Autodesk University 2009 LEARN. CONNECT. EXPLORE.

# Upgrading Your Autodesk® Inventor® Add-Ins to Use the New Ribbon User Interface

Brian Ekins – Autodesk

**Virtual Session** This session will introduce you to the features of the new ribbon user interface introduced in Inventor 2010. You'll learn about the new ribbon-related functionality in the Inventor programming interface and how to use it to add support for the ribbon interface to your add-ins. Because Inventor users can choose to use either the classic or ribbon interface in Inventor, you'll also learn how your add-in can provide support for both.

**About the Speaker:**
Brian is a designer for the Autodesk Inventor® programming interface. He began working in the CAD industry over 25 years ago in various positions, including CAD administrator, applications engineer, CAD API designer, and consultant. Brian was the original designer of the Inventor® API and has presented at conferences and taught classes throughout the world to thousands of users and programmers.

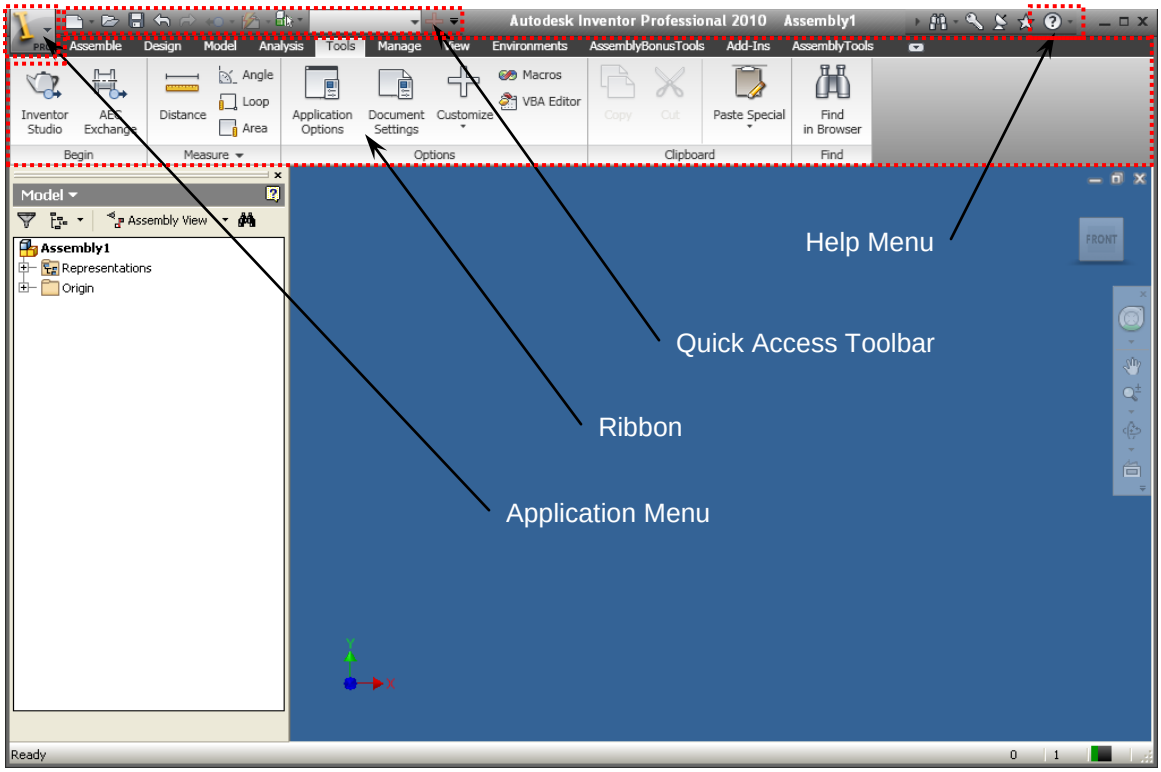
brian.ekins@autodesk.com

Inventor 2010 introduced a new user-interface called the *ribbon* user-interface. This ended up being somewhat controversial with people either liking or hating it. For those that don't like the ribbon, Inventor 2010 still supports the ability to use the old user-interface, now known as the *classic* user-interface. Luckily for me, this paper doesn't have to discuss the merits or shortcomings of the ribbon user-interface. The fact is that, love it or hate it, the ribbon is here and as an Inventor Add-In writer you need to be aware of what's required to support it. In this paper I'll address the following questions:

1. From a user's perspective, what is the ribbon user-interface?
2. Can my VBA macros work with the ribbon?
3. Will my existing add-ins work with the ribbon?
4. How does the API support the ribbon?
5. What do I need to consider when adding my buttons to the ribbon?
6. What do I do about the classic interface?

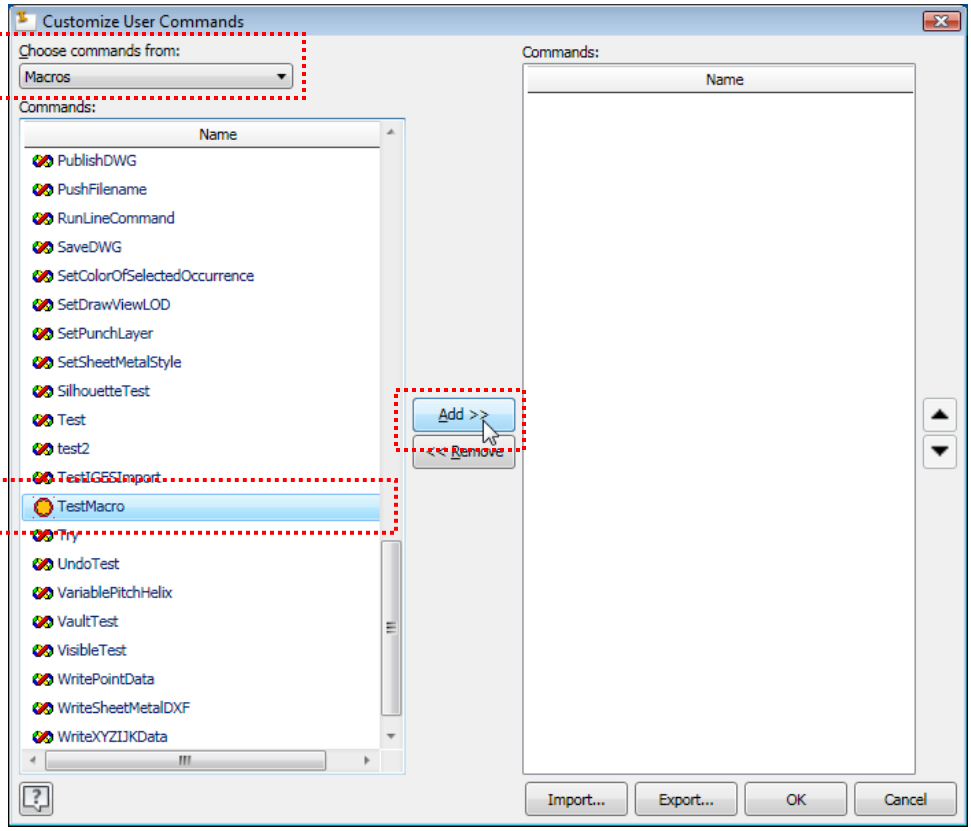
## What is the Ribbon User-Interface?

The ribbon user-interface is a new format for presenting the set of Inventor commands to the end-user. It's very similar to the user-interface that Microsoft introduced in Office 2007. The picture below shows Inventor using the ribbon interface. When comparing it with the classic interface there are several things missing; the menu, the standard toolbar, and the panel menu. The new interface has four primary components, as labeled below. This paper isn't intended to be an introduction to the ribbon for the end-user but I did want to emphasize that the structure is completely different from the classic interface; with the only thing the same between them being the browser.

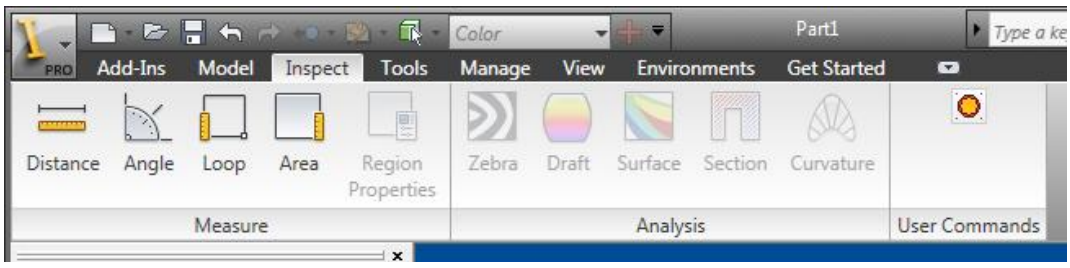


## VBA Macros in the Ribbon

Before looking at programming the ribbon let's look at how the ribbon supports VBA macros. In the classic interface, users can use the **Customize** command to add buttons for their VBA macros. They can drag and drop the buttons for their macros anywhere in the user-interface. The ribbon provides somewhat similar capability using the **Customize User Commands** command shown below. This command is accessed by right-clicking anywhere within the ribbon and running the command from the context menu. In the combo box on the upper-left you can choose "Macros" to limit the list to the available VBA macros. You can then select the desired macro in the list and click the "Add >>" button to add it to the command list on the right.



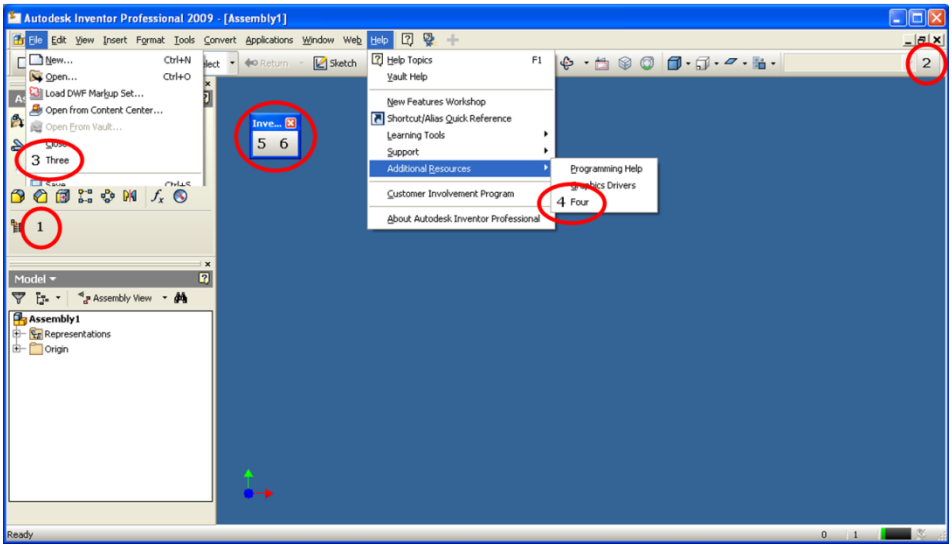

When you add a command, a new panel called "User Commands" is created in the active taband the command is added to that panel, as shown below. You can specify the icons for VBA macros just the same as you did with the classic interface. See my blog article for a detailed description of how to do that. http://modthemachine.typepad.com/my_weblog/2008/11/creating-buttons-for-vba-macros.html



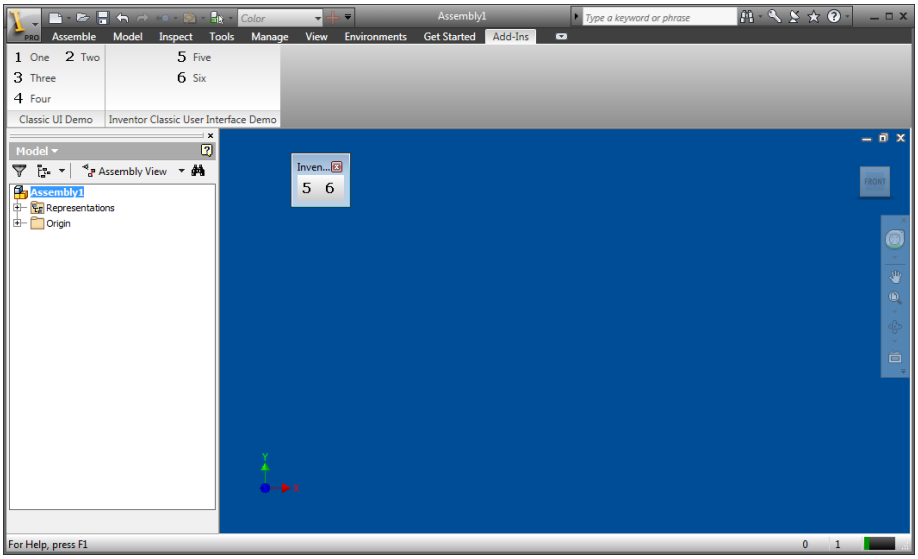
## How Does My Existing Add-In Work With the Ribbon?

Let's assume you have an add-in that you wrote for Inventor 2009 or earlier and it creates buttons in Inventor's user-interface. The question is: what do you need to do to have your add-in work in Inventor 2010's ribbon interface? The answer is: nothing. Your existing add-in will run and its user-interface will be available in the ribbon interface. However, it's likely that your add-in's commands won't be positioned where you would like them to be.

Below is a picture pointing out the commands that a sample add-in added to the classic interface in Inventor 2009. They've been added to several areas of the classic interface; menus (File and Help), default toolbar, panel bar, and a custom toolbar. The add-in chose their position so the commands would be available at the time they're needed and could be easily found.

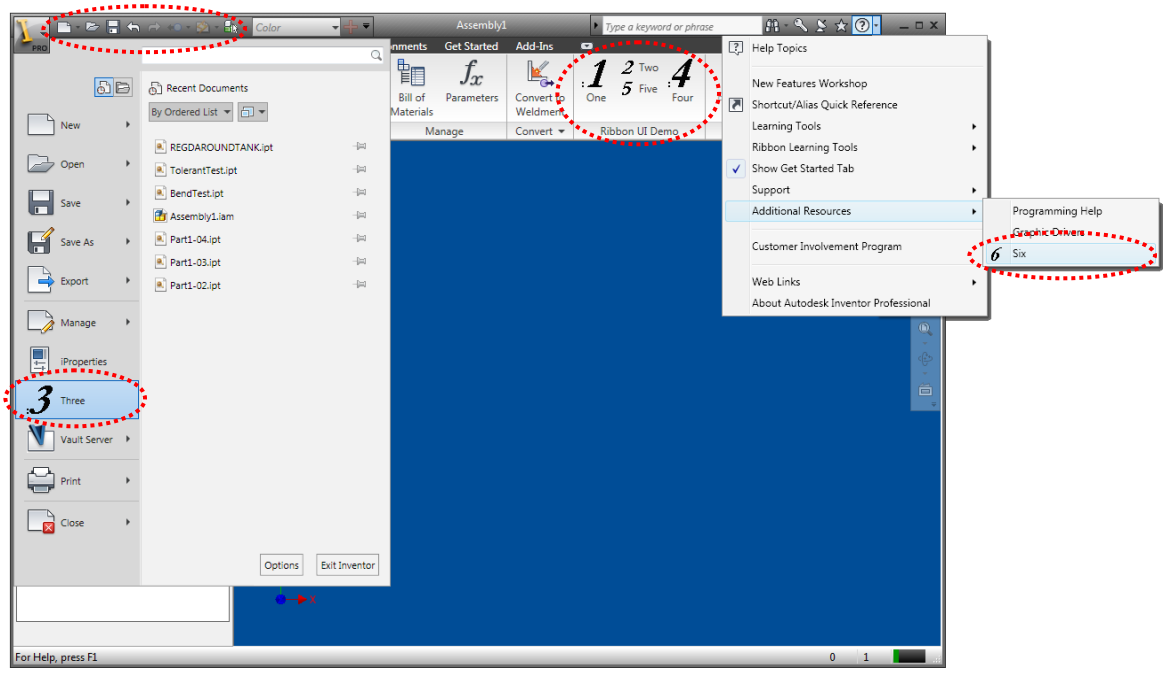



Here's the result of running the same add-in in Inventor 2010. Notice that all 6 commands are available but they've all been created within the "Add-Ins" tab of the ribbon. Commands 5 and 6 are also still available in their own toolbar.



When doing the automatic conversion from the classic to the ribbon interface, Inventor is smart enough to know that most of the original commands were added to areas of the API that were associated with the assembly environment. Because of that the contents of the Add-In tab will change depending on what type of document is currently active. The picture above shows the result when an assembly is active. If a part is active then only command 4 is available, since it was in the Help menu which is the same for all environments.

The picture below shows a better result which is achieved by modifying the add-in so it has ribbon specific code. In the example, most of the commands have been added to a new panel in the Assembly tab of the Assembly ribbon. One command is on the Application menu and the last one is on the Help menu. You can also add commands to the QAT (Quick Access toolbar) although that's not demonstrated by this example.



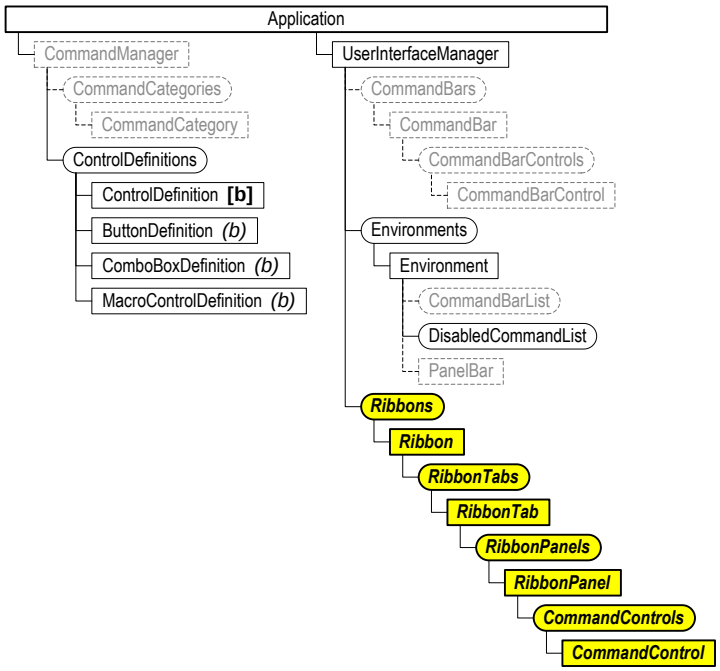

One possible issue that's worth mentioning is the case where you have an add-in that will be used by people running either Inventor 2009 or Inventor 2010. You can't have an Add-In that will run in Inventor 2009 and that can still support the newer API functions in Inventor 2010. This means you can't have a single add-in that will work in Inventor 2009 and still fully support the ribbon in Inventor 2010. There are two solutions to this. The first is to write the add-in for Inventor 2009 and its classic interface and rely on the automatic ribbon insertion that Inventor performs. The second is to have two versions of your add-in; one for Inventor 2009 and another for Inventor 2010, where you've taken advantage of the new API functionality to support the ribbon.

## Ribbon Terminology and the API

Before getting into the coding details of how to add support for the ribbon let's take a high level look at the components of the ribbon and the corresponding API objects. One advantage of the ribbon over the classic interface, from a programming perspective, is that has a well defined structure, which makes it more intuitive and easier to program.

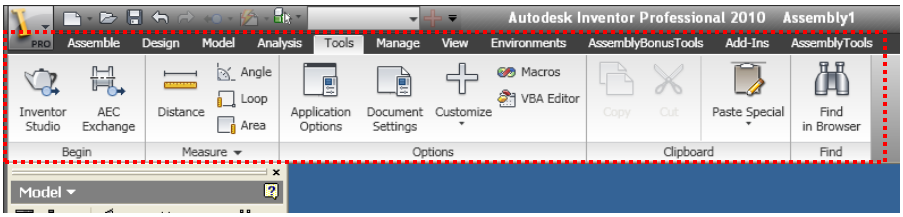
## Ribbon API Object Hierarchy

Below is the full object hierarchy for the portion of Inventor's API that works with the user-interface. The highlighted items are the new objects added to support the ribbon. The grayed items are those that are specific to the classic interface. The others are objects that are used by both the classic and ribbon interfaces.




## Ribbon

The ribbon is the entire area highlighted below and is represented by the Ribbon API object.

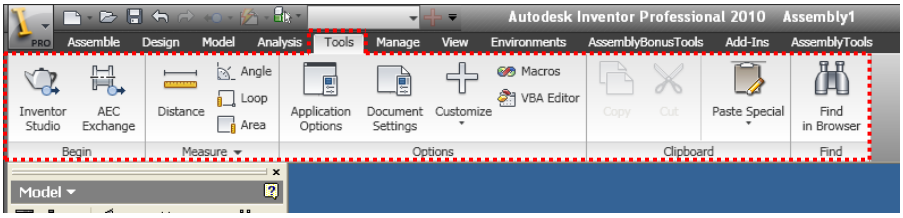



Internally there are seven different ribbons, primarily one for each document type. You can't create or delete ribbons but you can edit the seven existing ribbons by adding and removing elements from the ribbon. You typically access a specific ribbon through the API using the ribbon's internal name. The internal names for the ribbons are:

- ZeroDoc (Displayed when there aren't any documents open)
- Part
- Assembly
- Drawing
- Presentation
- iFeatures
- UnknownDocument (Used for notebook and drawing view orientation environments.)

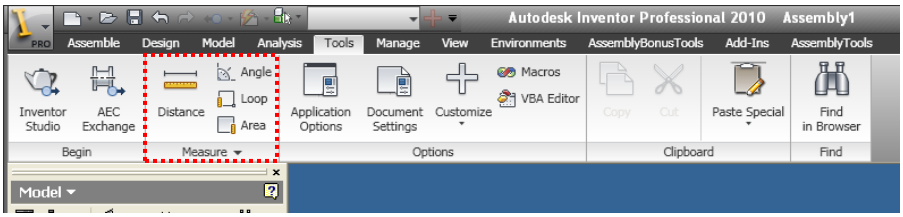
### Ribbon Tab

Each ribbon consists of a set of tabs. The Tools tab of the Assembly ribbon is highlighted below. These are represented in the API by the RibbonTab object. There can be any number of tabs, although there's a practical limit to what will fit on the screen. Tabs can be visible or not. Each tab also has an internal name that can be used to find it.



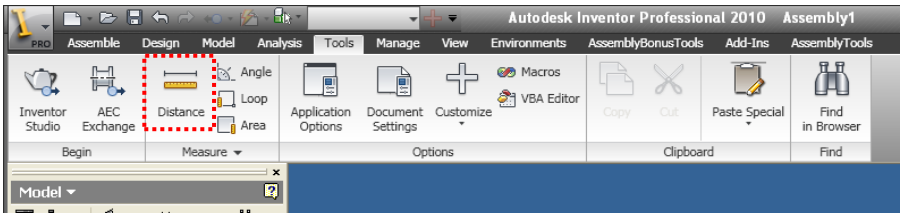

### Ribbon Panel

Each tab consists of a set of panels. The Measure panel of the Tools tab is highlighted below. These are represented in the API by the RibbonPanel object. There can be any number of panels in a tab, again with a practical limit, and they can be visible or not. Each panel also has an internal name that can be used to find it.




### Command Control

Each tab contains a set of controls. The button control for the Distance command is highlighted below. All controls are represented in the API by the CommandControl object. Each control also has an internal name that can be used to find it; however their internal name isn't explicitly defined but is inherited from the control definition it references.




## Determining Where to Add Your Buttons

The first step in supporting the ribbon is deciding where you want to insert your add-in is buttons into the ribbon. My suggestion is to try and put yourself in the position of the user of your add-in and imagine where they will most likely look for your commands. For example, if your add-in has a command that draws a custom slot shape in a sketch you would probably want to insert your button on the Draw panel of the Sketch tab of the Part, Assembly, and Drawing ribbons. If your command is unique from the rest of Inventor's commands and doesn't really fit in you might want to create a new tab or a panel to logically separate it from the rest of Inventor's commands. In addition to the ribbon you can also add your commands to the Application and Help menus and the QAT. The main objective is to have your button located where users will intuitively look for it.

Another factor that you need to consider when positioning your commands is the overall layout of the ribbon once your command is added. The reality is that there is a limited width to the screen and each button takes some space. Some ribbons are fuller than others so if logically your button could go in more than one location you might want to choose the one with the most room available.

## Getting the Internal Names of the Ribbon Components

It was previously mentioned that the various components of the ribbon have internal names that can be used to find specific ribbon components. You may have been asking yourself; how do you know what the internal names are? The program below is the best answer to that question. It allows you to get an up to date listing of the names of all the components in the current configuration of the ribbon.

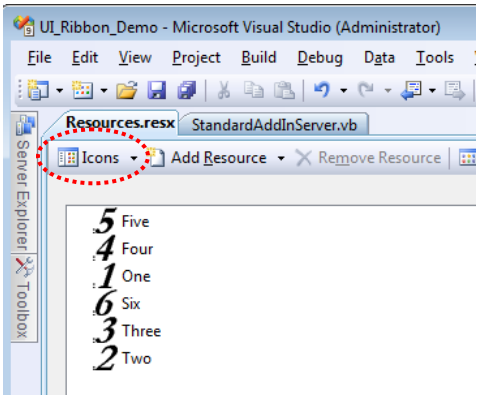
```
Public Sub PrintRibbon()
    Open "C:\temp\RibbonNames.txt" For Output As #1

    Dim oUIManager As UserInterfaceManager
    Set oUIManager = ThisApplication.UserInterfaceManager

    Print #1, "File Controls (Application Menu)"
    Call PrintControls(oUIManager.FileBrowserControls, "", 1)
    Print #1, "----------------------------------------------------------------------"

    Print #1, "Help Controls"
    Call PrintControls(oUIManager.HelpControls, "", 1)
    Print #1, "----------------------------------------------------------------------"

    Dim oRibbon As Ribbon
    For Each oRibbon In oUIManager.Ribbons
        Print #1, "Ribbon: " & oRibbon.InternalName

        Print #1, "    QAT controls"
        Call PrintControls(oRibbon.QuickAccessControls, "            ", 0)

        Dim oTab As RibbonTab
        For Each oTab In oRibbon.RibbonTabs
            Print #1, "    Tab: " & oTab.DisplayName & ", " & oTab.InternalName & _
                      ", Visible: " & oTab.Visible

            Dim oPanel As RibbonPanel
            For Each oPanel In oTab.RibbonPanels
                Print #1, "        Panel: " & oPanel.DisplayName & ", " & _
                           oPanel.InternalName & ", Visible: " & oPanel.Visible

                Call PrintControls(oPanel.CommandControls, "            ", 0)

                If oPanel.SlideoutControls.Count > 0 Then
                    Print #1, "            --- Slideout Controls ---"
                    Call PrintControls(oPanel.SlideoutControls, "            ", 0)
                End If
            Next
        Next

    Print #1, "----------------------------------------------------------------------"
    Next
    On Error GoTo 0

    Close #1

    MsgBox "Result written to: C:\temp\RibbonNames.txt"
End Sub
```

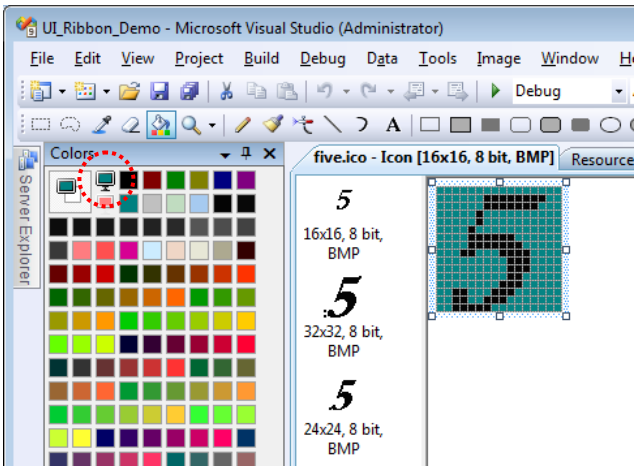
```
Private Sub PrintControls(Controls As CommandControls, _
                          LeadingSpace As String, _
                          Level As Integer)
    Dim oControl As CommandControl
    For Each oControl In Controls
        If oControl.ControlType = kSeparatorControl Then
            Print #1, LeadingSpace & Space(Level * 4) & "Control: Seperator"
        Else
            Print #1, LeadingSpace & Space(Level * 4) & "Control: " & _
                      oControl.DisplayName & ", " & oControl.InternalName & _
                      ", Visible: " & oControl.Visible

            If Not oControl.ChildControls Is Nothing Then
                Call PrintControls(oControl.ChildControls, LeadingSpace, Level + 1)
            End If
        End If
    Next
End Sub
```

## Creating Your Icons

Just like with the classic interface, the ribbon supports two sizes of icons. The small icons in the ribbon are the same as in the classic, 16x16 pixels. However the size of the large icons for the ribbon is 32x32, whereas for classic it was 24x24. There are a lot of options for creating your icons. I would recommend using .ico files for your icons. I prefer icons because you can specify a transparent background and you can have multiple sizes within a single file. For example, I can have 16x16, 24x24, and 32x32 icons all defined within a single .ico file. You can create and edit icons using Visual Studio (but not the Express edition) and there are other editors (some of them free) available on the internet.

Although not required, I would also recommend adding the icons as resources to your project. This simplifies maintenance and delivery of your add-in. The pictures below show the resources for the demonstration add-in. On the left, the “Icons” resource type has been selected and six different icons are displayed. On the right, one of the icons has been selected for edit and it shows that there are three different sizes for that icon.




To create icons of the correct sizes in Visual Studio, use the **New Image Type** command that’s available in the context menu. There are existing sizes for the 16x16 and 32x32 icons but you will need to create a custom size for the 24x24. You should choose 24 bit color since Inventor supports it. Using the color that’s circled in the picture to the right you can define a transparent background.

## Don't Forget the Classic Interface

Inventor gives the user the choice of using the classic or ribbon interface. This is good for those users that don't like the ribbon, but it does complicate writing add-ins because your add-in will need to support both the classic and ribbon interfaces. The easiest way to support both is to do what was described earlier and only write code to support the classic interface and then rely on Inventor to automatically add it to the ribbon. However as was already discussed, this will usually not provide the best result. This paper discusses what you need to do to fully support both interfaces.

## Adding Ribbon Support to Your Add-In

Now we can start looking at the code changes needed to support the ribbon. In both the ribbon and classic interfaces, the first thing you need to do is create the control definitions. Control definitions remain unchanged when programming for the ribbon. The only change you'll need to make in that section of code is to make sure you use the 32x32 pixel icon for the large size instead of the 24x24 pixel icon.

Below is the top portion of the Activate method of the sample add-in. The code that's specific to the ribbon is highlighted. This is where it uses the new InterfaceStyle property to check if Inventor is using the ribbon or classic interface. Based on the return value it sets a variable for the icon size to be 32 or 24 indicating the size of a large icon (32x32 for ribbon or 24x24 for classic).

```
Public Sub Activate(...) Implements Inventor.ApplicationAddInServer.Activate
    ' Initialize AddIn members.
    m_inventorApplication = addInSiteObject.Application

    ' Get the ClassID for this add-in and save it in a
    ' member variable to use wherever a ClientID is needed.
    m_ClientID = AddInGuid(GetType(StandardAddInServer))

    ' Get a reference to the UserInterfaceManager object.
    Dim UIManager As Inventor.UserInterfaceManager = _
                                        m_inventorApplication.UserInterfaceManager

    ' Set the large icon size based on whether it is the classic or ribbon interface.
    Dim largeIconSize As Integer
    If UIManager.InterfaceStyle = InterfaceStyleEnum.kRibbonInterface Then
        largeIconSize = 32
    Else
        largeIconSize = 24
    End If
```

The next step is to create the control definitions. The code below illustrates doing this for the first control definition in the sample program. The two statements before the AddButtonDefinition call are creating IPictureDisp objects from the icons. The first statement gets an IPictureDisp for the small icon by specifying the resource named "One" and getting the icon whose size is 16 by 16 pixels. The second statement also specifies the resource named "One" but uses the variable that was set earlier to indicate which size icon to use. The AddButtonDefinition call is unchanged from previous versions of Inventor. The Add-In goes through this same process for the rest of the buttons.

```
Dim controlDefs As ControlDefinitions = _
                                m_inventorApplication.CommandManager.ControlDefinitions

Dim smallPicture As stdole.IPictureDisp = _
            Microsoft.VisualBasic.Compatibility.VB6.IconToIPicture( _
            New System.Drawing.Icon(My.Resources.One, 16, 16))

Dim largePicture As stdole.IPictureDisp = _
            Microsoft.VisualBasic.Compatibility.VB6.IconToIPicture( _
            New System.Drawing.Icon(My.Resources.One, largeIconSize, largeIconSize))

m_button1Def = controlDefs.AddButtonDefinition("One", "UIRibbonSampleOne", _
                                            CommandTypesEnum.kNonShapeEditCmdType, _
                                            m_ClientID, , , smallPicture, largePicture)
```

Just to illustrate other options, the code below demonstrates reading an image from a file on disk instead of using a resource. A .png image file is used because it allows you to specify a transparent background.

```
largePicture = Microsoft.VisualBasic.Compatibility.VB6.ImageToIPicture( _
                                    System.Drawing.Image.FromFile("C:\Temp\test.png"))
```

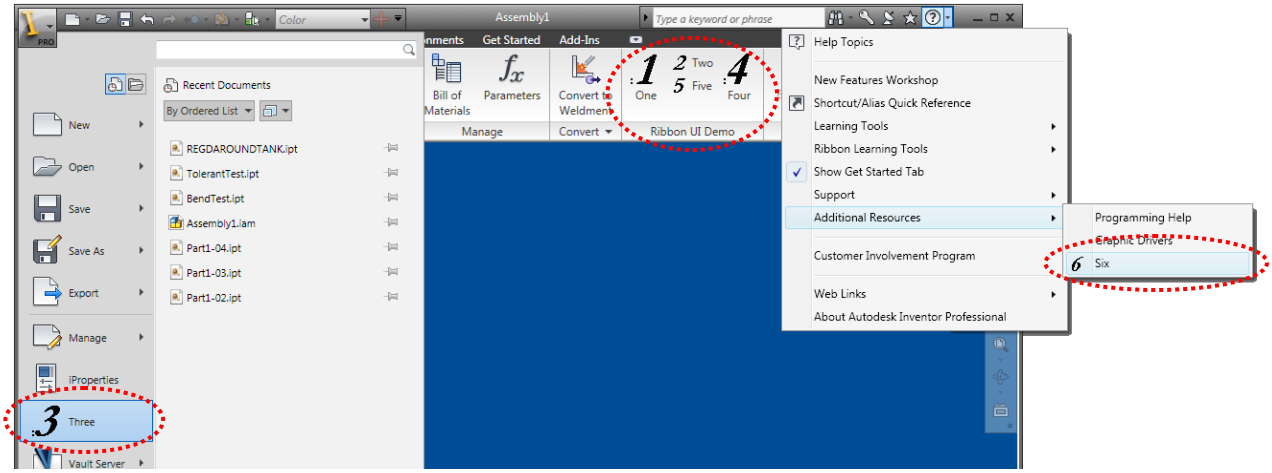
### Adding Your Commands to the User Interface

The next step is to create your controls within the user interface. The section of code that does this depends on the firstTime flag that is passed into the Activate method. Remember that this was done in the classic interface because Inventor remembers the controls you've created from one session to another so you only need to create them once. This is also the intention for the ribbon interface but it doesn't currently behave that way and doesn't remember the changes your add-in makes to the ribbon. What currently happens with the ribbon is that every time you add-in starts the first time flag will always have a value of True. Because the customization that the user is allowed to do with the ribbon is very limited this doesn't end up causing any problems.

Below is the portion of code from the Activate method of the sample add-in that checks the firstTime value and creates the appropriate user-interface. The sample has one function for creating the ribbon interface and another for classic. The code that creates the classic interface is unchanged from how you would have created the interface in Inventor 2009 before the ribbon interface existed. The interesting function for us is the CreateOrUpdateRibbonUserInterface function.

```
If firstTime Then
    If UIManager.InterfaceStyle = InterfaceStyleEnum.kRibbonInterface Then
        CreateOrUpdateRibbonUserInterface()
    ElseIf UIManager.InterfaceStyle = InterfaceStyleEnum.kClassicInterface Then
        CreateClassicInterface()
    End If
End If
```

Before looking at the code that handles the ribbon interface, let's look at the result of the code. It creates the six buttons as shown in the picture below; with four of them being in the assembly ribbon, one in the Application menu and one in the Help menu.




Here's the code that does the work. Because of the structure of the ribbon, the code is fairly simple. The first thing it does it get the assembly ribbon. It does this by getting the ribbon named "Assembly". It then gets the Assemble tab on the ribbon. This example then creates a new pane on the Assemble tab called "Ribbon UI Demo". It could have gotten an existing tab instead but this sample creates one. Finally, on the new panel it adds the four buttons.

```
Private Sub CreateOrUpdateRibbonUserInterface()
    ' Get a reference to the UserInterfaceManager object.
    Dim UIManager As Inventor.UserInterfaceManager
    UIManager = m_inventorApplication.UserInterfaceManager

    ' Get the assembly ribbon.
    Dim assemblyRibbon As Inventor.Ribbon = UIManager.Ribbons.Item("Assembly")

    ' Get the "Assemble" tab.
    Dim assembleTab As Inventor.RibbonTab
    assembleTab = assemblyRibbon.RibbonTabs.Item("id_TabAssemble")

    ' Create a new panel on the Assemble tab.
    Dim panel As Inventor.RibbonPanel
    panel = assembleTab.RibbonPanels.Add("Ribbon UI Demo", _
                                         "RibbonUIDemoPanel", m_ClientID)

    ' Add the buttons to the tab, with the first and last ones
    ' being large and the other two small.
    panel.CommandControls.AddButton(m_button1Def, True)
    panel.CommandControls.AddButton(m_button2Def)
    panel.CommandControls.AddButton(m_button5Def)
    panel.CommandControls.AddButton(m_button4Def, True)
```

The last portion of the CreateOrUpdateRibbonUserInterface function adds the commands to the Application and Help menus. The UserInterfaceManager supports the FileBrowserControls property that returns a CommandControls collection which contains all of the controls in the Application menu. It inserts button 3 immediately after the command whose name is "AppiPropertiesWrapperCmd", which is

the iProperties command. It then uses the HelpControls property to get the list of controls in the help menu and then gets the list of controls in the "Additional Resources" pop-up and adds button 6 to that list.

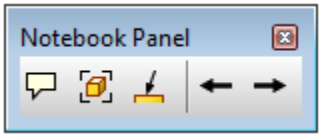
```
    ' Add the command to the File controls.
    Dim fileControls As CommandControls = UIManager.FileBrowserControls
    fileControls.AddButton(m_button3Def, , , "AppiPropertiesWrapperCmd")


    ' Add a button to the Help menu within the "Additional Resources" pop-up.
    Dim additionalResourcePopUp As CommandControl
    additionalResourcePopUp = UIManager.HelpControls.Item("Additional Resources")
    additionalResourcePopUp.ChildControls.AddButton(m_button6Def)
End Sub
```

## Extras

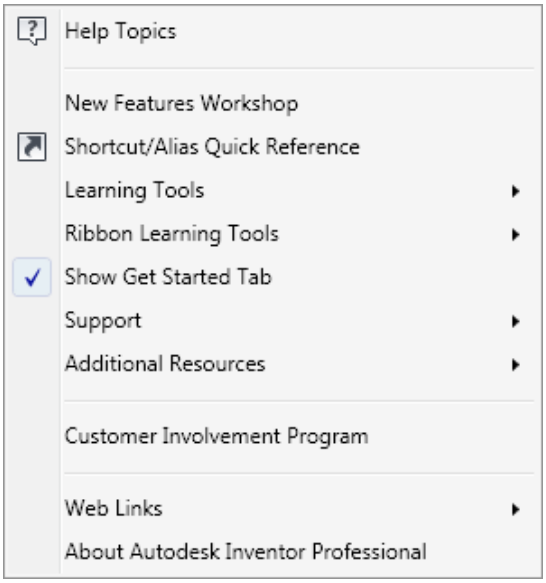
What's covered above can be considered the minimum needed for most add-ins to support the ribbon. Here are some other features of the ribbon that are also good to know about.

### Seperators

Separators are used to provide a visual divider between commands to provide a visual grouping of similar commands. These were used frequently in the classic interface. The toolbar below is an example where there's a separator between the third and fourth icons. The API supported defining separators by using the GroupBegins property of the CommandBarControl object. Setting this property to True resulted in a separator appearing immediately before that control (it was the beginning of the next group of commands).




Separators are also supported by the ribbon although they're not as common since the panels serve the same purpose of grouping commands. They're most common in the Application and Help menus. The picture below shows the help menu in the ribbon with its three separators.



Separators in the ribbon are handled differently in the API than they were in the classic interface. Instead of the separator being a property of one of the buttons it is now a separate control. To create a separator you use the AddSeparator method of the CommandControls object. It doesn't have an associated control definition.

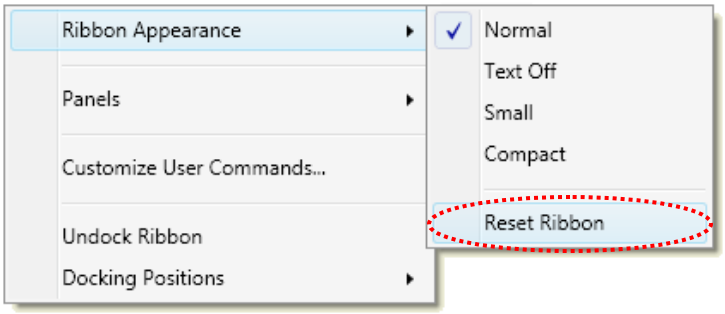
### Context Menu

There's not much to say regarding the context menu other than nothing has changed for it as a result of the ribbon interface. It still uses a CommandBar object to define its contents.

### Handling Resets in the Ribbon Interface

In both the classic and ribbon interfaces, the user can reset the interface to restore it to its original state. This has the side effect of removing any customization that add-ins have done. Because of the way add-ins work they tend to feel like standard Inventor functionality to the user. Because of this most users expect a reset to take Inventor back to its initial interface plus any customization that add-ins have done to add their buttons. To accomplish this, an add-in needs to react to a reset by recreating whatever customization it did. An add-in does this by listening to events. This is described below for both the ribbon and classic interfaces.

In the ribbon interface the user can use the **Reset Ribbon** command to reset the entire ribbon interface back to its initial state. The **Reset Ribbon** command is invoked through the ribbon's context menu as shown below.



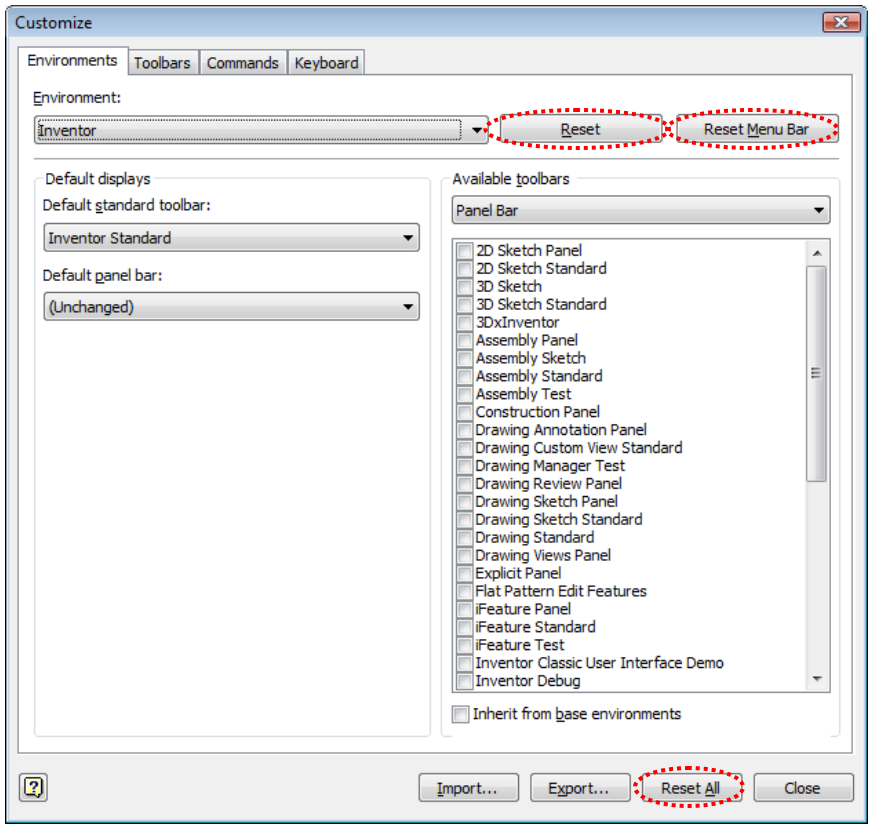

For an add-in to handle the **Reset Ribbon** command it needs to listen to the OnResetRibbonInterface event. All you need to do in response to this event is exactly what you did when the add-in was executed for the first time. The code below illustrates this by calling the same function that was called in the Activate method of the add-in.

```
Private Sub m_userInterfaceEvents_OnResetRibbonInterface(...)
    CreateOrUpdateRibbonUserInterface()
End Sub
```

### Handling Resets in the Classic Interface

Unfortunately, handling resets in the classic interface isn't as simple as it is for the ribbon. Even though all well behaved add-ins should react to an interface reset, most of the add-ins that have been created do not. Surprisingly this has not been a big issue. I think this is most likely because the reset commands are rarely used. You can decide if you want to do the extra work to support a reset. Because it's so simple to support reset for the ribbon there's no good reason not to. However, it is a fair amount of work in the classic interface. Because of the amount of work and the realistic likelihood that the code would never be used I would consider handling reset for the classic interface as optional. Below is a description of what you need to do for the classic interface.

The thing that makes it harder in the classic interface is that there are different types of resets. Through the Customize dialog the user can reset individual toolbars, individual environments, or everything (all toolbars and all environments). This added flexibility for the user means you have to handle more cases. The Environments tab of the Customize dialog is shown below. Notice the three reset buttons that are highlighted. The Reset button will reset the selected environment. The Reset Menu Bar button will reset the toolbar that represents the menu for the selected environment, so it's really just a toolbar reset. The Reset All button will reset all environments and all toolbars. On the Toolbars tab the user can reset individual toolbars.




To support these various types of resets, Inventor provides two events; OnResetCommandBars and OnResetEnvironments. A collection of the command bars or environments that have been reset is provided through the event. The collection may only have one command bar or environment in it in the case of a single reset, or it might have all of the environments and toolbars in the case of a Reset All. Below is the code from the sample add-in for the OnResetCommandBars event.

```
Private Sub m_userInterfaceEvents_OnResetCommandBars( _
                                ByVal CommandBars As Inventor.ObjectsEnumerator, _
                                ByVal Context As Inventor.NameValueMap) _
                                   Handles m_userInterfaceEvents.OnResetCommandBars

   ' Get the assembly environment.
   Dim UIManager As Inventor.UserInterfaceManager
   UIManager = m_inventorApplication.UserInterfaceManager
   Dim asmEnvironment As Environment
   asmEnvironment = UIManager.Environments.Item("AMxAssemblyEnvironment")
```

```
   For Each commandBar As Inventor.CommandBar In CommandBars
      ' Special case for each command bar.
      If commandBar Is asmEnvironment.PanelBar.DefaultCommandBar Then
         commandBar.Controls.AddButton(m_button1Def)
      End If

      If commandBar Is asmEnvironment.DefaultToolBar Then
         commandBar.Controls.AddButton(m_button2Def)
      End If

      If commandBar Is _
      asmEnvironment.DefaultMenuBar.Controls.Item("AssemblyFileMenu").CommandBar Then
         ' Add a button to the File menu right after the Close button.
         commandBar.Controls.AddButton(m_button3Def, _
                                commandBar.Controls.Item("AppFileCloseCmd").Index + 1)
      End If

      Dim helpBar As CommandBar
      helpBar = asmEnvironment.DefaultMenuBar.Controls.Item("AppHelpMenu").CommandBar
      If commandBar Is _
      helpBar.Controls.Item("AppHelpAdditionalResourcesMenu").CommandBar Then
         ' Add a button to the Help menu at the bottom.
         Dim additionalResourceButtonBar As CommandBar
         additionalResourceButtonBar = _
                    helpBar.Controls.Item("AppHelpAdditionalResourcesMenu").CommandBar
         additionalResourceButtonBar.Controls.AddButton(m_button4Def)
      End If

      If commandBar.InternalName = "UISampleRegular" Then
         ' Add two buttons to it.
         commandBar.Controls.AddButton(m_button5Def)
         commandBar.Controls.AddButton(m_button6Def)

         ' Make the command bar visible.
         commandBar.Visible = True
      End If
   Next
End Sub
```

Below is the code to from the sample add-in that handles resetting environments.

```
Private Sub m_userInterfaceEvents_OnResetEnvironments( _
                                 ByVal Environments As Inventor.ObjectsEnumerator, _
                                 ByVal Context As Inventor.NameValueMap) _
                                    Handles m_userInterfaceEvents.OnResetEnvironments

   Dim UIManager As Inventor.UserInterfaceManager
   UIManager = m_inventorApplication.UserInterfaceManager
   Dim asmEnvironment As Environment
   asmEnvironment = UIManager.Environments.Item("AMxAssemblyEnvironment")

   Dim commandBar As CommandBar
   Try
      commandBar = UIManager.CommandBars.Item("UISampleRegular")
   Catch ex As Exception
      commandBar = UIManager.CommandBars.Add("Inventor Classic User Interface Demo", _
                  "UISampleRegular", CommandBarTypeEnum.kRegularCommandBar, m_ClientID)
   End Try

   ' Add the new command bar to the panel bar list.
   asmEnvironment.PanelBar.CommandBarList.Add(commandBar)
End Sub
```

Finally, here's the code that is called from the Activate method when the first time flag is True, to initially create the add-in's interface within the classic interface.

```
Private Sub CreateClassicInterface()
    ' Get the assembly environment.
    Dim UIManager As Inventor.UserInterfaceManager
    Set UIManager = m_inventorApplication.UserInterfaceManager
    Dim asmEnv As Environment
    asmEnv = UIManager.Environments.Item("AMxAssemblyEnvironment")

    ' Add a button to the assembly panel bar.
    Dim asmPanel As Inventor.CommandBar = asmEnvironment.PanelBar.DefaultCommandBar
    asmPanel.Controls.AddButton(m_button1Def)

    ' Add a button to the main tool bar of the assembly environment.
    asmEnvironment.DefaultToolBar.Controls.AddButton(m_button2Def)

    ' Add a button to the File menu right after the Close button.
    Dim fileBar As CommandBar
    Set filebar = asmEnv.DefaultMenuBar.Controls.Item("AssemblyFileMenu").CommandBar
    fileBar.Controls.AddButton(m_button3Def, _
                                   fileBar.Controls.Item("AppFileCloseCmd").Index + 1)

    ' Add a button to the Help menu at the bottom.
    Dim helpBar As CommandBar
    Set helpBar =  asmEnv.DefaultMenuBar.Controls.Item("AppHelpMenu").CommandBar
    Dim additionalResourceButtonBar As CommandBar
    Set additionalResourceButtonBar = _
                    helpBar.Controls.Item("AppHelpAdditionalResourcesMenu").CommandBar
    additionalResourceButtonBar.Controls.AddButton(m_button4Def)

    Dim commandBar As CommandBar = Nothing

    ' Need to check first to see if the command bar already exists.  If the user
    ' does a reset the command bar isn't deleted.
    Try
        commandBar = UIManager.CommandBars.Item("UISampleRegular")
    Catch ex As Exception
    End Try

    If commandBar Is Nothing Then
        ' The command bar doesn't exist, so create it.
        commandBar = UIManager.CommandBars.Add( _
                         "Inventor Classic User Interface Demo", "UISampleRegular", _
                         CommandBarTypeEnum.kRegularCommandBar, m_ClientID)
    End If

    ' Add two buttons to it.
    commandBar.Controls.AddButton(m_button5Def)
    commandBar.Controls.AddButton(m_button6Def)

    ' Make the command bar visible.
    commandBar.Visible = True

    ' Add the new command bar to the panel bar list.
    asmEnvironment.PanelBar.CommandBarList.Add(commandBar)
End Sub
```