using Spinner – Code Behind

To help us manipulate the spinner I’ve created a helper class that encapsulates all the work regarding the ribbon framework. To use it, create a *RibbonSpinner* instance, passing to the constructor the *Ribbon* instance and command ID of the spinner:

```
private RibbonItems _ribbonItems;
private RibbonSpinner _spinner;

public Form1()
{
    InitializeComponent();
  _ribbonItems = new RibbonItems(_ribbon);
    _spinner = _ribbonItems.Spinner;
}
```

Now you can manipulate the spinner properties easily, by setting them on the *_spinner* instance, for example:

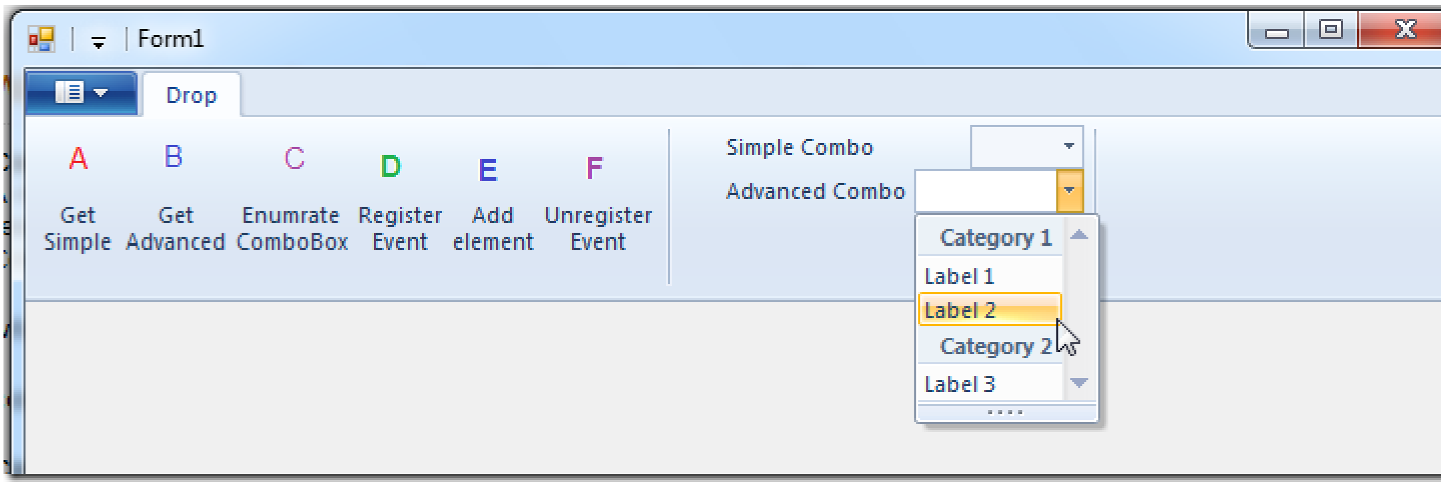
```
private void InitSpinner()
{
    _spinner.DecimalPlaces = 2;
    _spinner.DecimalValue = 1.8M;
    _spinner.TooltipTitle = "Height";
    _spinner.TooltipDescription = "Enter height in meters.";
    _spinner.MaxValue = 2.5M;
    _spinner.MinValue = 0;
    _spinner.Increment = 0.01M;
    _spinner.FormatString = " m";
    _spinner.RepresentativeString = "2.50 m";
    _spinner.Label = "Height:";
}
```

Where can we get it?
You can find a working sample that demonstrates using a spinner control at Windows Ribbon for WinForms under sample "05-Spinner".

Part 8 – ComboBox

ComboBox Control

A ribbon ComboBox control is basically the normal ComboBox control that we all love, but with the additional feature of dividing the items into categories. A category is not an item and cannot be selected from the ComboBox. It is only used to organized the items.



ComboBox Properties

Every ribbon control has properties that defines the way it looks and behaves. Here is a quick review of ComboBox properties, divided into logical groups:

ComboBox Value Related Properties (internal details)

- Items Source – The list of ComboBox items. It is exposed as an *IUICollection* where every element in the collection is of type: *IUISimplePropertySet*. More on this later.
  Property Identifier: UI_PKEY_ItemsSource
- Categories – The list of categories. Also exposed as an IUICollection of *IUISimplePropertySet* elements.
  Property Identifier: UI_PKEY_Categories
- Selected Item – The index of the selected item in the ComboBox. If nothing is selected returns *UI_Collection_InvalidIndex*, which is a fancy way to say -1.
  Property Identifier: UI_PKEY_SelectedItem
- String Value – The current string in the ComboBox. This can be a string that isn't one of the possible items in the ComboBox, in case the ComboBox has *IsEditable* set to

true.
Property Identifier: UI_PKEY_StringValue

ComboBox Appearance Related Properties

- Representative String – A string that represents the common value for the ComboBox. This is used to calculate the width of the ComboBox, so you should set here the longest string you forecast. Note that it doesn't have to be an actual value, it can be also: "XXXXXXXX".
Property Identifier: UI_PKEY_RepresentativeString

Common Appearance & Image Properties

See these sections at Windows Ribbon for WinForms, Part 7 – Spinner

Using ComboBox – Ribbon Markup

As always, a command should be defined:

```
<Command Name="cmdComboBox2" Id="1019" />
```

The views section:

```
<Application.Views>
  <Ribbon>
    <Ribbon.Tabs>
      <Tab>
        <Group>
          <ComboBox CommandName="cmdComboBox2"
                    IsAutoCompleteEnabled="true"
                    IsEditable="true"
                    ResizeType="VerticalResize" />
        </Group>
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

ComboBox Attributes:

- *CommandName* – Name of the command attached to this ComboBox.
- *IsAutoCompleteEnabled* – Flag that indicated whether to complete the words as you write.
- *IsEditable* – Flag that indicates whether to allow free writing in the ComboBox.
- *ResizeType* – Allow resize of the ComboBox. Can be *NoResize* or *VerticalResize*.

Using ComboBox – Code Behind

In a similar way to the spinner control, I've created a helper classes that encapsulates the interaction between the ComboBox and ribbon framework. To use the ComboBox, create a *RibbonComboBox* instance, passing to the constructor the *Ribbon* instance and command ID of the ComboBox:

```
private RibbonItems _ribbonItems;
private RibbonComboBox _comboBox2;
public Form1()
{
    InitializeComponent();
```

```
  _ribbonItems = new RibbonItems(_ribbon);
    _comboBox2 = _ribbonItems.ComboBox2;
    _comboBox2.RepresentativeString = "XXXXXXXXXXX";
}
```

Note: We set the *RepresentativeString* property BEFORE the initializing the ribbon framework. This is because for some reason the framework reads this property only once, when the ribbon is initialized. This means that if you change it after initialization it will have no affect since the framework doesn't reads this property anymore. By the way, according to the current documentation of the ribbon framework, this property is not part of the ComboBox, but as mentioned earlier, this property controls the width of the ComboBox.

In the next code snippet, you can see how to use another helper class, named *GalleryItemPropertySet*. This class represents a container for properties of a single element in an *IUICollection*.

Adding categories and items to the ComboBox is done like this:

```
private void Form1_Load(object sender, EventArgs e)
{
  // set combobox2 label
    _comboBox2.Label = "Advanced Combo";

    // set _comboBox2 categories
    IUICollection categories2 = _comboBox2.Categories;
    categories2.Clear();
    categories2.Add(new GalleryItemPropertySet() { Label="Category 1",
CategoryID=1 });
    categories2.Add(new GalleryItemPropertySet() { Label="Category 2",
CategoryID=2 });
  // set _comboBox2 items
    IUICollection itemsSource2 = _comboBox2.ItemsSource;
    itemsSource2.Clear();
    itemsSource2.Add(new GalleryItemPropertySet() { Label="Label 1",
CategoryID=1 });
    itemsSource2.Add(new GalleryItemPropertySet() { Label="Label 2",
CategoryID=1 });
    itemsSource2.Add(new GalleryItemPropertySet() { Label="Label 3",
CategoryID=2 });
}
```

Note: Adding items and categories can be done only AFTER the Ribbon Framework has been initialized.

IUICollection Events

Objects that implements *IUICollection* interface usually expose an *OnChanged* event that is called when the collection has changed due to: Insert item, Remove item, Replace item, Reset collection.

This event is exposed using the standard COM events mechanism, namely: *IConnectionPointContainer*, *IConnectionPoint and Advise()*.

To help the user to avoid these issues altogether, I've created the *UICollectionChangedEvent* class, which attaches to a given IUICollection and exposes the ChangedEvent event as a normal .NET event.

Following is an example of using it:

```
private RegisterEvent()
{
    _uiCollectionChangedEvent = new UICollectionChangedEvent();
    _uiCollectionChangedEvent.Attach(_comboBox1.ItemsSource);
    _uiCollectionChangedEvent.ChangedEvent +=
        new
EventHandler<UICollectionChangedEventArgs>(_ChangedEvent_ChangedEvent);
}

void _ChangedEvent_ChangedEvent(object sender, UICollectionChangedEventArgs e)
{
    MessageBox.Show("Got ChangedEvent event. Action = " + e.Action.ToString());
}
```

Note: There is no *ChangedEvent* event for the ComboBox. Only for *IUICollection*, which is completely different.

Update (27.10.2009): The ComboBox itself has 3 events, *ExecuteEvent*, *PreviewEvent* and *CancelPreviewEvent*. The *ExecuteEvent* event can be used as a "Selected Change" event. See this future post.

Summary
You can find a working sample that demonstrates using a ComboBox control at Windows Ribbon for WinForms under sample "06-ComboBox".

Part 9 – Changing Ribbon Colors

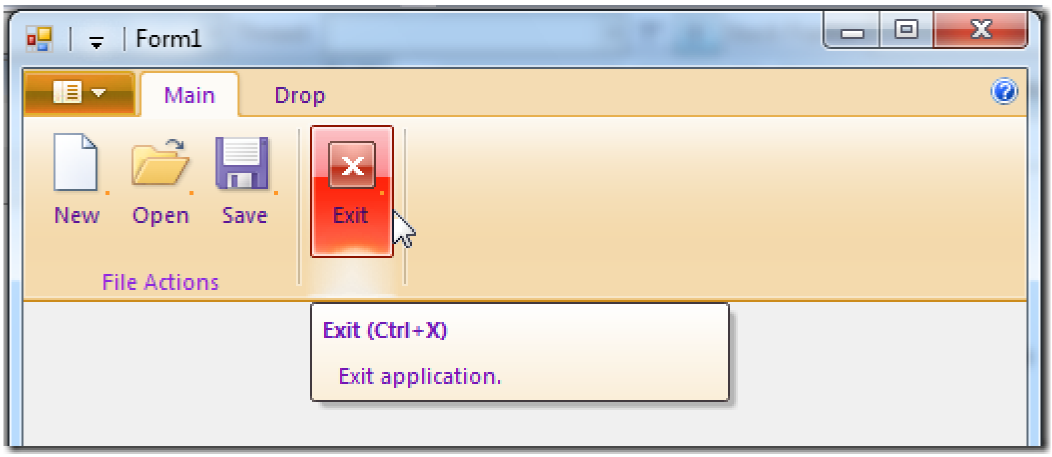
Introduction to the feature

The feature I want to talk about today is how to change the ribbon general colors. Note that you can't change the colors of a specific ribbon item, only the entire ribbon.

There are 3 colors we can change:

- Background Color
- Highlight Color
- Text Color

Here is an example of a colored ribbon:



How to do it?
I've added a new method to the *RibbonLib.Ribbon* class in my Windows Ribbon for WinForms library.
Following is an example of how to use it:

```
private void Form1_Load(object sender, EventArgs e)
{
    // set ribbon colors
   _ribbon.SetColors(Color.Wheat, Color.IndianRed, Color.BlueViolet);
}
```

Behind the scenes
What the SetColors method actually does is:

- Get *IPropertyStore* interface from the *IUIFramework* (which represents the ribbon framework)
- Create 3 PropVariant variables that will hold the 3 colors we want to set
- Convert the colors: RGB –> HSL –> HSB –> uint, see next section.
- Set the relevant properties with the converted colors values

```
public void SetColors(Color background, Color highlight, Color text)
{
   if (_framework == null)
   {
      return;
   }

   IPropertyStore propertyStore = (IPropertyStore)_framework;
   PropVariant backgroundColorProp = new PropVariant();
   PropVariant highlightColorProp = new PropVariant();
   PropVariant textColorProp = new PropVariant();
    uint backgroundColor = ColorHelper.RGBToUInt32(background);

   uint highlightColor = ColorHelper.RGBToUInt32(highlight);

   uint textColor = ColorHelper.RGBToUInt32(text);
```

```
    backgroundColorProp.SetUInt(backgroundColor);
    highlightColorProp.SetUInt(highlightColor);
    textColorProp.SetUInt(textColor);

    propertyStore.SetValue(ref RibbonProperties.UI_PKEY_GlobalBackgroundColor,
                    ref backgroundColorProp);
    propertyStore.SetValue(ref RibbonProperties.UI_PKEY_GlobalHighlightColor,
                    ref highlightColorProp);
    propertyStore.SetValue(ref RibbonProperties.UI_PKEY_GlobalTextColor,
                    ref textColorProp);
    propertyStore.Commit();
}
```

Color Format
I didn’t dive into the colors format world, so I’ll just give a quick reference:

RGB is the well-known color format that us, developers, like and understand.

RGB can be converted to HSL. This should be a common transformation. or as Microsoft wrote here, “easily accomplished with most photo editing software”. In this page Microsoft also gives the formulas for converting HSL to HSB.

You can find code for transforming RGB to HSL and vice versa at W3C or Wikipedia. The transformations are not lossless.

Finally, HSB is converted to uint by just OR-ing the values, much like RGB to uint conversion. I’ve encapsulated all these details in a helper class named *RibbonLib.ColorHelper*.

A new sample, named *07-RibbonColor*, summarize the details of this post. Find it on the project site.

Part 10 – Working with Images

In this post we’ll review the ribbon framework images terminology and see how to set images both statically and dynamically in your WinForms application.

More details can be found at Specifying Ribbon Image Resources on MSDN.

Large Images vs. Small Images
Many ribbon controls allow you to specify an image. For example: *Button*, *ComboBox*, and *Spinner*.
Most of these controls have two properties, one for a large image and one for small. The Ribbon framework will choose one of these sizes according to the available screen space and your definitions for group scaling.

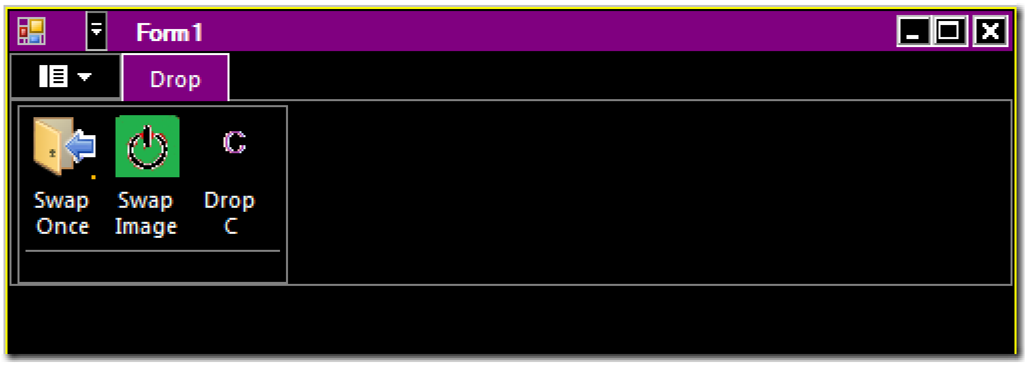
Large image is usually of size 32×32 pixels and Small image is usually of size 16×16 pixels.
I say usually, because this can change. The actual image size should be dependent on your chosen resolution and DPI settings. Microsoft recommended sizes for images are as follows:

DPI	Small Image	Large Image
96 dpi	16×16 pixels	32×32 pixels
120 dpi	20×20 pixels	40×40 pixels
144 dpi	24×24 pixels	48×48 pixels
192 dpi	32×32 pixels	64×64 pixels

The images for a ribbon control are exposed via the *LargeImage* and *SmallImage* properties.

High Contrast Mode
High Contrast is a windows accessibility feature designed for people who have vision impairment. It can be turned on/off by pressing: Left ALT + Left SHIFT + PRINT SCREEN.

The mode's main affect is changing the system colors, so that near colors have high contrast. Now, in order to support high contrast mode in your application, the ribbon framework exposes two extra properties: *LargeHighContrastImage* and *SmallHighContrastImage* which allows you to set images specifically for this mode. Here is an example of how an application usually looks in high contrast mode:



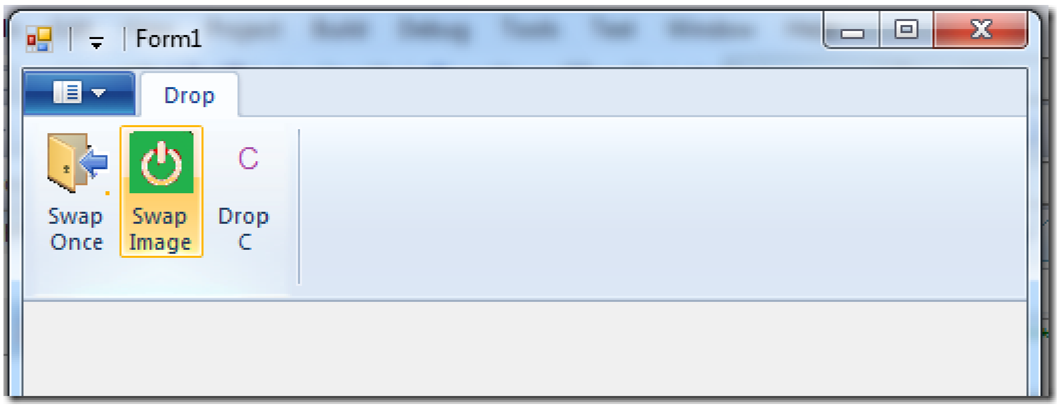
Setting Images Statically
So we've mentioned that we have 4 image properties: *LargeImage*, *SmallImage*, *LargeHighContrastImage and SmallHighContrastImage. And that the size of the images depends on the current system settings.*
So, we need a way to supply the application different images for these scenarios. Here it is:

```
<Command Name="cmdCut" Id="1008" LabelTitle="Cut">
  <Command.LargeImages>
    <Image Source="res/CutLargeImage32.bmp" MinDPI="96" />
    <Image Source="res/CutLargeImage40.bmp" MinDPI="120" />
    <Image Source="res/CutLargeImage48.bmp" MinDPI="144" />
    <Image Source="res/CutLargeImage64.bmp" MinDPI="192" />
  </Command.LargeImages>
  <Command.SmallImages>
    <Image Source="res/CutSmallImage16.bmp" MinDPI="96" />
    <Image Source="res/CutSmallImage20.bmp" MinDPI="120" />
    <Image Source="res/CutSmallImage24.bmp" MinDPI="144" />
    <Image Source="res/CutSmallImage32.bmp" MinDPI="192" />
  </Command.SmallImages>
  <Command.LargeHighContrastImages>
    <Image Source="res/CutLargeImage32HC.bmp" MinDPI="96" />
    <Image Source="res/CutLargeImage40HC.bmp" MinDPI="120" />
    <Image Source="res/CutLargeImage48HC.bmp" MinDPI="144" />
    <Image Source="res/CutLargeImage64HC.bmp" MinDPI="192" />
  </Command.LargeHighContrastImages>
  <Command.SmallHighContrastImages>
    <Image Source="res/CutSmallImage16HC.bmp" MinDPI="96" />
    <Image Source="res/CutSmallImage20HC.bmp" MinDPI="120" />
    <Image Source="res/CutSmallImage24HC.bmp" MinDPI="144" />
    <Image Source="res/CutSmallImage32HC.bmp" MinDPI="192" />
  </Command.SmallHighContrastImages>
</Command>
```

If you don't specify all these images, the ribbon framework will use the available images and resize them according to his needs. Of course, providing the images yourself is the way to get the best results.

Setting Images Dynamically

In this section we'll see how to dynamically set the images for a button. The end result will look like this:



This time the image doesn't help, you need to run it yourself to see the code at work.
The "Swap Once" button demonstrates the simplest way to set the *LargeImage* property programmatically.
The "Swap Image" button demonstrates how to set the image according to the recommended size.

I've added a new function to the *RibbonLib.Ribbon* class, named *ConvertToUIImage. Here is how you use it:*

```
void _buttonDropA_OnExecute(object sender, ExecuteEventArgs e)
{
    // load bitmap from file
    Bitmap bitmap = new System.Drawing.Bitmap(@"..\..\Res\Drop32.bmp");
    bitmap.MakeTransparent();
  // set large image property
    _buttonDropA.LargeImage = _ribbon.ConvertToUIImage(bitmap);
}
```

If you want to set an image which has the correct size according to the current DPI settings, in order to avoid the ribbon framework from resizing your image, you should check the value of *SystemInformation.IconSize.Width.*
Large images size should be (SystemInformation.IconSize.Width x SystemInformation.IconSize.Width) and small images size should be (SystemInformation.IconSize.Width/2) x (SystemInformation.IconSize.Width/2).

Here is an example for setting an image according to windows settings:

```
void _buttonDropB_OnExecute(object sender, ExecuteEventArgs e)
 {
     List<int> supportedImageSizes = new List<int>() { 32, 48, 64 };
   Bitmap bitmap;
     StringBuilder bitmapFileName = new StringBuilder();
   int selectedImageSize;
     if (supportedImageSizes.Contains(SystemInformation.IconSize.Width))
     {
         selectedImageSize = SystemInformation.IconSize.Width;
     }
     else
     {
         selectedImageSize = 32;
     }
   exitOn = !exitOn;
     string exitStatus = exitOn ? "on" : "off";
   bitmapFileName.AppendFormat(@"..\..\Res\Exit{0}{1}.bmp", exitStatus,
selectedImageSize);
   bitmap = new System.Drawing.Bitmap(bitmapFileName.ToString());
     bitmap.MakeTransparent();
   _buttonDropB.LargeImage = _ribbon.ConvertToUIImage(bitmap);
 }
```

Behind the scenes
What *ConvertToUIImage* method actually does is creating an instance of a ribbon framework COM object named *UIRibbonImageFromBitmapFactory*, which implements *IUIImageFromBitmap*. This interface supplies a function for wrapping a given HBITMAP (handle for bitmap) as an *IUIImage* interface.
The ribbon image properties work with these instances of *IUIImage*. Note that the actual creation of *UIRibbonImageFromBitmapFactory* is done in the *RibbonLib.Ribbon InitFramework* method.

```
public IUIImage ConvertToUIImage(Bitmap bitmap)
{
    if (_imageFromBitmap == null)
    {
        return null;
    }
  IUIImage uiImage;
    _imageFromBitmap.CreateImage(bitmap.GetHbitmap(), Ownership.Transfer, out
uiImage);
  return uiImage;
}
```

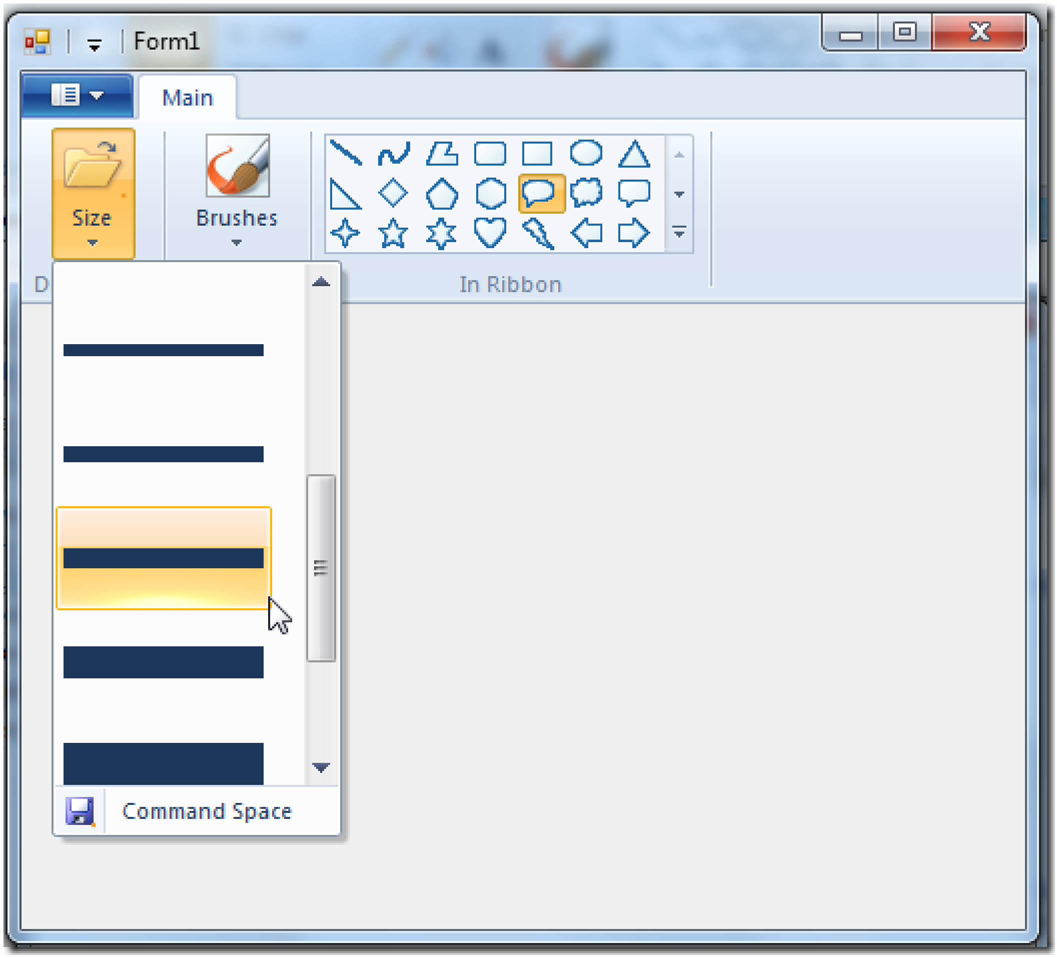
Bonus
Similar to my implementation of helper classes for *Spinner* and *ComboBox* ribbon controls, I've added helper classes for *Tab*, *Group* and *Button* controls. These helpers let you change properties of tabs, groups and buttons easily. The button class also exposes an *ExecuteEvent* event, which facilitate the way you respond to button clicks.

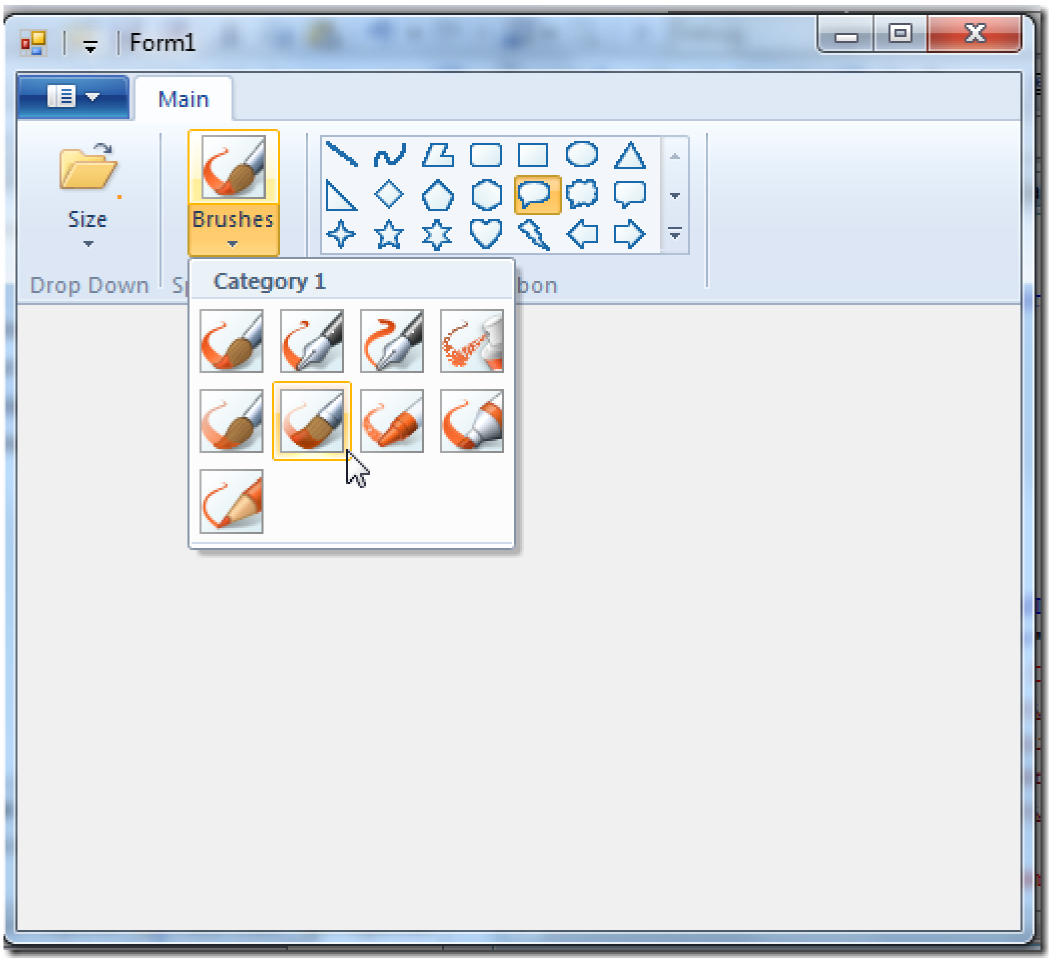
As always, the result of this post is yet another example of using ribbon features in WinForms applications. Find it at Windows Ribbon for WinForms.

Part 11 – DropDownGallery, SplitButtonGallery and InRibbonGallery

In this post I'll show you how to use the different galleries available with the Windows Ribbon Framework.

The result of this post is a new sample named "09-Galleries" that you can find on the Windows Ribbon for WinForms project page. It looks like this:





Item Galleries vs. Command Galleries

The galleries that we will soon review comes in two flavors: item galleries and command galleries. In this section will see what's the difference between these two.

Item Galleries

- Gallery items are "simple" items, just text and image, similar to items in a ComboBox.
- Items have an index and there is a concept of "Selected Item".
- Item galleries support preview. This means you get a "preview" notification (event) when you move over an item and also a "cancel preview" notification when you cancel the selection.
- A single item in an item gallery has the following properties: Label, Image and Category ID.

Note: A ComboBox ribbon control is also an item gallery.

Command Galleries

- Gallery items are actually commands.
- The Commands are not indexed and thus doesn't have the concept of "Selected Item".
- Command galleries doesn't support preview.
- A single item in a command gallery has the following properties: Command ID, Command Type and Category ID.

Note: Both gallery types have the concept of categories, see ComboBox post for more details.

Gallery Types

Now we will review the 3 gallery types we have; each one can be an item gallery or command gallery.

Note that according to MSDN a ComboBox is also considered to be a gallery which is always an item gallery. Since we already review the ComboBox in a previous post, I won't mention it here.

> DropDownGallery
> Just a button that a click on it displays a list of items. The button itself has no action.
> In the first image above, the leftmost control is a *DropDownGallery*.
>
> SplitButtonGallery
> Two buttons, one that is used as a default action and another one that opens a list of items.
> In the second image above, the middle control is a *SplitButtonGallery*.

Note: We already saw in a previous post that the main difference between a *DropDown* and a *SplitButton* is that a *SplitButton* has a default item.

> InRibbonGallery
> The items are displayed inside the ribbon; no button is needed.
> In the images above, the rightmost control is an *InRibbonGallery*.

Command Space

One additional feature you should know about, in the sake of completeness, is that all 3 galleries have a command space. This is a section on the bottom of the control that contains statically defined commands.

In the first image above, the *DropDownGallery* has a button defined in its command space.

Using Galleries

In the next sections I'll show you how to define galleries in ribbon markup and use the new helper classes for easily operate on them. Note that since galleries have many options. I will demonstrate only a representative subset.

Using DropDownGallery – Ribbon Markup

Here I'll show you how to use the *DropDownGallery* as an item gallery with a command space.

Commands section:

```
<Application.Commands>
  <Command Name='cmdTabMain' Id='1000' LabelTitle='Main' />
  <Command Name='cmdGroupDropDownGallery' Id='1001' LabelTitle='Drop Down' />
  <Command Name='cmdDropDownGallery' Id='1002' LabelTitle='Size' >
    <Command.LargeImages>
      <Image>Res/Open32.bmp</Image>
    </Command.LargeImages>
    <Command.SmallImages>
      <Image>Res/Open16.bmp</Image>
    </Command.SmallImages>
  </Command>
  <Command Name='cmdCommandSpace' Id='1003' LabelTitle='Command Space' >
    <Command.LargeImages>
      <Image>Res/Save32.bmp</Image>
    </Command.LargeImages>
    <Command.SmallImages>
```

```
      <Image>Res/Save16.bmp</Image>
    </Command.SmallImages>
  </Command>
  …
</Application.Commands>
```

Views section:

```
<Application.Views>
  <Ribbon>
    <Ribbon.Tabs>
      <Tab CommandName='cmdTabMain'>
        <Group CommandName='cmdGroupDropDownGallery' SizeDefinition='OneButton'>
          <DropDownGallery CommandName='cmdDropDownGallery'
                           TextPosition='Hide'
                           Type='Items'>
            <DropDownGallery.MenuLayout>
              <FlowMenuLayout Columns='1' Rows='5' Gripper='None' />
            </DropDownGallery.MenuLayout>
            <DropDownGallery.MenuGroups>
              <MenuGroup>
                <Button CommandName='cmdCommandSpace' />
              </MenuGroup>
            </DropDownGallery.MenuGroups>
          </DropDownGallery>
        </Group>
        …
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

In the view part you define the use of a *DropDownGallery* along with its layout (how many columns and rows you want) and command space.
What makes this gallery an item gallery is setting the *Type* attribute to '*Items*'.
More details about *DropDownGallery* attributes can be found on MSDN.

Using DropDownGallery – Code Behind

Create an instance of *RibbonLib.Controls.DropDownGallery* helper class and register some of its events:

```
private Ribbon _ribbon = new Ribbon();
private RibbonDropDownGallery _dropDownGallery;
public Form1()
{
    InitializeComponent();
  _dropDownGallery = new RibbonDropDownGallery(_ribbon,
(uint)RibbonMarkupCommands.cmdDropDownGallery);
  _dropDownGallery.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_dropDownGallery_ExecuteEvent);
    _dropDownGallery.PreviewEvent += new
EventHandler<ExecuteEventArgs>(_dropDownGallery_OnPreview);
    _dropDownGallery.CancelPreviewEvent += new
EventHandler<ExecuteEventArgs>(_dropDownGallery_OnCancelPreview);
}
void _dropDownGallery_OnCancelPreview(object sender, ExecuteEventArgs e)
{
    Console.WriteLine("DropDownGallery::OnCancelPreview");
}
void _dropDownGallery_OnPreview(object sender, ExecuteEventArgs e)
{
    Console.WriteLine("DropDownGallery::OnPreview");
}
void _dropDownGallery_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    Console.WriteLine("DropDownGallery::ExecuteEvent");
}
```

Add items to the *DropDownGallery*:

```
private void Form1_Load(object sender, EventArgs e)
{
  FillDropDownGallery();
}
private void FillDropDownGallery()
{
    // set label
    _dropDownGallery.Label = "Size";
  // set _dropDownGallery items
    IUICollection itemsSource = _dropDownGallery.ItemsSource;
    itemsSource.Clear();
    foreach (Image image in imageListLines.Images)
    {
        itemsSource.Add(new GalleryItemPropertySet()
                        {
                            ItemImage = _ribbon.ConvertToUIImage((Bitmap)image)
                        });

    }
}
```

Note that I'm using here a standard *ImageList* control to supply the bitmaps for the *DropDownGallery*.
GalleryItemPropertySet is a helper class that represents a single item in an item gallery.

Using SplitButtonGallery – Ribbon Markup
Here I'll show you how to use the *SplitButtonGallery* as a command gallery.
Commands section:

```
<Command Name='cmdGroupSplitButtonGallery' Id='1004' LabelTitle='Split Button'
/>
<Command Name='cmdSplitButtonGallery' Id='1005' LabelTitle='Brushes' >
  <Command.LargeImages>
    <Image>Res/Brush1.bmp</Image>
  </Command.LargeImages>
</Command>
```

Views section:

```
…
<Group CommandName="cmdGroupSplitButtonGallery" SizeDefinition="OneButton">
  <SplitButtonGallery CommandName="cmdSplitButtonGallery"
                      TextPosition="Hide" Type="Commands" HasLargeItems="true">
    <SplitButtonGallery.MenuLayout>
      <FlowMenuLayout Columns="4" Rows="3" Gripper="None"/>
    </SplitButtonGallery.MenuLayout>
  </SplitButtonGallery>
</Group>
…
```

In the view part you define the use of a *SplitButtonGallery* along with its layout (how many columns and rows you want), and optionally, a command space.
What makes this gallery a command gallery is setting the *Type* attribute to '*Commands*'.
More details about *SplitButtonGallery* attributes can be found on MSDN.

Using SplitButtonGallery – Code Behind
Create an instance of *RibbonLib.Controls.SplitButtonGallery* helper class.

```
private Ribbon _ribbon = new Ribbon();
private RibbonSplitButtonGallery _splitButtonGallery;
private RibbonButton[] _buttons;
public Form1()
{
    InitializeComponent();
  _splitButtonGallery = new RibbonSplitButtonGallery(_ribbon,
(uint)RibbonMarkupCommands.cmdSplitButtonGallery);
}
```

Add items to the *SplitButtonGallery*:

```
private void Form1_Load(object sender, EventArgs e)
{
  FillSplitButtonGallery();
}
private void FillSplitButtonGallery()
{
    // set label
    _splitButtonGallery.Label = "Brushes";
  // prepare helper classes for commands
    _buttons = new RibbonButton[imageListBrushes.Images.Count];
    uint i;
    for (i = 0; i < _buttons.Length; ++i)
    {
        _buttons[i] = new RibbonButton(_ribbon, 2000 + i)
                          {
                              Label = "Label " + i.ToString(),
                              LargeImage = _ribbon.ConvertToUIImage((Bitmap)
imageListBrushes.Images[(int) i])
                          };
    }
  // set _splitButtonGallery categories
    IUICollection categories = _splitButtonGallery.Categories;
    categories.Clear();
    categories.Add(new GalleryItemPropertySet() { Label = "Category 1",
CategoryID = 1 });
  // set _splitButtonGallery items
    IUICollection itemsSource = _splitButtonGallery.ItemsSource;
    itemsSource.Clear();
    i = 0;
    foreach (Image image in imageListBrushes.Images)
    {
        itemsSource.Add(new GalleryCommandPropertySet()
                            {
                                CommandID = 2000 + i++,
                                CommandType = CommandType.Action,
                                CategoryID = 1
                            });
    }
  // add default item to items collection
    itemsSource.Add(new GalleryCommandPropertySet()
                        {
                            CommandID =
(uint)RibbonMarkupCommands.cmdSplitButtonGallery,
                            CommandType = CommandType.Action,
                            CategoryID = 1
                        });
}
```

GalleryCommandPropertySet is a helper class that represents a single item in a command gallery.

Important: If you don't add the default item to the items list of a *SplitButtonGallery*, the items will appear twice! This is probably a bug.

Update (18.11.2009): The updated version of the Ribbon class provides an implementation for IUICommandHandler, so the user doesn't need to implement Execute and UpdateProperty methods anymore.

–

Using InRibbonGallery – Ribbon Markup

Here I'll show you how to use the *InRibbonGallery* as an item gallery.

Commands section:

```
<Command Name='cmdGroupInRibbonGallery' Id='1006' LabelTitle='In Ribbon' />
<Command Name='cmdInRibbonGallery' Id='1007' />
```

Views section:

```
…
<Group CommandName="cmdGroupInRibbonGallery"
SizeDefinition="OneInRibbonGallery">
  <InRibbonGallery CommandName="cmdInRibbonGallery" Type="Items"
                   MaxRows="3" MaxColumns="7">
  </InRibbonGallery>
```

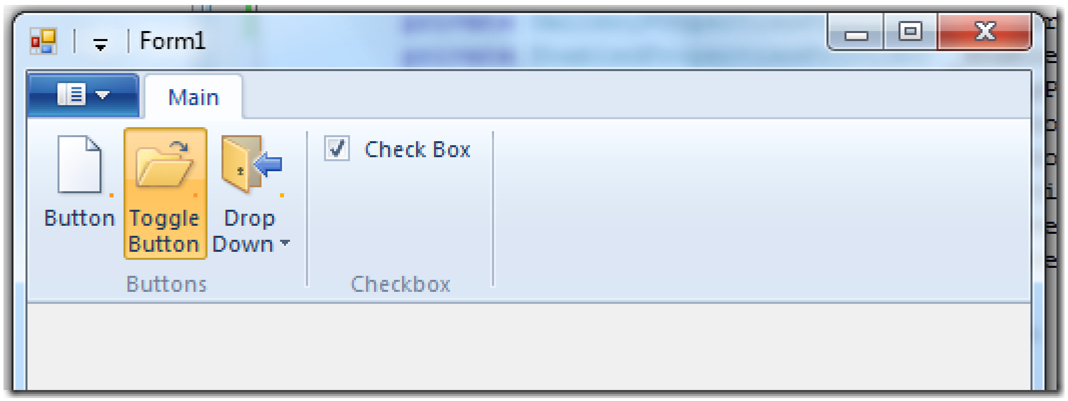
```
</Group>
…
```

In the view part you define the use of an *InRibbonGallery* along with its layout. Note that *InRibbonGallery* has more control on how to layout its items.
More details about *InRibbonGallery* attributes can be found on MSDN.

Using InRibbonGallery – Code Behind
Similar to *DropDownGallery* code. This post is long enough.

Part 12 – CheckBox and ToggleButton

CheckBox and ToggleButton
In short, I've added support for *CheckBox* and *ToggleButton* ribbon controls.
A new sample, named “10-CheckBox” has been added to the project site. The result look like this:



Using CheckBox and ToggleButton – Ribbon Markup

Commands and Views sections:

```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns='http://schemas.microsoft.com/windows/2009/Ribbon'>
  <Application.Commands>
  <Command Name="cmdToggleButton"
              Id="1002"
              LabelTitle="Toggle Button">
      <Command.LargeImages>
        <Image>Res/Open32.bmp</Image>
      </Command.LargeImages>
      <Command.SmallImages>
        <Image>Res/Open16.bmp</Image>
      </Command.SmallImages>
    </Command>
  <Command Name="cmdCheckBox"
              Id="1003"
              LabelTitle="Check Box">
      <Command.LargeImages>
        <Image>Res/Save32.bmp</Image>
      </Command.LargeImages>
      <Command.SmallImages>
        <Image>Res/Save16.bmp</Image>
      </Command.SmallImages>
    </Command>
 </Application.Commands>
 <Application.Views>
    <Ribbon>
      <Ribbon.Tabs>
        <Tab>
          <Group>
            <ToggleButton CommandName="cmdToggleButton" />
          </Group>
          <Group CommandName="cmdGroupCheckBox">
            <CheckBox CommandName="cmdCheckBox" />
          </Group>
        </Tab>
      </Ribbon.Tabs>
    </Ribbon>
  </Application.Views>
</Application>
```

Using CheckBox and ToggleButton – Code Behind

Initializing:

```
private Ribbon _ribbon;
private RibbonToggleButton _toggleButton;
private RibbonCheckBox _checkBox;
public Form1()
{
    InitializeComponent();
  _ribbon = new Ribbon();
    _toggleButton = new RibbonToggleButton(_ribbon,
(uint)RibbonMarkupCommands.cmdToggleButton);
    _checkBox = new RibbonLib.Controls.RibbonCheckBox(_ribbon,
(uint)RibbonMarkupCommands.cmdCheckBox);
  _button.ExecuteEvent += new EventHandler<ExecuteEventArgs>
(_button_ExecuteEvent);
}
```

Update (18.11.2009): The updated version of the Ribbon class provides an implementation for IUICommandHandler, so the user doesn't need to implement Execute and UpdateProperty methods anymore.

Get / Set CheckBox status:

```
void _button_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    MessageBox.Show("checkbox check status is: "
+  _checkBox.BooleanValue.ToString());
}
```

Windows Ribbon for WinForms Library Internal Design Issues

Warning: the rest of this post is extremely boring. It discusses internal details of my ribbon library implementation. This doesn't change anything for the user of the library. Also, it has nothing to do with the checkbox feature.

So now that I'm the only one left, I can discuss some internal details of the library. I've just made a major refactoring of the ribbon library.

The ribbon library is composed of:

- Windows Ribbon Framework API wrappers
- Main ribbon class
- Helper classes for different ribbon controls, such as Button, ComboBox, DropDownGallery etc.

In the old version of the RibbonLib, the controls helper classes had lots of duplicated code. For instance, each control that had images attached to it (LargeImage, SmallImage, … properties) needed to handle those images in the same way (manipulating internal variables, notifying the parent ribbon that an image has invalidated etc.). Now, I don't know about you, but whenever I copy-paste code I always get this strange feeling that something is wrong. So, after several sample projects I couldn't take it anymore and decided to redesign this section.

What I've done was encapsulate common code in classes, so they can be reused in several controls without duplicating code.

So now every ribbon control is composed from several properties' providers (like *ImagePropertiesProvider* and *TooltipPropertiesProvider*) and events providers (like *ExecuteEventsProvider* and *PreviewEventsProvider*).

Note: The following sentence is hard, but there is an example right after, so be strong.

Each provider component has its own interface which the using-control also implements by delegating the execution to the component.

For example, I have an *ImagePropertiesProvider* class that implements the interface *IImagePropertiesProvider*, which exposes 4 image properties (*LargeImage*, *SmallImage*, …).
In my *Button* helper class, I create a member variable of type *ImagePropertiesProvider* and make the Button class also implement *IImagePropertiesProvider* by calling the corresponding methods on the member variable.

This is one of those cases where multiple inheritance was really missing. As a substitute, what I did was used multiple inheritance of interfaces, aggregation and delegation pattern. So, the controls still have some duplicated code (the delegation code) but this code is really simple and has no logic in it.

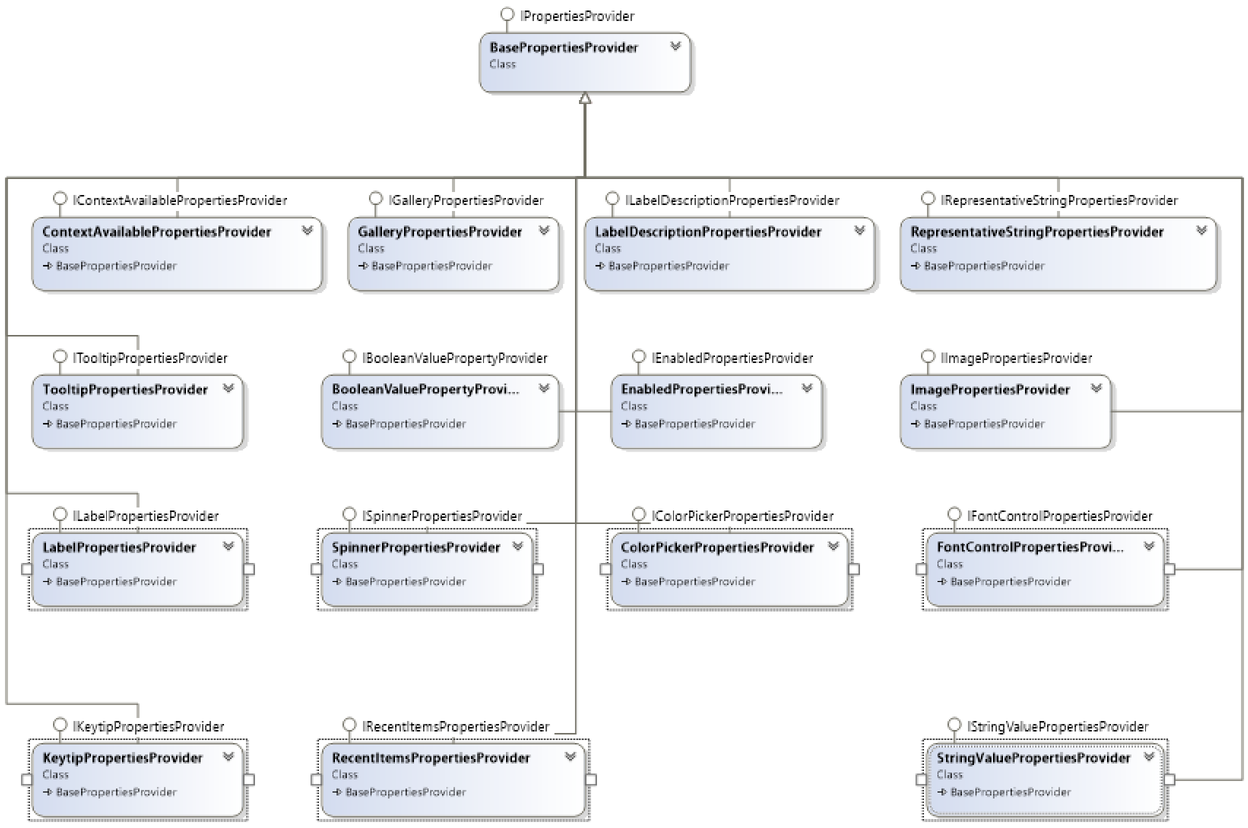
Also, each ribbon control has an *Execute* and *UpdateProperty* methods it needs to implement according to the properties and events it exposes. So, I've made a general implementation which resided in the *BaseRibbonControl* class that search the implementation of a property or event inside one of the registered providers, thus simplifying this code section in every control (now it doesn't exist, the simplest way I know).

The results of all these changes are:

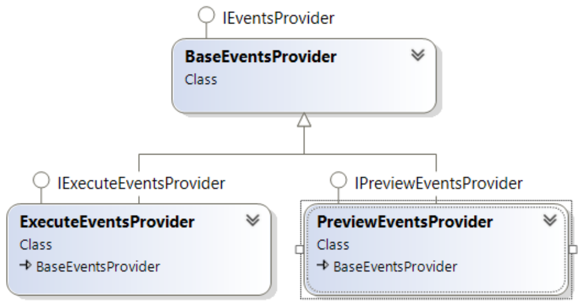
- Shorter code with less duplications.

- Developing of new helper classes is extremely simple.
- Cool class diagrams.

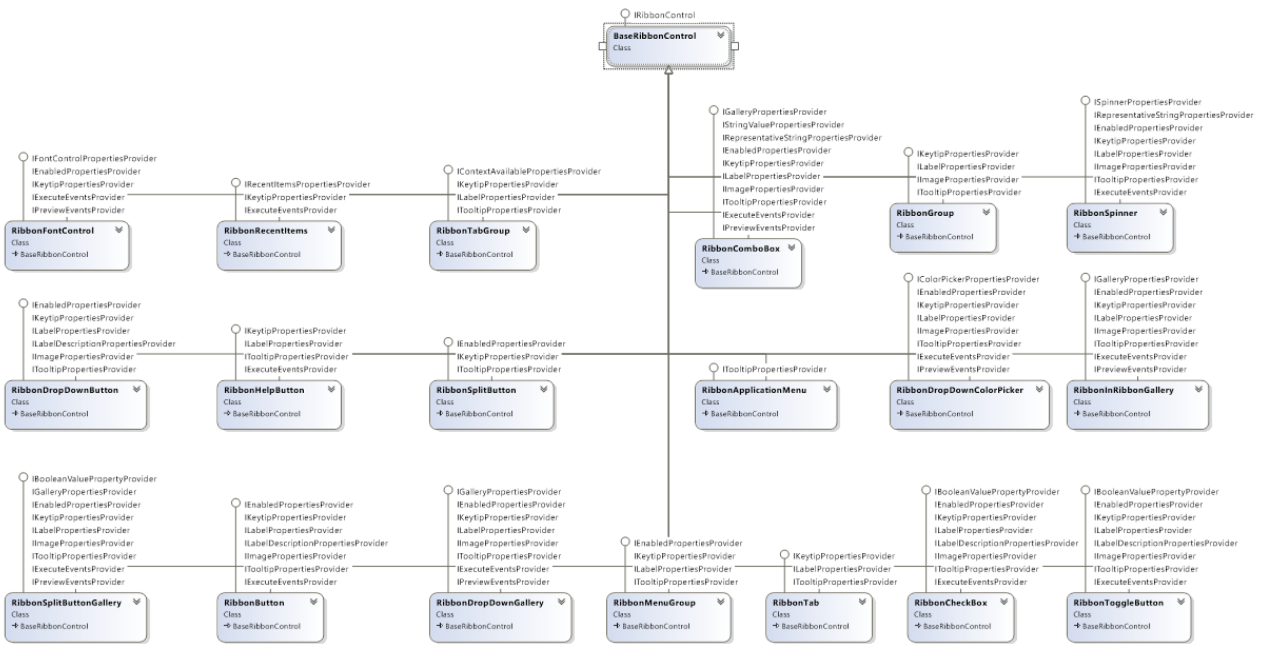
Properties Providers Class Diagram


IPropertiesProvider
BasePropertiesProvider
Class
IContextAvailablePropertiesProvider
ContextAvailablePropertiesProvider
IGalleryPropertiesProvider
GalleryPropertiesProvider
ILabelDescriptionPropertiesProvider
LabelDescriptionPropertiesProvider
IRepresentativeStringPropertiesProvider
RepresentativeStringPropertiesProvider
ITooltipPropertiesProvider
TooltipPropertiesProvider
IBooleanValuePropertyProvider
BooleanValuePropertyProvi...
IEnabledPropertiesProvider
EnabledPropertiesProvi...
IImagePropertiesProvider
ImagePropertiesProvider
ILabelPropertiesProvider
LabelPropertiesProvider
ISpinnerPropertiesProvider
SpinnerPropertiesProvider
IColorPickerPropertiesProvider
ColorPickerPropertiesProvider
IFontControlPropertiesProvider
FontControlPropertiesProvi...
IKeytipPropertiesProvider
KeytipPropertiesProvider
IRecentItemsPropertiesProvider
RecentItemsPropertiesProvider
IStringValuePropertiesProvider
StringValuePropertiesProvider
Class
BasePropertiesProvider

Events Providers Class Diagram

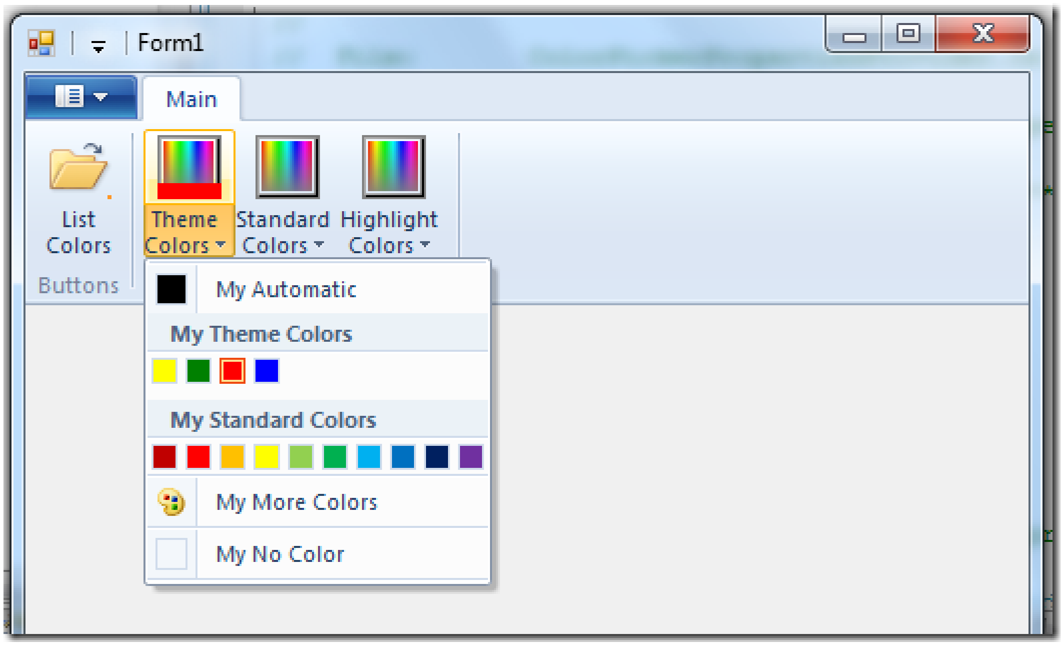


Ribbon Controls Class Diagram

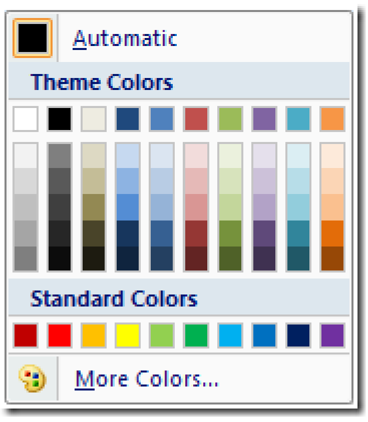

IRibbonControl
BaseRibbonControl
Class
IFontControlPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
IExecuteEventsProvider
IPreviewEventsProvider
RibbonFontControl
Class
BaseRibbonControl
IRecentItemsPropertiesProvider
IKeytipPropertiesProvider
IExecuteEventsProvider
RibbonRecentItems
Class
BaseRibbonControl
IContextAvailablePropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
ITooltipPropertiesProvider
RibbonTabGroup
Class
BaseRibbonControl
IGalleryPropertiesProvider
IStringValuePropertiesProvider
IRepresentativeStringPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
IPreviewEventsProvider
RibbonComboBox
Class
BaseRibbonControl
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
RibbonGroup
Class
BaseRibbonControl
ISpinnerPropertiesProvider
IRepresentativeStringPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
RibbonSpinner
Class
BaseRibbonControl
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
ILabelDescriptionPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
RibbonDropDownButton
Class
BaseRibbonControl
IKeytipPropertiesProvider
ILabelPropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
RibbonHelpButton
Class
BaseRibbonControl
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ITooltipPropertiesProvider
RibbonSplitButton
Class
BaseRibbonControl
ITooltipPropertiesProvider
RibbonApplicationMenu
Class
BaseRibbonControl
IColorPickerPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
IPreviewEventsProvider
RibbonDropDownColorPicker
Class
BaseRibbonControl
IGalleryPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
IPreviewEventsProvider
RibbonInRibbonGallery
Class
BaseRibbonControl
IBooleanValuePropertyProvider
IGalleryPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
IPreviewEventsProvider
RibbonSplitButtonGallery
Class
BaseRibbonControl
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
ILabelDescriptionPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
RibbonButton
Class
BaseRibbonControl
IGalleryPropertiesProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
IPreviewEventsProvider
RibbonDropDownGallery
Class
BaseRibbonControl
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
ITooltipPropertiesProvider
RibbonMenuGroup
Class
BaseRibbonControl
IKeytipPropertiesProvider
ILabelPropertiesProvider
ITooltipPropertiesProvider
RibbonTab
Class
BaseRibbonControl
IBooleanValuePropertyProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
ILabelDescriptionPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
RibbonCheckBox
Class
BaseRibbonControl
IBooleanValuePropertyProvider
IEnabledPropertiesProvider
IKeytipPropertiesProvider
ILabelPropertiesProvider
ILabelDescriptionPropertiesProvider
IImagePropertiesProvider
ITooltipPropertiesProvider
IExecuteEventsProvider
RibbonToggleButton
Class
BaseRibbonControl

Part 13 – DropDownColorPicker

Windows Ribbon for WinForms library now supports DropDownColorPicker control.
The result of this post is a yet another sample, "11-DropDownColorPicker", found on the project site.



DropDownColorPicker Control
The drop-down color picker looks like this:



DropDownColorPicker Properties (internal details)
Following is the list of properties which are unique for DropDownColorPicker control. The rest of the properties have been reviewed in previous posts.

- Color – The selected color.
  Property Identifier: UI_PKEY_Color
- ColorType – The type of selected color. Can be: NoColor, Automatic or RGB (meaning specific color).
  Property Identifier: UI_PKEY_ColorType
- AutomaticColorLabel – Defines the label for the “Automatic” color button.
  Property Identifier: UI_PKEY_AutomaticColorLabel
- MoreColorsLabel – Defines the label for the “More colors…” button.
  Property Identifier: UI_PKEY_MoreColorsLabel
- NoColorLabel – Defines the label for the “No color” button.
  Property Identifier: UI_PKEY_NoColorLabel
- RecentColorsCategoryLabel – Defines the label for the “Recent colors” category.
  Property Identifier: UI_PKEY_RecentColorsCategoryLabel
- StandardColorsCategoryLabel – Defines the label for the “Standard colors” category.
  Property Identifier: UI_PKEY_StandardColorsCategoryLabel
- ThemeColorsCategoryLabel – Defines the label for the “Theme colors” category.
  Property Identifier: UI_PKEY_ThemeColorsCategoryLabel

Note: The different labels in the drop down color picker allows you to localize the control.

For more details on DropDownColorPicker control, check out Drop-Down Color Picker on MSDN.

Using DropDownColorPicker – Ribbon Markup

Commands and Views sections:

```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns='http://schemas.microsoft.com/windows/2009/Ribbon'>
  <Application.Commands>
    <Command Name="cmdDropDownColorPickerThemeColors"
             Id="1002"
             LabelTitle="Theme Colors">
      <Command.LargeImages>
        <Image>Res/Colors32.bmp</Image>
      </Command.LargeImages>
    </Command>
  </Application.Commands>
 <Application.Views>
    <Ribbon>
      <Ribbon.Tabs>
        <Tab>
          <Group>
            <DropDownColorPicker CommandName="cmdDropDownColorPickerThemeColors"
                                 ColorTemplate="ThemeColors"/>
          </Group>
        </Tab>
      </Ribbon.Tabs>
    </Ribbon>
  </Application.Views>
</Application>
```

Using DropDownColorPicker – Code Behind

Initializing:

```
private Ribbon _ribbon;
private RibbonDropDownColorPicker _themeColors;
public Form1()
{
    InitializeComponent();
  _ribbon = new Ribbon();
    _themeColors = new RibbonDropDownColorPicker(_ribbon,
(uint)RibbonMarkupCommands.cmdDropDownColorPickerThemeColors);
}
private void Form1_Load(object sender, EventArgs e)
{
    InitDropDownColorPickers();
}
private void InitDropDownColorPickers()
{
    // common properties
    _themeColors.Label = "Theme Colors";
    _themeColors.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_themeColors_ExecuteEvent);
  // set labels
    _themeColors.AutomaticColorLabel = "My Automatic";
    _themeColors.MoreColorsLabel = "My More Colors";
    _themeColors.NoColorLabel = "My No Color";
    _themeColors.RecentColorsCategoryLabel = "My Recent Colors";
    _themeColors.StandardColorsCategoryLabel = "My Standard Colors";
    _themeColors.ThemeColorsCategoryLabel = "My Theme Colors";
  // set colors
    _themeColors.ThemeColorsTooltips = new string[] { "yellow", "green", "red",
"blue" };
    _themeColors.ThemeColors = new Color[] { Color.Yellow, Color.Green,
Color.Red, Color.Blue };
}
```

Respond to selected color event:

```
void _themeColors_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    MessageBox.Show("Selected color is " + _themeColors.Color.ToString());
}
```

Windows Ribbon for WinForms Library Update
Warning: Internal details ahead. Not related to DropDownColorPicker in any way.

Microsoft recently released PreviewRibbon. It is a sample application that helps you design a ribbon enabled application by giving you a preview of how your markup will look. The application is built as a command line tool that accepts a ribbon markup xml file and displays the result. It is written in C# using WinForms and thus can be used as a sample of how to use the ribbon in a .NET application, which is similar to what I've been trying to do.

This is the place to mention RibbonExplorer by Alon Fliess, which is another great tool for previewing a ribbon and manipulating its properties, written in C++/ATL.

Since both Windows Ribbon for WinForms and PreviewRibbon projects use the ribbon thru COM Interop and written in C#, they solve similar problems. So, I've decided to review the PreviewRibbon code to learn how they solved some of the issues I've faced.

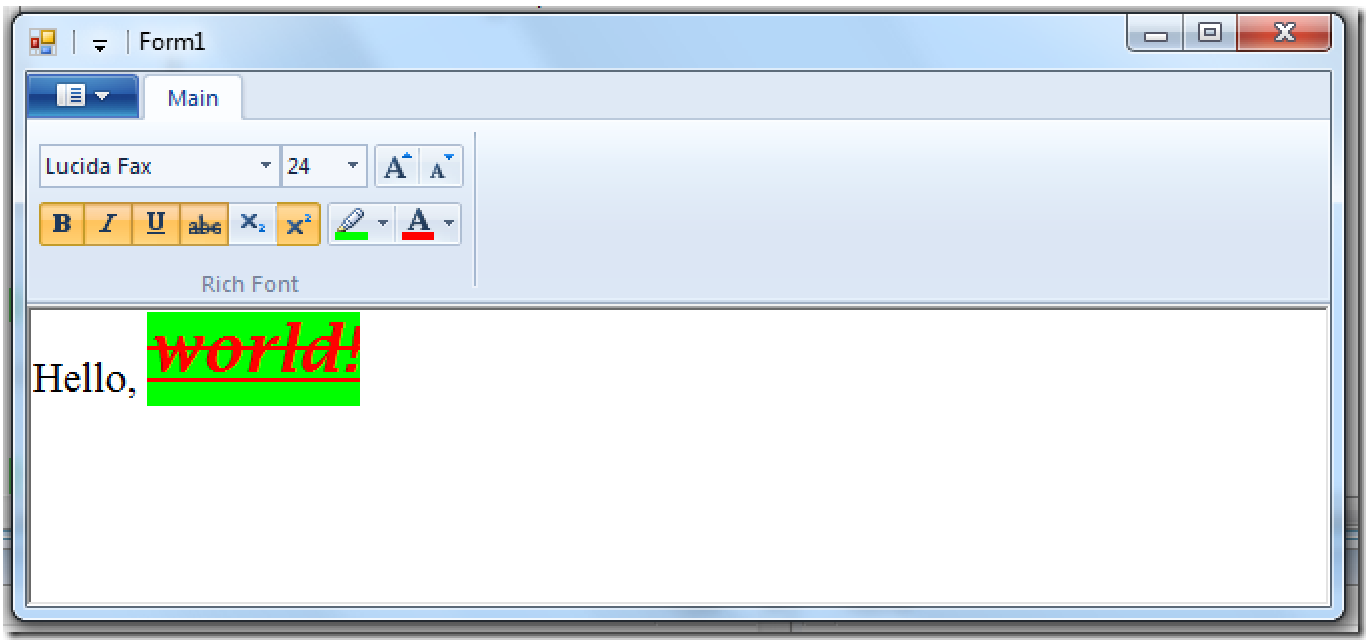
Update 25.02.2020: I cannot find PreviewRibbon or RibbonExplorer anymore on the web.

The end result of the review is as follow:

- I've created my own version of *PropertyKey* class, which is basically a copy of the one PreviewRibbon use, only I've separated the *PropertyKey* definition from ribbon related code.
- I've created my own version of *PropVariant* class.
  Decimal value handling is taken from PreviewRibbon (well done!).
  Vectors handling is taken from Windows API Code Pack.
- As a small bonus, since now I have my own versions of *PropertyKey* and *PropVariant*, the project doesn't depend anymore on Windows API Code Pack.
- Changed the naming convention of the ribbon enums (removed UI_ prefix) and ribbon property keys (removed UI_PKEY_ prefix).

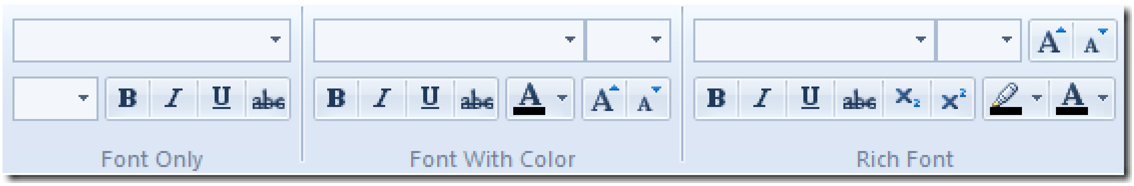
Part 14 – FontControl

Windows Ribbon for WinForms library now supports FontControl control.
The result of this post is a yet another sample, “12-FontControl”, found on the project site.



FontControl Control
FontControl is another special control provided by the Windows Ribbon Framework.
It allows you to choose font family, size, colors and related effects.
It has three types, each exposing a little more functionality then the other:

- Font Only
- Font with Color
- Rich Font



Check Font Control on MSDN for full details on the differences between the types.

FontControl Properties (internal details)
Following is the list of properties which are unique for FontControl control. The rest of the properties have been reviewed in previous posts.

- FontProperties – This property is of type *IPropertyStore* and holds all the font specific properties, like Size, Bold, Underline, etc.
  In the FontControl helper class I use this property internally to access the other properties but do not expose it to the user, since it has no use other than being an access point to the other properties.
  Property Identifier: UI_PKEY_FontProperties

- ChangedProperties – This property contains all the recently changed properties.
  The FontControl doesn't expose it but provides it as one of the parameters in the Execute / Preview / CancelPreview events. For example, if you click on the "Bold" button, the Execute event will be called and the "this" property will contain only the Bold property.
  Property Identifier: UI_PKEY_FontProperties_ChangedProperties
- Family – The selected font family name.
  Property Identifier: UI_PKEY_FontProperties_Family
- Size – The size of the font.
  Property Identifier: UI_PKEY_FontProperties_Size
- Bold – Flag that indicates whether bold is selected.
  Property Identifier: UI_PKEY_FontProperties_Bold
- Italic – Flag that indicates whether italic is selected.
  Property Identifier: UI_PKEY_FontProperties_Italic
- Underline – Flag that indicates whether underline is selected.
  Property Identifier: UI_PKEY_FontProperties_Underline
- Strikethrough – Flag that indicates whether strikethrough is selected (sometimes called Strikeout).
  Property Identifier: UI_PKEY_FontProperties_Strikethrough
- VerticalPositioning – Flag that indicates which one of the Subscript and Superscript buttons are selected, if any.
  Property Identifier: UI_PKEY_FontProperties_VerticalPositioning
- ForegroundColor – Contains the text color if ForegroundColorType is set to RGB. The FontControl helper class expose this property as a .NET Color and handles internally the conversion to and from COLORREF structure.
  Property Identifier: UI_PKEY_FontProperties_ForegroundColor
- ForegroundColorType – The text color type. Valid values are *RGB* and *Automatic*.
  If *RGB* is selected, the user should get the color from the ForegroundColor property.
  If *Automatic* is selected the user should use *SystemColors.WindowText*.
  The *FontControl* helper class doesn't expose the ForegroundColorType property. Instead it implements the color selection algorithm internally (i.e. return correct color according to the type property).
  Property Identifier: UI_PKEY_FontProperties_ForegroundColorType
- BackgroundColor – Contains the background color if BackgroundColorType is set to RGB. The FontControl helper class expose this property as a .NET Color and handles internally the conversion to and from COLORREF structure.
  Property Identifier: UI_PKEY_FontProperties_BackgroundColor
- BackgroundColorType – The background color type. Valid values are *RGB* and *NoColor*.
  If *RGB* is selected, the user should get the color from the BackgroundColor property.
  If *NoColor* is selected the user should use *SystemColors.Window*.
  The *FontControl* helper class doesn't expose the ForegroundColorType property. Instead it implements the color selection algorithm internally (i.e. return correct color according to the type property).
  Property Identifier: UI_PKEY_FontProperties_BackgroundColorType
- DeltaSize – Indicated whether the "Grow Font" or "Shrink Font" buttons were pressed. This property is only available as part of the *ChangedProperties* property and is not exposed by

the FontControl helper class.
Property Identifier: UI_PKEY_FontProperties_DeltaSize

Using FontControl – Ribbon Markup

Commands and Views sections:

```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns='http://schemas.microsoft.com/windows/2009/Ribbon'>
    <Application.Commands>
    <Command Name="cmdTabMain" Id="1001" LabelTitle="Main" />
    <Command Name="cmdGroupRichFont" Id="1002" LabelTitle="Rich Font" />
    <Command Name="cmdRichFont" Id="1003" Keytip="F" />
  </Application.Commands>

    <Application.Views>
        <Ribbon>
      <Ribbon.Tabs>
        <Tab CommandName="cmdTabMain">
          <Group CommandName="cmdGroupRichFont" SizeDefinition="OneFontControl">
            <FontControl CommandName="cmdRichFont" FontType="RichFont" />
          </Group>
        </Tab>
      </Ribbon.Tabs>
    </Ribbon>
    </Application.Views>
</Application>
```

More details on FontControl attributes can be found on MSDN.

Using FontControl – Code Behind

The following code shows the basic steps of using a ribbon FontControl which stays in sync with the selected text in a standard .NET RichTextBox control.

- Initializing:

```
private Ribbon _ribbon;
private RibbonFontControl _richFont;

public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();
    _richFont = new RibbonFontControl(_ribbon,
(uint)RibbonMarkupCommands.cmdRichFont);

    _richFont.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_richFont_ExecuteEvent);
```

```
    _richFont.PreviewEvent += new
EventHandler<ExecuteEventArgs>(_richFont_OnPreview);
    _richFont.CancelPreviewEvent += new
EventHandler<ExecuteEventArgs>(_richFont_OnCancelPreview);
}
```

- Setting RichTextBox properties when FontControl has changed:

```
void _richFont_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    // skip if selected font is not valid
    if ((_richFont.Family == null) ||
         (_richFont.Family.Trim() == string.Empty) ||
         (_richFont.Size == 0))
    {
        return;
    }

    // prepare font style
    FontStyle fontStyle = FontStyle.Regular;
    if (_richFont.Bold == FontProperties.Set)
    {
        fontStyle |= FontStyle.Bold;
    }
    if (_richFont.Italic == FontProperties.Set)
    {
        fontStyle |= FontStyle.Italic;
    }
    if (_richFont.Underline == FontUnderline.Set)
    {
        fontStyle |= FontStyle.Underline;
    }
    if (_richFont.Strikethrough == FontProperties.Set)
    {
        fontStyle |= FontStyle.Strikeout;
    }

    // set selected font
    richTextBox1.SelectionFont =
        new Font(_richFont.Family, (float)_richFont.Size, fontStyle);

    // set selected colors
    richTextBox1.SelectionColor = _richFont.ForegroundColor;
    richTextBox1.SelectionBackColor = _richFont.BackgroundColor;

    // set subscript / superscript
    switch (_richFont.VerticalPositioning)
    {
        case FontVerticalPosition.NotSet:
            richTextBox1.SelectionCharOffset = 0;
            break;

        case FontVerticalPosition.SuperScript:
            richTextBox1.SelectionCharOffset = 10;
            break;

        case FontVerticalPosition.SubScript:
            richTextBox1.SelectionCharOffset = -10;
            break;
    }
}
```

Note: RichTextBox doesn't support Subscript and Superscript natively. What it does support is setting the character offset, so this is what I use to simulate the required behavior.

- Adding support for preview while changing font family and size:

```
void _richFont_OnPreview(object sender, ExecuteEventArgs e)
{
    PropVariant propChangesProperties;
    e.CommandExecutionProperties.GetValue(ref
RibbonProperties.FontProperties_ChangedProperties, out propChangesProperties);
    IPropertyStore changedProperties =
(IPropertyStore)propChangesProperties.Value;

    UpdateRichTextBox(changedProperties);
}

void _richFont_OnCancelPreview(object sender, ExecuteEventArgs e)
{
    IPropertyStore fontProperties =
(IpropertyStore)e.CurrentValue.PropVariant.Value;

    UpdateRichTextBox(fontProperties);
}

private void UpdateRichTextBox(IPropertyStore propertyStore)
{
    RibbonLib.FontPropertyStore fontPropertyStore = new
RibbonLib.FontPropertyStore(propertyStore);
    PropVariant propValue;

    FontStyle fontStyle = richTextBox1.SelectionFont.Style;
    string family = richTextBox1.SelectionFont.FontFamily.Name;
    float size = richTextBox1.SelectionFont.Size;

    if (propertyStore.GetValue(ref RibbonProperties.FontProperties_Family, out
propValue) == HRESULT.S_OK)
    {
        family = fontPropertyStore.Family;
    }
    if (propertyStore.GetValue(ref RibbonProperties.FontProperties_Size, out
propValue) == HRESULT.S_OK)
    {
        size = (float)fontPropertyStore.Size;
    }

    richTextBox1.SelectionFont = new Font(family, size, fontStyle);
}
```

Note: Only font family and font size should support preview since only they have attached combo boxes.

- Updating FontControl when text selection changes in RichTextBox:

```
private void richTextBox1_SelectionChanged(object sender, EventArgs e)
{
    // update font control font
    if (richTextBox1.SelectionFont != null)
    {
        _richFont.Family = richTextBox1.SelectionFont.FontFamily.Name;
        _richFont.Size = (decimal)richTextBox1.SelectionFont.Size;
        _richFont.Bold = richTextBox1.SelectionFont.Bold ?
                         FontProperties.Set : FontProperties.NotSet;
        _richFont.Italic = richTextBox1.SelectionFont.Italic ?
                         FontProperties.Set : FontProperties.NotSet;
        _richFont.Underline = richTextBox1.SelectionFont.Underline ?
                         FontUnderline.Set : FontUnderline.NotSet;
        _richFont.Strikethrough = richTextBox1.SelectionFont.Strikeout ?
                         FontProperties.Set : FontProperties.NotSet;
    }
    else
    {
        _richFont.Family = string.Empty;
        _richFont.Size = 0;
        _richFont.Bold = FontProperties.NotAvailable;
        _richFont.Italic = FontProperties.NotAvailable;
        _richFont.Underline = FontUnderline.NotAvailable;
        _richFont.Strikethrough = FontProperties.NotAvailable;
    }

    // update font control colors
    _richFont.ForegroundColor = richTextBox1.SelectionColor;
    _richFont.BackgroundColor = richTextBox1.SelectionBackColor;

    // update font control vertical positioning
    switch (richTextBox1.SelectionCharOffset)
    {
        case 0:
            _richFont.VerticalPositioning = FontVerticalPosition.NotSet;
            break;

        case 10:
            _richFont.VerticalPositioning = FontVerticalPosition.SuperScript;
            break;
        case -10:
            _richFont.VerticalPositioning = FontVerticalPosition.SubScript;
            break;
    }
}
```

Part 15 – Use Ribbon as External DLL

Using Windows Ribbon for WinForms just got a lot easier.

Warning: Boring post. Talks about changes in the ribbon library and their reasons.

But first, let me start by asking for your forgiveness.
I'm trying to create a library which will be easy to use and so the last change to the library wasn't backward compatible. Namely, classes names and interfaces have changed.
Rest assured that every single change I made is making the library a little easier to use. However, for all those who have started using the library, I say: sorry. [This is what you get for using a project in BETA]

So, what have changed?

Classes names
I've added a "Ribbon" prefix to all the ribbon controls helper classes. This is to prevent collision with the standard WinForms controls, like Button, ComboBox, etc.

Yea, I know, this is why we have namespaces. However, having a WinForms project with both a button and a ribbon button is not a rare case. When this happen, the user can't add "using RibbonLib.Controls;" so almost every other line is cluttered with "RibbonLib.Controls" prefix, for example:

```
RibbonLib.Controls.Button ribbonButton = new RibbonLib.Controls.Button(_ribbon,
commandId);
```

Instead of:

```
RibbonButton ribbonButton = new RibbonButton(_ribbon, commandId);
```

IUICommandHandler Implementation
I've added a general implementation of *IUICommandHandler* (*Execute* and *UpdateProperty* functions) to the *Ribbon* class, so that these methods should not be implemented anymore by the user. This means the user doesn't need to write anymore the following code in its *Form* class:

```
public HRESULT Execute(uint commandId, ExecutionVerb verb, PropertyKeyRef key,
PropVariantRef currentValue, IUISimplePropertySet commandExecutionProperties)
{
    switch (commandId)
    {
        case (uint)RibbonMarkupCommands.cmdDropDownColorPickerGroup:
            _groupColors.Execute(verb, key, currentValue,
commandExecutionProperties);
            break;
    case (uint)RibbonMarkupCommands.cmdButtonsGroup:
            _groupButtons.Execute(verb, key, currentValue,
commandExecutionProperties);
            break;
    case (uint)RibbonMarkupCommands.cmdButtonListColors:
            _buttonListColors.Execute(verb, key, currentValue,
commandExecutionProperties);
            break;
    case (uint)RibbonMarkupCommands.cmdDropDownColorPickerThemeColors:
            _themeColors.Execute(verb, key, currentValue,
commandExecutionProperties);
            break;
    case (uint)RibbonMarkupCommands.cmdDropDownColorPickerStandardColors:
            _standardColors.Execute(verb, key, currentValue,
commandExecutionProperties);
            break;
    case (uint)RibbonMarkupCommands.cmdDropDownColorPickerHighlightColors:
            _highlightColors.Execute(verb, key, currentValue,
commandExecutionProperties);
            break;
    }
  return HRESULT.S_OK;
}
public HRESULT UpdateProperty(uint commandId, ref PropertyKey key,
PropVariantRef currentValue, ref PropVariant newValue)
{
    switch (commandId)
    {
        case (uint)RibbonMarkupCommands.cmdDropDownColorPickerGroup:
            _groupColors.UpdateProperty(ref key, currentValue, ref newValue);
            break;
    case (uint)RibbonMarkupCommands.cmdButtonsGroup:
            _groupButtons.UpdateProperty(ref key, currentValue, ref newValue);
            break;
    case (uint)RibbonMarkupCommands.cmdButtonListColors:
            _buttonListColors.UpdateProperty(ref key, currentValue, ref
newValue);
            break;
    case (uint)RibbonMarkupCommands.cmdDropDownColorPickerThemeColors:
            _themeColors.UpdateProperty(ref key, currentValue, ref newValue);
            break;
    case (uint)RibbonMarkupCommands.cmdDropDownColorPickerStandardColors:
            _standardColors.UpdateProperty(ref key, currentValue, ref newValue);
            break;
    case (uint)RibbonMarkupCommands.cmdDropDownColorPickerHighlightColors:
            _highlightColors.UpdateProperty(ref key, currentValue, ref
newValue);
            break;
    }
  return HRESULT.S_OK;
}
```

The new implementation, which resides in the *Ribbon* class just delegates the call to the correct ribbon control, according to the command ID:

```
public virtual HRESULT Execute(uint commandID, ExecutionVerb verb,
PropertyKeyRef key,
                               PropVariantRef currentValue,
                               IUISimplePropertySet commandExecutionProperties)
{
    if (_mapRibbonControls.ContainsKey(commandID))
    {
        _mapRibbonControls[commandID].Execute(verb, key, currentValue,
                                              commandExecutionProperties);
    }
  return HRESULT.S_OK;
}
public virtual HRESULT UpdateProperty(uint commandID, ref PropertyKey key,
                                      PropVariantRef currentValue,
                                      ref PropVariant newValue)
{
    if (_mapRibbonControls.ContainsKey(commandID))
    {
        _mapRibbonControls[commandID].UpdateProperty(ref key, currentValue,
                                                     ref newValue);
    }
  return HRESULT.S_OK;
}
```

The *_mapRibbonControls* is an internal dictionary that contains all the ribbon controls helper classes the user have created in the main form.

Note: I’ve made these functions *virtual* so that if some user wants direct access to the notifications from the ribbon framework, he can still get them by deriving from the *Ribbon* class and overriding these methods.

Support for Ribbon External DLL

Now you can have your ribbon resource reside in an external dll instead of inside the application executable, as a native resource. This issue was a problem for developers who needed the native resource for other uses (like setting the application icon).

In fact, I’ve made this the default behavior of the ribbon. So now when you call the simplest form of *Ribbon.InitFramework*, the ribbon library tries to load the ribbon from *your_app_name.ribbon.dll* and only if it fails to find this file it will revert back to the previous behavior, that is, try to load ribbon from your executable native resource.

Of course, you can provide your own dll name to load, or even load it yourself and pass the dll handle (=what returns from LoadLibrary) to the different overloads of *Ribbon.InitFramework*. This allows you to implement a custom ribbon loading mechanism which can be useful if you wish to load different ribbons on different scenarios, e.g. add localization support (different locale has different ribbon).

What this means for the user of the ribbon library is that he needs to add one more step to the pre-build event. This step will create the ribbon resource dll from the resource file created by previous steps.
So, the pre-build events for projects that uses the ribbon library are now:

```
"%PROGRAMFILES%\Microsoft SDKs\Windows\v7.0\Bin\UICC.exe"
"$(ProjectDir)RibbonMarkup.xml" "$(ProjectDir)RibbonMarkup.bml"
/res:"$(ProjectDir)RibbonMarkup.rc"
"%PROGRAMFILES%\Microsoft SDKs\Windows\v7.0\Bin\rc.exe" /v
"$(ProjectDir)RibbonMarkup.rc"
cmd /c "("$(DevEnvDir)..\..\VC\bin\vcvars32.bat") &&
("$(DevEnvDir)..\..\VC\bin\link.exe" /VERBOSE /NOENTRY /DLL
/OUT:"$(ProjectDir)$(OutDir)$(TargetName).ribbon.dll"
"$(ProjectDir)RibbonMarkup.res")"
```

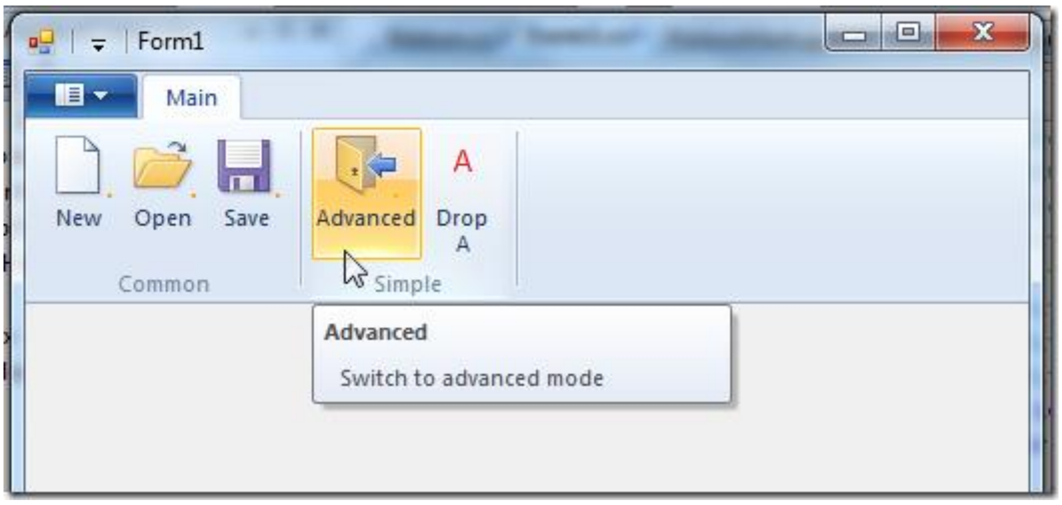
I know, this looks like a pile of junk, but actually what's written is:

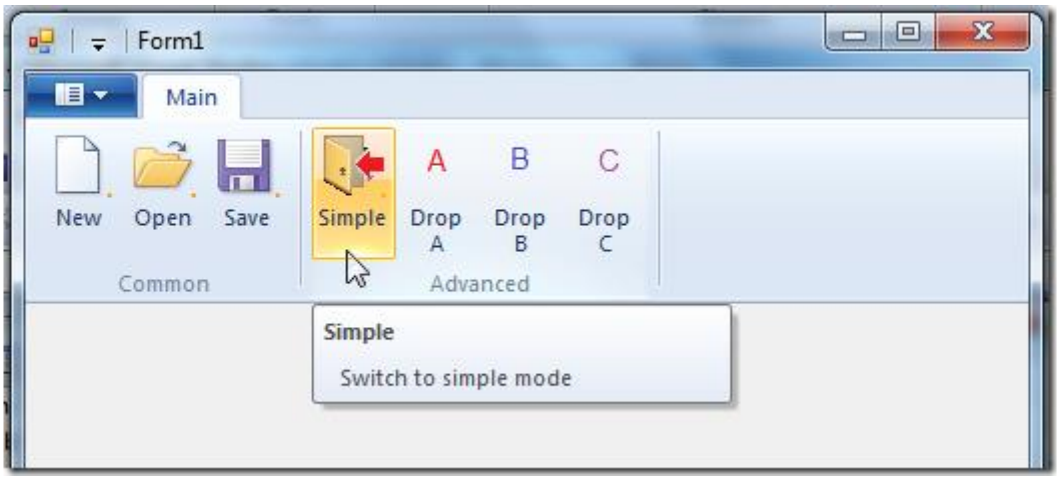
- **UICC** – please convert RibbonMarkup.xml to RibbonMarkup.rc
- **rc** – please convert RibbonMarkup.rc to RibbonMarkup.res
- **link** – please convert RibbonMarkup.res to YourAppName.ribbon.dll

I've updated all the ribbon library samples to use the ribbon resource dll.

Part 16 – ApplicationModes

Windows Ribbon for WinForms library now supports Application Modes.
The result of this post is a yet another sample, "13-ApplicationModes", found on the project site.





What are application modes?
It is best to explain using examples. Applications sometimes have different “modes” in which they show different GUI, for example:

- Simple mode VS Advanced mode
- Regular editor mode VS Print mode

The ribbon framework support changing its GUI according to the current application mode. In order to use the ribbon application modes, you need to:

- Set the available application modes for each ribbon item. This is done in design time.
- Set the current application mode. This is done in run time.

To summarize, application modes is a feature that allows the ribbon to change its GUI according to the current application context.

More details on this subject can be found at Reconfiguring the Ribbon with Application Modes on MSDN.

Using ApplicationModes extra remarks

- You can set up to 32 different application modes, each identified by a number between 0 and 31.
- Modes can coexist, meaning you can set both simple mode and advanced mode as active at the same time. Internally the current application modes are represented by a single 32bit variable (which represents a Boolean array of size 32), thus explaining why you can only have 32 modes.

- Mode 0 is the default mode. So, if you don't set the ApplicationModes attribute, 0 is the default.
- At least one mode should be set at all times. You can't disable all the modes (the framework will just ignore your last set).

Using ApplicationModes – Ribbon Markup

Following is an example of the views section where you set the ApplicationModes attribute:

```
<Application.Views>
  <Ribbon>
    <Ribbon.Tabs>
      <Tab CommandName="cmdTabMain" ApplicationModes="0,1">
        <Group CommandName="cmdGroupCommon"
               SizeDefinition="ThreeButtons"
               ApplicationModes="0,1">
          <Button CommandName="cmdButtonNew" />
          <Button CommandName="cmdButtonOpen" />
          <Button CommandName="cmdButtonSave" />
        </Group>
        <Group CommandName="cmdGroupSimple"
               SizeDefinition="TwoButtons"
               ApplicationModes="0">
          <Button CommandName="cmdButtonSwitchToAdvanced" />
          <Button CommandName="cmdButtonDropA" />
        </Group>
        <Group CommandName="cmdGroupAdvanced"
               SizeDefinition="FourButtons"
               ApplicationModes="1">
          <Button CommandName="cmdButtonSwitchToSimple" />
          <Button CommandName="cmdButtonDropA" />
          <Button CommandName="cmdButtonDropB" />
          <Button CommandName="cmdButtonDropC" />
        </Group>
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

In this example we create a tab with 3 groups in it: Common, Simple and Advanced.
The common group should always appear so we set its *ApplicationModes* attribute to "0,1"
The simple group should only appear in simple mode (0). Similarly, the advanced group should only appear in advanced mode (1).
Note that the tab element should appear in both modes, so you must also set its *ApplicationModes* attribute to "0,1".

Note that you have to set an ApplicationMode to each possible Element if you use *ApplicationModes* for only a few elements. Otherwise you can get a Shutdown of your application when you switch the ApplicationMode.

ApplicationModes can be set on the following elements:

- Core tabs (as opposed to contextual tabs).
- Groups which are children of core tabs.
- Button, SplitButton, DropDownButton, SplitButtonGallery and DropDownGallery but only when those controls are in the application menu.

ApplicationModes – code behind

Following are two ribbon buttons, “simple” and “advanced”, each changes the current application mode:

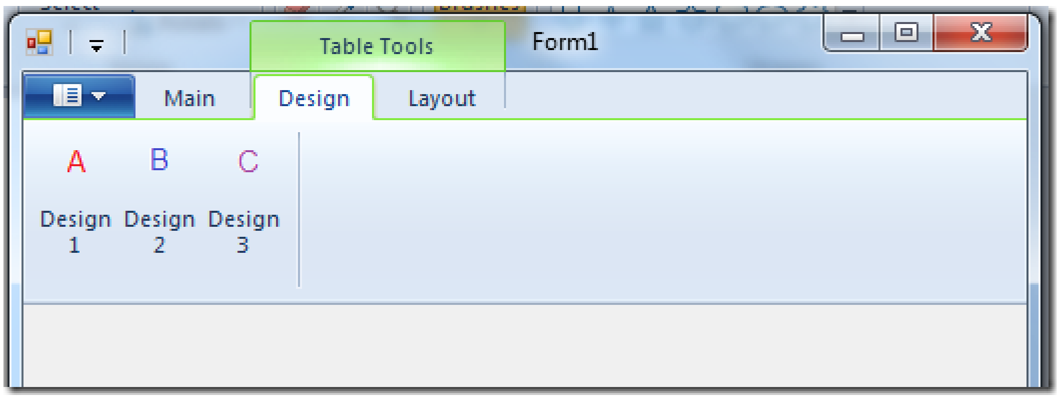
```
private Ribbon _ribbon;
private RibbonButton _buttonSwitchToAdvanced;
private RibbonButton _buttonSwitchToSimple;

public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();
    _buttonSwitchToAdvanced = new RibbonButton(_ribbon,
                                (uint)RibbonMarkupCommands.cmdButtonSwitchToAdva
nced);
    _buttonSwitchToSimple = new RibbonButton(_ribbon,
                                (uint)RibbonMarkupCommands.cmdButtonSwitchToSimp
le);

    _buttonSwitchToAdvanced.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_buttonSwitchToAdvanced_ExecuteEvent);
    _buttonSwitchToSimple.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_buttonSwitchToSimple_ExecuteEvent);
}

void _buttonSwitchToAdvanced_ExecuteEvent(object sender, ExcecuteEventArgs e)
{
    _ribbon.SetModes(1);
}

void _buttonSwitchToSimple_ExecuteEvent(object sender, ExcecuteEventArgs e)
{
    _ribbon.SetModes(0);
}
```

The *Ribbon.SetModes* method is just a simple wrapper that converts a byte array to a compact 32bit integer and pass it to the relevant framework function:

```
public void SetModes(params byte[] modesArray)
{
    // check that ribbon is initialized
    if (!Initalized)
    {
        return;
    }

    // calculate compact modes value
    int compactModes = 0;
    for (int i = 0; i < modesArray.Length; ++i)
    {
        if (modesArray[i] >= 32)
        {
            throw new ArgumentException("Modes should range between 0 to 31.");
        }

        compactModes |= (1 << modesArray[i]);
    }

    // set modes
    Framework.SetModes(compactModes);
}
```

Part 17 – ContextualTabs

Windows Ribbon for WinForms library now supports Contextual Tabs.
The result of this post is a yet another sample, "14-ContextualTabs", found on the project site.



What are contextual tabs?
Contextual tabs are additional tabs that appear when you enable their context.
For example, in word, when you select a table in your document, you get two additional tabs (Design and Layout) with commands relevant only to tables.

The basic working unit is a *TabGroup*, which is a group of contextual tabs with the same context. A *TabGroup* has a property named *ContextAvailable* (Property Identifier: *UI_PKEY_ContextAvailable*) which can have the following values:

- Active – context is currently available and the tab group should be active (= “selected”)
- Available – context is currently available (tabs are not necessarily active).
- NotAvailable – context is currently not available.

More details on this subject can be found at Displaying Contextual Tabs on MSDN.

Using ContextualTabs – Ribbon Markup

Following is the *views* section for defining contextual tabs. The *commands* section is straightforward.

```
<Application.Views>
  <Ribbon>
    <Ribbon.ContextualTabs>
      <TabGroup CommandName='cmdTabGroupTableTools'>
        <Tab CommandName='cmdTabDesign'>
          <Group CommandName='cmdGroupDesign' SizeDefinition='ThreeButtons'>
            <Button CommandName='cmdButtonDesign1' />
            <Button CommandName='cmdButtonDesign2' />
            <Button CommandName='cmdButtonDesign3' />
          </Group>
        </Tab>
        <Tab CommandName='cmdTabLayout'>
          <Group CommandName='cmdGroupLayout' SizeDefinition='TwoButtons'>
            <Button CommandName='cmdButtonLayout1' />
            <Button CommandName='cmdButtonLayout2' />
          </Group>
        </Tab>
      </TabGroup>
    </Ribbon.ContextualTabs>
    <Ribbon.Tabs>
      <Tab CommandName='cmdTabMain'>
        <Group CommandName='cmdGroupMain' SizeDefinition='TwoButtons'>
          <Button CommandName='cmdButtonSelect' />
          <Button CommandName='cmdButtonUnselect' />
        </Group>
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

Here we define a single TabGroup for “Table Tools”, with two contextual tabs, “Design” and “Layout”.

Each tab has some buttons in it.
In addition, we define in the main tab two buttons that we will use to set and unset the "Table Tools" context.

Using ContextualTabs – Code Behind

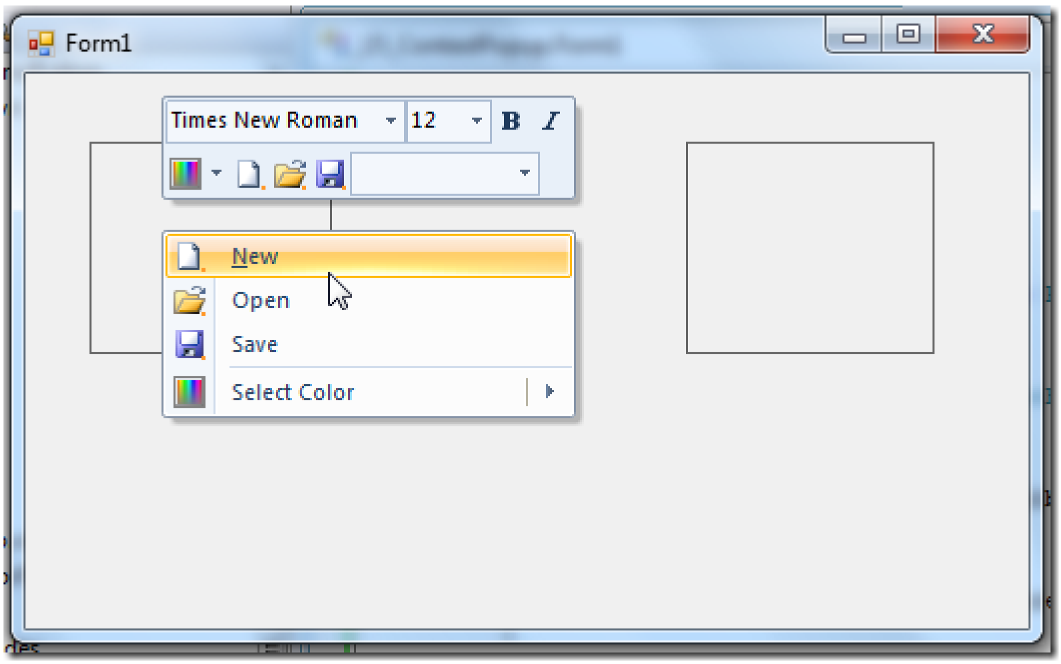
Following is an example of setting the context for a TabGroup thus making it visible.

```
private Ribbon _ribbon;
private RibbonTabGroup _tabGroupTableTools;
private RibbonButton _buttonSelect;
private RibbonButton _buttonUnselect;

public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();
    _tabGroupTableTools = new RibbonTabGroup(_ribbon,
                        (uint)RibbonMarkupCommands.cmdTabGroupTableTools);
    _buttonSelect = new RibbonButton(_ribbon,
                        (uint)RibbonMarkupCommands.cmdButtonSelect);
    _buttonUnselect = new RibbonButton(_ribbon,
                        (uint)RibbonMarkupCommands.cmdButtonUnselect);

    _buttonSelect.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_buttonSelect_ExecuteEvent);
    _buttonUnselect.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_buttonUnselect_ExecuteEvent);
}

void _buttonSelect_ExecuteEvent(object sender, ExcecuteEventArgs e)
{
    _tabGroupTableTools.ContextAvailable = ContextAvailability.Active;
}

void _buttonUnselect_ExecuteEvent(object sender, ExcecuteEventArgs e)
{
    _tabGroupTableTools.ContextAvailable = ContextAvailability.NotAvailable;
}
```

Here we just register to the ribbon buttons execute events and set the context of the TabGroup accordingly. Of course, in a real application you might use a more sophisticated logic for setting the context.

Part 18 – ContextPopup

Windows Ribbon for WinForms library now supports using a Context Popup.
The result of this post is a yet another sample, "15-ContextPopup", found on the project site.



What is a Context Popup?
A context popup is a combination of a small toolbar (called MiniToolbar) and a context menu, which a ribbon enabled application can provide. This popup usually appears when right-clicking an application surface and it usually provide common operation relevant for the clicked surface. Basically, it's just another way to get to your ribbon commands. In fact, since the MiniToolbar (the upper part of the context popup) can't be access using the keyboard, Microsoft strongly recommends that every command should be also available using the standard ribbon interface.

More details can be found at Context Popup on MSDN.

Using ContextPopup – Ribbon Markup
Following is an example of defining a context popup. Only the *Views* section is presented since the *Commands* section is obvious. The result of this markup is shown in the image above.

```
<Application.Views>
  <ContextPopup>
    <ContextPopup.MiniToolbars>
      <MiniToolbar Name="MiniToolbar">
        <MenuGroup>
          <FontControl CommandName="cmdFontControl"/>
        </MenuGroup>
        <MenuGroup>
          <DropDownColorPicker CommandName="cmdDropDownColorPicker" />
          <Button CommandName="cmdButtonNew" />
          <Button CommandName="cmdButtonOpen" />
          <Button CommandName="cmdButtonSave" />
          <ComboBox />
        </MenuGroup>
      </MiniToolbar>
    </ContextPopup.MiniToolbars>
    <ContextPopup.ContextMenus>
      <ContextMenu Name="ContextMenu">
        <MenuGroup>
          <Button CommandName="cmdButtonNew" />
          <Button CommandName="cmdButtonOpen" />
          <Button CommandName="cmdButtonSave" />
        </MenuGroup>
        <MenuGroup>
          <DropDownColorPicker CommandName="cmdDropDownColorPicker" />
        </MenuGroup>
      </ContextMenu>
    </ContextPopup.ContextMenus>
    <ContextPopup.ContextMaps>
      <ContextMap CommandName="cmdContextMap"
                  ContextMenu="ContextMenu"
                  MiniToolbar="MiniToolbar" />
    </ContextPopup.ContextMaps>
  </ContextPopup>
  <Ribbon>
  </Ribbon>
</Application.Views>
```

Things to note:

1. In the *ContextPopup.MiniToolbars* element you can define a list of available mini toolbars (upper parts).
2. In the *ContextPopup.ContextMenus* element you can define a list of available context menus (lower parts).
3. The actual definition of the context popup resides in the *ContextPopup.ContextMaps* element, where you can define a list of context popups. For each context popup you need to specify its upper part and lower part.

Using ContextPopup – Code Behind

In order to show the predefined context popups I present here two ways: the recommended way and the convenient way.

The recommended way

Use the method, *Ribbon.ShowContextPopup* to show a given context popup in a certain location, usually as a response to right clicking:

```
private Ribbon _ribbon;
```

```
public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();

    // recommended way
    panel2.MouseClick += new MouseEventHandler(panel2_MouseClick);
}

void panel2_MouseClick(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
    {
        System.Drawing.Point p = panel2.PointToScreen(e.Location);
        _ribbon.ShowContextPopup((uint)RibbonMarkupCommands.cmdContextMap, p.X,
p.Y);
    }
}
```

Note that the ShowContextPopup gets the location in screen coordinates, so a transformation using the *Control.PointToScreen* method is required.

The convenient way

If you are used to program client applications in .NET you must be familiar with ContextMenuStrip class. Every WinForms control has a *ContextMenuStrip* property where you can attach an instance of a predefined *ContextMenuStrip* instance, thus letting the .NET framework handle the opening of the context menu for you. Unfortunately, the *ContextMenuStrip* property works only with *ContextMenuStrip* instances (An IContextMenuStrip interface would be great...), so you can't easily replace the .NET implementation of a context menu with our new ribbon-based implementation.

Following is a hack you can use to achieve this. I've created a *RibbonContextMenuStrip* class that inherits from *ContextMenuStrip* and hacked my way into showing the ribbon context menu instead of the .NET context menu. The advantage is that you have an easier interface for working with the ribbon context menu:

```
private Ribbon _ribbon;
private RibbonContextMenuStrip _ribbonContextMenuStrip;

public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();
    _ribbonContextMenuStrip = new RibbonContextMenuStrip(_ribbon,
(uint)RibbonMarkupCommands.cmdContextMap);

    // convenient way
    panel1.ContextMenuStrip = _ribbonContextMenuStrip;
}
```

As you can see from the code, it is shorter and use the standard .NET way for setting context menus on controls.

Let’s see the “magic” in action:

```
public class RibbonContextMenuStrip : ContextMenuStrip
{
    private uint _contextPopupID;
    private Ribbon _ribbon;

    public RibbonContextMenuStrip(Ribbon ribbon, uint contextPopupID)
        : base()
    {
        _contextPopupID = contextPopupID;
        _ribbon = ribbon;
    }

    protected override void OnOpening(CancelEventArgs e)
    {
        _ribbon.ShowContextPopup(_contextPopupID,
                                 Cursor.Position.X,
                                 Cursor.Position.Y);
        e.Cancel = true;
    }
}
```

Note: The *RibbonContextMenuStrip* is **NOT** part of the Windows Ribbon for WinForms library. The code is available as part of the sample “15-ContextPopup”. It works but I don’t like it.

Ribbon Properties
This section has nothing to do with context popup.

The ribbon library now supports 3 general properties for the ribbon UI:

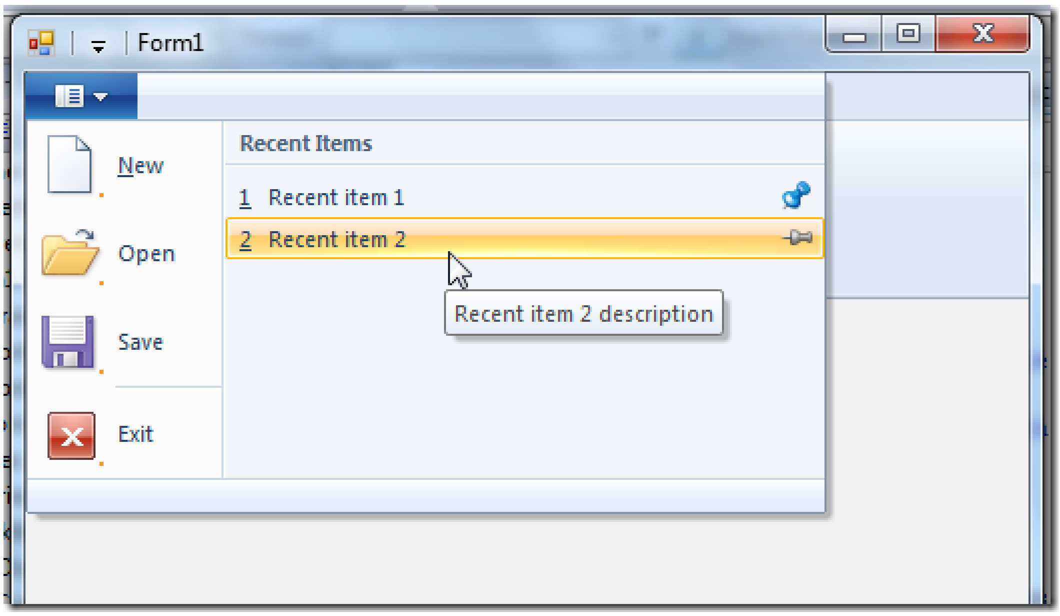
- Minimized – Specifies whether the ribbon is in a collapsed or expanded state.
  Property Identifier: *UI_PKey_Minimized*
- Viewable – Specifies whether the ribbon user interface (UI) is in a visible or hidden state.
  Property Identifier: *UI_PKey_Viewable*
- QuickAccessToolbarDock – Specifies whether the quick access toolbar is docked at the top or at the bottom.
  Property Identifier: *UI_PKey_QuickAccessToolbarDock*

Using these properties is as simple as:

```
private void Form1_Load(object sender, EventArgs e)
{
    _ribbon.Viewable = false;
}
```

Part 19 – RecentItems

Windows Ribbon for WinForms library now supports working with recent items in the application menu.
The result of this post is a yet another sample, “16-RecentItems”, found on the project site.



What are recent items?
Recent items are items in a list which appears in the application menu. They don’t have to be file names and they doesn’t have to be recent, although it is recommended.

Every item has 3 properties:

- Label – Item name, usually file name without path
- Label Description – Item tooltip, usually full filename path
- Pinned – Boolean that indicates whether the recent item should not be removed from the list

More details can be found at Recent Items on MSDN.

Using RecentItems – Ribbon Markup
Commands section:

```
<Application.Commands>
  …
  <Command Name="cmdRecentItems" Id="1005" LabelTitle="Recent Items" />
</Application.Commands>
```

Views section:

```
<Application.Views>
  <Ribbon>
    <Ribbon.ApplicationMenu>
      <ApplicationMenu CommandName="cmdApplicationMenu">
        <ApplicationMenu.RecentItems>
          <RecentItems 
CommandName="cmdRecentItems" EnablePinning="true" MaxCount="7" />
        </ApplicationMenu.RecentItems>
        …
      </ApplicationMenu>
    </Ribbon.ApplicationMenu>
  </Ribbon>
</Application.Views>
```

Things to note:

- The “Recent Items” label can be changed to whatever you need (e.g. “Days of the week”).
- Setting *EnablePinning* attribute to false will hide the pins from the application menu.
- *MaxCount* attribute specify how many items to display on the application menu.

Using RecentItems – Code Behind
Initialization:

```
private Ribbon _ribbon;
private RibbonRecentItems _ribbonRecentItems;

List<RecentItemsPropertySet> _recentItems;

public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();
    _ribbonRecentItems = new RibbonRecentItems(_ribbon,
(uint)RibbonMarkupCommands.cmdRecentItems);

    _ribbonRecentItems.ExecuteEvent += new
EventHandler<ExecuteEventArgs>(_recentItems_ExecuteEvent);
}

private void Form1_Load(object sender, EventArgs e)
{
    InitRecentItems();
}

private void InitRecentItems()
{
    // prepare list of recent items
    _recentItems = new List<RecentItemsPropertySet>();
    _recentItems.Add(new RecentItemsPropertySet()
                     {
                         Label = "Recent item 1",
                         LabelDescription = "Recent item 1 description",
                         Pinned = true
                     });
    _recentItems.Add(new RecentItemsPropertySet()
                     {
                         Label = "Recent item 2",
                         LabelDescription = "Recent item 2 description",
                         Pinned = false
                     });

    _ribbonRecentItems.RecentItems = _recentItems;
}
```

RibbonRecentItems is the helper class for working with the recent items feature. It has a property named *RecentItems* of type *IList<RecentItemsPropertySet>*. This property contains the list of the recent items. Note that it is the user responsibility for providing this list and update it when needed (add / remove items, change pinned state).

Responding to a click on an item:

```
void _recentItems_ExecuteEvent(object sender, ExcecuteEventArgs e)
{
    if (key.PropertyKey == RibbonProperties.RecentItems)
    {
        // go over recent items
        object[] objectArray = (object[])e.CurrentValue.PropVariant.Value;
        for (int i = 0; i < objectArray.Length; ++i)
        {
            IUISimplePropertySet propertySet = objectArray[i] as
IUISimplePropertySet;

            if (propertySet != null)
            {
                PropVariant propLabel;
                propertySet.GetValue(ref RibbonProperties.Label,
                                     out propLabel);
                string label = (string)propLabel.Value;

                PropVariant propLabelDescription;
                propertySet.GetValue(ref RibbonProperties.LabelDescription,
                                     out propLabelDescription);
                string labelDescription = (string)propLabelDescription.Value;

                PropVariant propPinned;
                propertySet.GetValue(ref RibbonProperties.Pinned,
                                     out propPinned);
                bool pinned = (bool)propPinned.Value;

                // update pinned value
                _recentItems[i].Pinned = pinned;
            }
        }
    }
    else if (key.PropertyKey == RibbonProperties.SelectedItem)
    {
        // get selected item index
        uint selectedItem = (uint)e.CurrentValue.PropVariant.Value;

        // get selected item label
        PropVariant propLabel;
        e.CommandExecutionProperties.GetValue(ref RibbonProperties.Label,
                                              out propLabel);
        string label = (string)propLabel.Value;

        // get selected item label description
        PropVariant propLabelDescription;
        e.CommandExecutionProperties.GetValue(ref
RibbonProperties.LabelDescription,
                                              out propLabelDescription);
        string labelDescription = (string)propLabelDescription.Value;

        // get selected item pinned value
        PropVariant propPinned;
        e.CommandExecutionProperties.GetValue(ref RibbonProperties.Pinned,
                                              out propPinned);
        bool pinned = (bool)propPinned.Value;
    }
}
```

I know, some explanations are in order.

The *ExecuteEvent* event is called on two occasions:

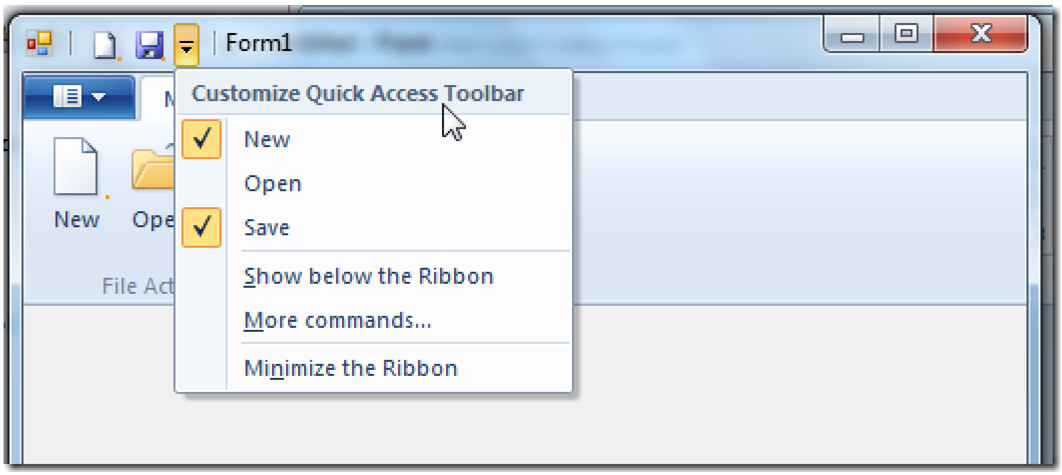
1. When the user clicks on one of the items.
2. When the user changes the pinned status of several items and then closes the menu (either by selecting one of the items or by clicking outside the menu).

When the user clicks on an item, the *e.CurrentValue* argument contains the index of the selected item and e.*CommandExecutionProperties* argument contains the properties of the selected item. The above code shows how to extract them.

When the user changes the pinned status of several items, the e.C*urrentValue* argument contains the new status of the items. It is the user responsibility to update the items in its own list. otherwise the user's change won't appear the next time he opens the menu.

Part 20 – QuickAccessToolbar

Windows Ribbon for WinForms library now supports working with the ribbon quick access toolbar. The result of this post is a yet another sample, "17-QuickAccessToolbar", found on the project site.



What is Quick Access Toolbar (QAT)?
Quick access toolbar resides on the left of the window title. The user can save their common ribbon commands he wants to easily access. A user can add a ribbon button (or toggle button or checkbox) to the QAT by right clicking it and select "Add to Quick Access Toolbar".

The application developer can specify a set of "default buttons" (In the above image: new, open and save). This is done using ribbon markup. Also, the application developer can add commands dynamically using code.

One more feature of the ribbon is the ability to save and load the list of commands. Using this feature to save and load the settings between application sessions provides the user with a consistent UI experience.

More details can be found at Quick Access Toolbar on MSDN.

Using QuickAccessToolbar – Ribbon Markup

Following is an example of a *views* section that uses *QuickAccessToolbar*:

```
<Application.Views>
  <Ribbon>
    <Ribbon.QuickAccessToolbar>
      <QuickAccessToolbar
CommandName='cmdQAT' CustomizeCommandName='cmdCustomizeQAT'>
        <QuickAccessToolbar.ApplicationDefaults>
          <Button
CommandName="cmdButtonNew" ApplicationDefaults.IsChecked="true" />
          <Button
CommandName="cmdButtonOpen" ApplicationDefaults.IsChecked="false" />
          <Button
CommandName="cmdButtonSave" ApplicationDefaults.IsChecked="false" />
        </QuickAccessToolbar.ApplicationDefaults>
      </QuickAccessToolbar>
    </Ribbon.QuickAccessToolbar>
    <Ribbon.Tabs>
      <Tab CommandName="cmdTabMain">
        <Group CommandName="cmdGroupFileActions" SizeDefinition="ThreeButtons">
          <Button CommandName="cmdButtonNew" />
          <Button CommandName="cmdButtonOpen" />
          <Button CommandName="cmdButtonSave" />
        </Group>
      </Tab>
    </Ribbon.Tabs>
  </Ribbon>
</Application.Views>
```

As you can see, the QuickAccessToolbar element defines two command names. The first, identified by the attribute "*CommandName*", is the command for the actual quick access toolbar. The second, identified by the attribute "*CustomizeCommandName*" is the command for the button "More Commands…" which you can see in the image above on the menu. The "More Commands…" button is optional, but if specified, can be used to open an application defined dialog for selecting more commands into the QAT.

Another thing you can see in the markup is the use of *QuickAccessToolbar.ApplicationDefaults* element which let the developer specify which items will be in the QAT popup menu. Every item can be marked with *IsChecked* attribute, in which case the item will appear on the QAT.

Using QuickAccessToolbar – Code Behind

Initialization:

```
private Ribbon _ribbon;
private RibbonQuickAccessToolbar _ribbonQuickAccessToolbar;

public Form1()
{
    InitializeComponent();

    _ribbon = new Ribbon();
    _ribbonQuickAccessToolbar = new RibbonQuickAccessToolbar(_ribbon,
                                        (uint)RibbonMarkupCommands.cmdQAT,
                                        (uint)RibbonMarkupCommands.cmdCustomizeQ
AT);

    // register to the QAT customize button
    _ribbonQuickAccessToolbar.ExecuteEvent +=
                    new new
EventHandler<ExecuteEventArgs>(_ribbonQuickAccessToolbar_ExecuteEvent);
}
```

Note that the customize QAT button is optional so if you didn't declare it you should use the other constructor of *RibbonQuickAccessToolbar* class.

Handling the customize QAT button ("More Commands…"):

```
void _ribbonQuickAccessToolbar_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    MessageBox.Show("Open customize commands dialog..");
}
```

The application developer should probably load a dialog that allows the user to select commands and then set them by manipulating the commands list.

Manipulating commands list:

```
void _buttonNew_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    // changing QAT commands list
    IUICollection itemsSource = _ribbonQuickAccessToolbar.ItemsSource;
    itemsSource.Clear();
    itemsSource.Add(new GalleryCommandPropertySet()
                    { CommandID = (uint)RibbonMarkupCommands.cmdButtonNew });
    itemsSource.Add(new GalleryCommandPropertySet()
                    { CommandID = (uint)RibbonMarkupCommands.cmdButtonOpen });
    itemsSource.Add(new GalleryCommandPropertySet()
                    { CommandID = (uint)RibbonMarkupCommands.cmdButtonSave });
}
```

This code is very similar to the code for manipulating the command list of a gallery.

Saving and loading ribbon settings:

```
private Stream _stream;

void _buttonSave_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    // save ribbon QAT settings
    _stream = new MemoryStream();
    _ribbon.SaveSettingsToStream(_stream);
}

void _buttonOpen_ExecuteEvent(object sender, ExecuteEventArgs e)
{
    // load ribbon QAT settings
    _stream.Position = 0;
    _ribbon.LoadSettingsFromStream(_stream);
}
```

You can save the settings in any .NET stream object, usually it should be saved into a file or the registry.

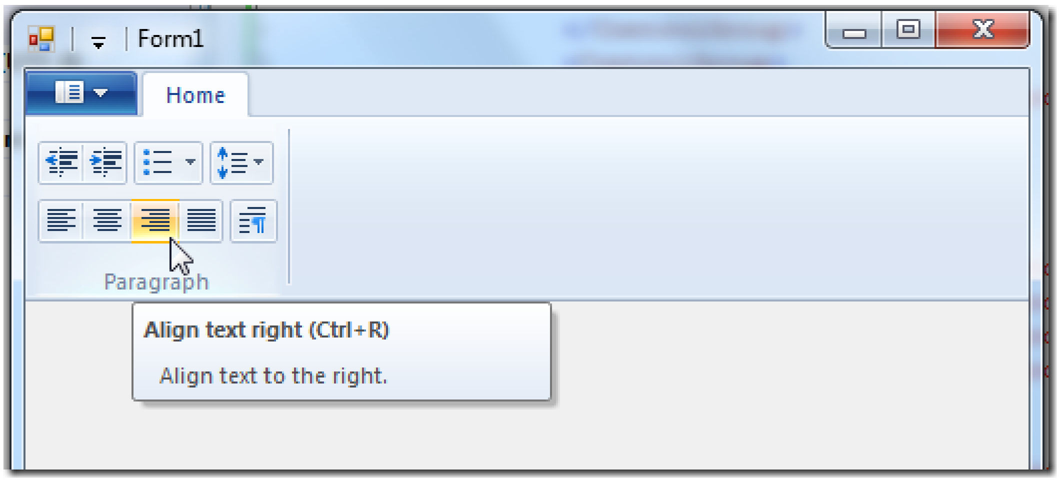
The settings which are saved are: QAT commands list, ribbon QuickAccessToolbarDock property and ribbon Minimized property.

Part 21 – SizeDefinition

After reviewing the MSDN documentation for the Windows Ribbon Framework I've discovered there is only one subject I haven't covered in my ribbon posts. This post comes to rectify this issue.

This post is about how to define custom size definitions for ribbon group elements. The post is entirely about ribbon markup, so no changes to the Windows Ribbon for WinForms library.

Nevertheless, I've uploaded a new sample "18-SizeDefinition" to the project site. In this sample I create the paragraph group which you can find in WordPad application. What's special about this group is the custom layout it represents.



Align text right (Ctrl+R)
Align text to the right.

What is SizeDefinition?
SizeDefinition is the ribbon markup element which allows us, developers, to control the layout of controls in a group. Every such definition is called layout template.

Every group can scale to the following sizes: Large, Medium, Small and Popup. This allows the ribbon framework to show the UI even when we don't have a lot of screen space.

Note: Scaling, which has an important impact on how your group will look was already reviewed in Windows Ribbon for WinForm, Part 6 – Tabs, Groups and HelpButton.

Every layout template includes:

- List of controls participating in the group
- A definition of a layout for a given group size.

Predefined Layout Templates
Microsoft has provided predefined common layout template so we can use them on our groups without having to specify the exact layout. Up until now, all the previous samples used them.

Reminder:

```
<Group CommandName="cmdGroupFileActions" SizeDefinition="ThreeButtons">
  <Button CommandName="cmdButtonNew" />
  <Button CommandName="cmdButtonOpen" />
  <Button CommandName="cmdButtonSave" />
</Group>
```

The “ThreeButtons” is a name of a predefined layout template that handles the layout for 3 button controls.

Here is a boring list of available predefined templates:

- OneButton
- TwoButtons
- ThreeButtons
- ThreeButtons-OneBigAndTwoSmall
- ThreeButtonsAndOneCheckBox
- FourButtons
- FiveButtons
- FiveOrSixButtons
- SixButtons
- SixButtons-TwoColumns
- SevenButtons
- EightButtons
- EightButtons-LastThreeSmall
- NineButtons
- TenButtons
- ElevenButtons
- OneFontControl
- OneInRibbonGallery
- InRibbonGalleryAndBigButton
- InRibbonGalleryAndButtons-GalleryScalesFirst
- ButtonGroups
- ButtonGroupsAndInputs
- BigButtonsAndSmallButtonsOrInputs

Their exact layout can be found at Customizing a Ribbon Through Size Definitions and Scaling Policies on MSDN.

Defining Custom Layout Templates

Custom layout templates can be defined in two ways: Inline and standalone.

<u>Standalone definition</u>

Standalone means you define the layout once, under a *Ribbon.SizeDefinitions* element and then use its name in your group definition, exactly like the predefined layout templates. For example:

Defining the named, standalone, custom layout:

```
<Ribbon.SizeDefinitions>
  <SizeDefinition Name="ParagraphLayout">
    <ControlNameMap>
      <ControlNameDefinition Name="button1" />
      <ControlNameDefinition Name="button2" />
      <ControlNameDefinition Name="button3" />
      <ControlNameDefinition Name="button4" />
      <ControlNameDefinition Name="button5" />
      <ControlNameDefinition Name="button6" />
      <ControlNameDefinition Name="button7" />
      <ControlNameDefinition Name="button8" />
      <ControlNameDefinition Name="button9" />
    </ControlNameMap>
    <GroupSizeDefinition Size="Large">
      <Row>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button1" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button2" IsLabelVisible="false" />
        </ControlGroup>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button3" IsLabelVisible="false" />
        </ControlGroup>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button4" IsLabelVisible="false" />
        </ControlGroup>
      </Row>
      <Row>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button5" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button6" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button7" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button8" IsLabelVisible="false" />
        </ControlGroup>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button9" IsLabelVisible="false" />
        </ControlGroup>
      </Row>
    </GroupSizeDefinition>
  </SizeDefinition>
</Ribbon.SizeDefinitions>
```

Although this looks intimidating, this is actually pretty simple. First, the *ControlNameMap* element is a definition of placeholder's controls used in the layout. In our example we define 9 controls.

Then comes the layout definition. This is done in a *GroupSizeDefinition* element, where we set as an attribute what is the group scale size we are defining. Remember that different groups sizes will have different layouts. In our example we define a layout only for large size.

Then we use the *Row* elements to specify that our layout comes in two lines (three lines is the maximum).

In every row we use *ControlGroup* elements to specify grouping of controls. Controls which are in the same group have no spacing between them.

Using the custom layout is very simple:

```
<Group CommandName="cmdGroupParagraph" SizeDefinition="ParagraphLayout">
  <Button CommandName="cmdDecreaseIndent" />
  <Button CommandName="cmdIncreaseIndent" />
  <SplitButton>
    <Button CommandName="cmdStartList" />
  </SplitButton>
  <DropDownButton CommandName="cmdLineSpacing">
    <Button />
  </DropDownButton>
  <Button CommandName="cmdAlignLeft" />
  <Button CommandName="cmdAlignCenter" />
  <Button CommandName="cmdAlignRight" />
  <Button CommandName="cmdJustify" />
  <Button CommandName="cmdParagraph" />
</Group>
```

<u>Inline definition</u>

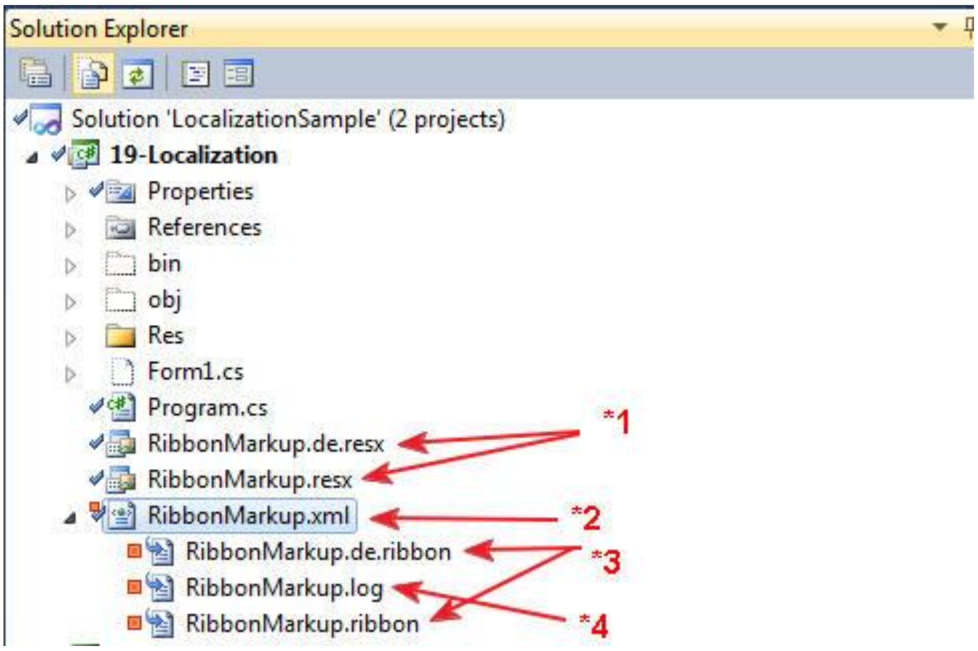
Inline means you write the layout definition inside your actual group definition. Here is the same example, only an Inline version:

```
<Group CommandName="cmdGroupParagraph">
  <SizeDefinition>
    <ControlNameMap>
      <ControlNameDefinition Name="button1" />
      <ControlNameDefinition Name="button2" />
      <ControlNameDefinition Name="button3" />
      <ControlNameDefinition Name="button4" />
      <ControlNameDefinition Name="button5" />
      <ControlNameDefinition Name="button6" />
      <ControlNameDefinition Name="button7" />
      <ControlNameDefinition Name="button8" />
      <ControlNameDefinition Name="button9" />
    </ControlNameMap>
    <GroupSizeDefinition Size="Large">
      <Row>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button1" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button2" IsLabelVisible="false" />
        </ControlGroup>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button3" IsLabelVisible="false" />
        </ControlGroup>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button4" IsLabelVisible="false" />
        </ControlGroup>
      </Row>
      <Row>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button5" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button6" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button7" IsLabelVisible="false" />
          <ControlSizeDefinition ControlName="button8" IsLabelVisible="false" />
        </ControlGroup>
        <ControlGroup>
          <ControlSizeDefinition ControlName="button9" IsLabelVisible="false" />
        </ControlGroup>
      </Row>
    </GroupSizeDefinition>
  </SizeDefinition>
  <Button CommandName="cmdDecreaseIndent" />
  <Button CommandName="cmdIncreaseIndent" />
  <SplitButton>
    <Button CommandName="cmdStartList" />
  </SplitButton>
  <DropDownButton CommandName="cmdLineSpacing">
    <Button />
  </DropDownButton>
  <Button CommandName="cmdAlignLeft" />
  <Button CommandName="cmdAlignCenter" />
  <Button CommandName="cmdAlignRight" />
  <Button CommandName="cmdJustify" />
  <Button CommandName="cmdParagraph" />
</Group>
```

Part 22 – Localization

Add Ribbon XAML(RibbonMarkup.xml) and Localization Files (RibbonMarkup.resx)

For more details about Ribbon XAML, look at MSDN or the CodePlex samples mentioned above.

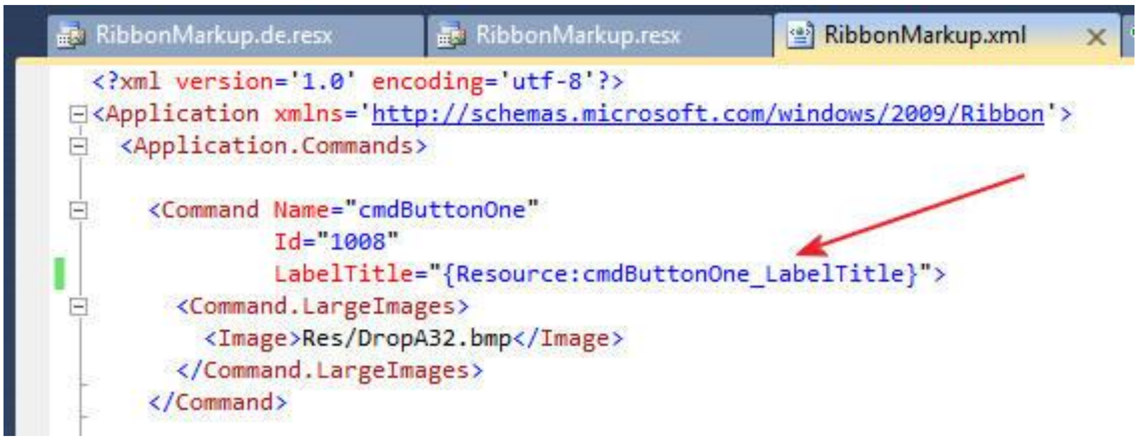


1. The localization information for the Ribbon text. The file name must be equal to the Ribbon XAML file
2. The Ribbon XAML file
3. The Ribbon embedded resource generated by custom build tool
4. The Ribbon log file generated by custom build tool
5. The custom build tool name to generate *3 and *4

How Localization Works

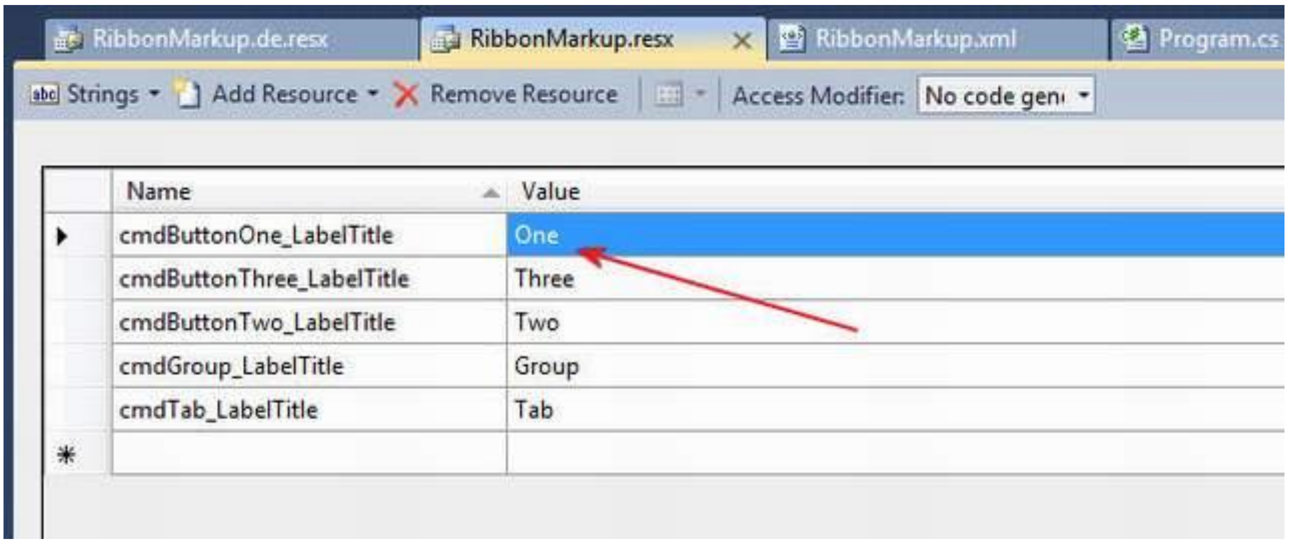
Ribbon UI information must be defined in the Ribbon XAML file and localization information in the ResX file. The custom tool is searching for "`{Resource`" tags in the XAML file and replacing it using information from ResX file. Multiple *.*ribbon* files are generated - one for each ResX file. All these *.*ribbon* files have to set to Embedded Resources in the Build Property of your project.

Define localized text inside Ribbon XAML using {Resource:<ResourceKey>} notation.



```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns='http://schemas.microsoft.com/windows/2009/Ribbon'>
  <Application.Commands>

    <Command Name="cmdButtonOne"
             Id="1008"
             LabelTitle="{Resource:cmdButtonOne_LabelTitle}">
      <Command.LargeImages>
        <Image>Res/DropA32.bmp</Image>
      </Command.LargeImages>
    </Command>
```

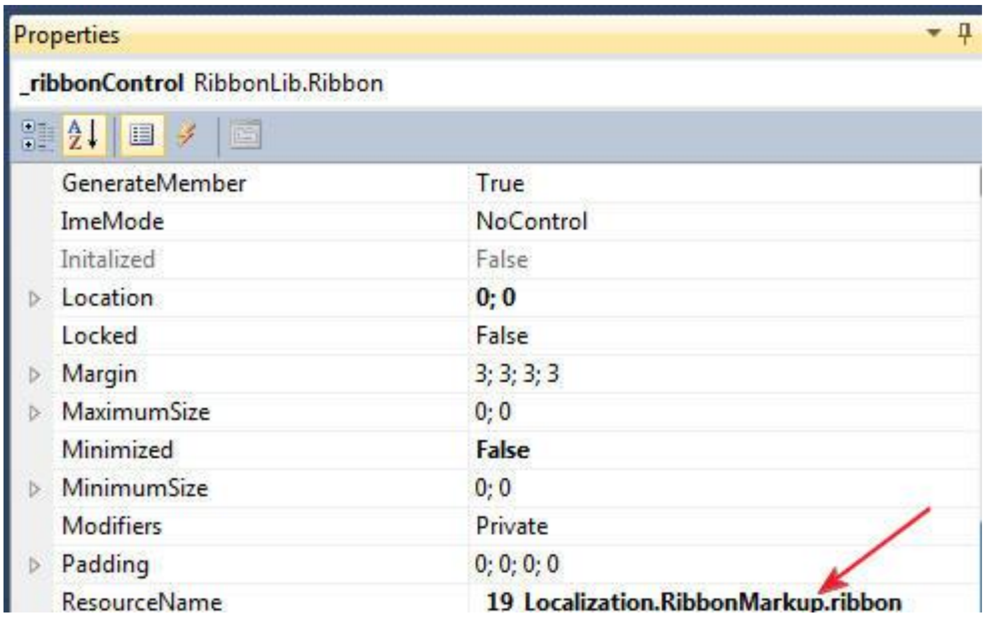
Add localized resources to ResX file



Name	Value
cmdButtonOne_LabelTitle	One
cmdButtonThree_LabelTitle	Three
cmdButtonTwo_LabelTitle	Two
cmdGroup_LabelTitle	Group
cmdTab_LabelTitle	Tab

Specify the Ribbon UI in the Ribbon control properties

In the properties of the Ribbon control, we must specify the default ".*ribbon*" file generated by the `RibbonGenerator` build tool (rgc.exe or RibbonPreview.exe). The file must be an embedded resource in your application. You can find it under your Ribbon XAML file.

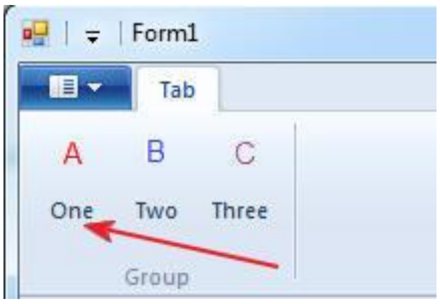


GenerateMember	True
ImeMode	NoControl
Initalized	False
Location	0; 0
Locked	False
Margin	3; 3; 3; 3
MaximumSize	0; 0
Minimized	False
MinimumSize	0; 0
Modifiers	Private
Padding	0; 0; 0; 0
ResourceName	19 Localization.RibbonMarkup.ribbon

Result looks like...

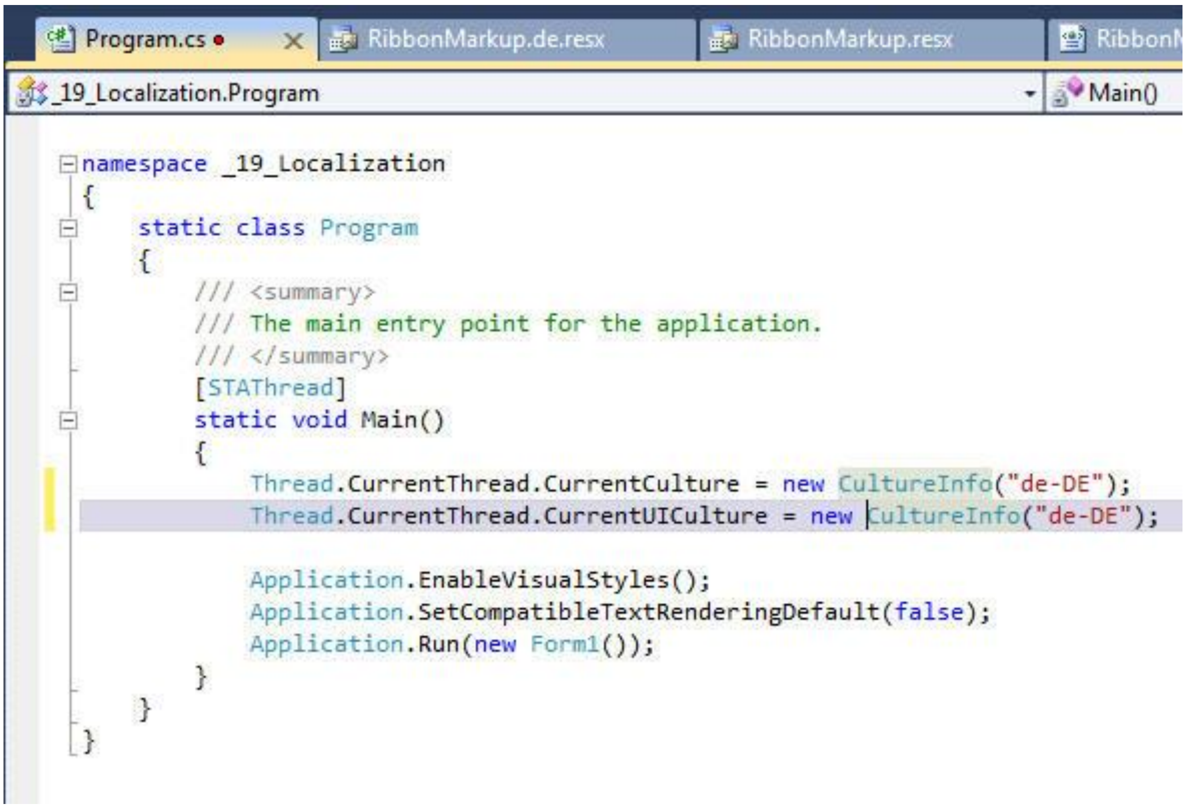
Let's see how the result looks like. In our sample, we have two Ribbon resources: Default and German.

Localized by default culture



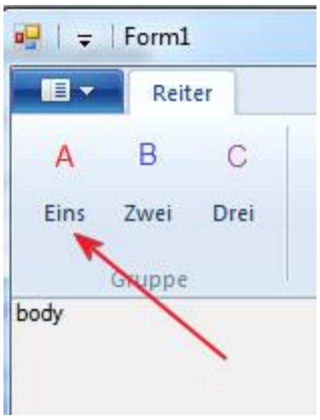
Specify culture information

In order to use a localized Ribbon, the application's current culture must be set by changing the `CurrentCulture` property of the current thread.



```
namespace _19_Localization
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
            Thread.CurrentThread.CurrentUICulture = new CultureInfo("de-DE");

            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

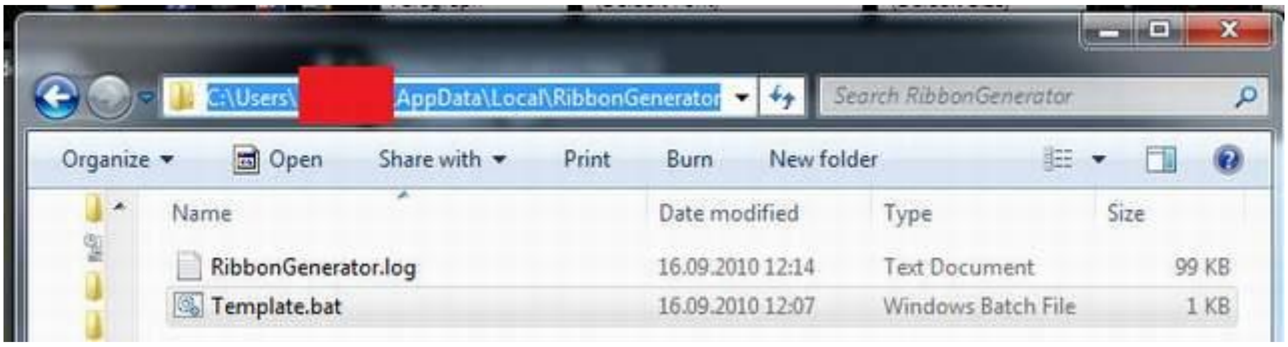
Localized by German culture



Part 23 – UIRibbon Tools

RibbonGenerator Custom Tool for .ribbon File Generation (rgc.exe)

The custom tool does the same job as the pre-build event script in the CodePlex samples. In short: It generates the "*.ribbon*" files. More: The custom tool creates one *.ribbon* file for each *.ResX* file to support localization. It needs Windows SDK binary tools and Visual Studio C++ tools be present on the system, because it just creates and executes batch files. The template of the batch file will be created on the first run in the user's app data folder (C:\Users\<user>\AppData\Local\RibbonGenerator\Template.bat). This template can be customized by you, but it should work without modifications.



Name	Date modified	Type	Size
RibbonGenerator.log	16.09.2010 12:14	Text Document	99 KB
Template.bat	16.09.2010 12:07	Windows Batch File	1 KB

Template.bat

If the `RibbonGenerator` custom tool will not work on your system but all requirements are installed, probably you need to customize the template.*bat* file. Here you can see the content of the Template.bat on a Windows 10 Professional, 64 Bit with installed Visual Studio 2019 C++ Tools and the Windows 10 SDK.

```
"C:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86\UICC.exe" "{XmlFilename}"
"{BmlFilename}" /res:"{RcFilename}" /header:"{HeaderFilename}"

"C:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86\rc.exe" /v "{RcFilename}"

cmd /c "("C:\Program Files (x86)\Microsoft Visual
Studio\2019\Community\VC\Auxiliary\Build\vcvars32.bat") && ("C:\Program Files
(x86)\Microsoft Visual
Studio\2019\Community\VC\Tools\MSVC\14.24.28314\bin\Hostx86\x86\link.exe" /VERBOSE
/NOENTRY /DLL /MACHINE:X86 /OUT:"{DllFilename}" "{ResFilename}")"
```

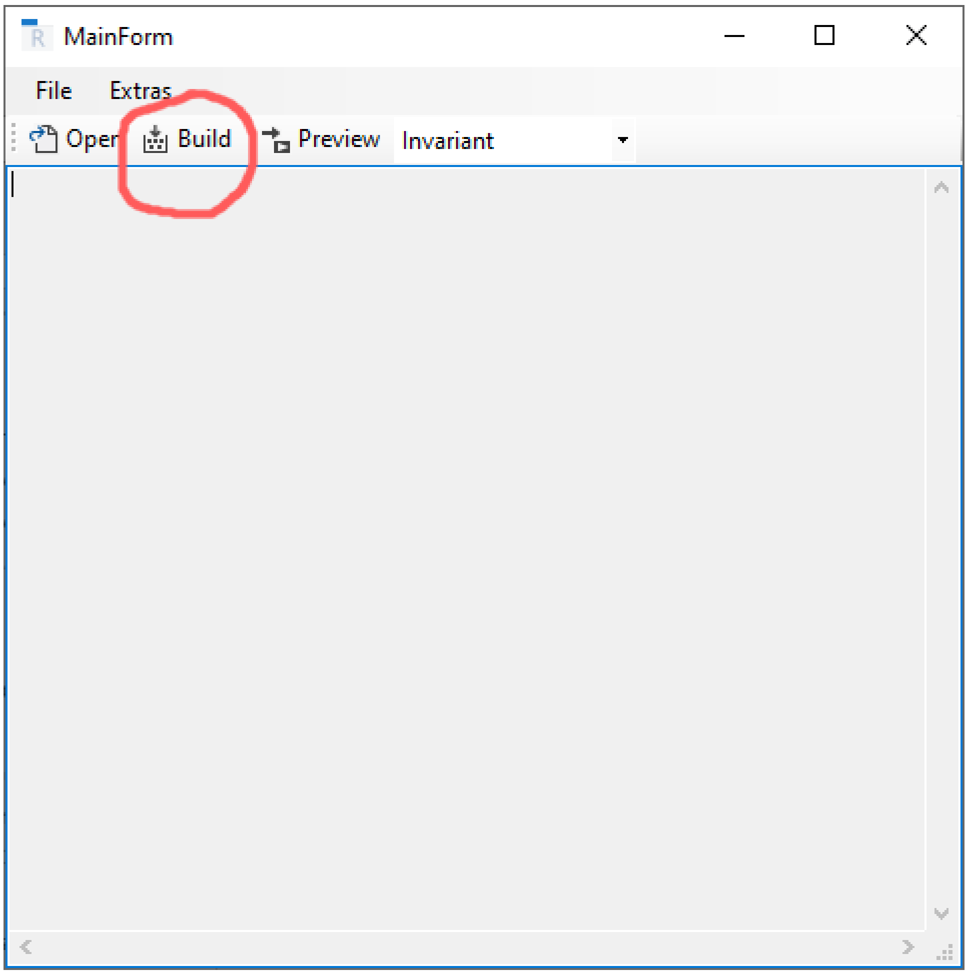
As you can see, the **Template.bat** contains process steps that will execute while the custom tool is running. If you delete the template.bat file, it will be recreated on the next run. To ensure, that the `RibbonGenerator` will work correctly, install the Windows SDK and the Visual Studio C++ Tools first, otherwise delete or customize the *template.bat* file.

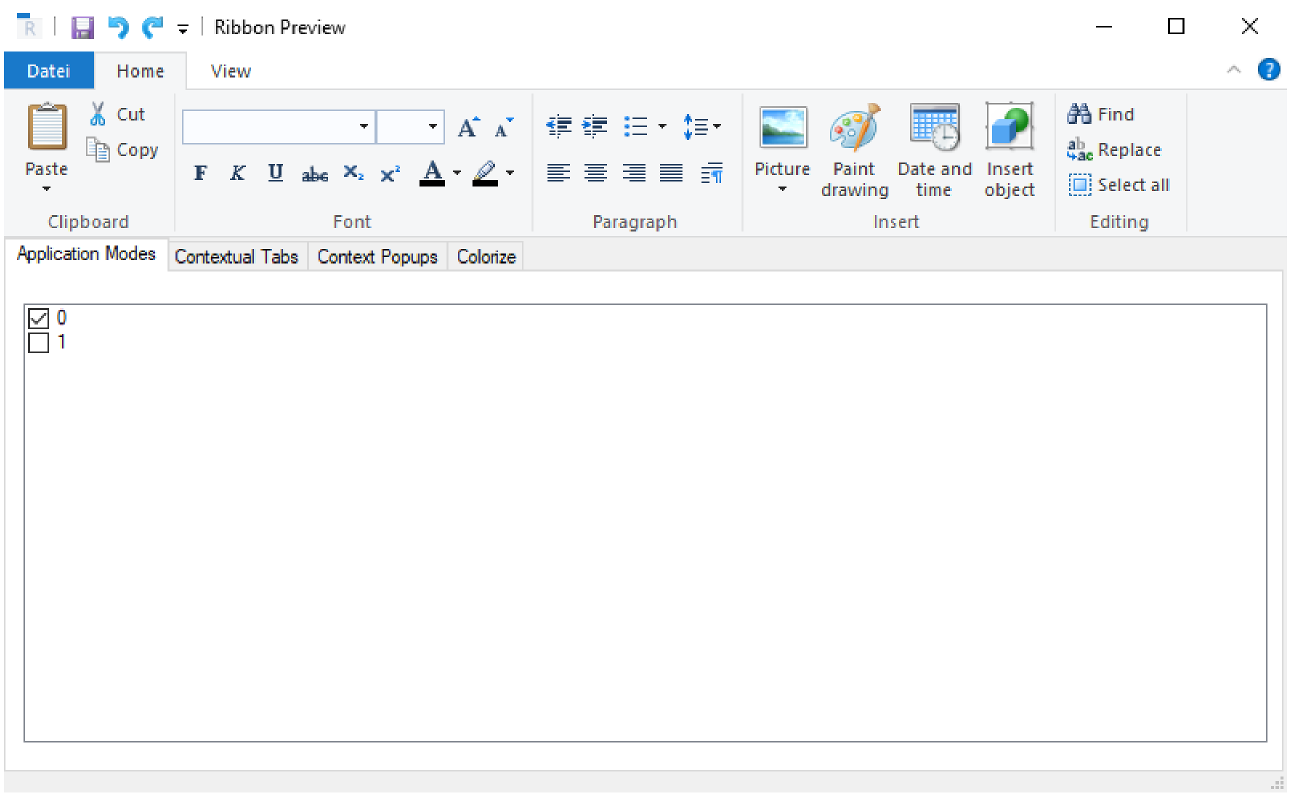
Todo ?

Part 24 – Ribbon Designer

Just a Delphi based Designer for building the RibbonMarkup.xml, see RibbonDesigner.exe
The toolchain for building and preview are not working with this RibbonDesigner.

For this we developed a program named RibbonPreview (In Setup RibbonPreview.msi).


MainForm
File
Extras
Open
Build
Preview
Invariant


Ribbon Preview
Datei
Home
View
Paste
Cut
Copy
Clipboard
Font
Paragraph
Picture
Paint drawing
Date and time
Insert object
Insert
Find
Replace
Select all
Editing
Application Modes
Contextual Tabs
Context Popups
Colorize
0
1

Part 25 – Miscellaneous

Wrapper class RibbonItems

The custom build tool generates a file named RibbonItems.Designer.cs. In this file you get a wrapper class for all Ribbon items one defined in the RibbonMarkup.xml file. You should not modify this file. The wrapper class is defined as a partial class. So, you can extend this class by a file named RibbonItems.cs with the same namespace and class name. The namespace of the wrapper class is “RibbonLib.Controls” and the class name is RibbonItems. Don’t forget the partial Keyword at the class declaration. The constructor of the wrapper class RibbonItems should be called in the Form constructor after InitializeComponent(). The parameter of the constructor is the Ribbon Control which is placed to the Form. In the file RibbonItems.cs you can define the whole logic including events for the Ribbon items. In your user defined RibbonItems.cs you can also define additional constructors with some more parameters if you need them in your logic. These constructors have to call the standard constructor with the Ribbon Control as parameter.

Hint: If you have more than one Ribbon Control for different Forms in the project, then you should name the RibbonMarkup.xml to RibbonMarkup1.xml, RibbonMarkup2.xml, ... RibbonMarkup9.xml because we got different classes of RibbonItems(x).Designer.cs .

Todo ?

Part 26 – News

Todo ?