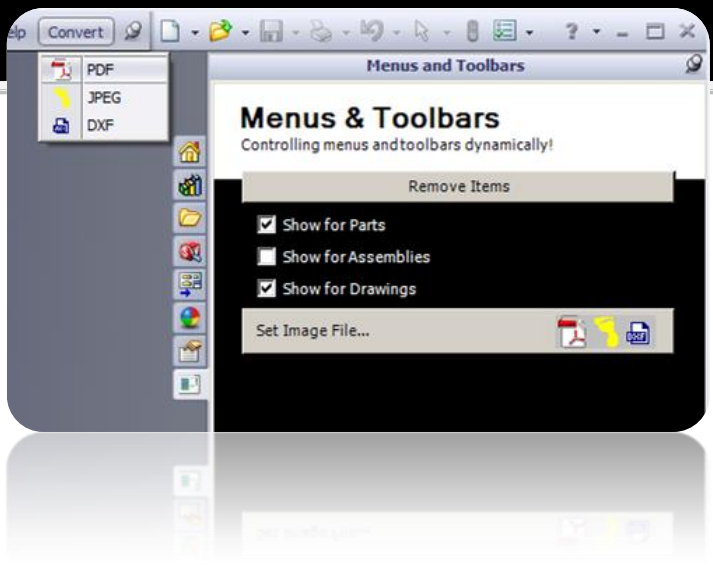
Menus & Toolbars

angelsix.com



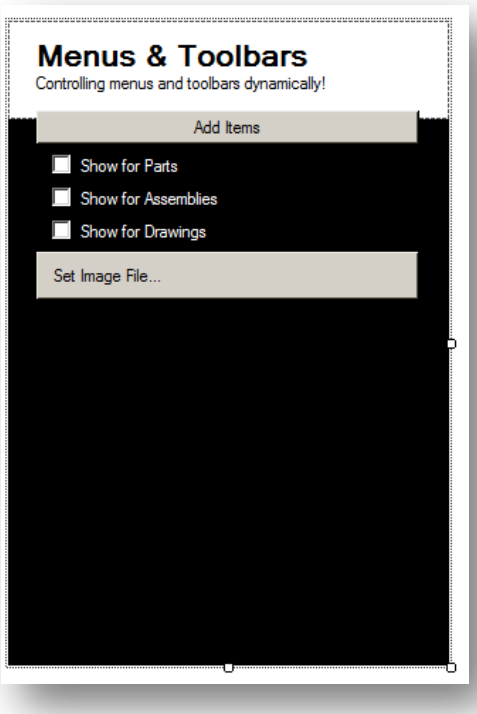
OK if you haven't done so already read the first tutorial on creating a SolidWorks Add-in or if you already know how to then download the code files from the tutorial and open them up. We will start with the add-in from the previous tutorial as the base, so we already have an add-in that loads a Taskpane for us.

So we can get some dynamic control over the menus and toolbars we will add controls to our Taskpane to do all the work.

Overview

What this tutorial is going to show you how to do is create command items (items that serve a dual purpose as a menu item and/or a toolbar item), and then to toggle the visibility settings for the items based on model types, and to dynamically change the icon file used in the toolbar.

Creating the UI



Open up the Visual Studio Solution, and open up the SWTaskPaneHost user control and delete any existing controls.

The command items are going to be hard-coded to begin with to allow easier understanding of the code to create menu and toolbar controls and to not confuse them with the code that is used to create them from user information dynamically.

We provide the user with an option of where to show the items based on the model type and to select an image to use for them.

Create a button on the control and give it the text "Add Items" and the name bAddItems, a button with the text "Set Image File..." and the name bSetImageFile, and then add 3 checkboxes and give each the text "Show for Parts", "Show for Assemblies" and "Show for Drawings" and

names cbShowForParts, cbShowForAssemblies and cbShowForDrawings respectively.

To add event handlers to the controls (functions that get called on specific events) we can do it simply in this case by double-clicking each control. By default, Visual Studio will add an event handler for the default event of the control when it is double-clicked. In the case of a button it is the Click event, and in the case of a Checkbox it is the CheckedChanged event. The Click event of the button is the one we need.

Double-click the Add Items button and it will take you to the code file in the newly created function. Go back to the form designer and repeat the step for the Set Image File button control and once done you will have the following new functions in the code file ready for use:

```
private void bAddItems_Click(object sender, EventArgs e)
{

}

private void bSetImageFile_Click(object sender, EventArgs e)
{

}
```

Getting Connected

Before we can do much in our host control we need to get access to SolidWorks - to do this add a new function called Connect that requires a SolidWorks object and an integer to be passed in. Also add new private variables to store the passed in value, so all in all it will look like this:

```
private SldWorks mSWApplication;
private int mCookie;


public void Connect(SldWorks swApp, int cookie)
{
    // Store the SolidWorks instance passed in
    mSWApplication = mSWApplication;
    mCookie = cookie;
}
```

Now we have access to the SolidWorks application and cookie once this function gets called. We must call this function from the main SolidWorks Add-in class to pass in the SolidWorks application and cookie.

Open up the SWIntegration code file and in the UISetup function where we create a new instance of our Taskpane control, change the Taskpane Title parameter, and add an extra line to call the Connect function and pass in the active SolidWorks application and cookie variables like so:

```
private void UISetup()
{
    mTaskpaneView = mSWApplication.CreateTaskpaneView2(string.Empty, "Menus And
Toolbars");
    mTaskpaneHost =
(SWTaskpaneHost)mTaskpaneView.AddControl(SWTaskpaneHost.SWTASKPANE_PROGID, "");
    mTaskpaneHost.Connect(mSWApplication, mSWCookie);
}
```

With the Connect function called as soon as the control is added to the Taskpane means we are always ready to use the SolidWorks instance anywhere in our Taskpane class for the events we added, as the UI events can only be called after the host is added so there is no need for any sort of checking whether the variable is initialized or not.

The Initial State

Back in the SWTaskpaneHost code file; change the constant SWTASKPANE_PROGID to a new name so it doesn't clash with the other tutorial as it is a new add-in. I gave it the new name of "AngelSix.SWTaskPane_MenusAndToolbars".

When our Taskpane first loads there will be no menus added as we haven't done any code for that yet. We want the user to select the settings for the menu (the 3 checkboxes and the image) and then click the Add Items button in order to add the menu items.

What isn't very well documented in SolidWorks is the fact that once you create a Command Group and call the Activate function on it (which sets it up and shows it based on your settings) you cannot then edit anything about the Command Group. Instead you have to completely remove and re-add the group to do this. So if you were to create some form of tool for the user it may be of benefit to make note of or warn them that they must set the information before adding the items. Alternately you can just automatically remove and re-add the group every time a settings changed (by monitoring the CheckChanged events and Click events and calling the Remove and Add functions to remove and re-add based on the new UI settings), it is left up to you. For this tutorial we will just provide the functionality to the Add Items button.

If you were to launch our add-in now (feel free to test it at this point) the Taskpane would be loaded, and all the UI items would be visible, but with no functionality.

Once they click the Add Items button, we want to add the SolidWorks menu and toolbar items using the settings and image the user have specified. Once the items are added we no longer want the user to be able to re-add the same items. We could just disable the button to prevent and further clicks, however to show more functionality of the SolidWorks code we will change the function of the button to be a Remove Items button, and if they clicked it again, it would remove the items from SolidWorks.

Because we are adding a dual function to the Add Items button, when it is clicked we need to know what state we are in as to whether we should be adding or removing items, and also to change the text of the button. When the control is first created no items will exist so the button will be in an Add state. To signify this, add a new private variable to the SWTaskpaneHost file called mCommandItemsAdded. The default value of an uninitialized Boolean is false, and because this variable is for checking whether the command items are added yet (which they are not) then it is already in the correct state of false:

```
private bool mCommandItemsAdded;
```

Go to the event handler for when the Add Items button is clicked and let's add the code and functions to give us the desired functionality:

```
private void bAddItems_Click(object sender, EventArgs e)
{
    // Have we added items yet?
    if (mCommandItemsAdded)
    {
        // Yes, so this click is to remove them
        RemoveCommandItems();
        // Update the text and variable
        bAddItems.Text = "Add Items";
        mCommandItemsAdded = false;
    }
    else
    {
        // No, so this click is to add them
        AddCommandItems();
        // Update the text and variable
        bAddItems.Text = "Remove Items";
        mCommandItemsAdded = true;
    }
}
```

The first thing we do when the button is clicked is to check the current state of the variable to see if the command items have been added already. If they have then we call the function RemoveCommandItems, change the button text to "Add Items" as that is what the next click will do, and update the variable to indicate there are no items added so on the next click we know the new state. If the items are not already added we do the opposite and add them, change the text to Remove Items and update the variable. That is all the functionality we need for the button so it is time to create the AddCommandItems and RemoveCommandItems functions.

Adding and Removing the Menus & Toolbars

The main part of this tutorial happens here – we dynamically add and remove menu and toolbar items.

There are several ways to add menu and toolbar items in SolidWorks including functions like AddToolbar4, AddMenu and AddMenuPopupItem3 etc… But they are all pretty messy and unmanageable for dynamic creation and are really more of a legacy thing. The best way to control UI content in SolidWorks is to use the Command Manager. Not only can a Command item be invoked in API using the RunCommand method, but they can be added as menu and/or toolbar items, enabled/disabled and displayed based on model type and other parameters so are very configurable.

For this first tutorial we will cover the adding/removing and displaying based on model types and enable/disable the items. Once you understand this it should be very easy to expand to register them to be run via the RunCommand.

To add command items we first need the Command Manager from SolidWorks. We could get a hold of this when the user clicks the Add Items button, but that would be a waste of processing time as we only need to get a hold of it once, not every time we want to use it. This would be similar to opening a new instance of SolidWorks every time you wanted to create a part.

A good place to get hold of the Command Manager would be when we get passed on the SolidWorks instance and the cookie. Add a private variable and get the command manager like so:

```
private CommandManager mCommandManager;

public void Connect(SldWorks swApp, int cookie)
{
    // Store the SolidWorks instance passed in
    mSWApplication = swApp;
    mCookie = cookie;

    // One-time get the command manager for our add-in
    mCommandManager = mSWApplication.GetCommandManager(mCookie);
}
```

Simple enough – now we are free to use the mCommandManager object when the user clicks the Add Items button.

The Add Items functions

When the user clicks the Add Items button for the first time our code above will call into the AddItems function which we haven't created yet, so let's create that now.

Start by added a few new variables to the class to keep track of the Command Group information:

```
private CommandGroup mCommandGroup;
private int mCommandGroupId = 1;
private string imageLocation;
```

The CommandGroup object is to store the group that gets created, which effectively represents a main menu top-level item like File, Edit, Tools etc… or a Toolbar such as Layers or Features.

The integer is a unique ID that is set and used by ourselves when adding/removing the command group from the manager.

The string is the location of the image to use when creating the command group that will be used for the menu and toolbar items. This gets set later when we implement the Set Image File button.

When creating a new Command Group we need to set the image location, which is a string, and the type of documents it should be displayed in, which is of type swDocTemplateTypes_e in the SwConsts library. So to keep the UI settings clean from the code that creates command groups we will use the AddCommandItems function to get the UI information and convert it into the information required by the Command Group, and then pass that into another function that takes those parameters and creates the group.

Create the AddCommandItems function and retrieve the UI settings information like so:

```
private void AddCommandItems()
{
    // Show when nothing is displayed by default
    int docDisplay = (int)swDocTemplateTypes_e.swDocTemplateTypeNONE;
    // If selected, show for parts
    if (cbShowForParts.Checked)
        docDisplay |= (int)swDocTemplateTypes_e.swDocTemplateTypePART;
    // If selected, show for assemblies
    if (cbShowForAssemblies.Checked)
        docDisplay |= (int)swDocTemplateTypes_e.swDocTemplateTypeASSEMBLY;
    // If selected, show for drawings
    if (cbShowForDrawings.Checked)
        docDisplay |= (int)swDocTemplateTypes_e.swDocTemplateTypeDRAWING;

    // Add the items
    AddCommandItems(docDisplay, imageLocation);
}
```

What this does is set the document display variable, which dictates where the group is displayed, by default to None. This is a bit misleading as None doesn't mean it never gets displayed it actually means it gets displayed when there are no models open. If we didn't add this item then if nothing was open the group would not be displayed. If you wish to it to never be displayed then set the docDisplay variable to 0 instead.

After that we then check each checkbox and if it is checked, add the document type to the display variable so it will be displayed for that item. We convert the swDocTemplateTypes_e enumerators to integers as that is what the Command Group function requires, and use the OR syntax which is a pipe bar in C# to effectively add the variable to it. Without going into binary to explain this just accept that it works for now.

Finally we call a new function with the same name that takes in an integer and a string as parameters which will create the group using the settings passed in.

By creating a second function with the same name but different parameters we are creating what is called an overloaded function/method. This just means that the function can be called by passing in any of the functions parameters. We have 2 functions, the above one with no parameters and this new one which will have an integer and a string so we could call it in 2 ways:

```
AddCommandItems();
AddCommandItems(123, "Hello World");
```

And both are valid. Anyway, back to the tutorial – all we need to do to create a Command Group now is to use the Command Manager to create the group, and then set a few items on it, and activate it:

```
private void AddCommandItems(int documentTypes, string imageFile)
{
    // Create a new command group which is the main menu item, and the toolbar title
    mCommandGroup = mCommandManager.CreateCommandGroup(mCommandGroupId, "Convert", "",
"", -1);

    // Set the group to have both a menu and a toolbar
    // You could easily add new parameters and UI items to allow
    // the user to dynamically alter these settings
    mCommandGroup.HasMenu = true;
    mCommandGroup.HasToolbar = true;
    mCommandGroup.ShowInDocumentType = documentTypes;
    mCommandGroup.SmallIconList = mCommandGroup.SmallMainIcon imageFile;
    mCommandGroup.LargeIconList = mCommandGroup.LargeMainIcon = imageFile;

    // We want all command items to be a menu and toolbar item
    int itemType = (int)(swCommandItemType_e.swMenuItem |
swCommandItemType_e.swToolbarItem);

    mCommandGroup.AddCommandItem2("PDF", 0, "", "Save model as PDF", 0, "PDFItemClicked",
"PDFEnable", 1001, itemType);
    mCommandGroup.AddCommandItem2("JPEG", 1, "", "Save model as JPEG", 1,
"JPEGItemClicked", "JPEGEnable", 1002, itemType);
    mCommandGroup.AddCommandItem2("DXF", 2, "", "Save model as DXF", 2, "DXFItemClicked",
"DXFEnable", 1003, itemType);

    mCommandGroup.Activate();
}
```

We start by creating a new group using the CreateCommandGroup function of the manager and passing in the unique ID we specified as a new class variable, and giving it a title of "Convert", with no tooltip or hint (tooltip displays on hover, hint displays in the SolidWorks status bar at the bottom left) and a position of -1 which means at the very right-end of the main menu.

Then we set the properties HasMenu and HasToolbar to true so the group shows up as both a menu item at the top, and optionally as a toolbar. As the note suggests these could easily be passed in as 2 new Boolean variables to the function and set based on UI checkboxes on the form, but for now we just want them to always show in both.

Next we set the ShowInDocumentType property that we created from our UI checkboxes earlier, and then finally set all images for the group to the image location variable (we will set this next when the user clicks the Set Image File button).

That is all the user defined information set, so the rest is just hard coded. To display something useful add a few command items with any details you wish, remembering to set the image index if you are using a multi-image file. I will cover the AddCommandItem2 function in detail later.

The last step is to activate the group which commits all of our settings in the group and creates it in SolidWorks. Once this is done, no changes can be made to the information we set so if we alter anything we need to remove and re-add the group to do so.

Selecting a BMP file

We allow the user to select an image file to use for the menus and toolbars. We have already written the code to use the imageLocation variable as the image in the created Command Group so all we need to do is to actually set the imageLocation variable to a location of a bmp file and we are good to go.

Back in the event handler for the Set Image File button, add the following code to allow the user to browse for a bmp file:

```
private void bSetImageFile_Click(object sender, EventArgs e)
{
    using (OpenFileDialog ofd = new OpenFileDialog())
    {
        ofd.Filter = "Bitmap (*.bmp) | *.bmp";
        ofd.Title = "Select menu image file";
        if (ofd.ShowDialog() == DialogResult.OK)
        {
            imageLocation = ofd.FileName;
            bSetImageFile.Image = Image.FromFile(imageLocation);
        }
    }
}
```

When the user clicks the Set Image file button we create a new OpenFileDialog object which is the default file browsing dialog of Windows. We then set the title of the dialog and the filter to only allow selection of bitmap files.

Then we show the dialog and check if the user clicked OK (Select) in the dialog meaning they successfully selected a file, and if so set the imageLocation variable to the file they selected, and set the Image property on the Set Image File button to the image they selected so we can see what they selected too. That's it.

The RemoveCommandItem function

All that is required to do to remove the command group is the following:

```
private void RemoveCommandItems()
{
    mCommandManager.RemoveCommandGroup(mCommandGroupId);
}
```

Passing in the ID we used when creating the group will remove it. Job done!

The CommandItem explained

Above I showed you the code for creating command items (which are menu items and toolbar buttons) but did not explain them.

```
// We want all command items to be a menu and toolbar item
int itemType = (int)(swCommandItemType_e.swMenuItem | swCommandItemType_e.swToolbarItem);

mCommandGroup.AddCommandItem2("PDF", 0, "", "Save model as PDF", 0, "PDFItemClicked",
"PDFEnable", 1001, itemType);
mCommandGroup.AddCommandItem2("JPEG", 1, "", "Save model as JPEG", 1, "JPEGItemClicked",
"JPEGEnable", 1002, itemType);
mCommandGroup.AddCommandItem2("DXF", 2, "", "Save model as DXF", 2, "DXFItemClicked",
"DXFEnable", 1003, itemType);
```

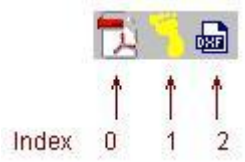
When you add a new Command Item one of the things you have to specify is whether the item is a menu item and/or a toolbar item. We want our items to appear in both the menu and toolbar so we create a new variable with both toolbar and menu enumerators set so that we do not have to keep specifying it for each item as it is a bit long, then we just pass that in as the last parameter.

The first parameter is the text of the item. This is the text for the menu item or toolbar button.

Next is the position in the command group, so 0, 1 and 2 are the order the items appear in the menu or toolbar.

The third and forth items are again the tooltip and descriptions for the items.

The fifth item is the image index. This is the position in the image file specified that this item looks for the 24x24 or 16x16 image from the image strip. An index of 0 looks at the first location in the image file to the left. An index of 1 looks at the second image location, which starts at the 25th pixel across to the right for a 24 pixel high image strip, or the 17th pixel for a 16 pixel high image. For example here is an image strip used, and the numbers show the image index. The image you specify must be 16px or 24px height, and a multiple of the height long, so each block is a new image.



The Callback and Enable Functions

The sixth and seventh items are the Callback and Enable function names respectively. All this means is that when the user clicks the item, the Callback function will be called, and when the item is above to be shown (for example when the menu is about to be dropped down) the Enable function gets called to determine whether or not to enable the item.

The functions are string values and are the name of a function in your **SWAddin** class (which is not this class we are in (the Taskpane control class), so the functions must be declared in the class that inherits the SwAddin interface, which in our case is the SWIntegration class file.

The functions MUST be public for them to be found and called by SolidWorks, so make sure they are not declared as private.

The Enable function should be a function that returns an integer, and the integer value it returns determines the state:

If your method returns...	Then the SolidWorks software...
0	Deselects and disables the item
1	Deselects and enables the item; this is the default state if no update function is specified
2	Selects and disables the item
3	Selects and enables the item
4	Hides the item

The eighth parameter is just another unique ID for the item. The Id's MUST be unique not just to the Command Items but also to the Command Group, so they cannot have an ID the same as the ID of the Command Group (hence why I started at 1001 not 1). Although you can have the same ID odd things will happen if you do that, so don't!

So if you take a look at the 3 items we added the Callback functions are specified as:

```
"PDFItemClicked"
"JPEGItemClicked"
"DXFItemClicked"
```

This means that SolidWorks will look for a function called PDFItemClicked in the **SWAddin** class of our add-in, which is the SWIntegration class file in our code, NOT the SWTaskpaneHost class file we actually created the items in.

Open up the SwIntegration class and add the functions so the names are the same as those specified, and make sure they are public:

```
public void PDFItemClicked()
{
    MessageBox.Show("PDF");
}
public void JPEGItemClicked()
{
    MessageBox.Show("JPEG");
}
public void DXFItemClicked()
{
    MessageBox.Show("DXF");
}
```

Now when you click the items these functions will be called – feel free to do whatever you like in them.

The Enable functions are specified as:

```
"PDFEnable"
"JPEGEnable"
"DXFEnable"
```

Again, add the following functions to the SWIntegration class:

```
public int PDFEnable()
{
    // Let's disable PDF
    return 0;
}
public int JPEGEnable()
{
    // Let's leave JPEG as default
    return 1;
}
public int DXFEnable()
{
    // Let's hide DXF all together
    return 4;
}
```

Now if you launch your add-in you can specify your settings, browse for an image and click Add Items to add the new Command Group. Straight away you will notice the Convert menu item appear at the top if it is set to be visible for the current state (if nothing is open it will always be). If you drop-down the menu you will notice 3 items (we specified to hide the DXF item in the DXFEnable function, however in true SolidWorks fashion it doesn't behave as it should). The PDF will be disabled due to the PDFEnable function, and the JPEGEnable will be enabled. You will also note that the DXFEnable returning 4 also has other issues as it doesn't obey its own rules when models are open and instead enables the item, so stick with 0 and 1 for enabling/disabling items.

Clicking the items will run the associated Callback functions, and in our case that means a simple Message Box will be displayed based on the selection.

And that's about it for this tutorial folks, I hope that you have understood everything as it turned out to be a bit longer tutorial than I expected so tried to keep everything as simple as possible at the same time as showing you some customization and power to the user to dynamically specify menu settings.

It wouldn't be hard then to remove the hard-coded Command Items and also generate those based on user specifications. Perhaps having them browse for a text file then parsing that for the CommandItem parameters and creating them on the fly for example. The limit is only your imagination.

One final note: - We should technically be doing a `Marshal.ReleaseComObject` on the CommandManager object and CommandGroup object as we got those from SolidWorks and they need disposing, but to correctly implement a disposal structure I would have to show you how to implement the IDisposable interface and all the rest of it, and for now I think this tutorial is long enough, so don't worry about that too much it's not going to affect anything with such a small add-in anyway, just be aware of it.

Until next time, enjoy!