How to Customize the MapInfo Pro Ribbon Interface

MapInfo Pro x64 (MIPro) introduced a new User Interface (UX) based on the Microsoft Ribbon library replacing the Win32 Menu and ButtonPad interface. While many of the original MapBasic commands and functions used in the older interface can still run and appear in a Legacy Ribbon tab group, most users will want to get familiar with the new Ribbon interface and start building their customized ribbon controls using either .NET or MapBasic. This document will help describe how to customize the new Ribbon UX in MapBasic only.

There are several sample applications provided with the MapBasic download to help get users started, and there is also an excellent Ribbon Library for MapBasic available from the Community Downloads page.

Components

The Ribbon UX has several major components that will be broken down in sections below starting from the top-level IMapInfoPro interface, then breaking out into its dependent interfaces to help explain how these components work in the new Ribbon. Below is a quick list of major components provided as an overview that will be described in more detail under those sections.

IMapInfoPro – The MapInfo Application Interface returned as an instance from call to SystemInfo(SYS_INFO_IMAPINFOAPPLICATION)

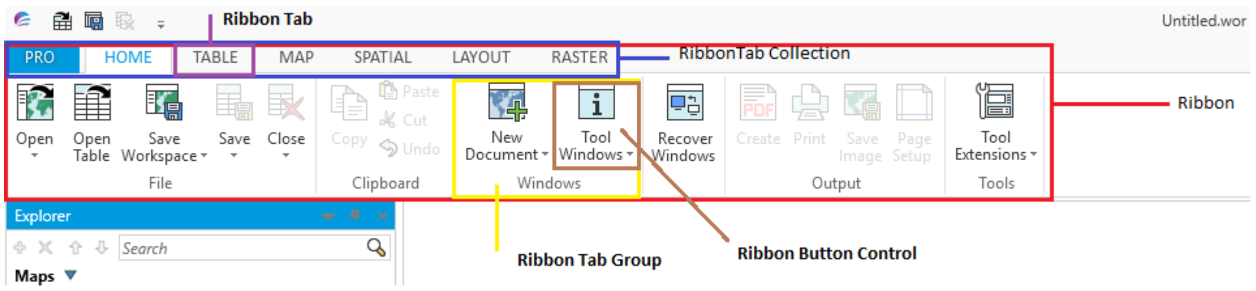
Ribbon: The Main Ribbon Interface from which all Ribbon controls inherit.

Ribbon Tab Collection: Collection of Ribbon Tabs for grouping related control groups under the Ribbon (e.g.: Home, Table, Map, Spatial, etc.).

Ribbon Tab: Individual member of Ribbon Tab Collection.

Ribbon Tab Group: Group of Ribbon controls under a specific Ribbon Tab that are typically related tasks.

Ribbon Button Control: the actual control (e.g.: RibbonButton, Ribbon Tool Button, Ribbon Split Button, Menu, or Drop Down controls, etc.) that reside within the Ribbon Tab Group.



Other controls that are part of the Ribbon interface such as Ribbon Button Panels, Ribbon Button Checkboxes, Ribbon Gallery Controls, Ribbon Contextual Tabs, Ribbon Menu Items, Stack Controls, and others that will be addressed later in this document. Getting an understanding of programming these basic Ribbon controls in MapBasic should make further customizations to the ribbon interface much easier to understand.

New Variable Types

MapInfo Pro introduced some new variable types in recent versions to allow better integration with .NET assemblies and extending MapBasic to take advantage of them. The first of these types is **This**, which allows users to access an instance of a .NET object. You can use **This** to hold a .NET reference type and call methods on it. MIPro treats it as a simple integer value stored in 4 bytes. The new Extensibility methods for MIPro's Ribbon UX require use of **This** for many of its operations.

Another new variable type, **IntPtr**, is a platform specific type to represent a pointer or a handle. This helps to write applications that will work on both win32 & x64 versions of MIPro. In MIPro x64 this is treated as a LargeInt and in Win32 MIPro it is treated as an Integer. There is the **RefPtr** variable type that represents a reference to a .NET object. This holds a .NET reference type and can pass it to a method in .NET code. MIPro treats it as a simple integer value stored in 4 bytes.

Naming Conventions and Syntax for Interface Methods and Properties.

This document may refer to interfaces by a common name, but each section will have the Interface name users can find within the Extensibility Reference provided with MapBasic Help to get more information. Interface names are always preceded by a capital 'I' to identify them as such (e.g.: IRibbon, IRibbonTabCollection, etc.). In the Extensibility Reference, users can expand MapInfo.Types under the Contents tab and these Interfaces will be listed in alphabetical order. Within IMapInfoPro.def file, the declared methods are grouped in sections for each Interface (e.g.: MapInfo.Types.IRibbon). For each interface in the Extensibility Reference, users can click on either a specific method or property and there will be a syntax tab showing C#, VB, or MapBasic syntax for this method or property. The MapBasic method name and parameters found in IMapInfoPro.def will be under this MapBasic tab.

In order to access any Ribbon UX interfaces within a MapBasic program, one must include IMapInfoPro.def in the header of the MapBasic program file (*.mb). This *.def file contains a few hundred methods that can be used to customize the Ribbon interface. The number of methods in IMapInfoPro.def file appears overwhelming, however, many of these property methods for lower-level controls (e.g.: GetRbnBtnCtrlName, that gets the control name for a RibbonButton) are actually inherited from a higher level interface called IMapInfoControl, and users can call the shorter named property method of GetMICtrlName() for the same RibbonButton. Longer and more concise method names are provided so users can write their MapBasic programs with more descriptive methods of what control is being acted upon in their program.

***NOTE: There has always been a 32-character limit for method names in MapBasic despite some of the method names in IMapInfoPro.def exceeding this limit; however, the first 32 characters are unique so it does not cause compilation or runtime errors when these extra characters are clipped during compilation.

There is also an Enums.def file included with MapBasic that contains enumerated defines for control types, window types, events, and control properties. MapBasic programmers must either provide the numeric constant, or define name, when using many of these methods and code is a lot more readable when using defines to describe what property or control, etc., is being used within these methods.

Example creating a new Split Button control:

```
Dim SplitButton1 as This
SplitButton1 = MICtrlCollAddStrStrInt(groupControlColl, "SplitBtn", "DoStuff",
ControlType_SplitButton)
```

Versus

```
SplitButton1 = MICtrlCollAddStrStrInt(groupControlColl, "SplitBtn", "DoStuff", 10)
```

The method used above will be explained later, but readers of this code will know a lot quicker that a new Split Button control is being added to a RibbonControlCollection without having to know what the enumerated type of 10 means. Control Properties such as alignment, width, height, and orientation, etc., are also a lot more recognizable using defines (e.g.: Orient_Vertical) rather than their numeric constants.

MapBasic.def should also be included and it's defines used for more readable code as well. It is a lot easier to understand a call to get the character set of a table with it's define like TableInfo (<*tablename*>, TAB_INFO_CHARSET) than if using the numeric constant like TableInfo (<*tablename*>, 48). If calling any predefined MapInfo command IDs such as 102 for opening a table from a ribbon control, users should also consider including Menu.def and use the define M_FILE_OPEN instead. For built-in cursors and DrawModes used with Ribbon Tool Buttons, these constants can be accessed by including Icons.def.

If you want to use icons on controls within the Ribbon UX users must create an instance of a GenericUriParser and register this instance with the RegisterUriParser method. This is accomplished with a call to RegisterUriParser() like below. It's use will be explained in the section for **Ribbon Controls**. This .NET System function is used for getting resources out of .NET assemblies.

```
Call RegisterUriParser(New_GenericUriParser(1), "pack", -1)
```

MapInfo Pro Application (IMapInfoPro)

Before any customization of the Ribbon UX can occur, an instance of the MapInfoPro (MIPro) application itself must be obtained in MapBasic with a call to SystemInfo(SYS_INFO_IMAPINFOAPPLICATION) or 20. The variable type returned here is **This**, a .NET specific variable, and refers to an instance of IMapInfoPro, the Top-Level interface for controlling MIPro through add-ins (e.g. Ribbon Customization apps). This method is only for use with x64 versions of MIPro. There are several other interfaces that are accessible from IMapInfoPro, other than the Ribbon, which control other parts of the MIPro application such as the Backstage, Quick Access Tool Bar, Status Bar, Docking Manager, etc. This document will focus on the Ribbon interface first to get users familiar with Extensibility methods to customize the MIPro Ribbon UX. Other Interfaces will be explained afterward.

'Get an instance of the MapInfoPro Application Interface. This should be at least a module level variable in most cases.

```
Dim mapinfoApplication as This
mapinfoApplication = SystemInfo(SYS_INFO_IMAPINFOAPPLICATION)  'or 20
```

Ribbon (IRibbon)

In order to begin customizing the Ribbon interface, users must get a Ribbon (IRibbon) instance from the IMapInfoPro Interface in order to access the main ribbon inside MIPro. This is done through a call to GetRibbon() method where an instance of IMapInfoPro is passed as its only parameter. See IMapInfoPro.Ribbon Property topic for more information.

```
Dim Ribbon as This
Ribbon = GetRibbon(mapinfoApplication)
```

A Ribbon interface (IRibbon) has several properties that are available, most importantly the IRibbonTabCollection interface, which is the collection of Ribbon Tabs such as Home, Table, Map, Spatial, etc., seen in the basic MIPro Ribbon UX. To get access to the Ribbon Tabs users must first get a collection of these Ribbon Tabs by calling the method GetTabsColl() and passing the Ribbon instance obtained earlier. See IRibbon.Tabs Property for more information.

```
Dim RibbonTabColl as This
RibbonTabColl = GetTabsColl(Ribbon)
```

After accessing the RibbonTabCollection, users should do something with this collection by either Adding, Removing, or Modifying Ribbon Tabs within it. Ribbon customization typically uses multiple Get, Set, or Create (aka: Constructor) methods that are very similar in their usage throughout the Extensibility interface for MIPro. The usage pattern should become readily apparent as these methods are described in this guide. Often the very first parameter in a given method is an Interface (IRibbon, IRibbonTab, etc.) that has to be obtained in order to act on it with Get or Set methods. When debugging Ribbon UX customization programs, one of the very first things to check is whether an Interface is Null before calling a subsequent method that uses it as its first parameter. If a Constructor or Get… method did not create or return a valid instance of an Interface object, then trying to use it in a subsequent call will cause an exception to be thrown. Instances of these NET objects can also be set to a NULL_PTR within MapBasic (e.g.: within an EndHandler procedure), so make sure these objects are not null (in MapBasic, a NULL_PTR equals 0) before attempting to use them. MapBasic language does not support true null values, however for the Extensibility interfaces a NULL_PTR type was implemented to allow setting interfaces to null in order to dispose them correctly.

Ribbon Tabs (IRibbonTab)

After getting an instance of the RibbonTabCollection there are methods for adding or inserting RibbonTabs into, or removing them from, the RibbonTabCollection. In this example, a New Ribbon Tab will be added using only string parameters. The first parameter for this method is the instance of RibbonTabCollection that will receive the added Ribbon Tab, followed by a Tab Name, and an optional Caption property for the Tab.

```
Dim RibbonTab as This
RibbonTab = RbnTabCollAddStrStr (RibbonTabColl, "MBAddIn", "MB Tab")
```

***Note the naming convention of these methods in IMapInfoPro.def file. The method names typically contain the parameter types in order of their use within the method name.

There are other methods where you can add a tab to a RibbonTabCollection using a RefPtr from a previously created IRibbonTab. Users can also Insert a newly created RibbonTab into the collection at a specified index in the RibbonTabCollection like below.

```
'Create a new RibbonTab instance to use as a RefPtr.  This is a Constructor method to create an
IRibbonTab interface.  See MapBasic Tab under IRibbonTabCollection.Create Method.
RibbonTab = RbnTabCollCreate(RibbonTabColl,"MyTab","TestTab")
'Add it to end of the RibbonTabCollection as a RefPtr
Call RbnTabCollAddRefPtr(RibbonTabColl, RibbonTab)
'Or Insert it into RibbonTabCollection in position 1 of this Ribbon Tab.
Call RbnTabCollInsertIntRefPtr(RibbonTabColl, 1, RibbonTab)
```

Users can also call RbnTabCollRemove or RbnTabCollRemoveAt methods to remove a RibbonTab from the RibbonTabCollection. If Removing a Ribbon tab from a RibbonTabCollection at a specific index, then get its index inside the collection first.

```
Dim iRet as integer
iRet = RbnTabCollIndexOf(RibbonTabColl, RibbonTab)
'then use returned index value to remove the RibbonTab from the collection.
Call RbnTabCollRemoveAt (RibbonTabColl, iRet)
```

If you have a RefPtr to a known RibbonTab, then just remove it using RbnTabCollRemove() and check logical return value which is TRUE if successful.

```
Dim bRet as logical
bRet = RbnTabCollRemove(RibbonTabColl, RibbonTab)
```

There are other property methods to modify IRibbonTab type objects such as adding KeyTips, ToolTips, selected state, caption, etc. See IRibbonTab Interface in Extensibility reference for more information.

Ribbon Control Groups (IRibbonControlGroup)

Once a specific RibbonTab has been obtained or created, users should then get or create RibbonControlGroups (IRibbonControlGroup) for that Tab. Under the Home Tab in MIPro's Ribbon, there are groups such as File, Clipboard, Windows, Output, and Tools. These RibbonControlGroups help organize related controls under each Ribbon Tab. In order to add, remove, or modify a RibbonControlGroup under a Ribbon Tab, users first need to get the RibbonControlGroupCollection (IRibbonControlGroupCollection) by calling the method GetRbnTabGrps() and passing the instance of a specific RibbonTab.

```
Dim ribbonGroupsColl as This
ribbonGroupsColl = GetRbnTabGrps(RibbonTab)
```

Just like RibbonTabCollections, users can add or remove RibbonControlGroups to a RibbonTab using similar methods. Here a new RibbonControlGroup named 'MBCtrlGroup' with a caption of 'Addin' will be added to the RibbonTab created earlier and placed at the end of the collection.

```
Dim ribbonGroup as This
```

```
ribbonGroup = RbnCtrlGrpCollAddStrStr(ribbonGroupsColl, "MBCtrlGroup", "Addin")
```

RibbonControlGroups can also be inserted at specific indexes within the RibbonTabCollections as well. Here a new RibbonControlGroup is inserted in first position of control groups under this RibbonTab.

```
ribbonGroup =RbnCtrlGrpCollInsertIntStrStr(ribbonGroupsColl, 1,"MBCtrlGroup", "Addin")
```

RibbonControlGroups can also be removed by index, or by passing a RefPtr of an already existing IRibbonControlGroup and then checking the logical return value from the method where TRUE means it was successful.

```
bRet = RbnCtrlGrpCollRemove(ribbonGroupsColl, ribbonGroup)
```

A Ribbon Control Group can also be removed using a specific index, once that index of a RibbonControlGroup in the collection is known. Get the ribbonGroup index value (iRet) from ribbonGroupsColl then pass this integer index value to method RbnCtrlGrpCollRemoveAt() and check the logical return value for TRUE if the group was removed successfully from collection like below.

```
iRet = RbnCtrlGrpCollIndexOf(ribbonGroupsColl, ribbonGroup)
Call RbnCtrlGrpCollRemoveAt(ribbonGroupsColl, iRet)
```

Once users have drilled their way down to a specific Ribbon Control Group, several types of controls can be added, removed, or modified within this group. In order to do this, users need to get its collection as well. This collection is also an IRibbonControlGroupCollection, however, it's easier to refer to this collection as an IMapInfoControlCollection. After getting this collection of controls, users can add, remove, or modify controls within the group that actually do things. See IMapInfoControlCollection Interface for more details.

```
Dim groupControlColl as This
groupControlColl = GetRbnCtrlGrpCtrls(ribbonGroup)
```

Different Method Names for same Operations in MapBasic

Recall earlier statements for use of more descriptive method names in IMapInfoPro.def and that these longer-named methods are inherited from a higher-level Interface, but are provided to make it more explicit what is being used if needed. To expose these interfaces in MapBasic within IMapInfoPro.def, the entire .NET UX object model was exported to IMapInfoPro.def with unique method names for every Interface within it. This resulted in numerous MapBasic methods for lower-level controls based on their Interface. Most lower-level ribbon controls are inherited from IMapInfoControl interface, so methods from either the lower level Ribbon Control or IMapInfoControl interfaces can be used interchangeably. If more descriptive names are needed for program clarity, then using the lower level method names may be desired, but not necessary.

The method of GetRbnCtrlGrpCtrls() is one example of a more descriptive named method that is inherited from a higher level IControlGroup interface. If users wanted to call a more generic method name that can be used with other control collections (not just RibbonControlGroups) they could just call a more generic GetCtrlGrpCtrls() and get the same collection of controls.

Example using method from IControlGroup interface:

```
groupControlColl = GetCtrlGrpCtrls(ribbonGroup)
```

Now let's add a basic RibbonButton Control to the Control Collection just obtained.

Ribbon Controls

There are several Ribbon Control types for use within a Ribbon Control Group. The most common of these are:

- RibbonButton (IRibbonButtonControl)
- RibbonToolButton (IRibbonToolButtonControl)
- RibbonMenuItem (IRibbonMenuItemControl)

- DropDownButton (IRibbonDropDownControl)
- SplitButton (IRibbonSplitControl)
- GalleryControl (IRibbonGalleryControl)
- CustomControl (IRibbonCustomControl)
- TextBlock (ITextBlock)
- Image (IImageControl)
- CheckBox (IRibbonCheckBox)
- RadioButton (IRibbonRadioButton)

RibbonButton (IRibbonButtonControl)

To add a simple RibbonButton (IRibbonButtonControl) to the Control Group Collection (IMapInfoControlCollection) call this method below which takes the Control collection as first parameter, a control Name, a control Caption, and an enumerated Control Type.
*** See enums.def and MapInfo.Types ControlType Enumeration in Extensibility Reference

```
dim button1 as This
button1 = MICtrlCollAddStrStrInt(groupControlColl, "OpenBtn", "Open", ControlType_Button)
```

There should now be a button control interface created and ready for use. If the application was compiled and run as is, the control would just exist but could not be used for anything until some properties were assigned and perhaps an icon so users could see it and know it's purpose.

Make it a Large size button:

```
call SetRbnBtnCtrlIsLarge(button1, TRUE)
```

Set a KeyTip to the control so when Alt key is pressed this letter shortcut appears on the control:

```
Call SetRbnBtnCtrlKeyTip(button1, "OZ")
```

Icons

This is where RegisterUriParser method would be called to set up a parser to retrieve icons from NET assemblies. A user would pass the newly created button1 instance followed by a constructor for a new Uri to retrieve an icon image resource with the filename of "openTable_32x32.png" within MIPro's StyleResources NET assembly. These icons can be set to a couple different size properties. See LargeIcon or SmallIcon link under IRibbonButtonControl Properties in Extensibility Reference for more details. These properties are inherited from the IImageControl Interface.

This example is assigning a larger sized image icon from MIPro's StyleResources NET assembly to this newly added Ribbon Button. If unsure of whether an image resource exists within a NET assembly users can try using the freeware ILSPY tool available online.

```
Call SetRbnBtnCtrlLargeIcon(button1,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/openTable_32x32.png
", 0))
```

Users can also access image resources from NET assemblies in other specified directories. In this example, the icon image within ImageResources.dll is located in an Images subdirectory of the MapBasic Application folder to be used with this control.

```
Call SetRbnBtnCtrlSmallIcon(button1, New_Uri("pack://addinorigin:,,,/" + ApplicationDirectory$() +
"\\Images\\ImageResources.dll;component/Images/close16.png", 0))
```

Tool Tips

Now if a user wants a ToolTip to appear with information about the button's purpose when hovering over the control, a new instance of a ToolTip can be constructed and assigned to this control. ToolTips allow multiple properties for customization. A new instance of a Ribbon ToolTip (See MapInfoRibbonToolTip Class) has to be created to do this (MapInfoRibbonToolTip Constructor), and then set some properties to use with this new ToolTip.

Create & Set the button tooltip for button1
```
Dim button1ToolTip as This
button1ToolTip = New_MapInfoRibbonToolTip()
```

Set Title Text for ToolTip
```
Call SetMIRbnToolTipToolTipText (button1ToolTip, "Open")
```

Set a Description of what the Tool does
```
Call SetMIRbnToolTipToolTipDescription (button1ToolTip, "Open Table")
```

Set a description to be used if\when the tool is in a disabled state (e.g.: how to enable it)
```
Call SetMIRbnToolTipToolTipDisabledText (button1ToolTip, "Always Enabled")
```

Now assign this ToolTip to the Control button
```
Call SetRbnBtnCtrlToolTip(button1, button1ToolTip)
```

Button Commands and Handlers
Control buttons should do something, so this button will be used to launch the Open dialog in MIPro.
If including Menu.def users can pass the define, else use the actual numeric constant

Set MapInfo command to the button
```
Call SetRbnBtnCtrlCmdID(button1, M_FILE_OPEN)  ‘or 102
```

If a user clicks on this button, or invokes the KeyTip, the Open dialog should appear for user to open a table into MIPro. Users can also set a Calling Handler for use with this Control Button. If a handler subroutine exists in MapBasic application, this control button can access that handler code instead of a pre-defined ID within MIPro or some addin.
```
Call SetRbnBtnCtrlCallingHandler(button1, “MyHandlerSub”)

Sub MyHandlerSub
       Run Menu Command M_FILE_OPEN
End Sub
```

Key Gestures
A Key gesture (Shortcut) can also be assigned to this Button control. If this Key Gesture is used by MIPro already for another control, then it should be ignored. Use Command Editor Tool in MIPro to see which key gestures are already being used.

```
Call SetRbnBtnCtrlKeyGesture(button1,"Ctrl+O")
```

There are numerous other properties that can be Set or returned (Get) on Ribbon Controls at this level. See the IRibbonButtonControl Interface or IMapInfoControl Interface topic for a listing of these Property methods. At this level users can choose to call the more descriptive method name that contains the abbreviated control type or the shorter method name from the parent Interface (IMapInfoControl) that applies to most controls at this level. For example getting the Horizontal Content Alignment property of a Ribbon Button Control can be done by calling:

```
Dim iRet as integer
iRet = GetRbnBtnCtrlHorzContentAlign(button1)
```

Or
```
iRet = GetMICtrlHorzContentAlign(button1)
```

Returned value of iRet will be one of the following enumerations for alignment properties:
- Horiz_Align_Left 0
- Horiz_Align_Center 1

- Horiz_Align_Right 2
- Horiz_Align_Stretch 3

These two functions will do the **exact same thing**, however one uses a more detailed method name from the IRibbonButtonControl interface and the latter uses the more generic name from the parent IMapInfoControl interface. It is a matter of user preference only.

RibbonToolButton (IRibbonToolButtonControl)

A Ribbon Tool Button Control is created for use on a document window only such as a Map, Browser, Layout, or Redistricter window. Examples of Tool Button controls are the Ruler, Info, and the various Select tools or Spatial Object tools. To create these Tool Buttons on the Ribbon UX, users would go through all the steps to create a RibbonButtonControl described earlier except for the following methods described that are unique to Ribbon Tool Buttons. See IRibbonToolButtonControl Interface for more information.

After getting the Control Collection, a ToolButton can be added to it just like a Ribbon Button earlier. Notice the different control type of ControlType_ToolButton. See Enums.def or ControlType Enumeration in Extensibility Reference for a listing of Control Types available.

```
Dim ToolButton1 as This
ToolButton1 = MICtrlCollAddStrStrInt(groupControlColl, "ZoomIn", "Zoom In", ControlType_ToolButton)
```

Setting control size and a KeyTip

```
Call SetRbnToolBtnCtrlIsLarge (ToolButton1, TRUE)
Call SetRbnToolBtnCtrlKeyTip (ToolButton1, "ZI")
```

Set the Tool button icon when control is large size

```
Call SetRbnToolBtnCtrlLargeIcon(ToolButton1,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/zoomIn_32x32.png",
0))
```

Set the Tool button icon when control is a smaller size

```
Call SetRbnToolBtnCtrlSmallIcon(ToolButton1,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/zoomIn_16x16.png",
0))
```

Create & Set the ToolButton1 tooltip

```
Dim ToolButton1ToolTip as This
ToolButton1ToolTip = New_MapInfoRibbonToolTip()
Call SetMIRbnToolTipToolTipText (ToolButton1ToolTip, "Zoom-In Tool")
Call SetMIRbnToolTipToolTipDescription (ToolButton1ToolTip, "Zoom into map")
Call SetMIRbnToolTipToolTipDisabledText (ToolButton1ToolTip, "Enabled only when map window is
open")
Call SetRbnToolBtnCtrlToolTip (ToolButton1, ToolButton1ToolTip)
```

Now assign a MapInfo command to the button

```
Call SetRbnToolBtnCtrlCmdID (ToolButton1, M_TOOLS_EXPAND)  'or 1705
```

Tool Button Cursors

There are two methods to set a cursor for a Tool Button. One (SetRbnToolBtnCtrlCursorId) uses an integer that is a built in MapInfo Cursor ID that users can find listed in ICONS.def. The second method (SetRbnToolBtnCtrlCursor) uses a string that provides a way to pass the resource id and a specific resource DLL that contains that ID for use as a tool cursor. Users can also pass built-in cursor IDs from Icons.def as a string.

Using a cursor resource ID 136 as a string from a FILE called gcsres32.dll

```
Call SetRbnToolBtnCtrlCursor(ToolButton1, "136 FILE gcsres32.dll")
```

*** NOTE: Cursors can be obtained from Win32 resource DLL's and used in MapInfoPro x64 versions. Icon resources must be in x64 NET assemblies.

Can also pass the built in cursor ID's used by MIPro as a string

```
Call SetRbnToolBtnCtrlCursor (ToolButton1, "129")    'this is MI_CURSOR_ZOOM_IN
```

Or as an integer define from Icons.def:

```
Call SetRbnToolBtnCtrlCursorId(ToolButton1, MI_CURSOR_ZOOM_IN)
```

DrawMode

Assign a DrawMode to the Tool Button. DrawModes are ways that Tool Buttons can be used such as just a single click (DM_CUSTOM_POINT) or drawing an object (e.g.: DM_CUSTOM_LINE, DM_CUSTOM_POLYGON, etc.). See Icons.def for list of supported DrawModes. Only Tool Buttons have DrawModes.

```
Call SetRbnToolBtnCtrlDrawMode (ToolButton1, DM_CUSTOM_RECT)
```

Modifier Keys

Set a Modifier key for this Tool Button if desired. This affects whether shift and control keys allows "rubber-band" drawing if the user drags the mouse while using this Tool Button.

```
SetRbnToolBtnCtrlBModifierKeys(ToolButton1, FALSE)
```

Set its initial enabled state

```
Call SetRbnToolBtnCtrlEnabled(ToolButton1, TRUE)
```

RibbonMenuItem (IRibbonMenuItemControl)

To create these Ribbon Menu Item controls on the Ribbon UX, users would go through all the steps to create Buttons or Tool Buttons described earlier except for the following methods described that are unique to Ribbon Menu Items.

After getting the Control Collection, a Ribbon Menu Item can be added to it just like a RibbonButton earlier. What is unique about Ribbon Menu Items from these other controls is that these Menu Items can be nested (e.g.: Sub Menus). Most of the other placement and similar properties of other controls apply to Menu Items. See IRibbonMenuItemControl Interface for more information.

```
Dim MenuItem, MenuItemToolTip as This
MenuItem = MICtrlCollAddStrStrInt(groupControlColl, "MenuOpen", "Open
Table",ControlType_RibbonMenuItem)
Call SetRbnMenuItemCtrlKeyTip (MenuItem, "OT")
```

Set the Menu icon, ToolTips, and CommandID or CallingHandler in same manner as described earlier

```
Call SetRbnMenuItemCtrlCmdID(MenuItem, M_FILE_OPEN)   'or 102
Call SetRbnMenuItemCtrlCallingHandler(MenuItem, "MyHandlerSub")
```

Additionally, if a user does not want icons appearing with menu items then set this property to FALSE
Call SetRbnMenuItemCtrlIsIconEnabled(MenuItem, TRUE)

If user wants to create sub menus, then a control group collection from this MenuItem can be obtained, then just like with previous controls a new MenuItem control can be added, inserted to this collection that becomes a Sub Menu of the original MenuItem. Each MenuItem can have its own collection providing ability to create Sub Menus several layers deep if needed. Can use either the descriptive method name or higher-level common method name if desired.

```
Dim SubMenu1, MenuCtrlColl as This

MenuCtrlColl = GetRbnMenuItemCtrlCtrls(MenuItem)
```

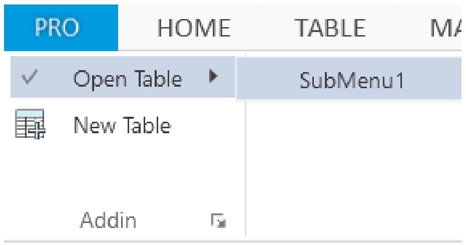
OR

```
MenuCtrlColl = GetCtrlGrpCtrls(MenuItem)
```

Add it to the submenu collection as a menu item

```
SubMenu1 = MICtrlCollAddStrStrInt(MenuCtrlColl, "SubMenu1", " SubMenu1",ControlType_RibbonMenuItem)
```

Ribbon Menu Item with a Sub Menu should look like this:



DropDownButton (IRibbonDropDownControl)

To create these Ribbon Drop Down controls on the Ribbon UX, users would go through steps to create a RibbonMenuItem described earlier except for the following methods described that are unique to Ribbon Drop Down controls. Examples of Ribbon Drop down buttons are the Styles and Select Tools buttons under the Map Tab.

After getting the Control Collection, a Ribbon Drop Down control can be added to it just like earlier. What is unique about Ribbon Drop Down buttons from these other controls is that these controls typically use a collection of Ribbon Buttons or Ribbon Tool Buttons accessed by clicking on the Drop Down control. Most of the other placement and similar properties of other ribbon controls apply to Drop Down buttons. See IRibbonDropDownControl Interface for more information.

```
Dim DropDown as This
DropDown = MICtrlCollAddStrStrInt(groupControlColl, "DropDownBtn", "DropIt",
ControlType_DropDownButton)
```

As done earlier with these controls, users can assign an icon, set alignment, KeyTips, etc., to this drop down button. They do not launch commands themselves but provide access to a collection of controls such as Tool or RibbonButtons that expand or collapse underneath them.

As done earlier get controls collection from an instance of the DropDown button and then add some RibbonButtons to it.

```
Dim DropControlsColl as This
DropControlsColl = GetRbnDropDownCtrlCtrls(DropDown)
```

Now add three RibbonButtons to the collection of this Ribbon Drop Down control:

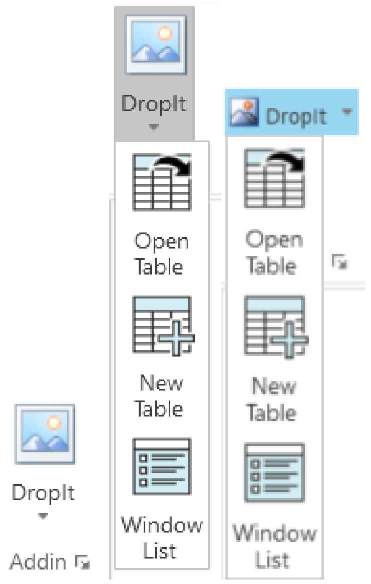
```
Dim btnOpen, btnNew, btnCustom as This
btnOpen = MICtrlCollAddStrStrInt(DropControlsColl, "OpenBtn", "Open Table", ControlType_Button)
Call SetRbnBtnCtrlLargeIcon(btnOpen,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/openTable_32x32.png
", 0))
Call SetRbnBtnCtrlCmdID (btnOpen, M_FILE_OPEN)  'or 102
btnNew = MICtrlCollAddStrStrInt(DropControlsColl, "NewBtn", "New Table", ControlType_Button)
Call SetRbnBtnCtrlLargeIcon (btnNew,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Table/newTable_32x32.png",
0))
Call SetRbnBtnCtrlCmdID (btnNew, M_FILE_NEW)  'or 101
btnCustom = MICtrlCollAddStrStrInt(DropControlsColl, "CustomBtn", "Window List",
ControlType_Button)
```

```
Call SetRbnBtnCtrlLargeIcon (btnCustom,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Window/windowList_32x32.png
", 0))
```

Set this one to call a custom handler that does something specific required by users.

```
Call SetRbnBtnCtrlCallingHandler  (btnCustom, "CustomHandler")
```

A DropDown Control would end up similar to this in the MapInfo Ribbon (collapsed and expanded state and also using an Extra Small Icon, e.g.: SetRbnDropDownCtrlIsExtraSmall).



SplitButton (IRibbonSplitControl)

A Ribbon Split Button is essentially the same control as a Ribbon DropDown button EXCEPT, a Ribbon Split Button is able to display the Last Used Command (last control that was clicked) from its drop-down.

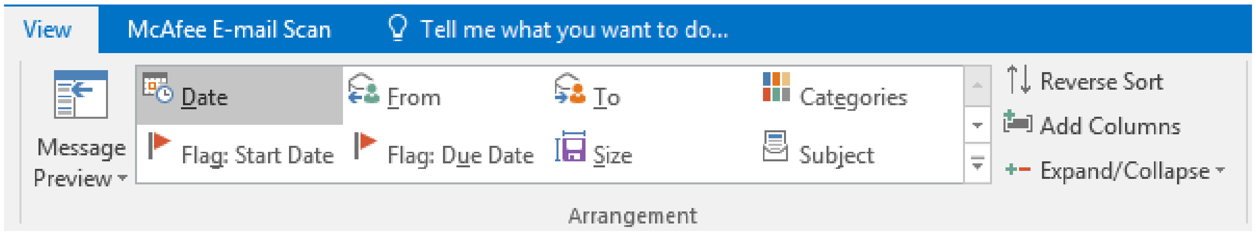
Add a Split button to the ribbon group

```
Dim SplitButton1 as This
SplitButton1 = MICtrlCollAddStrStrInt(groupControlColl, "SplitBtn", "SplitButton",
ControlType_SplitButton)
Call SetRbnSplitCtrlRememberLastCmd(SplitButton1, TRUE)
```

Now whenever a user clicks a button from the drop down list, that last used button will be shown as the current button in the ribbon for this control. If user selected the Window List control created in last example, that control would now be the current icon shown instead of the original DropIt button.

GalleryControl (IRibbonGalleryControl)

Gallery Controls are one of the more complex controls to create on the ribbon, as they require several more properties to set them up. Gallery Controls allow users to organize numerous controls inside separate expandable groups within this Gallery that can be expanded or collapsed and scrolled through if needed. It permits users to present a lot more controls for use without taking up a lot of space on the ribbon with separate Ribbon, DropDowns, Split buttons, Menus, Tools, etc. If users are familiar with MS Outlook, that application has multiple Gallery Controls in its ribbon for View Tab Arrangement settings and several Quick Steps. Example from MS Outlook here:



MIPro allows users to create the same control in its new Ribbon UX as well. Gallery Controls also remember the last used control by default. After getting the Control Group Collection as done with previous examples a new Gallery Control (IRibbonGalleryControl) is added to the collection.

Add a Ribbon Gallery Control to the Ribbon Group Control Collection

```
Dim GallCtrl as This
GallCtrl = MICtrlCollAddStrStrInt(GroupControlColl, "MyGallery", "Operations",
ControlType_GalleryControl)
```

Set InLine Property for the Ribbon Gallery Control. If set to FALSE, it will appear similar to a DropDown control.

```
Call SetRbnGalleryCtrlInline(GallCtrl,TRUE)
```

Allows for Manipulation of the Gallery control or not (this is a Windows touch screen property):

```
Call SetRbnGalleryCtrlIsManipulationEnabled(GallCtrl,TRUE)
```

Gallery Controls can have additional menu items at bottom. If FALSE, these are hidden from view.

```
Call SetRbnGalleryCtrlIsMenuIconBarEnabled(GallCtrl,TRUE)
```

Create a border color around the gallery frame

```
Call SetRbnGalleryCtrlBorderBrush(GallCtrl,"65535")
```

Set a KeyTip for the Gallery Control

```
Call SetRbnGalleryCtrlKeyTip(GallCtrl,"GC")
```

If the Gallery Control has multiple groups with several icons in each group, then it can be useful at times to filter out controls that are not needed so users can find them easily. Setting up three Filters on this Gallery Control for “All”, “Table”, and “Options” groups.

```
Call RbnGalleryCtrlAddFilter(GallCtrl,"All")
Call RbnGalleryCtrlAddFilter(GallCtrl,"Table")
Call RbnGalleryCtrlAddFilter(GallCtrl,"Options")
```

Once the initial properties and filters have been defined for the Gallery Control, users can now begin adding Control groups to the Gallery Control, then add the actual working controls to these Control groups.

'Get Gallery Groups collection from the Gallery Control:

```
Dim GroupControlColl as This
GallCtrlGroup = GetRbnGalleryCtrlGrps(GallCtrl)
```

Define the two Gallery Groups in Gallery Control

```
Dim TableGroup, OptionsGroup as This
```

Define the Collection of controls in each of these Gallery Groups

```
Dim TableGroupCtrls, OptionsGroupCtrls as This
```

Define the GalleryControlItems for each collection of controls (Table and Options groups)

```
Dim TableGallItem1, TableGallItem2, TableGallItem3 as This
Dim OptionsGallItem1, OptionsGallItem2, OptionsGallItem3 as This
```

Add Groups to the Gallery Group Collection

```
TableGroup = MIGalleryGrpCollAddStrStr (GallCtrlGroup, "Table", "Table Group")
OptionsGroup = MIGalleryGrpCollAddStrStr (GallCtrlGroup, "Options", "Options Group")
```

Get Collection of Gallery Group Controls from Table Group

```
TableGroupCtrls = GetRbnGalleryGrpCtrls (TableGroup)
```

Get Collection of Gallery Group Controls from Options Group

```
OptionsGroupCtrls = GetRbnGalleryGrpCtrls (OptionsGroup)
```

Construct Filter index arrays for the Gallery Control groups (All, Table, and Options filters), then add items to these index arrays.

```
Dim FilterIndex1, FilterIndex2 as This
FilterIndex1 = New_Int32List()
FilterIndex2 = New_Int32List()
Call Int32ListAdd(FilterIndex1, 0)
Call Int32ListAdd(FilterIndex1, 1)
Call Int32ListAdd(FilterIndex2, 0)
Call Int32ListAdd(FilterIndex2, 2)
```

Assign the Filter indexes to each group so if user selects the Table Filter, the Gallery Control will only show controls in the Table control group.

```
Call SetRbnGalleryGrpFilterIndex(TableGroup, FilterIndex1)
```

Do same for the Options control group and its index.

```
Call SetRbnGalleryGrpFilterIndex(OptionsGroup, FilterIndex2)
```

Add a Gallery Control Item to the Collection of Table Group controls (TableGroupCtrls).

```
TableGallItem1 = MICtrlCollAddStrStrInt (TableGroupCtrls, "NewTable", "New Table",
ControlType_GalleryItem)
```

Create a new ToolTip and assign to TableGallItem1 control

```
Dim TableCtrlToolTip1  as This
TableCtrlToolTip1 = New_MapInfoRibbonToolTip()
Call SetMIRbnToolTipToolTipText (TableCtrlToolTip1, "New")
Call SetMIRbnToolTipToolTipDescription (TableCtrlToolTip1, "New Table")
Call SetMIRbnToolTipToolTipDisabledText (TableCtrlToolTip1, "Always Enabled")
Call SetRbnGalleryItemToolTip (TableGallItem1, TableCtrlToolTip1)
```

Set other Ribbon Gallery Item properties needed for this control such as Command ID or CallingHandler, the icon used for this control, etc. See IRibbonGalleryItem Interface for more information.

'Proceed to create other Gallery Control items (e.g.: TableGallItem2 and TableGallItem3) and add them to the TableGroupCtrls Collection. Proceed to add Options Group Controls to the OptionsGroupCtrls collection and assign similar properties in same manner.

Gallery Control Menu Items (IRibbonGalleryControl.MenuItems)

Gallery Controls can have additional menu items at the bottom. They are simply Ribbon Menu Items attached to Gallery Controls such as those under the New Document gallery control in Home Tab of MIPro used for window related commands (clone, redraw, and status bar).

Get the MenuItem collection from the Gallery Control. See IRibbonGalleryControl.MenuItems Property for more information.

```
Dim GallMenuItems  as This
GallMenuItems = GetRbnGalleryCtrlMenuItems(GallCtrl)
```

Add a new Gallery Menuitem to show or Hide the status bar to this collection

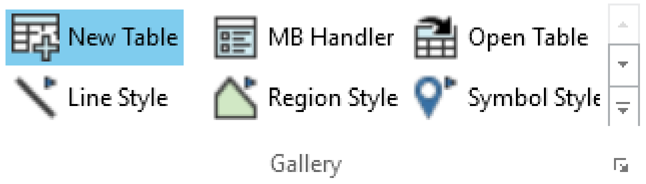
```
Dim mnuHideBar as This
mnuHideSBar = MICtrlCollAddStrStrInt (GallMenuItems, "HideSBar", "Hide Status Bar",
ControlType_RibbonMenuItem)
```

Then set a Tooltip and Icon for the Menu Item as well as any other Menu Item properties required. Set MapInfo Menu command id to the menu to Show\Hide status bar

```
Call SetRbnMenuItemCtrlCmdID(mnuHideSBar, M_WINDOW_STATUSBAR)  or 616
```

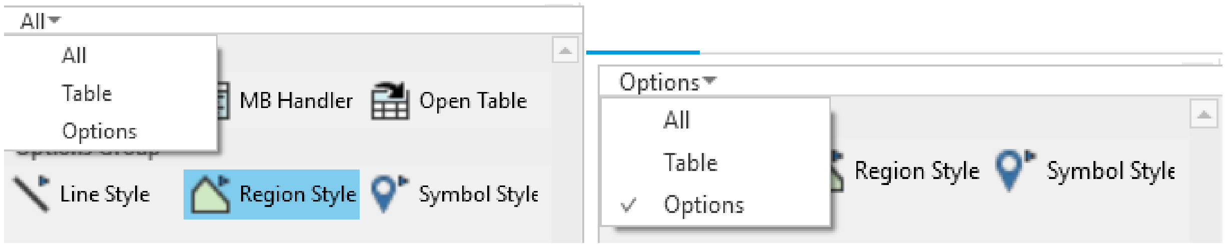
Gallery Control Examples

In collapsed state, also remembers last used command:



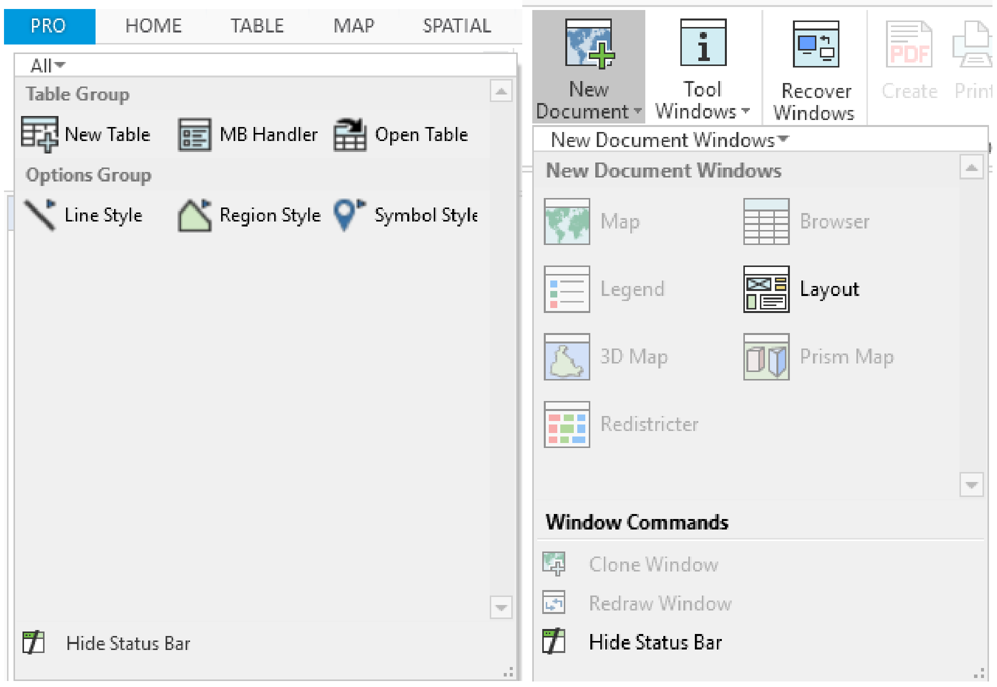
Showing optional filters to display only certain controls as needed.

Filter shown with only Options group on right.



Showing expanded state with additional Gallery Menu Control that can be added to it at left.

If InLine property equals FALSE then Gallery Controls appear more like a Drop Down control at right:



CustomControl (IRibbonCustomControl)

Custom Controls are essentially third-party NET User Controls. They are added to a ribbon like earlier controls and have most of the same properties except for CommandID's and CallingHandlers. Custom controls require using SetContent and Mouse related methods that are going to be defined by the control itself.

To Add a Custom Control to the ribbon group in MapBasic

```
CustomCtrl = MICtrlCollAddStrStrInt(groupControlColl, "CustomCtrlBtn", "DoStuff",
ControlType_CustomControl)
```

TextBlock (ITextBlock)

Text Blocks are simple controls that can be used, for example, in the Status bar by manipulating its enabled state.

To set up a Text Block in the status bar, users first need to get an instance of the MIPro Application like earlier:

```
Dim mapinfoApplication as This
mapinfoApplication = SystemInfo(SYS_INFO_IMAPINFOAPPLICATION)
```

Now get Status Bar Interface from mapinfoApplication instance.

```
Dim sBar as This
sBar = GetStatusBar(mapinfoApplication)
```

Status bars, like other controls, have a collection that can be accessed.

```
sBarCtrls = GetIStatusBarCtrls(sBar)
```

Add a TextBlock control to this collection, then proceed to set its properties as needed. See ITextBlock Interface in Extensibility Help for more information on its properties. Control should now display as text in the MIPro Status Bar.

```
Dim TextBlk as This
TextBlk = MICtrlCollAddStrStrInt(sBarCtrls,"CustomStatusBarItem", "My Text Block",
ControlType_TextBlock)
```

Image (IImageControl)

An ImageControl is essentially the same thing that has been added to all the controls previously created that use icons in their display. It just allows display of an image on the ribbon. Control should now display as an image in the MIPro Ribbon Bar. Assuming user has drilled down to the Ribbon Control Collection level, just add this control to that collection. Set properties on the Image control, such as an icon from a NET assembly, plus any others as needed. See IImageControl Interface in Extensibility Help for more information on its properties.

Add a button to the ribbon group.

```
Dim image1 as This
image1 = MICtrlCollAddStrStrInt(groupControlColl, "ImageControl", "MyImage", ControlType_Image)
Call SetIImageLargeIcon (image1,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/openTable_32x32.png
", 0))
```

CheckBox (IRibbonCheckBox)

A Ribbon CheckBox is a basic control that allows users to change its state (Checked\Unchecked) in order to affect some operation in their application. Assuming user has drilled down to the Ribbon Control Collection level, just add this CheckBox control to that collection. Set properties on this control as needed such as ToolTip, enabled state, KeyTip, etc. See IRibbonCheckBox Interface in Extensibility Help for more information on its available properties.

```
button1 = MICtrlCollAddStrStrInt(groupControlColl, "CheckBox", "CheckMate", ControlType_CheckBox)
```

Set it to checked. If unchecked set it to FALSE.

```
Call SetRbnCheckBoxIsChecked(button1, TRUE)
```

If the checkbox is checked or TRUE, it can be used to invoke a command, so set properties for CommandID or CallingHandler.

RadioButton (IRibbonRadioButton)

A Ribbon Radio Button is another basic control that allows users to change its state (Checked\Unchecked) in order to affect some operation in their application. RadioButtons are created in groups as these controls are used together for exclusive actions. This means if one RadioButton is checked, another in the same group is Unchecked. Assuming user has drilled down to the Ribbon Control Collection level users can add Radio Button groups to the collection. See IRibbonRadioButton Interface in Extensibility Help for more information on its available properties.

Add a button to the ribbon group

```
Dim button1 as This
button1 = MICtrlCollAddStrStrInt(groupControlColl, "OpenBtn", "Open", ControlType_RadioButton)
```

Set Radio group name to be used with both controls

```
Call SetRbnRadioBtnGrpName(button1, "My Radio Buttons")
```

Set it to checked, the other button will be unchecked

```
Call SetRbnRadioBtnIsChecked(button1, TRUE)
```

Create a tooltip and there properties for this Radio Button

Set MapInfo command to the button – Lock scale with a mapper

```
call SetICmdCtrlCmdId(button1, M_VIEW_LOCK_SCALE) 'or 838
```

Add a second button to the ribbon group

```
dim button2 as This
button2 = MICtrlCollAddStrStrInt(groupControlColl, "CustomHandlerBtn", "MB
Handler",ControlType_RadioButton)
```

Set a Calling Handler to use with this radio button if it's checked

```
SetICmdCtrlCallingHandler(button2, "MySubHandler")
```

Context Menus (IContextMenus)

Context menus are menus that allow users to interact with document windows through a Right-Click on the window itself to bring up a slate of menu items to perform actions related to the window type. The MapMiniToolBar is part of the IContextMenus Interface which will be covered in next section. See IContextMenus Interface in Extensibility Help for more information on its available properties.

To access ContextMenus in MIPro first get an instance of the IMapInfoPro interface if one does not already exist (e.g.: mapinfoApplication = SystemInfo(SYS_INFO_IMAPINFOAPPLICATION))

From the IMapInfoPro interface get its collection of Context Menus (IContextMenus).

```
Dim ContextMenus as This
ContextMenus = GetContextMenus(mapinfoApplication)
```

In this example, the Mapper window's context menu will be modified so an instance of the Map Window Context Menus will be obtained. See IContextMenus.GetContextMenu Method. The second parameter in the ICntxtMnusGetCntxtMnu() method is an enumerated Window Menu type define from enums.def. See ContextMenuId Enumeration for more details. This menu will only appear after a right-click in the Mapper window.

```
Dim MapContextMenu  as This
MapContextMenu = ICntxtMnusGetCntxtMnu(ContextMenus, MenuId_MapperShortcut)
```

Once a user has an instance of the Mapper Context Menu, the Map Context Menu Collection needs to be accessed in order to add context menu items to it. With MapBasic this method is duplicated under IContextMenu and IControlGroup interfaces with different names, so either of these methods produce same results.

```
Dim MapMenus as This
MapMenus = GetICntxtMnuCtrls(MapContextMenu)
```

OR

```
MapMenus  = GetCtrlGrpCtrls(MapContextMenu)
```

Once the collection of Map Menu Items is accessed, add a new Context menu to the MapMenus collection. Provide a control name and the actual caption to appear on the menu. The last parameter is a

```
Dim mnuMapViewEntire as This
mnuMapViewEntire = MICtrlCollAddStrStrInt (MapMenus, "MapViewEntire", "View All",
ControlType_ContextMenuItem)
```

See IMapInfoControlCollection Interface for more information on methods to add controls to collections, and ControlType Enumeration and Enums.def for info on Control types.

Context Menu Tool Tips

Creating a Tool Tip for Context Menus is a little confusing as the methods used to define it are from the Ribbon, not ContextMenu interfaces. Call the constructor method as done with other controls earlier, then call SetICntxtMnuMenuItemToolTip() method to assign the Ribbon Tool tip to the specific Context menu.

```
Dim MenuToolTip as This
MenuToolTip = New_MapInfoRibbonToolTip()
Call SetMIRbnToolTipToolTipText (MenuToolTip, "View Entire layer")
Call SetMIRbnToolTipToolTipDescription (MenuToolTip, "Display an individual or all map layer(s).")
Call SetMIRbnToolTipToolTipDisabledText (MenuToolTip, "Enabled only when map window is open")
Call SetICntxtMnuMenuItemToolTip (mnuMapViewEntire, MenuToolTip)
```

Assign Large and small icons to the context menu item

```
Call SetICntxtMnuMenuItemSmallIcon (mnuMapViewEntire,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/zoomToEntireLayer_1
6x16.png", 0))
Call SetICntxtMnuMenuItemLargeIcon (mnuMapViewEntire,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/zoomToEntireLayer_3
2x32.png", 0))
```

The Menu Item needs to do something when clicked so set a MapInfo Menu command id to the menu

```
Call SetICntxtMnuMenuItemCmdID(mnuMapViewEntire, M_MAP_ENTIRE_LAYER)
 ' or 807 for view entire layer
```

Set other properties for Context Menu item as required. See IContextMenuMenuItem Interface for more details.

Context menus like Ribbon Menu Items can have sub menus as well. A new Context Sub menu item can be added to a Context menu item like this:

```
Dim mnuTop, menuItems, mnuSubMenu as This
```

Add this Context menu to the MapMenus Colleciton like that done earlier.

```
mnuTop = MICtrlCollAddStrStrInt (MapMenus, "TopMenu", "ExpandThis", ControlType_ContextMenuItem)
```

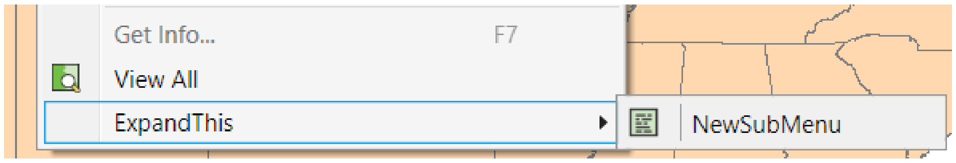
Get a Collection from this Context menu item to add sub menus to it.

```
MenuItems = GetICntxtMnuMenuItemCtrls(mnuTop)
```

Add the Sub Menu to this collection then set whatever properties on the sub menu as needed.

```
mnuSubMenu = MICtrlCollAddStrStrInt (MenuItems, "SubMenu", "NewSubMenu",
ControlType_ContextMenuItem)
```

Context Menu with Sub Menu example:



MapMiniToolBar (IMapMiniToolBar)

The MapMiniToolBar is a set of controls derived from the Context Menus Interface (IMapInfoPro.ContextMenus) that allow users to access a suite of tools while on the mapper without having to go back to the Ribbon Map tab or another control on the Ribbon elsewhere. The MapMiniToolBar is access through a Right-Click on the Map Window itself. It is not available on other document window types.

To access the MapMiniToolBar, get the Context Menus (IContextMenus) from an instance of IMapInfoPro as done earlier for Context Menus, then get an instance of the MapMiniToolBar from instance of IContextMenus (See IContextMenus Interface)

```
Dim ContextMenus, MiniBar as This
ContextMenus = GetContextMenus(mapinfoApplication)
MiniBar = GetICntxtMnusMapMiniToolBar(ContextMenus)
```

To modify the MapMiniToolBar as done with other controls, first get a collection of controls from an instance of MapMiniToolBar. See IMapMiniToolBar Interface for more information.

```
Dim MiniBarColl as This
MiniBarColl = GetIMapMiniToolBarCtrls(MiniBar)
```

OR

```
MiniBarColl = GetCtrlGrpCtrls(MiniBar)
```

To add a control to the MapMiniToolBar, create a new StackPanel control (IStackPanel) to add to the MapMiniToolBar that will hold a new RibbonToolButton control for an Info Tool. This control will be added to the MiniBar Control collection just acquired.

```
Dim StackPanel as This
StackPanel = MICtrlCollAddStrStrInt(MiniBarColl, "MyPanel", "MyPanel", ControlType_StackPanel)
```

Like with other Ribbon controls, to add more controls to this StackPanel, first need to get its collection as well.

```
Dim StackPanelColl as This
StackPanelColl = GetIStackPanelCtrls (StackPanel)
```

Now add a RibbonToolButton to the StackPanel collection

```
Dim MiniButton as This
MiniButton = MICtrlCollAddStrStrInt(StackPanelColl, "MyButton", "InfoTool", ControlType_ToolButton)
```

Since this is going to be an Info Tool button, set the button icons for it to use same ones in the Map Ribbon Tab.

```
Call SetRbnToolBtnCtrlLargeIcon(MiniButton,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/infoTool_32x32.png"
, 0))
Call SetRbnToolBtnCtrlSmallIcon(MiniButton,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Mapping/infoTool_16x16.png"
, 0))
```

Make it a small to fit on stack panel better

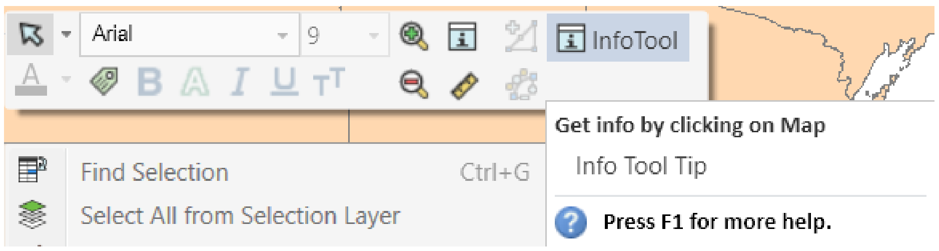
```
Call SetRbnToolBtnCtrlIsExtraSmall (MiniButton, TRUE)
```

Create & Set the MiniToolBar button tooltip.

```
Dim MiniButtonToolTip as This
MiniButtonToolTip = New_MapInfoRibbonToolTip()
Call SetMIRbnToolTipToolTipText (MiniButtonToolTip, "Info Tool Tip")
Call SetMIRbnToolTipToolTipDescription (MiniButtonToolTip, "Get info by clicking on Map")
Call SetMIRbnToolTipToolTipDisabledText (MiniButtonToolTip, "Enabled only when map window is open")
Call SetRbnToolBtnCtrlToolTip (MiniButton, MiniButtonToolTip)
```

Set a MapInfo command ID, Cursor, and DrawMode to this Info Tool Button

```
Call SetRbnToolBtnCtrlCmdID (MiniButton, M_TOOLS_PNT_QUERY)   ' or 1707
Call SetRbnToolBtnCtrlCursorId (MiniButton, MI_CURSOR_FINGER_UP)
Call SetRbnToolBtnCtrlDrawMode (MiniButton, DM_CUSTOM_POINT)
```



Get info by clicking on Map
Info Tool Tip
Press F1 for more help.

Status Bar

The Status Bar in MapInfoPro is the control that resides at the bottom of the Main application window. It is used to show text information or provide access to other controls (Notification windows, Menus, progress bars, etc.) that keep users informed on the state of an application.

To access the Status Bar, get an instance of it from an instance of IMapInfoPro. See IStatusBar Interface for details.

```
Dim sBar as This
sBar = GetStatusBar (mapinfoApplication)
```

Can set the Status Bar visibility, add a tooltip, enable or disable it. Most users will want to add some control to the StatusBar and to do that will need to get a collection of it's controls, then add a control to that collection.

```
sBarCtrls = GetIStatusBarCtrls(sBar)
```

Add a Text Block control to the status bar

```
Dim TextBlk as This
TextBlk = MICtrlCollAddStrStrInt(sBarCtrls,"CustomStatusBarItem", "This is a Text Block",
ControlType_TextBlock)
```

Notification Object (NotificationObject)

This is used to show a Notification 'bubble' such as an Error, Warning, or some other kind of information and display it for a set time in seconds. It's typically used with an event handler (e.g.: NotificationEventArgs). See NotificationObject Class and NotificationType Enumeration for more information.

Notification Objects have a constructor method for creation. Once an instance of IMapInfoPro is obtained, users can call the constructor for a New Notification object and it has no parameters.

```
Dim Notify as This
Dim pt as Point
pt.X = 200  'This is number of points units onscreen for X & Y location to display this
notification.
pt.Y = 200
```

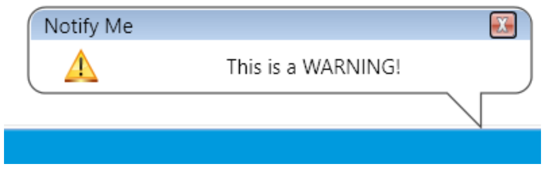
Construct a new Notification object

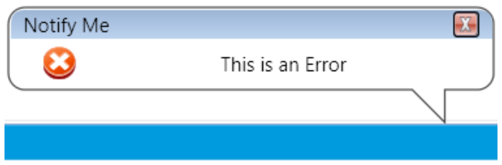
```
Notify = New_NotificationObject()
```

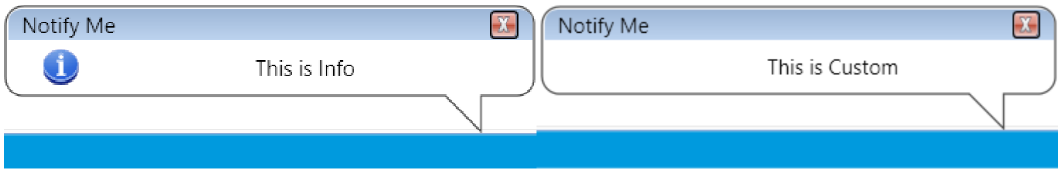
Define a title for it, the message, and type of warning (Info, Warning, Error, None, or Custom), set its location with a Point object, and how many seconds to display it, then make call to show it on the main application window.

```
Call SetNotificationObjectTitle(Notify,"Notify Me")
Call SetNotificationObjectMessage(Notify,"This is a Warning!")
Call SetNotificationObjectType(Notify, Notify_Warning)  'or 2 from Enums.def
Call SetNotificationObjectNotificationLocation(Notify,pt)
Call SetNotificationObjectTimeToShow(Notify, 30)
Call ShowNotification(mapinfoApplication, Notify)
```

Examples:







Ribbon Contextual Tabs (IRibbonContextualTabGroup)

Ribbon Contextual Tabs and Groups are used to display groups of Ribbon controls that only need to be shown when required. The Layer Tools Contextual Tab Group with Layer Styles and Labels tabs and their controls is shown when user selects a Layer in the Explorer Window is one example in MIPro.

To create a Contextual Tab Group and its associated Ribbon Tabs first get an instance of IMapInfoPro, then get an instance of IRibbon like that done for all the ribbon controls done earlier.

Now get collection of Context Tab Groups from the IRibbon instance (see IRibbon.ContextualTabGroups Property for more information)

```
Dim RibbonContextTabGroupColl as This
RibbonContextTabGroupColl = GetCntxtTabGrpsColl(Ribbon)
```

Now that an instance of IRibbonContextualTabGroupCollection Interface exists, users can Add, Create, Insert, or Remove Ribbon Contextual Tab Groups to this collection. In this example, a New Ribbon Contextual Tab Group (IRibbonContextualTabGroup) will be added to this collection.

Call constructor method to create this Context Group then Add it to the Collection.

```
Dim ContextTabGrp as This
ContextTabGrp = RbnCntxtTabGrpCollCreate (RibbonContextTabGroupColl,"ContextTabGroup", "Context
Tools")
Call RbnCntxtTabGrpCollAddRefPtr (RibbonContextTabGroupColl,ContextTabGrp)
```

Once the Context Group is created and added to the collection, set some properties on it to make it stand out from the other Ribbon Tabs when in use. Assign a different background color to do that.

```
Call SetRbnCntxtTabGrpBackColorArgb(ContextTabGrp, RED)
```

When the Context Tab group needs to be shown in Ribbon (such as selecting a Layer), make a call to its Visibility property and make it enabled if\when certain conditions are met. It can also be made to always show in Ribbon regardless.

```
Call SetRbnCntxtTabGrpVisible(ContextTabGrp, TRUE)
Call SetRbnCntxtTabGrpEnabled(ContextTabGrp, TRUE)
```

Now that the Context Ribbon Group exists, the actual Context tabs need to be added, so get the Context Tab Collection from this Context Tab Group.

```
Dim ContextTabColl as This
ContextTabColl = GetRbnCntxtTabGrpTabs(ContextTabGrp)
```

Now add two Context Tabs to this Context Tab Collection.

```
Dim ContextTab, ContextTab2 as This
ContextTab = RbnTabCollAddStrStr(ContextTabColl, "MyContextTab", "Context Tab")
ContextTab2 = RbnTabCollAddStrStr(ContextTabColl, "MyContextTab2", "Context Tab2")
```

Now get the Ribbon Tab Groups from each of these Ribbon Context Tabs. Get its collection first.

```
Dim CTabGroupColl, CTabGroupColl2, CTabGroup, CTabGroup2 as This
Dim groupControlColl, groupControlColl2 as This
CTabGroupColl = GetRbnTabGrps(ContextTab)
```

Add a new group this collection.

```
CTabGroup = RbnCtrlGrpCollAddStrStr(CTabGroupColl, "CtxtGroup", "Context Ribbon Group")
```

Get Ribbon Group controls collection from this Tab group

```
groupControlColl = GetRbnCtrlGrpCtrls(CTabGroup)
```

Now start adding actual controls to this group control collection

```
Dim btnOpen, btnNew as This
btnOpen = MICtrlCollAddStrStrInt(groupControlColl, "OpenCmd", "Open", ControlType_Button)
btnNew = MICtrlCollAddStrStrInt(groupControlColl, "NewBtn", "New Table", ControlType_Button)
```

Now as done with the RibbonButton in earlier examples, set icons to these button controls, then set a CommandID or CallingHandlers, size, width, etc.

Since this was a Context Group a second context tab should be added in the same manner as the first context tab.

Get Ribbon Tab Groups from context tab 2

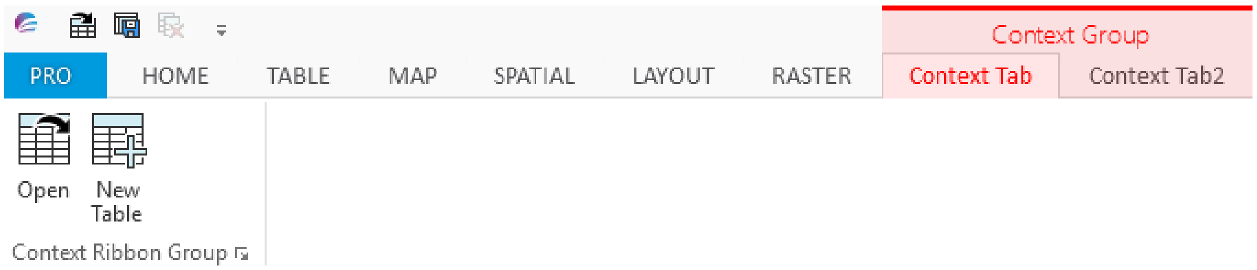
```
CTabGroupColl2 = GetRbnTabGrps(ContextTab2)
```

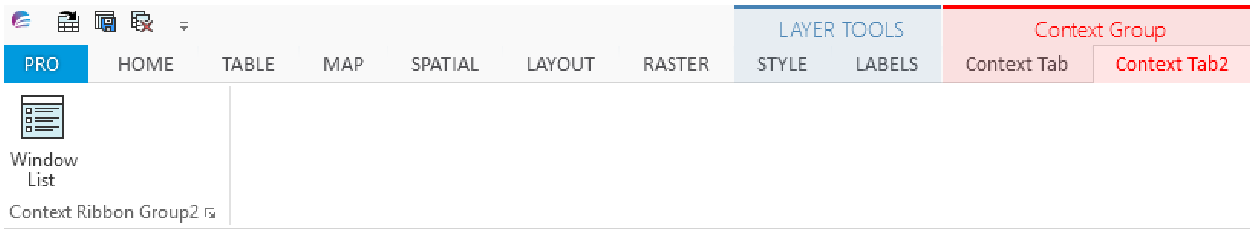
Add a new group to Ribbon Tab Groups.

```
CTabGroup2 = RbnCtrlGrpCollAddStrStr(CTabGroupColl2, "CtxtGroup2", "Context Ribbon Group2")
```

Get Ribbon Group controls collection, then add a button control to the group collection.

```
groupControlColl2 = GetRbnCtrlGrpCtrls(CTabGroup2)
dim btnCustom as This
btnCustom = MICtrlCollAddStrStrInt(groupControlColl2, "CustomBtn", "Window List",
ControlType_Button)
```





Ribbon Backstage (IRibbonBackStage)

The BackStage of MIPro (the PRO tab) is part of the User Interface for supporting numerous application settings (e.g. Options, Licensing, Help, Links to other products on the Web, Exiting, and access to Add-ins installed with MIPro. It is very similar to the contents found under the File tab of the ribbon interface used in Microsoft Word. MIPro allows users to customize the backstage if needed.

After getting an instance of IMapInfoPro, an instance of the IRibbonBackStage can be obtained.

```
Dim mapinfoApplication, BackStageRibbon as This
mapinfoApplication = SystemInfo(SYS_INFO_IMAPINFOAPPLICATION)
BackStageRibbon = GetRibbonBackStage(mapinfoApplication)
```

If users want to change its name from "Pro", they can call this method:

```
Call SetRbnBackStageCaption(BackStageRibbon, "Settings")
```

Can also call other property methods similar to other ribbon tab controls as needed.

The BackStage (IRibbonBackStage) contains a list of BackStage Tabs shown down its left side (e.g.: About, Licensing, Options, Help, Exit, etc.). Users can either create custom Backstage Tabs and content, or add to or modify existing Backstage Tabs and content. Developers of OEM and MapInfo RunTime applications that want to restrict access to certain BackStage Tabs, or options within those Backstage Tabs, can set Visible or Enabled states on these controls as well.

Existing BackState Tabs and Content sections.

- AddTabContent – Adds an entirely new BackStage Tab to the BackStage for adding custom content.
- GetBackStageOptionTab – Gets an instance of IBackStageOptionTab, a special instance of the BackStage Options Tab only. There are certain things that only apply to Options Tabs that will be described later.
- GetBackStageTab (RbnBackStageGetBackStageTab in MapBasic) – Gets an instance of other BackStage tabs (About, Help, Products, Options, or Addins only).

Adding a New content section to Add-Ins Backstage Tab

```
IconLarge =
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/options_64x64.p
ng", 0)
IconSmall =
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/loadExtension_1
6x16", 0)
Dim AddINTab, NewContent as This
```

Get the Add Ins Backstage Tab from the BackStage Ribbon. See IRibbonBackStage.GetBackStageTab Method for more info. Also see BackStageTabs Enumeration topic for which Backstage Tabs may be retrieved from the BackStage ribbon to be modified. This method example is using the define from Enums.def for the AddIns Tab.

```
AddINTab = RbnBackStageGetBackStageTab(BackStageRibbon, BackStageTabs_AddIns)
```

Add a new content section to this Add Ins Tab. It allows for either a small or large icon to be passed as Uri's from a NET assembly defined above.

```
NewContent = TabAddContent(AddINTab, "NewContent", "Try This", "New Content", "Something New",
IconSmall, IconLarge)
```

Adding a New content section to Options BackStage Tab

The Options BackStage Tab is a special BackStage tab that has certain restrictions on it as it contains all the core system settings controls for MIPro application. Developers should only be adding or modifying their own options sections and controls in this Options Backstage Tab.

```
Dim optionsTab, NewPrefs as This
Get the Options Backstage Tab from the BackStage Ribbon. Note the method name below is specifically
for Options. See IRibbonBackStage.GetBackStageOptionTab Method for more info.
optionsTab = RbnBackStageGetBackStageOptionTab (BackStageRibbon)
```

Add a new content section to the Options Tab with its own group name then add a Button control to this group that opens the MapInfo Preferences dialog as an example. The example below is one of two methods for adding Option controls to the Option Tab. See IBackStageOptionTab.AddOption Methods for more details.

```
optPrefs = IBackStageOptionTabAddOptInGrpName(optionsTab, "PrefGroup", "NewPrefs", "New
Preferences", "New Preferences", ControlType_Button)
```

Set the command ID for this control to open the Preferences dialog (from menu.def)

```
Call SetICmdCtrlCmdId(optPrefs, M_EDIT_PREFERENCES)
```

Add a caption for the button

```
Call SetRbnItemCaption(optPrefs, "Preference Settings")
```

Set large and mall icons to be used for this button

```
call SetIImageCtrlLargeIcon(optPrefs,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/systeminfo_32x3
2.png", 0))
call SetIImageCtrlSmallIcon(optPrefs,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/systeminfo_16x1
6.png", 0))
```

Custom BackStage Tabs and Content sections

Users can also create their own BackStage Tab then proceed to add content to it instead of adding content to existing Backstage Tabs.

Add a New BackStage Tab to the BackStage

```
Dim NewBackStageTab, IRibbonBackStageCtrls as This
IRibbonBackStageCtrls  = GetRbnBackStageCtrls (BackStageRibbon)
NewBackStageTab = MICtrlCollAddStrStrInt(IRibbonBackStageCtrls,"BackTab", "NewBackStageTab",
ControlType_BackStageTabItem)
```

Once an instance of NewBackStageTab exists, then users can set multiple properties for the Tab itself, and from there proceed to Add Content with these three methods:

- TabAddContent – see IBackStageTabItem.AddContentSection Method (String, String, String, String, Uri, Uri)
- TabAddContentAtIndex – See IBackStageTabItem.AddContentSection Method (String, String, String, String, Uri, Uri, Int32)
- IBackStageTabItemSetContent() *** Requires a NativeHandleContractInsulator for setting a custom User control. Not going to be described in this document for MapBasic users.

To create a Uri for use with these AddContent methods here's an example:

```
Dim IconLarge, IconSmall as This
IconLarge =
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/about_64x64.png
", 0)
IconSmall =
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/about_32x32",
0)
```

Now add this content (IBackStageTabSection) to the new Backstage Tab, then using same methods for adding controls to ribbon, get a collection of controls from this instance and add them to the section.

```
Dim NewContent as This
NewContent = TabAddContent(NewBackStageTab, "NewContent", "Try This", "New Content", "Something
New", IconSmall, IconLarge)
```

Docking Manager (IDockingManager)

The Docking Manager is the interface that makes MapInfoPro's windowing system so much different from the Win32 versions that preceded it. It replaces or adds to many older Set Window and WindowInfo, or WindowID methods in MapBasic that were specific to Windows32 only. It allows users to set specific window properties such as Docked, Tabbed, Hidden, AutoHidden, or Floating states. Other methods including creating Tab Groups of windows, and adding to, or removing from, these groups of tabbed windows. For one example, the older SysMenuClose in Set Window statement is now replaced by using the "CanClose" parameter in SetDockingManagerProperty() method. This topic will show a few of the most important methods used in the IDockingManager interface.

To get access to the IDockingManager manager interface, get an instance of it right from the IMapInfoPro interface.

```
Dim dockMgr as This
dockMgr = GetDockingManager(mapinfoApplication)
```

Nearly all of the docking manager methods require using an instance of a window (IWindowInfo) as a RefPtr, so one must be obtained using one of the methods like below and passing in the instance of IDockingManager. Can get an IWindowInfo instance using these two most common methods from either an ActiveWindow or the FrontDocumentWindow (e.g: Map, Browser, Layout, etc.).

First open a table and either map or browse it, then use the following methods:

```
Dim retWindow as This
retWindow = GetDockingManagerActiveWindow(dockMgr)
```

OR

```
retWindow = GetDockingManagerFrontDocumentWindow(dockMgr)
```

Once a reference to an IWindowInfo instance is obtained, users can get and set multiple properties on this window instance as needed such as its type, name, visibility, etc. See IWindowInfo Interface for more information.

To change properties of windows in the Docking manager system, users will want to call methods such as SetDockingManagerProperty() or GetDockingManagerProperty(). These methods take the instance of the Docking Manager, a Window instance, a Property Name, and a Property Value.

As mentioned earlier, the Set Window SysMenuClose command no longer applies in new windowing system, so users need to use the "CanClose" property on windows to disable the close button.

Example:

```
Call SetDockingManagerProperty(dockMgr, retWindow, "CanClose", "false")
If GetDockingManagerProperty(dockMgr, retWindow, "CanClose") = "false" then
     Print "Window close button is disabled"
End if
```

Another common use is that users can control the States of windows (Docking, Floating, Hidden, AutoHidden, or Tabbed) by passing in the appropriate parameter for "State" property. This example sets a window to the Floating state after it is opened (typically in docked state by default).

```
Call SetDockingManagerProperty(dockMgr, retWindow, "State", "float")
If GetDockingManagerProperty(dockMgr, retWindow, "State") = "float" then
Print "Window is in floating state"
End if
```

See the IDockingManager.SetDockingManagerProperty Method for information on the multiple properties that can be set on windows under the Docking Manager.

Integrating MapBasic tools into the Tool Manager

MapInfo Pro 64 bit introduced a new Tool Manager with a different interface to allow developers to add tools with better descriptions, icons, help topics, default actions, and versioning info, etc. The Extensibility methods and properties for integrating with the new Tool Manager derive from the IMapBasicApplication Interface. Under many of the Properties and Methods sections for IMapBasicApplication interface there is a MapBasic tab and in the remarks section an abbreviated method name can be used inside a MapBasic program as a declared Function or Sub to integrate a Tool into the Tool Manager of MapInfo Pro. These Properties and Methods will be detailed below:

IMapBasicApplication.IsAboutSupported

```
Sub AddIn_About
   Note "This is about a Sample MapBasic Application."
End Sub
```

IMapBasicApplication.DefaultCommandText

```
Function AddIn_DefaultCommandText() as string
   AddIn_DefaultCommandText = "Execute this command”
End Function
```

If function does not exist in MBX, or returns an empty string, then default text of ‘Default Command’ is used. This is the text shown for the default command context menu item.

IMapBasicApplication.Description

```
Function AddIn_Description() as String
   Note “This describes the tool in use”
End function
```

IMapBasicApplication.HasDefaultCommand

```
Sub AddIn_DefaultCommand
   call DoThisSub
   ‘This is a user sub or function called for running the actual tool code
End Sub
```

IMapBasicApplication.ImageUri

```
Function AddIn_ImageUri() As String
   AddIn_ImageUri =
"pack://application:,,,/MapInfo.StyleResources;component/Images/Application/runMapbasic_16x16.png"
End Function
```

This shows an icon in the Tool Manager for this tool

IMapBasicApplication.IsHelpSupported

```
Sub AddIn_Help
   Note "This is information to assist in running this tool"
 End Sub
```

IMapBasicApplication.Name

```
Function AddIn_Name() as string
   AddIn_Name = “Sample Application”
End Function
```

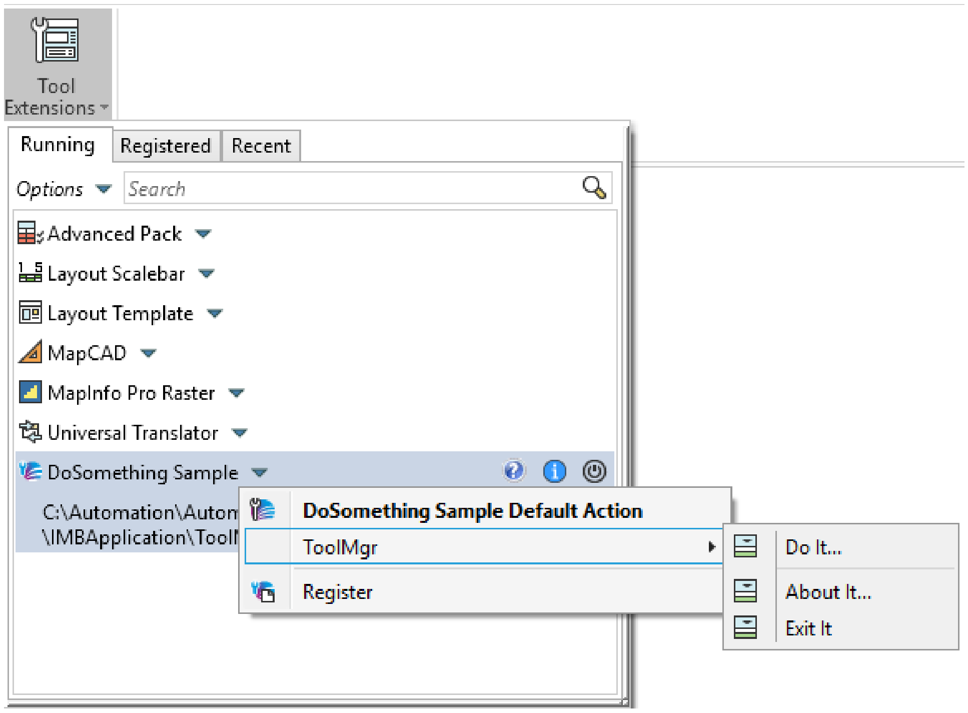
IMapBasicApplication.ShowInRunningPrograms

```
Function AddIn_HideRunningProgram() as Logical
   AddIn_HideRunningProgram = TRUE
End Function
```

NOTE: The ShowInRunningPrograms should only be used by MapInfoPro Runtime based applications that replace the MIPro UX so that these apps can't be unloaded accidentally via the Tool Manager.

IMapBasicApplication.Version

```
Function AddIn_Version() as string
       AddIn_Version = "1.0"
End Function
```



```
Include "mapbasic.def"
Include "IMapInfoPro.def"
Include "Enums.def"

Declare Sub Main
Declare Sub EndHandler
Declare Sub DoSomething()
Declare Function prompt_the_user() As Logical
Declare Sub About
Declare Sub bye
Declare Sub AddInLoadedHandler

Declare Sub AddIn_About
Declare Sub AddIn_Help
Declare Sub AddIn_DefaultCommand
Declare Function AddIn_Name() As String
Declare Function AddIn_Description() As String
Declare Function AddIn_Version() As String
Declare Function AddIn_ImageUri() As String
Declare Sub set_tools_menu(ByVal menuName as String)

Dim gsAppFilename, gsAppDescription as string
Dim mapinfoApplication, Ribbon, RibbonTabColl,  RibbonTab as This
Dim btnDoSomething, DoSomethingToolTip,  ribbonGroupsColl, ribbonDataGroup as This
Dim groupControlColl as This
Dim gVariable as Integer

'****************************************************************************
Sub Main()
onerror goto ErrorTrap
```

```
   Call RegisterUriParser(New_GenericUriParser(1), "pack", -1)

   'get MIPro interface
   mapinfoApplication = SystemInfo(SYS_INFO_IMAPINFOAPPLICATION)

   'Adding a separate ribbon button control to the Table Tab, Content section
   'Get Ribbon
   Ribbon = GetRibbon(mapinfoApplication)
   'Get RibbonTabs Collection
   RibbonTabColl = GetTabsColl(Ribbon)
   'Get Table Tab from Collection = #1
   RibbonTab = GetRbnTabCollItemInt (RibbonTabColl, 1)
   'Get the ribbon group collection.
   ribbonGroupsColl = GetRbnTabGrps(RibbonTab)
   'get group.
   ribbonDataGroup = GetRbnCtrlGrpCollItemInt (ribbonGroupsColl, 0)
   'Get Group controls collection
   groupControlColl = GetRbnCtrlGrpCtrls(ribbonDataGroup)
   'Add Control to this collection
   btnDoSomething = MICtrlCollAddStrStrInt(groupControlColl, "DoSomething", "Do Something",
ControlType_Button)

   call SetRbnBtnCtrlIsLarge(btnDoSomething, TRUE)
   'Create & Set the button tooltip
   DoSomethingToolTip = New_MapInfoRibbonToolTip()
   Call SetMIRbnToolTipToolTipText (DoSomethingToolTip, "Do Something")
   Call SetMIRbnToolTipToolTipDescription (DoSomethingToolTip, "Do Something")
   call SetMIRbnToolTipToolTipDisabledText (DoSomethingToolTip, "Should always be enabled")
   call SetRbnBtnCtrlToolTip(btnDoSomething, DoSomethingToolTip)

   'Set the button icon
   call SetRbnBtnCtrlLargeIcon(btnDoSomething,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/openUniversal_3
2x32.png", 0))
   call SetRbnBtnCtrlSmallIcon(btnDoSomething,
New_Uri("pack://application:,,,/MapInfo.StyleResources;component/Images/Application/openUniversal_1
6x16.png", 0))
   'Set Custom Handler to the button
   call SetRbnBtnCtrlCallingHandler (btnDoSomething, "DoSomething")
   gsAppFilename =  "ToolMgr.mbx"   ' name of MapBasic app file
   gsAppDescription = "ToolMgr"     ' short description of MB application
   Create Menu gsAppDescription as
      "&Do It..." Calling DoSomething,
       "(-",
       "&About It..." Calling about,
       "E&xit It" Calling bye

   Call set_tools_menu(gsAppDescription)
   Exit Sub
ErrorTrap:
   Note "Main: " + Err() + ": " + Error$()
   Resume Next
End Sub
'*********************************************************************
' Add a tools menu sub menu in the correct way
'*********************************************************************
Sub set_tools_menu(ByVal menuName as String)
  Alter Menu ID 4 Add
     menuName As MenuName
End Sub
'****************************************************************************
Sub DoSomething
Note "this sub launches the application"
End Sub
'****************************************************************************
```

```
'  About: This procedure displays an About dialog box.
Sub About
   Note "This Sample App demonstrates ToolManager code only"
End Sub
'****************************************************************************
'  BYE:  This procedure cleans up and shuts down the sample.
'        Called if the user chooses Exit from the DoSomething submenu. '
Sub Bye
    End Program
End Sub
'****************************************************************************
Sub AddInLoadedHandler
    print "DoSomething is Loaded"
End Sub
'****************************************************************************
Sub EndHandler
    dim bRet as logical
    bret = MICtrlCollRemove(groupControlColl, btnDoSomething)
    Ribbon = NULL_PTR
    RibbonTabColl = NULL_PTR
    RibbonTab = NULL_PTR
    ribbonGroupsColl = NULL_PTR
    ribbonDataGroup = NULL_PTR
    groupControlColl = NULL_PTR
    DoSomethingToolTip = NULL_PTR
    btnDoSomething= NULL_PTR
    end program
End Sub
'****************************************************************************
Function AddIn_Name() As String
AddIn_Name = "DoSomething Sample"
End Function
'****************************************************************************
Sub AddIn_About
    Note "This MapBasic application simply demonstrates ToolMgr code"
End Sub
'****************************************************************************
Sub AddIn_Help
    Note "Help info is found here"
End Sub
'****************************************************************************
Sub AddIn_DefaultCommand
    'Note "Default Command"
    call DoSomething
End Sub
'****************************************************************************
Function AddIn_ImageUri() As String
   AddIn_ImageUri =
"pack://application:,,,/MapInfo.StyleResources;component/Images/Application/runMapbasic_16x16.png"
End Function
'****************************************************************************
Function AddIn_Version() as string
    AddIn_Version = "1.0"
ENd Function
```