Creating a MicroStation CONNECT Edition WPF Addin

Downloading the SDK:

- First you MUST be a member of Bentley's Developer Network (BDN) in order to download the SDK. You must request to become a member. (Typically it's at no cost to DOTs)

  https://www.bentley.com/en/software-developers/bdn-inquiry-form

Use the Developers Shell:

After installing the SDK you will have the MicroStation Developer Shell (MicroStationDeveloperShell.bat). You will use this shell to set up the environment to accessing the SDK. Make sure to run it as admin.

Accessing Delivered Examples:

Examples can be found here.

C:\Program Files\Bentley\MicroStationCONNECTSDK\examples

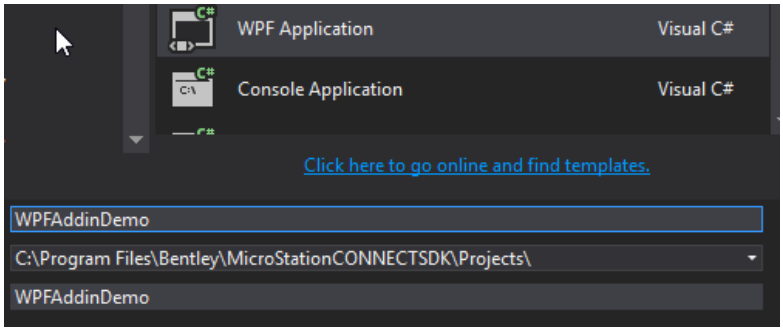
To build all examples run the buildallexamples.bat that's within the folder listed above from **WITHIN** the MicroStation Developer Shell.

Launch MicroStation Connect and use the Key-In dialog to load any of the examples. For example to load the WPFSample use the key-in MDL Load WPFSamples. This will load the Addin. Next use the Key-In browser and see all the available key ins for that Addin.

Steps to Create a WPF Addin.

This is the steps I figured out in order to get it to work. There could be a better way to do this.

- Open Visual Studio 2015 (this is the currently supported version of Visual Studio to use with the SDK) as Administrator as the file locations we will be saving to are inside the Program Files directory, which requires admin privileges.
- Create a new project, select WPF Application as the template under your desired programming language (I choose C#). Give your application a name and Location. I have made a Projects folder to store all my Addin.
- Open the properties for the newly created project and under the Application tab change the Output type from Windows Application to Class Library.



- Make sure the Application is targeting x64 (Properties > Build > General > Platform target:)
- To make debugging easy you can set the Output path to go to the mdlapps folder. Then when you build the addin the dll wont have to be copied to the directory so microstation can find it. An alternative would be to add the output directory to the **MS_ADDINPATH** variable. If you don't do one of those two options you will have a difficult time getting debugging to work. Below would be the Output Path (Properties > Build > Output > Output path:) to the mdlapps folder.

  "..\..\..\..\..\..\..\PROGRA~1\Bentley\MICROS~1\MICROS~1\mdlapps\"
- The WPFSample addin provided also added a Post-Build event command line parameter shown below. This will delete all files and subfolders within the obj folder after a successful build. Not sure why its there so I am NOT adding this but mentioning it for reference (Properties > Build Events > Post-build event command line)

  rd /s /q "$(SolutionDir)obj"
- Since we changed the output type to a class library we need to delete the App.xaml file and the App.config files. (these files define startup things which we don't need now that it's a dll.)
- Save and close Visual Studio.
- The examples have a batch file that is used to open the solution in visual studio. You could just open the solution directly from the MicroStation command shell instead of use the batch file. But the batch file ensures Visual Studio 2015 and Required Environment variables have been set. Below is the code to paste into the batch file (just replace the solution file name with whatever you named your project in the yellow highlighted areas below).

```
@echo off
@echo This batch file should be run from a VS2015 x64 Cross Tools Command Prompt
@echo Make sure that you have already run MicroStationDeveloperShell.bat in
@echo order to set the required environment variables.
IF DEFINED MSMDE GOTO checkforvs2015
ECHO MSMDE not defined. You need to run MicroStationDeveloperShell.bat first!
GOTO end
:checkforvs2015
IF DEFINED vs120comntools GOTO vardefined
ECHO Visual Studio 2015 is required for this project
GOTO end
:vardefined
rem At this point "vcvarsall.bat x86_amd64" or VS2015 x64 Cross Tools Command Prompt should already have been called
rem in order to compile successfully
@echo devenv.exe -useenv WPFAddinDemo.sln
start devenv.exe -useenv WPFAddinDemo.sln
:end
```

For ORD use this instead

```
@echo off
@echo This batch file should be run from a VS2015 x64 Cross Tools Command Prompt
@echo Make sure that you have already run OpenRoadsDesignerSDKDeveloperShell.bat in
@echo order to set the required environment variables.
IF DEFINED ORDE GOTO checkforvs2015
ECHO ORDE not defined. You need to run OpenRoadsDesignerSDKDeveloperShell.bat first!
GOTO end
:checkforvs2015
IF DEFINED vs120comntools GOTO vardefined
ECHO Visual Studio 2015 is required for this project
GOTO end
:vardefined
rem At this point "vcvarsall.bat x86_amd64" or VS2015 x64 Cross Tools Command Prompt should already have been called
rem in order to compile successfully
@echo devenv.exe -useenv WPFAddinDemo.sln
start devenv.exe -useenv WPFAddinDemo.sln
:end
```

- Next we will set up the make file and that the MicroStation command shell will use to build the Addin. The easiest way to do this is to copy one from the examples and modify it. I copied the

WPFSample.mke file and changed the name to whatever you named the project when you created it. I modified the file as shown below to make this step easy just copy the below text and change the line that says appName = ____ and make it equal to the name of your project.

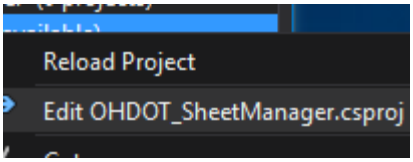
```
#--------------------------------------------------------------------------------
#
#    $Source: MstnExamples/WPF/WPFSample/WPFSample.mke $
#
#  $Copyright: (c) 2015 Bentley Systems, Incorporated. All rights reserved. $
#
#--------------------------------------------------------------------------------
DemoSrcDir   = $(_MakeFilePath)
PolicyFile = MicroStationPolicy.mki
appName    = WPFAddinDemo
MDLMKI = $(MSMDE)mki/
#--------------------------------------------------------------------------------------------
#  Includes
#--------------------------------------------------------------------------------------------
%include $(MDLMKI)mdl.mki
   SlnFile=$(appName).sln
%if $(TARGET_PROCESSOR_ARCHITECTURE)=="x64"
   Platform = x64
   OutSubDir = Winx64
%else
   Platform = x86
   OutSubDir = Winx86
%endif
%if $(TARGET_PROCESSOR_ARCHITECTURE)=="x64"
   OutSubDir = Winx64
%else
   OutSubDir = Winx86
%endif
#MSAddInTestProductDir = $(OutputRootDir)Product/Mstn/MicroStation/
always:
   | ************************************************
   | * SlnFile = $(SlnFile)
   | ************************************************
buildConfiguration = "$(Configuration)"
always:
   | ************************************************
   | * Processing $(appName) ...
   | * BuildConfiguration=$(buildConfiguration)
   | * ProductDir= $(MS)
   | ************************************************
#--------------------------------------------------------------------------------------------
#  Compile
#--------------------------------------------------------------------------------------------
%if defined(BMAKE_DELETE_ALL_TARGETS)
   Clean:
      devenv "$(SlnFile)" \/clean $(buildConfiguration)
%else
   %if defined (BMAKE_BUILD_ALL)
      BuildOp = \/rebuild
   %else
      BuildOp = \/build
   %endif
   BuildOrRebuild:
      devenv "$(SlnFile)" $(BuildOp) $(buildConfiguration)

%endif
```

- Now we are ready to open the solution via the MicroStation Command Shell (as admin). Switch to the directory of your project (use cd with the /d switch to change to a different mapped drive, ex. cd /d x:)

cd ..
cd Projects
cd WPFAddinDemo
openSln.bat (or you could just use WPFAddinDemo.sln)

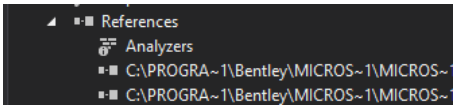
- Now we will add in some references to some MicroStation dlls. We might not need all of these referenced or might need more referenced but this is a good start see API hep doc for more info. To add these references we need to edit the csproj file. We can do this by right-clicking on the project and selecting unload project. Once unloaded you can right-click on the project again and select Edit (ProjectName).csproj.



Now in the csproj file find the ItemGroup that contains all the reference tags and add the following to that ItemGroup. **FOR ORD SDK switch $(MS) to $(ORD)\ everywhere seen below.**

```
<Reference Include="$(MS)ustation.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\Bentley.MicroStation.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\Bentley.MicroStation.WPF.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\Bentley.MicroStation.Interfaces.1.0.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.General.1.0.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.Windowing.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.Platform.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.General.1.0.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.ECObjects3.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.UI.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\ECFramework\Bentley.UI.Vendor.WPF.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Bentley.DgnPlatformNET.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\Bentley.RibbonView.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\Bentley.MicroStation.Ribbon.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Assemblies\Bentley.Interop.MicroStationDGN.dll">
  <Private>false</Private>
</Reference>
<Reference Include="$(MS)Bentley.DgnDisplayNet.dll">
  <Private>false</Private>
 </Reference>
```

Save the file, close it, then right-click on the project and select reload project. You should now see the references listed in the solution explorer under References.



- Now we need to set up the project so MicroStation can interact with it. Following the SDK examples we will start with creating a class named Addin.cs. Make sure you have the following using statements.

```
using System;
using System.Resources;
```

- Next copy and paste the below code into your Addin.cs class.
  - Make sure to change your namespace back to your projectname.
  - Everywhere you see WPFAddinDemoApp switch to an app name you want.

```
namespace WPFAddinDemo
{
 /*=============================================================================================**/
    ///
    /// <summary>Singleton AddIn class</summary>
    ///
/*==============+===============+===============+===============+===============+======*/
    [Bentley.MstnPlatformNET.AddInAttribute(MdlTaskID = "WPFAddinDemoApp")]
    public class WPFAddinDemoApp : Bentley.MstnPlatformNET.AddIn
    {
        private static WPFAddinDemoApp s_WPFAddinDemoApp;
        private static ResourceManager s_ResourceManager;
        /*---------------------------------------------------------------------------------**/
        /// <summary>Constructor for the AddIn</summary>
        /*-------+---------------+---------------+---------------+---------------+------*/
        private WPFAddinDemoApp
        (
        IntPtr mdlDesc
        )
            : base(mdlDesc)
        {
        }
        /*---------------------------------------------------------------------------------**/
        /// <summary>The Run method</summary>
        /*-------+---------------+---------------+---------------+---------------+------*/
        protected override int Run
        (
        string[] commandLine
        )
        {
            // save a reference to our addin to prevent it from being garbage collected.
            s_WPFAddinDemoApp = this;
            // Get the localized resources
            s_ResourceManager = Properties.Resources.ResourceManager;
            return 0;
        }
        /*---------------------------------------------------------------------------------**/
        /// <summary>Static method to get the AddIn</summary>
        /*-------+---------------+---------------+---------------+---------------+------*/
        internal static WPFAddinDemoApp Instance
        {
            get { return s_WPFAddinDemoApp; }
        }
        /*---------------------------------------------------------------------------------**/
        /// <summary>Static method to get the ResourceManager</summary>
        /*-------+---------------+---------------+---------------+---------------+------*/
        internal static ResourceManager ResourceManager
        {
            get { return s_ResourceManager; }
        }
    } // WPFAddinDemoApp class
} // namespace WPFAddinDemo
```

- Next we will set up some basic keyins for the addin. To set up keyins you need a keyins class and a xml file. First lets set up the keyin class by creating a new class named Keyins.cs. I just have two keyins, one for open and the other for close. The code is listed below. NOTE you will have errors until we add in the actual open and close methods.

```
class Keyins
    {
        /*-----------------------------------------------------------------------------------**/
        /// <summary>Open method for the keyin</summary>
        /*-------------+---------------+---------------+---------------+---------------+------*/
        public static void Open (string unparsed)
        {
            MainWindow.OpenWindow(unparsed);
        }

        /*-----------------------------------------------------------------------------------**/
        /// <summary>Close method for the keyin</summary>
        /*-------------+---------------+---------------+---------------+---------------+------*/
        public static void Close(string unparsed)
        {
            MainWindow.CloseWindow(unparsed);
        }
    }
```

- Now we need to create a xml file, following way the SDK examples named the file I named it WPFAddinDemo.commands.xml. this file sets up the keys in that you type into the MicroStation key in window and tells it what function to run, which points to the keyins class. To add a xml file right-click on the project and select Add > New Item. In the dialog that opens find XML File and select it and set the name and hit add. (As with everything else in the document change WPFAddinDemo to the desired name (projectname)). Copy and paste the code below into the xml file, again change everywhere you see WPFAddinDemo to your app name.

```
<?xml version="1.0" encoding="utf-8" ?>
<KeyinTree xmlns="http://www.bentley.com/schemas/1.0/MicroStation/AddIn/KeyinTree.xsd">
<RootKeyinTable ID="root">
<Keyword SubtableRef="props" CommandClass="MacroCommand" CommandWord="WPFAddinDemo">
<Options Required="true"/>
</Keyword>
</RootKeyinTable>

<SubKeyinTables>
<KeyinTable ID="props">
<Keyword CommandWord="OPEN" />
<Keyword CommandWord="CLOSE" />
</KeyinTable>
</SubKeyinTables>

<KeyinHandlers>
<KeyinHandler Keyin="WPFAddinDemo OPEN"   Function="WPFAddinDemo.Keyins.Open" />
<KeyinHandler Keyin="WPFAddinDemo CLOSE"  Function="WPFAddinDemo.Keyins.Close" />
</KeyinHandlers>
</KeyinTree>
```

- Next we need to fix the way this xml file is defined in the csproj file. The API Help doc says to add it as a Deflated EmbeddedResource but the WPFSample has it a just a Embedded Resource, so I am setting it to EmbeddedResource. To do this Right-click on the xml file and select properties. In the properties window set the Build Action to Embedded Resource.

- In addition to setting the xml file as an embedded resource we need to give it a logical name and subtype tag in order for MicroStation to find it and actually have the keyins show up in the keyin

browser. To do this we need to edit the csproj file again. Right-click on the project and unload the project again and then right-click and select edit …csproj. Find the item group that lists the EmbeddedResource and add the LogicalName and SubType tags listed below.

```
<ItemGroup>
    <EmbeddedResource Include="WPFAddinDemo.commands.xml" >
      <LogicalName>CommandTable.xml</LogicalName>
      <SubType>Designer</SubType>
    </EmbeddedResource>
  </ItemGroup>
```

- Now we need to add some code to attach the wpf window to the PowerPlatform. See API help doc for more info.
    - Add this using statement `using Bentley.MstnPlatformNET.WPF;`
    - Create a static variable for you window and another for the InteropHelper inside your MainWindow class.

```
public partial class MainWindow : Window
    {
        private static MainWindow s_window;
        private WPFInteropHelper m_wndHelper;
...
```

    - Now we will initialize the interophelper in the MainWindow method. Notice we are attaching to the class we made in the Addin.cs.

```
public MainWindow()
{
        InitializeComponent();
        /// PowerPlatform Integration
        // Create the MicroStation Interop Helper and Attach the Window
        m_wndHelper = new WPFInteropHelper(this);
        m_wndHelper.Attach(WPFAddinDemoApp.Instance, true, "WPFAddinDemoWindow");
}
```

- Almost done!! We still need to define the open and close methods the the keyins.cs are calling. This will be the methods that will specify how to open and close the main window.

```
/*--------------------------------------------------------------------------------------**/
/// <summary>Creates and opens the Window</summary>
/*--------------+---------------+---------------+---------------+---------------+------*/
public static void OpenWindow(string unparsed)
{
    if (null == s_window)
    {
        s_window = new MainWindow();
        s_window.Show();
    }
}
/*--------------------------------------------------------------------------------------**/
/// <summary>Closes the Window</summary>
/*--------------+---------------+---------------+---------------+---------------+------*/
public static void CloseWindow(string unparsed)
{
    if (null != s_window)
    {
        s_window.Close();
    }
}
```

- Lastly we want to make sure we properly detach from the power platform so we will override the OnClosed method to ensure this happens.

```
/*--------------------------------------------------------------------------------------**/
/// <summary>React to the Window closed</summary>
/*--------------+---------------+---------------+---------------+---------------+------*/
protected override void OnClosed(EventArgs e)
{
        base.OnClosed(e);
        // PowerPlatform Integration
        m_wndHelper.Detach();
        m_wndHelper.Dispose();
        s_window = null;
}
```

- We are now ready to build and debug the addin. Remember all builds should be done from the MicroStation command shell using the mke file.
    - To do a release build use the command, *bmake WPFAddinDemo.mke*
    - To do a debug build use the command, *bmake -a -ddebug WPFAddinDemo*
    - To make debugging easy define a start up application for the project. In the Projects Properties > Debug > Start Action. Select Start external program as MicroStation.
      C:\Program Files\Bentley\MicroStation CONNECT Edition\MicroStation\microstation.exe
      Now hit the Start button in visual studio and it should start up microstation. Set a break point somewhere in the project and use the following keys in to load the addin.
      Mdl load WPFAddinDemo
      WPFAddinDemo Open
    - To debug inside projectwise launch MicroStation from projectwise then in visual studio select Debug > Attach to Process… and select the running instance of microstation.

  NOTE that you will probably need to launch the MicroStation command shell NOT as admin and open your solution (NON admin) in order for you debugging to work properly. OR map your network drives as admin.

Dockable Windows

To create a dockable window the process is slightly different. The main difference is that a dockable window MUST be a UserControl. So we make a class that inherits DockableWindow. This Dockable Window is in the Bentley.MstnPlatformNET.WPF namespace. Now that our class is inheriting DockableWindow the attach method is within the class, so we don't need to define a WPFInteropHelper. Instead just set the Attach and Title on the class itself.

This section goes over how to set up a dockable window. To setup the project follow the steps listed in the Steps to Create a WPF Addin section.

- In your project add a user control, I named mine DockableContent, add some elements to it so we can see it in microstation.
- Create a new class file and name it as you wish. This class file will be the class that inherits DockableWindow, I named mine dockwindow.
- Add the DockableWindow inheritance to the class. (need `using Bentley.MstnPlatformNET.WPF;`)

```
class dockwindow : DockableWindow
```

- Inside the class create a static variable to hold the instance of the class. We will also set the content to our usercontrol and attach it to the powerplatform in the class constructor. Make sure to add a reference to the System.Drawing assembly and add this using statement

```
class dockwindow : DockableWindow
{
    static private dockwindow s_dockwindow;
    public dockwindow()
    {
        var userControl = new DockableContent();
        this.Content = userControl;

        this.Title = "Dockable Window";
        this.Attach(WPFAddinDemoApp.Instance, "dockWindow", new
System.Drawing.Size((int)userControl.MinWidth, (int)userControl.MinHeight));
    }
}
```

- Next add in the open and close methods and the onclosed override method just as we did in the Steps to Create a WPF addin section.

```
/*----------------------------------------------------------------------------------**/
/// <summary>Creates and opens the Window</summary>
/*------------+---------------+---------------+---------------+---------------+------*/
public static void OpenWindow(string unparsed)
{
    if (null == s_dockwindow)
    {
        s_dockwindow = new dockwindow();
        s_dockwindow.Show();
    }
}

/*----------------------------------------------------------------------------------**/
/// <summary>Closes the Window</summary>
/*------------+---------------+---------------+---------------+---------------+------*/
public static void CloseWindow(string unparsed)
{
    if (null != s_dockwindow)
    {
        s_dockwindow.Close();
    }
}
/*----------------------------------------------------------------------------------**/
/// <summary>React to the Window closed</summary>
/*------------+---------------+---------------+---------------+---------------+------*/
protected override void OnClosed(EventArgs e)
{
    base.OnClosed(e);
    // PowerPlatform Integration
    this.Detach();
    this.Dispose();
    s_dockwindow = null;
}
```

- Now just as we did in the Steps to Create a WPF addin section we need to add to our Keyins class to call these open and close methods.

```
public static void OpenDock(string unparsed)
{
    dockwindow.OpenWindow(unparsed);
```

```
        }

        public static void CloseDock (string unparsed)
        {
            dockwindow.CloseWindow(unparsed);
        }
```

- And lastly we need to add these keyins to the xml file.

```
<?xml version="1.0" encoding="utf-8" ?>
<KeyinTree xmlns="http://www.bentley.com/schemas/1.0/MicroStation/AddIn/KeyinTree.xsd">
  <RootKeyinTable ID="root">
    <Keyword SubtableRef="props" CommandClass="MacroCommand" CommandWord="WPFAddinDemo">
      <Options Required="true"/>
    </Keyword>
  </RootKeyinTable>

  <SubKeyinTables>
    <KeyinTable ID="props">
      <Keyword CommandWord="OPEN" />
      <Keyword CommandWord="CLOSE" />
      <Keyword CommandWord="OPENDOCK" />
      <Keyword CommandWord="CLOSEDOCK" />
    </KeyinTable>
  </SubKeyinTables>

  <KeyinHandlers>
    <KeyinHandler Keyin="WPFAddinDemo OPEN"    Function="WPFAddinDemo.Keyins.Open"  />
    <KeyinHandler Keyin="WPFAddinDemo CLOSE"   Function="WPFAddinDemo.Keyins.Close" />
    <KeyinHandler Keyin="WPFAddinDemo OPENDOCK" Function="WPFAddinDemo.Keyins.OpenDock" />
    <KeyinHandler Keyin="WPFAddinDemo CLOSEDOCK" Function="WPFAddinDemo.Keyins.CloseDock" />
  </KeyinHandlers>
</KeyinTree>
```

Dockable ToolBars

Setting up a Dockable ToolBar is a similar process as setting up a dockable Window. Both use UserControls. But instead of inheriting a DockableWindow it inherites a DockableTookbar. Attaching to the power platform is a little bit trickier as well.

This section goes over how to set up a dockable window. To setup the project follow the steps listed in the Steps to Create a WPF Addin section.

- In your project add a user control, I named mine docktoolbarContent.
  - Set the DesignHeight to 24 and add some content.
- Create a new class file and name it as you wish, I named mine dockToolbar. This class file will be the class that inherits DockableToolbar
- Add the DockableToolbar inheritance to the class. (need `using Bentley.MstnPlatformNET.WPF;`)

```
class dockwindow : DockableToolbar
```

- This class also needs to inherite a IGuiDockable (need `using BMG = Bentley.MstnPlatformNET.GUI;`)

```
class dockToolbar : DockableToolbar, BMG.IGuiDockable
```

- Inside the class create a static variable to hold the instance of the class. We will also set the content to our usercontrol and attach it to the powerplatform in the class constructor. Make sure to add a reference to the System.Drawing and System.Windows assemblies. We will add in the missing methods in the next step.

```
class dockToolbar : DockableToolbar, BMG.IGuiDockable
{
    static private dockToolbar s_dockToolbar;
```

```
        public dockToolbar()
        {
            var toolbarControl = new docktoolbarContent(); //usercontrol
            toolbarControl.VerticalContentAlignment = System.Windows.VerticalAlignment.Center;
            this.Content = toolbarControl;

            this.Title = "Dockable Toolbar";
            this.AttachingToHost += new BMG.AttachingToHostEventHandler(dockToolbar_AttachingToHost);
            this.DetachingFromHost += new EventHandler(dockToolbar_DetachingFromHost);

            this.Attach(WPFAddinDemoApp.Instance, "dockToolbar");
        }
    }
```

- We need to add in the attachingtohost and detachingfromhost methods. You can add to these methods as needed.

```
    void dockToolbar_AttachingToHost(object sender, BMG.AttachingToHostEventArgs e)
    {
        e.AttachPoint = new System.Drawing.Point(0, 0);
        //e.Handled = true;
        System.Diagnostics.Debug.WriteLine("dockToolbar_AttachingToHost");
    }
    void dockToolbar_DetachingFromHost(object sender, EventArgs e)
    {
        System.Diagnostics.Debug.WriteLine("dockToolbar_DetachingFromHost");
    }
```

- We still have an error with the inherited BMG.IGuiDockable. We need to implement some interfaces to handle the docking.

```
    #region IGuiDockable Members
    private System.Drawing.Size m_rejectedSize = System.Drawing.Size.Empty;
    public bool GetDockedExtent(BMG.GuiDockPosition dockPosition, ref BMG.GuiDockExtent extentFlag, ref
System.Drawing.Size dockExtent)
    {
        dockExtent.Height = this.CommonDockSize.Height;
        if (dockPosition == BMG.GuiDockPosition.Top ||
            dockPosition == BMG.GuiDockPosition.Bottom)
        {
            dockExtent.Width = (int)this.ActualWidth;
            extentFlag = BMG.GuiDockExtent.Specified;
        }
        else if (dockPosition == BMG.GuiDockPosition.NotDocked)
            extentFlag = BMG.GuiDockExtent.Specified;
        else
            extentFlag = BMG.GuiDockExtent.InvalidRegion;
        return true;
    }
    public bool WindowMoving(BMG.WindowMovingCorner corners, ref System.Drawing.Size newSize)
    {
        newSize.Height = CommonDockSize.Height;
        if (corners != BMG.WindowMovingCorner.LowerRight || m_rejectedSize.Equals(newSize))
        {
            m_rejectedSize = newSize;
            newSize.Width = (int)this.ActualWidth;
        }
        return true;
    }
    #endregion
```

- Next add in the open and close methods and the onclosed override method just as we did in the Steps to Create a WPF addin section.

```
        public static void OpenWindow(string unparsed)
        {
                if (null == s_dockToolbar)
                {
```

```
                s_dockToolbar = new dockToolbar();
                s_dockToolbar.Show();
            }
        }
        public static void CloseWindow(string unparsed)
        {
            if (null != s_dockToolbar)
            {
                s_dockToolbar.Close();
            }
        }
        protected override void OnClosed(EventArgs e)
        {
            base.OnClosed(e);
            // PowerPlatform Integration
            this.Detach();
            this.Dispose();
            s_dockToolbar = null;
        }
```

- Now just as we did in the Steps to Create a WPF addin section we need to add to our Keyins class to call these open and close methods.

```
public static void OpenDockToolbar(string unparsed)
 {
     dockToolbar.OpenWindow(unparsed);
 }

 public static void CloseDockToolbar(string unparsed)
 {
     dockToolbar.CloseWindow(unparsed);
 }
```

- And lastly we need to add these keyins to the xml file.

```
<?xml version="1.0" encoding="utf-8" ?>
<KeyinTree xmlns="http://www.bentley.com/schemas/1.0/MicroStation/AddIn/KeyinTree.xsd">
  <RootKeyinTable ID="root">
    <Keyword SubtableRef="props" CommandClass="MacroCommand" CommandWord="WPFAddinDemo">
      <Options Required="true"/>
    </Keyword>
```

```
  </RootKeyinTable>

  <SubKeyinTables>
    <KeyinTable ID="props">
      <Keyword CommandWord="OPEN" />
      <Keyword CommandWord="CLOSE" />
      <Keyword CommandWord="OPENDOCK" />
      <Keyword CommandWord="CLOSEDOCK" />
      <Keyword CommandWord="OPENDOCKTOOLBAR" />
      <Keyword CommandWord="CLOSEDOCKTOOLBAR" />
    </KeyinTable>
  </SubKeyinTables>

  <KeyinHandlers>
    <KeyinHandler Keyin="WPFAddinDemo OPEN"   Function="WPFAddinDemo.Keyins.Open" />
    <KeyinHandler Keyin="WPFAddinDemo CLOSE"  Function="WPFAddinDemo.Keyins.Close" />
    <KeyinHandler Keyin="WPFAddinDemo OPENDOCK"   Function="WPFAddinDemo.Keyins.OpenDock" />
    <KeyinHandler Keyin="WPFAddinDemo CLOSEDOCK"  Function="WPFAddinDemo.Keyins.CloseDock" />
    <KeyinHandler Keyin="WPFAddinDemo OPENDOCKTOOLBAR" Function="WPFAddinDemo.Keyins.OpenDockToolbar" />
    <KeyinHandler Keyin="WPFAddinDemo CLOSEDOCKTOOLBAR" Function="WPFAddinDemo.Keyins.CloseDockToolbar" />
  </KeyinHandlers>
</KeyinTree>
```

Tool Settings Host

ToolSettingsHost allows a UserControl to be placed into the Tool Settings Window when a tool is started. Currently, the Bentley.Interop.MicroStationDGN.IPrimitiveCommandEvents is subclassed to create a managed tool. But a DgnPrimitiveTool class will be available in "DgnDisplayNET" soon. An example will be provided at that time.

For this section we will use the same usercontrol we made for the Dockable Window section.

- Create a new class file, I named mine ToolSettings.
- Add `using BMI = Bentley.MstnPlatformNET.InteropServices;`
- Add `using Bentley.MstnPlatformNET.WPF;`
- Our class will inherit the Bentely.Interop.MicroStationDGN.IPrimitiveCommandEvents

```
class ToolSettings : Bentley.Interop.MicroStationDGN.IPrimitiveCommandEvents
```

- This part gets a little tricky. We will once again create our static variable to hold our class but the class constructor will set up a hosting class. This hosting class will inherite ToolSettingsHost from the Bentely.MstnPlatformNET.WPF namespace.
  So First lets add our public variables

```
class ToolSettings : Bentley.Interop.MicroStationDGN.IPrimitiveCommandEvents
{
    public static ToolSettings s_toolsettings;
    public ToolSettingsHost ToolSettingsHost { get; private set; }
   …
```

  Next we set up a method to initialize the ToolSettingsHost

```
public ToolSettings()
{
    ToolSettingsHost = new MyToolSettingsHost(this);
}
```

Next we add a class the inherits the ToolSettingsHost, which is what we are initializing the public ToolSettingsHost variable with. This subclass has a parameter for our main class. This is also where we add in our user control.

```
class MyToolSettingsHost : ToolSettingsHost
{
    private ToolSettings m_toolSettings;

    public MyToolSettingsHost(ToolSettings thismainclass)
    {
        m_toolSettings = thismainclass;

        var userControl = new DockableContent();
        this.Content = userControl;
        this.Title = "Tool Settings Dialog Takeover";
    }
};
```

- Now similar to the previous section, we need to implement the interfaces for the IPrimitiveCommandEvents

```
#region IPrimitiveCommandEvents Members
public void Start()
{
    ToolSettingsHost.Attach(WPFAddinDemoApp.Instance);
}
public void Cleanup()
{
    ToolSettingsHost.Detach();
    ToolSettingsHost.Dispose();
    ToolSettingsHost = null;
    s_toolsettings = null;
}
public void DataPoint(ref Bentley.Interop.MicroStationDGN.Point3d Point,
Bentley.Interop.MicroStationDGN.View View)
{
}
public void Dynamics(ref Bentley.Interop.MicroStationDGN.Point3d Point,
Bentley.Interop.MicroStationDGN.View View, Bentley.Interop.MicroStationDGN.MsdDrawingMode DrawMode)
{
}
public void Keyin(string Keyin)
{
}
public void Reset()
{
}
#endregion
```

- Now unlike the previous sections we will add one method for running this instead of an open, close, and onclosed methods.

```
public static void Run(string unparsed)
 {
    if (null == s_toolsettings)
    {
```

```
                s_toolsettings = new ToolSettings();
                BMI.Utilities.ComApp.CommandState.StartPrimitive(s_toolsettings, false);
            }
        }
```

- Now we need to define how to call this from a keyin by adding it to the Keyins class.

```
public static void OpenTool(string unparsed)
{
    ToolSettings.Run(unparsed);
}
```

- Lastly we need to add this new keyin to the xml file.

```
<?xml version="1.0" encoding="utf-8" ?>
<KeyinTree xmlns="http://www.bentley.com/schemas/1.0/MicroStation/AddIn/KeyinTree.xsd">
  <RootKeyinTable ID="root">
    <Keyword SubtableRef="props" CommandClass="MacroCommand" CommandWord="WPFAddinDemo">
      <Options Required="true"/>
    </Keyword>
  </RootKeyinTable>

  <SubKeyinTables>
    <KeyinTable ID="props">
       ......
      <Keyword CommandWord="OPENTOOL" />
    </KeyinTable>
  </SubKeyinTables>

  <KeyinHandlers>
      ......
    <KeyinHandler Keyin="WPFAddinDemo OPENTOOL"  Function="WPFAddinDemo.Keyins.OpenTool" />
  </KeyinHandlers>
</KeyinTree>
```

Other WPF capabilities

- **MstnRelayCommand** - provides an MVVM RelayCommand wrapper for an IMstnCommand, which is located in the MstnPlatformNET.Commands namespace.
- **RibbonBar** - the RibbonBar interface is supported and uses Telerik WPF RadRibbonView. When using the RibbonBar interface, the PowerPlatform main window is switched to a RadRibbonWindow with a child RadRibbonView. Using this RibbonBar class, applications may add their own tabs, groups and controls to the Ribbon. For more information, see @ RibbonBarInterface "Ribbon Bar Interface"
- **Icon** - provides interop with the named icon system in the Dialog Manager
- **ViewOverlay / ViewWidget (In development)** - provides a mechanism of overlaying a WPF window over a PowerPlatform View window as an overlay. The View Overlay may contain controls, which are then treated as View Widgets. When the widgets are not active, they are drawn into the View using QuickVision.

MVVM Setup

Coming Soon. But note additional steps need to be taken if you have elements that contain a <TransKit>

Contacts

For any questions, suggestions, or problems with this document please contact the ODOT Office of CADD and Mapping Services by use of the following form on the ODOT website:

https://odot.formstack.com/forms/cadd_servicerequest