SolidWorks 2008 API

Programming & Automation

Luke Malpass

AngelSix.com

SolidWorks API Series 1

Programming & Automation

Written by Luke Malpass
AngelSix.com

Published by AngelSix

©2011 Luke Malpass contact@angelsix.com

All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage or retrieval system, without prior written permission of the publisher.

Published by AngelSix – AngelSix.com

Second Edition

Trademark Information

SolidWorks and PDMWorks are registered trademarks of SolidWorks Corporation.

Excel is a registered trademark of Microsoft Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Contributors

Throughout this project I have received many enthusiastic responses and people contributing their own ideas and knowledge to areas of this book. It has been a great experience and an eye opening one.

Their enthusiasm towards this project has been a key driving factor in pushing me to do this. I just hope that I do not let any of them down.

Introduction

When I was first introduced to computers at the tender age of 9, I have always been intrigued how things worked. Not on a basic level of being told that if you write this portion of code, this will happen, but to know the reason for every line of code and its purpose. To know the reason why X comes before Y, and to analyse it and find its extremes, where it fails and to know its limitations.

It is this drive for deeper understanding that has pushed me in every aspect of my life, to fully understand computers, electronics, people, and ultimately the universe. Whenever I come across something I do not understand, I take it upon myself to learn; learn through observation, through trial-and-error, through others, through experience. So far I would like to think I have succeeded in all that which I set out to understand.

I have always had a unique way of thinking when it comes to computers and logic; I find most people learn programming through the process of finding a solution and remembering the answer, such as knowing that COS(1) = 0.5403, yet they do not know that

$$\cos x \equiv \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

.

Whereas I feel most people simply remember coding, I like to fully understand it, to the point where I could tell you what would happen with code before even compiling it; I often write 1000's lines of code before even compiling and testing it, as I already have in my head the architecture and theory of what is going to be output, and I simply translate my thoughts into writing.

By writing this book I hope to pass on my understanding of computers and programming to the next generation.

I would like to thank my parents for making me the person I am today, for giving me my morals and personality, and the drive in life that has lead me to this point.

This book will be my first, aimed at sharing with people my ways of thinking and programming and hopefully to enlighten at least one person's day. All feedback is greatly appreciated, please send comments to contact@angelsix.com.

This book presumes the reader has a basic understanding of computer programming on some level, such as the knowledge of **if**, **else**, **for**, **while** statements, conditional statements and general computer skills, and is savvy with SolidWorks.

I have tried to give the best explanation of all code provided on its purpose, and what the point of every line of code is. I hope you enjoy reading this book as much as I have enjoyed writing it.

CD CONTENT (case-sensitive link):

http://www.angelsix.com/CODE/SW2008.zip

Table of Contents

Table of Contents

The Basics

Recording your first macro

Writing a macro from scratch

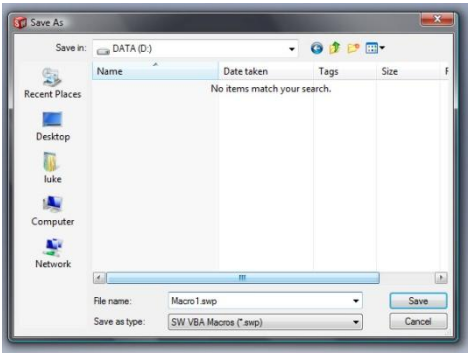
Recording yours first macro

Let's get right into the SolidWorks API and use the Macro Record function to get SolidWorks to create a working macro for us.

Start by opening SolidWorks, and before opening any file from the menu select **Tools->Macro->Record**. Most of the actions you perform will not be recorded by SolidWorks; it's just to get started.

For now we are simply going to create a new part. Select **File->New**, and then **File->Save**, and save the new empty part wherever you want.

That's it; now go to **Tools->Macro->Stop**. This will instruct SolidWorks to stop recording your actions, and subsequently save the newly created macro.

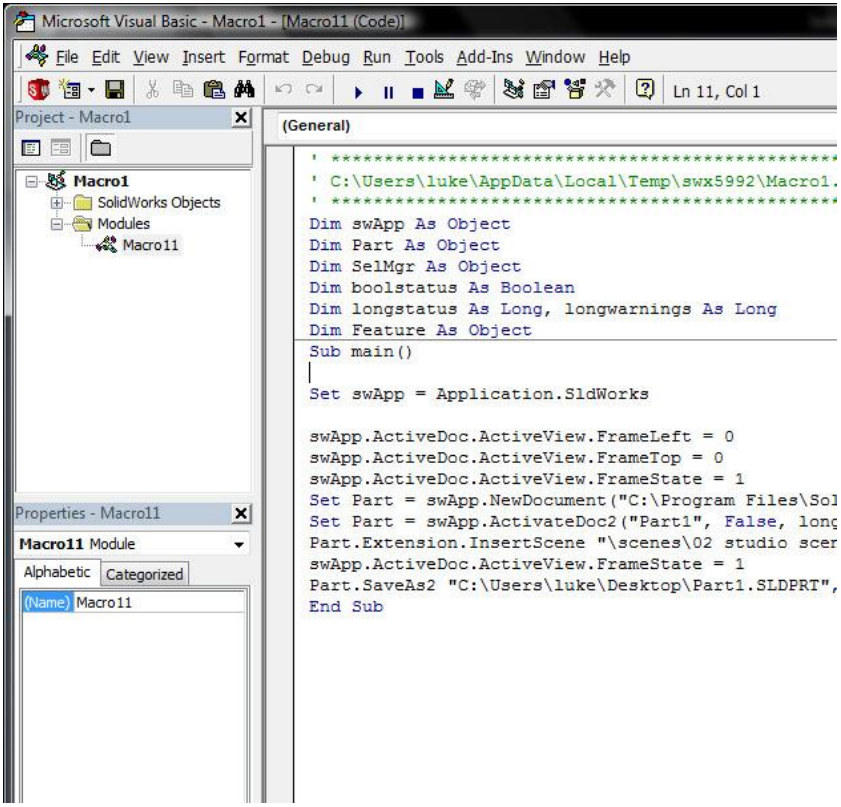


Save the macro where you would like.

Now you are returned to SolidWorks like normal. In order to see, and more importantly to edit and run the macro you just created, you need to go to **Tools->Macro->Edit…** Then select the macro you just saved. This will open up the Visual Basic for Applications development environment, or for short the VBE.

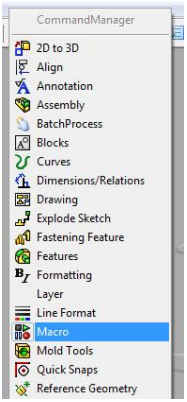
The first thing you will see is a lot of code to the right, with a tree-view panel to the left. If you are unfamiliar with the VBE, then just press F1 within the VBE environment and take a look at the help topics within there. Don't get too concerned with that right now, all

you need to know is that the panel on the left is where you will find your macro's files and the properties, and the remaining space to the right is where your currently selected files content from the left panel is displayed.



Notice in this screenshot the tree-view to the left labelled "Project – Macro1". This is the project explorer where all files related to the macro you are editing are shown. For now there is only one file, the module that SolidWorks created. And to the right, is that modules content.

Leave the VBE window open and go back to SolidWorks. Close down all files and on the top file menu, right-click and select Macro from the menu. This will save us having to go through the Tools menu all the time to access the Macro tools Run, Edit, Record etc... You should now notice that the Macros toolbar should now be visible somewhere on the window.



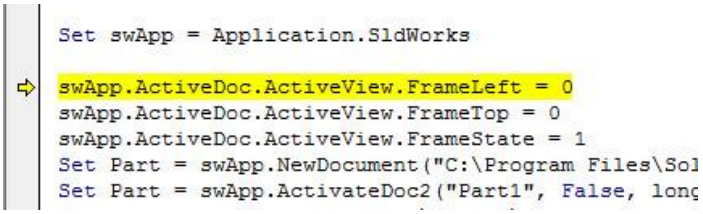
Before we get into understanding and editing this macro, let's first get this macro to run, check it is working, and to see exactly what it does.

With no files open, click the **Play** button on the macro toolbar. This will play the selected macro instantly. Select the macro you just created to run it.

Your very first lesson about using the built-in recording functionality of SolidWorks starts here. Trying to run this macro generates the following error:

```
Run-time error '91':
Object variable or With block variable not set
```

If you click debug, it will take you to the coding in the macro and highlight the error in yellow:



OK, so we know this line is causing the error. Without getting too involved for now, and to get you started, remove the lines so that you end up with the code below:

```
Dim swApp As Object
Dim Part As Object

Sub main()
Set swApp = Application.SldWorks

Set Part = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0#, 0#)
Part.SaveAs2 "C:\Users\luke\Desktop\Part1.SLDPRT", 0, False, False
End Sub
```

Now, click the Save button in the VBE environment to save the changes and run the macro again (by going back to SolidWorks and clicking the Run button or by clicking the Run button from within the VBE environment).

This time running the macro should create a new part and save it. If you still get errors double-check your code. Remember all code from this book is found of the accompanying CD.

Understanding the code

Now we have a working macro that creates a new part and saves it to a location. Let's now take a closer look at the code that did this and understand it.

Firstly, let me explain how macros interact with SolidWorks; in programming, assemblies can make themselves visible to other programs running on the system. This is called COM-Visibility. Basically this allows us to create an instance of the SolidWorks program (new or currently running) and access any functions that it makes visible to us. So, the first stage of any macro or program is to get an instance of SolidWorks to work with. This is done on this line:

```
Set swApp = Application.SldWorks
```

But before we get to this line, you will notice that there are several lines before this. Anything outside of the Sub Main() routine and not inside another Sub or Function, is classed as a global variable.

Variables are things such as numbers, Booleans (true, false), arrays, custom objects; this book presumes you have enough knowledge to know about variables, functions and conditional statements, if not search google, there are plenty of tutorials and resources available.

```
Dim swApp As Object
Dim Part As Object
```

Dim is the name used in VBA to declare a variable, following that is the variables name, then '**As**', and then its type.

Above, the macro has declared two variables. Two variables of type **Object**, which is a universal type that all other types come from; these will be used in the main() routine.

Firstly, once the macro is run, it enters the main() routine after declaring its global variables. The first line within the main function is the line which creates a new instance of the SolidWorks program through COM as we discussed previously:

```
Set swApp = Application.SldWorks
```

Now, the recorded macro then goes on to create a new document by calling the SolidWorks function **NewDocument**. In order to be able to find this newly created part after it is created, the **NewDocument** function returns what is called a "handle" to the new document it creates, so we can find it immediately. Any return value of a function is set to the variable to its left, so in this case the variable **Part** is set to the newly created document handle that is returned:

```
Set Part = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0#, 0#)
```

The first parameter to the **NewDocument** function is the template location, the other 3 are for drawing documents only.

Don't worry too much about the parameters until later; all will be explained.

Finally, the macro calls a function of the newly created **Part** variable called **SaveAs2**.

```
Part.SaveAs2 "C:\Users\luke\Desktop\Part1.SLDPRT", 0, False, False
```

The first thing you will notice about this call unlike the previous, is that this one is calling a function from the **Part** variable, whereas the previous was from the SolidWorks object itself (**swApp**). The second is that even though we are calling a function still, there are no parentheses. In VBA, if a function is called without ()'s then it is not returning a value. That is not to say that the function itself doesn't return a value, just that we are not making any use of it on this call.

Again, the **SaveAs2** function takes 4 parameters; this first is the name and location of where to save the file. The other 3 I shall cover later.

And that is it; you have just created your first SolidWorks Macro!

Writing a macro from scratch

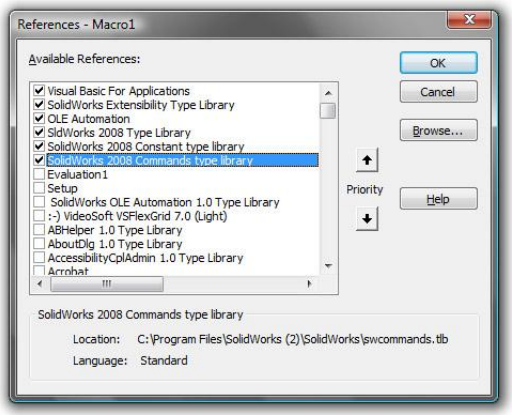
Now it is time to get deeper into the code, to understand every part of the code, line by line. Let's begin.

This time, from SolidWorks, select **New** from the **Macro Toolbar** and save the macro to your desired location. This will automatically open the VBE environment for you to start coding, as well as add some lines of code, and behind the scenes add references for you as well.

To truly start from scratch, delete the code from the main window. As well as the code, SolidWorks added a reference to itself, so I am going to explain how to add this yourself (something you will need to do later using Visual Studio).

In the VBE, go to **Tools->Reference…** This will open up all references to this macro. References are links to libraries that your macro is going to use. When we create new instance of SolidWorks in our code, or call functions such as **NewDocument** and **SaveAs2**, these are functions from the SolidWorks library. In order for our macro to know about these functions, we must reference the correct SolidWorks libraries first.

Although already checked, if they weren't, you would simply scroll down the list and check the **SolidWorks Types**, **Constants**, **Commands** and **Extensibility** libraries to add them to your project.



With a fresh clean empty macro ready to work with, we will begin by re-creating the recorded macro, but do it the correct way this time.

Creating variables of the correct type

One thing the record feature does not do when creating its code is to give variables their correct type. Instead it simply stamps them all as Objects. Of course, creating our own, we are going to do it right.

Start by declaring the variables we had in the first macro, only this time notice the types of the variables:

```
Dim swApp As SldWorks.SldWorks
Dim swPart As ModelDoc2
```

When creating a new instance of the SolidWorks application, it actually returns an object of type **SldWorks.SldWorks**. This is the main SolidWorks variable for the entire SolidWorks environment.

With this macro, we want to create a new part and save it, so we need another variable of type **ModelDoc2**. I have also renamed the variable to **swPart** for easier reading.

On to the actual main function where all code starts; press enter twice to give your code some space from the variables, and then type:

```
Sub main()
End Sub
```

We begin a **Sub** or **Function** with exactly that, followed by its name (in this case **main**), followed by parentheses and then within them any parameters, and finally, to close the Sub/Function, we type **End Sub**/**Function** respectively.

Within this sub is where we are going to start coding. Firstly, instead of creating an instance of the SolidWorks application using the usual recorded method, we are going to use a more portable method:

```
Set swApp = GetObject(“”, "SldWorks.Application")
```

Here, we are calling a function from the VBA library called **GetObject**. This function is used to get a COM object of a currently running process. The first parameter of this function is the path name, which is always a blank string or nothing at all, and the second one is the class name. The class name is the name that the COM object registers itself with, and SolidWorks registers itself as **SldWorks.Application**, so that is what we type.

Let’s stop here and add a little error handling. The **GetObject** function either returns an object of that class is, or nothing. If we test the variable as to whether it is nothing or not, then we know if we succeeded in getting a handle to a running SolidWorks application.

```
If swApp Is Nothing Then
MsgBox "Error gettings SolidWorks Handle"
Exit Sub
End If
```

All we have done here is test if **swApp** is **Nothing**, and if it is, meaning we failed to get a handle to any running SolidWorks applications, then display a message box error, and exit the sub effectively quitting the macro.

Now it's time to create a new part; in order to better understand what we need to do to create a part, go back to SolidWorks, leaving VBE open, and from the menu select **Help->API Help**, to open up the reference documentation for SolidWorks API calls. This comes in handy and is almost always the starting point for any new task.

In the help file, go to the search tab and type in "**NewDocument**" without quotations. In the results to the left double-click **SldWorks::NewDocument**. The help file simply uses a double colon instead of a period to signify that the right hand side is a method or property of the object to the left.

Reading the help gives us a brief description of what the **NewDocument** function does and requires. The help shows that the function takes 4 parameters:

```
SldWorks.NewDocument ( templateName, paperSize,
width, height )
```

The **templateName** is the full location of the template to create the document from (a drawing, part or assembly template). The **paperSize** is a special type of variable called an enumerator, but we do not need this variable for creating our part, and the same goes for **width** and **height**. One final note; the help file states that the return type is either a pointer to the dispatch object or **Nothing** if it fails. So we know from that, we can check for nothing again, just like before, to find out if the operation succeeded.

Now let's add the line of code to create a new part. For simplicity at this stage just type in the location of your required part template file; we could create a drop down list with all available templates of the currently installed templates, but that would just overwhelm the exercise at this point.

```
Set swPart = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0)
```

Now, let's check whether we managed to create a new part or not.

```
If swPart Is Nothing Then
MsgBox "Error creating new part"
Exit Sub
End If
```

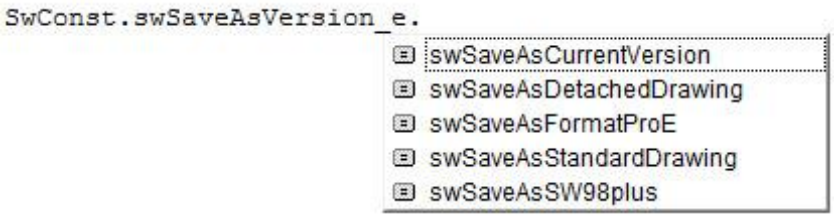
And finally, let's save this part to any location you select. Later on we will create some user interaction that prompts the user for a save location, but for now just type it in.

Start by searching the help file for "**SaveAs2**" like we saw in the first macro. Double-clicking on **ModelDoc2::SaveAs2** you will notice that the help file states that this function is obsolete in red writing, so click the suggested link of **ModelDoc2::SaveAs3**. This will again tell you that this is obsolete. Keep clicking until you get to **ModelDocExtension::SaveAs**. Now read the help to find out about the parameters required.

```
Retval = ModelDocExtension.SaveAs ( Name, Version,
Options, ExportData, Errors, Warnings)
```

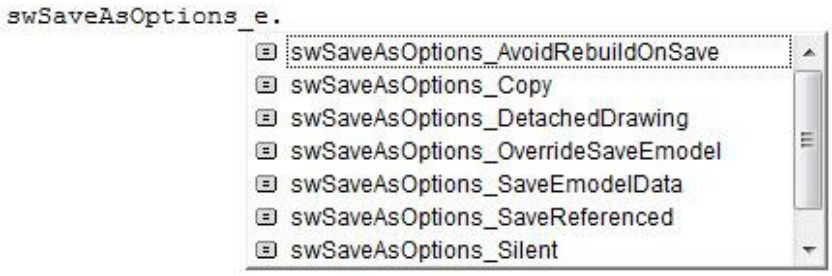
First, **Name** is declared as type **BSTR**, which is a string. This is again the name and location of where to save the file, which we will type in manually for now.

Now onto the **Version** parameter; the help states that this is of type **Long**, which is a number, yet the document says it is a format of type **swSaveAsVersion_e**. Let me explain; the type **swSaveAsVersion_e** is called an enumerator, or enum for short. This is basically a collection of numbers (**Long's**) given names for ease of understanding for the programmer. Instead of typing a number all the time you can type a descriptive name. This enum, and all of the others you will encounter for SolidWorks, is located in the **SolidWorks Constant** library we referenced at the very beginning, so all we need to do is access the constant library with **SwConst** name. As you type you will notice the menu popping up showing you all available properties and methods of the previous object. Drill down until you get to the **swSaveAsVersion_e,** and you will see the following options:



From this list select **swSaveAsCurrentVersion**. You will never use the other versions for the remainder of this book.

The next parameter, **Options,** is again an enum, this time of type **swSaveAsOptions_e,** so let's take a look at what possible values we have:



With enumerators being numbers once compiled, we can combine any combination of them into one variable, so long as the function taking it can process combinations. In this case it can, and if you wanted to pass two options, you would simply separate them like so:

```
swSaveAsOptions_Copy + swSaveAsOptions_Silent
```

By separating them by a + we can pass the two as one parameter. In this case we only need to pass one; **swSaveAsOptions_Silent**. This prevents any messages being shown to the user when saving the document. If you did not want to pass any options for any enum parameter, you pass the number 0 in its place.

The forth parameter is export data, which is purely used for exporting 3D PDFs at present, so we simply pass **Nothing** for this.

Notice for the last two parameters **Errors** and **Warnings**, the help file states they are outputs, not inputs. This means that whatever variables we pass in here will be updated and set to new values after the function has run, so we can check the values of these two variables to understand what has happened and if there are any errors or warnings.

Finally, the help file states this function returns a **Boolean** value (true/false) indicating whether the save succeeded or failed, so we can do another check for failure, and if so, we could access the **error** and **warning** variables for more information on the failure, but for now, we will just accept a failure as a failure.

Putting all this together, with error checking, we get:

```
Dim bRet As Boolean
Dim lErrors As Long
Dim lWarnings As Long
bRet = swPart.Extension.SaveAs("C:\Part1.SLDPRT",
swSaveAsCurrentVersion, swSaveAsOptions_Silent, Nothing, lErrors,
lWarnings)

If bRet = False Then
MsgBox "Error saving new part"
Exit Sub
End If
```

Take note: as the function we are using is from the **ModelDocExtension** object, not the **ModelDoc2** object we have, we have to select the **Extension** object from our **ModelDoc2** first.

You may notice I have also omitted the **SwConst** and **swSaveAsVersion_e** and **swSaveAsOptions_e**. This is not exactly required in VBA it just helps for clarity to start with.

And that is it. Save your macro and test it. If you get any errors review your code, or look at the complete macro on the accompanying CD. Find the final listing below.

```
Dim swApp As SldWorks.SldWorks
Dim swPart As ModelDoc2

Sub main()

Set swApp = GetObject("", "SldWorks.Application")

If swApp Is Nothing Then
MsgBox "Error gettings SolidWorks Handle"
Exit Sub
End If

Set swPart = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0)

If swPart Is Nothing Then
MsgBox "Error creating new part"
Exit Sub
End If

Dim bRet As Boolean
Dim lErrors As Long
Dim lWarnings As Long
bRet = swPart.Extension.SaveAs("C:\Part1.SLDPRT",
swSaveAsCurrentVersion, swSaveAsOptions_Silent, Nothing, lErrors,
lWarnings)

If bRet = False Then
MsgBox "Error saving new part"
Exit Sub
```

```
End If

End Sub
```

Hopefully you have learned quite a lot from this section. Although we have not covered many calls or seemingly done much with SolidWorks, we have covered a lot of technical ground that will provide a solid foundation for the rest of the book.

You will find once you know these main foundations, the rest is simply a case of calling the functions you would like, and through trial and error, or research or the aid of others, you find what you are after.

Connecting with Visual Studio Express

Download and Install Visual Studio Express

Connect to SolidWorks in C#

Connect to SolidWorks in VB.Net

Download and Install Visual Studio Express

Before you can begin you must download and install the Visual Studio software for your desired language.

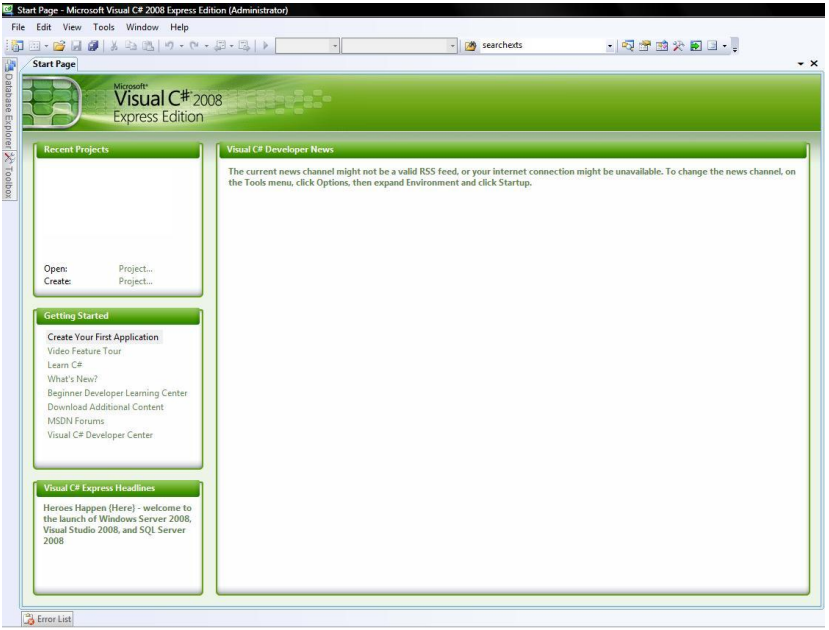
Go to http://www.microsoft.com/express/download/default.aspx and simply download and install either C# Express and/or VB.Net Express. All guides to installing them are on the site.

When you have installed the program, run it. You may be prompted to select layout style or preferred language, just select any.

Now you are ready to begin.

Connect to SolidWorks in C#

When you first open C# Express you are displayed with the following screen:

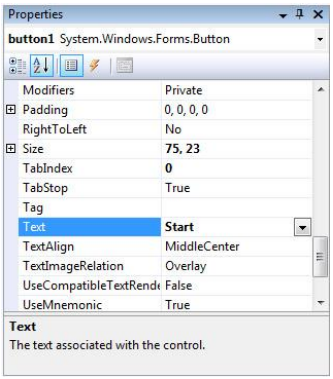


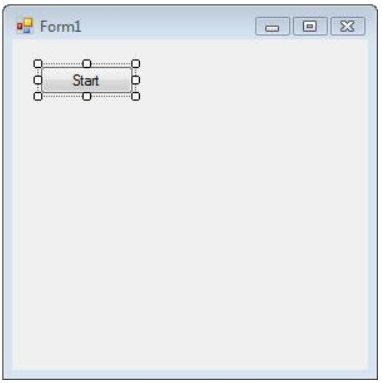
To get started, just select **File->New Project…** and select **Windows Forms Application**. In the **Name** box, enter any name you like and press **OK**.

This will then create your project and automatically create a blank form for you. We are going to create a button that on clicking will effectively initiate our "macro" as such.

To the left you will notice a toolbar icon. Click or hover over that to expand the menu. Find the **Button** item and drag it onto your form to create a new button. Once the button is on the form,

make sure it is selected. On your right will be the properties panel; in here is where the current selected items properties will appear. For our button we are going to change the text within it to "**Start**". Scroll down the properties until you find the property **Text,** and then type in "**Start**" without the quotations and press enter. The button will instantly update.



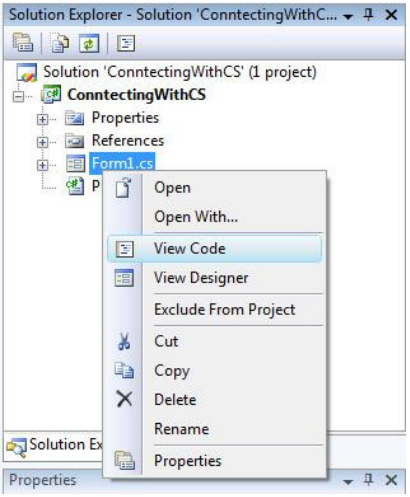


Event Handlers

Now all that is left to do is to add what is called an **Event Handler** to the button, so that when the user clicks it, something happens (an event, or more literally, a function in code, is called). To do this you can just double-click the button, but to understand it better I will show you how to do it another way.

In the property window where you just set the text of the button, notice the  icon. This switches between properties and events. Click it to take you to the buttons events (make sure the button is still selected). Now scroll down and find the event called **Click,** and double-click it. This will automatically create you the event handler code and take you to it.

You will notice you are now in a coding window not a visual design window. Let me explain; above the **Property Panel** to the right is the **Solution Explorer**. This is the same as the **Project Explorer** in VBA that showed all files in the macro. We use this like windows explorer to open, close, delete and rename files in our project. The only file you need to be aware of right now is the **Form1.cs** (or whatever you called your form. To open the form designer (where we added the button visually) just double-click **Form1.cs** in the **Solution Explorer**; to get to the coding behind the form right-click and select **View Code**.



Back in the code view, you will have some code already in the window to get you started. This is what Visual Studio created automatically when you selected to create a Windows Form project. Just like VBA, VS has added the required reference libraries for creating a form and added the required code.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

```
namespace ConntectingWithCS
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void button1_Click(object sender, EventArgs e)
      {

      }
   }
}
```

Let's go through each line of this code again so that you understand what is going on.

The only lines that can be outside of a class in VS are special declarations such as **#Regions,** pre-processor tags**, namespace**'s and **using** statements. The only one we are concerned about is the **using** statement.

The using Statement

In the C# language**, using** is a statement that simply tells the compiler that in any code within this file**,** we will be using classes and functions from the specified library. Strictly speaking**,** you can

remove all the **using** statements and still make your code work, but the advantage of **using** is that, like in VBA where I excluded typing **swConst** and **swSaveAsVersion_e**, it tells the compiler that if we typed **Environment** (which is actually a member of the **System** Library) it knows that we are accessing the **System.Environment** object.

Another example is that without typing **using System.Drawing**, we would have to write the following to access a **Point** object:

```
System.Drawing.Point
```

Whereas by having the **using System.Drawing** statement at the top of the code, we only need to type:

```
Point
```

So if brief, **using** statements simply save typing long-winded code statements.

Namespaces

Namespaces, again like **using** statements, are not strictly needed. To keep this short, you can remove the **namespace** statement and its open and closing curly braces or leave them in. For our usage throughout this book it will make no difference. They are basically used to group classes and coding blocks into a single accessible name, to be accessed like an object.

The next line is the declaration of the class for the form we created (the form with the button). This will always be generated automatically for our use so you do not need to type this part manually at any stage.

```
public partial class Form1 : Form
```

To explain; the **public** keyword states that this class is accessibly to any other part of code within the program. Do not worry too much about this; it does not play any important role for us. The **partial** statement means that this class has been split into multiple files. The other file in this case is the code that VS has created automatically to create our form, the button and the event handler links etc… To take a look at this file, go to the **Solution Explorer**, click the [+] next to the **Form1.cs** file and double-click the **Form1.Designer.cs** file to see its code. Do not worry if you do not understand this code because we are here to focus on SolidWorks programming.

The { } braces are used for all classes, functions, **if**, **else**, **while**, **do** and other statements in C# to determine the start and end of its block of code. This is equivalent to VBAs **If … Then** and **End If**, or **While… Wend**.

Within the main class (**Form1**), we find the following function:

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

This is called the constructor class, and this gets called as soon as the form is created. A constructor class has the same name as the class it is within (in this case **Form1**), and has no return value. The only function within it is a call to the function defined in the **Form1.Designer.cs** class that is created automatically by VS to create the form, create all the items like our button, add event handlers, set properties, and show the form. If you placed code after this **InitializeComponent** function call, that code would get run as soon as the form is created, so to the user it would appear that the code runs as soon as they run the program. As this is not the behaviour we want, we go on to create a function that is called on the click of a button instead.

The Button Click function

Visual Studio automatically created the following function for us when we chose to add an event handler for the Click event of the button in the previous stage.

```
private void button1_Click(object sender, EventArgs e)
{

}
```

A function in C# has the following structure:

```
[accessibility] [return type] [function name] (
[parameters] )
{   // Code here }
```

Accessibility is either **public**, **private** or **protected** as standard. For our usage we are going to be using **public** throughout.

Return type must always be defined, except for constructors and destructors. This is the type of variable that the function returns, such as a number (**Integer**, **Double**, **Float**) or text (**String**), or true/false (**Boolean**) as some examples. If your function does not return any value then the type is **void**.

The **function name** is anything you like, so long as it begins with a letter, and contains no special characters or spaces.

The parameters inside the function are defined in the following structure:

```
[pass as] [variable type] [variable name]
```

The **pass as** value can be **out** or **ref**, or left out completely. By default, if you pass a variable without stating **out** or **ref**, and that function then alters the variable inside its own code, once the function has run and returned, the variables value will return to what it was before the function call, so it remains unaffected. If this is not the behaviour you want (such as we will see later on), then you can pass the variable as **out** or **ref**. The **out** keyword means the variable you are passing must not have been assigned a value yet, and if you pass it as **ref**, it must have. You will see this used later.

The **variable type** is any type like the return value, such as a number, text, object etc... A variable parameter type cannot be **void**.

The **variable name** can be any name again, starting with a letter containing no special characters or reserved names or spaces.

So using that knowledge we know that this event handler function that VS has created for us when the button is clicked returns no value (**void**) back to its caller, is **private** so inaccessible to certain areas of code, is called **button1_Click**, and takes **2 parameters**.

The first parameter is of type **object**. This is because an event handler can be attached to many different objects, in this case it is our button, so this variable called **sender** is actually a reference to our button object. The second is a special type **EventArgs**; this variable contains information about the event. We do not need to do anything with either of these parameters so let's move on.

Adding the SolidWorks References

Before we can do anything, we must add the same references to SolidWorks that we did in VBA. In the **Solution Explorer**, right-click the **References** item and click **Add Reference…** Once the dialog appears, click the **COM** tab, and then from the list select the following items (holding **Ctrl** to select multiple in one go), and click **OK**.

SldWorks 2008 Type Library

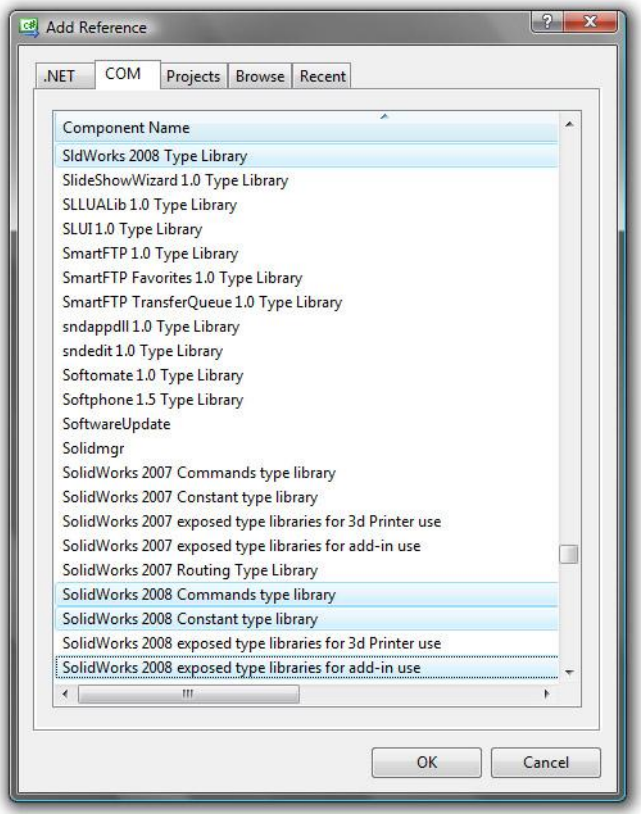
SolidWorks 2008 Commands type Library

SolidWorks 2008 Constant type Library

SolidWorks 2008 exposed type libraries for add-in use

And then finally, at the top of the code where the using statements are, add:

```
using SldWorks;
```



Writing our SolidWorks Code

We now have an entire C# program structure, ready to write our SolidWorks code inside our event handler function. All examples and codes done in C# from now on will presume this same structure, and only provide and describe the code that will be run inside this event handler, or wherever you decide to put it. So for our first example let's re-create our first written macro we made in VBA.

So first let's define our variables again; we can define these either inside this function, or inside the main class function. For example:

```
public partial class Form1 : Form
{
    SldWorks.SldWorks   swApp;
    ModelDoc2           swModel;
    ...
    private void button1_Click(object sender, EventArgs e) { ... }
}
```

Or inside the **button1_Click** function shown above...

```
    private void button1_Click(object sender, EventArgs e)
    {
        SldWorks.SldWorks swApp;
        ModelDoc2 swModel;
        ...
    }
```

For best practise in the future I will be writing the main variables in the main class not the event handler.

Declaring a variable in C# is a little different than VBA; you simply state the type first, followed by the name. As you can see, the two variables specified above are the same types of the ones we wrote in VBA.

With these variables defined, we move on to the main SolidWorks code. Take a look at the original VBA code; see the slight changes:

VBA

```
Set swApp = GetObject("", "SldWorks.Application")
If swApp Is Nothing Then
MsgBox "Error gettings SolidWorks Handle"
Exit Sub
End If

Set swPart = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0)

If swPart Is Nothing Then
MsgBox "Error creating new part"
Exit Sub
End If

Dim bRet As Boolean
Dim lErrors As Long
Dim lWarnings As Long
bRet = swPart.Extension.SaveAs("C:\Part1.SLDPRT",
swSaveAsCurrentVersion, swSaveAsOptions_Silent, Nothing, lErrors,
lWarnings)

If bRet = False Then
MsgBox "Error saving new part"
Exit Sub
End If
```

Now, let's do exactly this but in C# within our event handler button function.

Firstly, because C# does not use the **GetObject** function to acquire a handle to SolidWorks, we must use another function found in the **System.Runtime.Interop** library. Add the following line to the **using** section:

```
using System.Runtime.InteropServices;
```

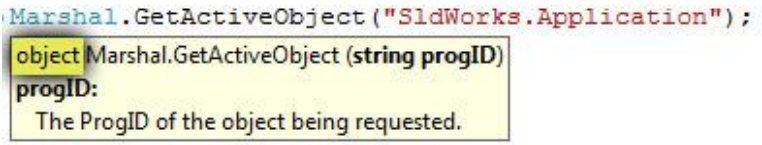
Now, to get the active SolidWorks object, we do this:

```
swApp =
(SldWorks.SldWorks)Marshal.GetActiveObject("SldWorks.Application");
```

Note that there is no such thing as a **Set** statement like in VBA, we do not need this. What we are doing here is calling the function **GetActiveObject** from the **Marshal** class in the **System.Runtime.InteropServices** library we added just. This function is identical to the VBA **GetObject** function.

Casting

Because C# is stricter on variable types, unlike VBA, we must do all the type conversions ourselves. Because the function **GetActiveObject** returns a variable of type **object**, we must convert it to the correct type, using a method called **Casting**.



To cast the object returned from **GetActiveObject**, we simply write the type of variable we would like it converting to inside parentheses before the method and after the equals sign. This does the job of converting the object to our variables type. In this case we typed **SldWorks.SldWorks**, which is the type of the **swApp** variable.

Try/Catch block

Now we wish to check if we have managed to get the SolidWorks object; however, this time we do this a little differently. Because the **GetActiveObject** function actually throws a system error if it fails we must try the function and catch any system errors that get thrown. If we do not handle this error our program will simply crash. We do this by placing the statement within a **try/catch** block.

```
try
{
   swApp =
(SldWorks.SldWorks)Marshal.GetActiveObject("SldWorks.Application");
}
catch
{
   MessageBox.Show("Error getting SolidWorks Handle");
   return;
}
```

When you place something inside a **try block**, your program will handle any system errors that are caused by any code you place within there, and instead of crashing your program, it passes them to the **catch block**. So in effect, we try the **GetActiveObject** function,

and if it succeeds our code continues to execute *after* the **catch block**. If it fails, our code jumps out of the **try block** at the point the error occurs, and goes straight to the **catch block** to process any code within there, not executing any code in the **try block** after the point of the error.

If we get the **handle**, we carry on to after the **catch block** with a workable **swApp** variable. If it fails, we show the user a message box and call the **return** statement, which returns the function, effectively ending it, so we proceed no further with the function.

Creating a new document

Now we have a workable **swApp** variable, we want to create a new document. To do this we call exactly the same function as we did before, with exactly the same parameters, setting the **swModel** variable at the same time.

```
swModel = (ModelDoc2)swApp.NewDocument(@"C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0);
```

The only thing different here is the **@** sign placed within the parentheses but before the opening quotes. All that this means is the following string contains back/forward slashes as literal characters instead of escape characters. If we did not put the **@** in, we would need to put **** for every single slash we wrote, because as standard **** is used to place special characters in strings, such as **\t** for a tab.

Also, we have done another casting from **object** to **ModelDoc2** as our variable is of type **ModelDoc2**.

When typing in any .Net language, you will find you get much better feedback from the Vs **IntelliSense** telling you about the object you are using.

Now to check that the document was created correctly:

```
if (swModel == null)
{
    MessageBox.Show("Error creating new part");
    return;
}
```

This statement says "if **swModel** is equal to **null** then". In C#, **==** means "is equal to". When setting variables in C# we use single equals sign (=), but when performing conditional checks such as "is equal to" we use double equals (==).

C#

```
==    Is Equal To
!=    Is Not Equal To
<=    Is Less Than or Equal To (Numeric)
>=    Is Greater Than or Equal To
<     Is Less Than
>     Is Greater Than
=     Set variable to left
*=    Multiple left variable by right
/=    Divide left variable by right
^=    XOR left variable by right
+=    Add right variable to right
-=    Subtract right variable from left
```

VB.Net

```
=      Is Equal To / Set variable to left
<>     Is Not Equal To
<=     Is Less Than or Equal To (Numeric)
>=     Is Greater Than or Equal To
<      Is Less Than
>      Is Greater Than
*=     Multiple left variable by right
/=     Divide left variable by right
^=     XOR left variable by right
+=     Add right variable to right
-=     Subtract right variable from left
```

Saving the part

Now it is time to save the part; in order to use constants such as the save as version and options, I have added the following to the **using** section, to save us having to type **SwConst** before every constant.

```
using SwConst;
```

Now we define some variables used in the **SaveAs** function, then attempt to save the part, and finally check if it succeeded.

```
int lErrors = 0, lWarnings = 0;
bool bRet = swModel.Extension.SaveAs(@"C:\Part1.SLDPRT",
(int)swSaveAsVersion_e.swSaveAsCurrentVersion,
(int)swSaveAsOptions_e.swSaveAsOptions_Silent, null, ref lErrors, ref
lWarnings);

if (!bRet)
```

```
{
    MessageBox.Show("Error saving new part");
}
```

Firstly, we defined the variables needed for the error and warning values. Then at the same time as declaring the variable **bRet** (true or false), we also set it by calling the **SaveAs** function.

A few differences this time compared to the VBA version. In VBA, to represent "nothing" we pas the special name **Nothing**, in C# the equivalent is **null**.

We also cast the constant **enums** to **int** as the function requests **int** values. And if you notice the **Intellisense tooltip** you see that it states that the **error** and **warning** variables should be passed as **ref**. If you recall earlier me telling you about **ref** and **out** keywords; by passing these two variables as **ref** means that when the **SaveAs** function alters them within its own code, the two variables can be set inside the **SaveAs** function. The easiest way to explain it is that if you pass variables without a **ref** or **out** keyword, they are copied to the function, not sent directly, so any alterations made are made to copies of the variables, not the originals. By passing them with **ref** or **out**, the originals are sent.

Notice we passed the two variables we defined for errors and warnings as **ref**, so as soon as the **SaveAs** function has been called the variables will be set to whatever **SaveAs** set them to. We could then check these variables after if the function returned **false** (meaning it failed to save). We also had to assign them before

passing them as they are passed as **ref**, not **out**; that is the reason for assigning them the value **o**.

Once the **SaveAs** function has been called and the return value has been saved in the **bRet** variable, we then check whether this **bRet** is true or false. If you place a Boolean value inside an **if** statement, there is no need to do a check such as **bRet == true** or **bRet == false**, you just put the variable in the parentheses. Because we do not want to check if **bRet** is true (success), but false, we place the not sign (!) before it to say "if **bRet** is not true", instead of "if **bRet** is true" so the code within the **if** statement will only run if **bRet** is false.

There is no need to place a **return** in this block as it is the end of our code within this function anyway. And that is it, go to **Build->Build Solution** to build your program, and then **Debug->Start Without Debugging** (or press F6, then Ctrl+F5) to run your program.

Make sure SolidWorks is open then go to your form and click the button to see the magic. If you have errors check you code with the code on the CD. Complete listing below:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using SldWorks;
```

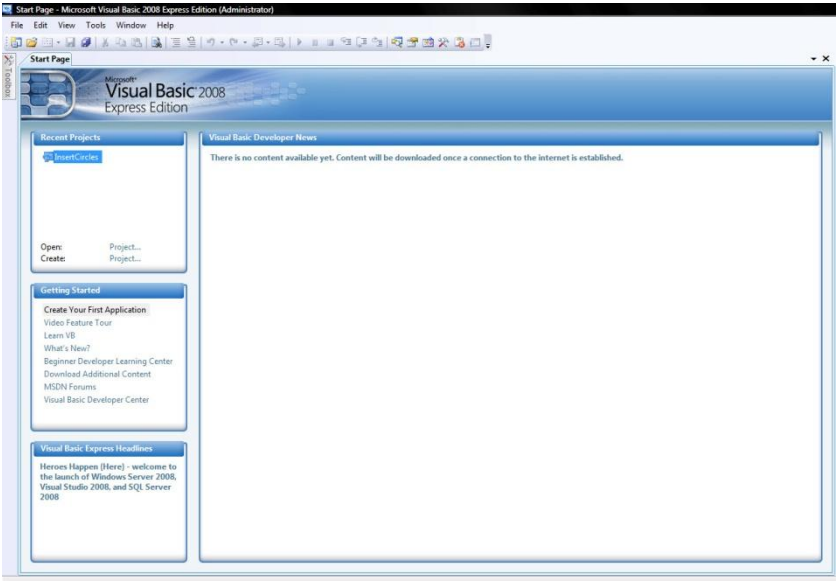
```
using SwConst;

namespace ConntectingWithCS
{
    public partial class Form1 : Form
    {
        SldWorks.SldWorks   swApp;
        ModelDoc2           swModel;

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                swApp =
(SldWorks.SldWorks)Marshal.GetActiveObject("SldWorks.Application");
            }
            catch
            {
                MessageBox.Show("Error getting SolidWorks Handle");
                return;
            }

            swModel = (ModelDoc2)swApp.NewDocument(@"C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0);
```

```
            if (swModel == null)
            {
                MessageBox.Show("Error creating new part");
                return;
            }

            int lErrors = 0, lWarnings = 0;
            bool bRet = swModel.Extension.SaveAs(@"C:\Part1.SLDPRT",
(int)swSaveAsVersion_e.swSaveAsCurrentVersion,
(int)swSaveAsOptions_e.swSaveAsOptions_Silent, null, ref lErrors, ref
lWarnings);

            if (!bRet)
            {
                MessageBox.Show("Error saving new part");
            }
        }
    }
}
```

One thing you may notice is that C# requires every statement line to end with a semi-colon (;).

Connecting to SolidWorks in VB.Net

When you first open VB.Net Express you are displayed with the following screen:

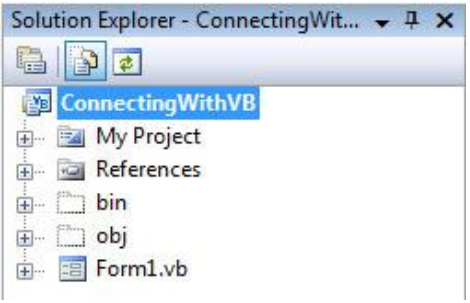


To get started, just select **File->New Project…** and select **Windows Forms Application**. In the **Name** box, enter any name you like and press **OK**.

This will then create your project and automatically create a blank form for you. We are going to create a button that on clicking will effectively initiate our "macro".

From this point on up until you add the event handler to the button, is exactly the same process as the C# section, so refer to that. Once you have created the event handler you will be taken to the coding section, but this time it looks rather different from the C# code.

You will also notice that in the **Solution Explorer,** by default, there is no **Reference** item. Just click the second icon in the explorer called "Show All Files" to show the **References** folder.



Add the SolidWorks references just like you did in the C# project. Now you are ready to begin.

VisualBasic.Net follows VBA much closer than C# so you may find it similar to the VBA example.

VB.Net does not use the **using** statements like C#, it uses **Imports** instead. So at the very top of the forms' code (right-click on **Form1.vb**->**View Code**) place this code to import the SolidWorks library in for use. Again this is not essential it just saves typing the library name before every variable.

```
Imports SldWorks
```

Let's define the usual variables by using the VB.Net syntax:

```
Dim [variable name] As [variable type]
```

Place the variables inside the main class function, not the button click function. You entire code so far should look like this:

```
Imports SldWorks

Public Class Form1

    Dim swApp As SldWorks.SldWorks
    Dim swModel As SldWorks.ModelDoc2

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click

    End Sub
End Class
```

To get an instance of SolidWorks, in the button click function place the following; it is exactly the same as VBA:

```
swApp = GetObject("", "SldWorks.Application")
```

And now we check if we managed to get the handle. This again is very similar to VBA so requires no explaining:

```
If swApp Is Nothing Then
    MsgBox("Error getting SolidWorks Handle")
    Exit Sub
End If
```

Exit Sub is the VB.Net equivalent of C#'s **return,** to effectively return from the current function or sub.

With a working SolidWorks application instance we will now attempt to create a new document and at the same time assign the new document to the **swModel** variable.

```
swModel = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0)
```

Like VBA, VB.Net does explicit object casting automatically, so we do not need to manually cast the returned **object** to **ModelDoc2** type, like we did not need to cast the **object** from **GetObject** to **SldWorks.SldWorks** above.

And again, we check if we managed to create the new part:

```
If swModel Is Nothing Then
    MsgBox("Error creating new part")
    Exit Sub
End If
```

Before we save, we will add the following to the **Imports** section to save us some typing:

```
Imports SwConst
```

And define the required variables and pass the **errors** and **warning** variables as references, like in C#. Only in VB.Net we do not need to state the keyword **ByRef** as VB.Net automatically passes the variables as references without any keywords.

```
Dim lErrors As Long = 0, lWarnings As Long = 0
Dim bRet As Boolean = swModel.Extension.SaveAs("C:\Part1.SLDPRT",
swSaveAsVersion_e.swSaveAsCurrentVersion,
swSaveAsOptions_e.swSaveAsOptions_Silent, Nothing, lErrors,
lWarnings)
```

When passing by reference we need to assign an initial value to the variables like in C#, this is why we set **lErrors** and **lWarnings** to 0.

Finally, we check if we managed to save the part.

```
If Not bRet Then
   MsgBox("Error saving new part")
End If
```

And that's it for VB.Net. Easy huh? Just press **Ctrl+F5** to build and start your project and test it again just like the C# version.

As usual, if you have any problems, double-check your code, and if all else fail, compare it to the working version on the CD.

This template will be used throughout the rest of the book just like the C# template, and only the coding within the event handler will be displayed and explained from now on.

Important Note: All examples on the CD are compiled with references to SolidWorks 2008 SP3.0. If for any reason you get errors when building them, remove and re-add your SolidWorks references and try again. This is a common problem for most people when using examples from another machine.

VB.Net

```
Imports SldWorks
Imports SwConst

Public Class Form1

    Dim swApp As SldWorks.SldWorks
    Dim swModel As SldWorks.ModelDoc2

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click

        swApp = GetObject("", "SldWorks.Application")

        If swApp Is Nothing Then
            MsgBox("Error getting SolidWorks Handle")
            Exit Sub
        End If

        swModel = swApp.NewDocument("C:\Program
Files\SolidWorks\data\templates\Part.prtdot", 0, 0, 0)

        If swModel Is Nothing Then
            MsgBox("Error creating new part")
            Exit Sub
        End If

        Dim lErrors As Long = 0, lWarnings As Long = 0
        Dim bRet As Boolean =
swModel.Extension.SaveAs("C:\Part1.SLDPRT",
```

```
swSaveAsVersion_e.swSaveAsCurrentVersion,
swSaveAsOptions_e.swSaveAsOptions_Silent, Nothing, lErrors,
lWarnings)

        If Not bRet Then
            MsgBox("Error saving new part")
        End If

    End Sub
End Class
```

Saving Drawing Sheets as DXF

Get Document Information

Displaying Document Information

Right, let's get into some interesting SolidWorks programming!

Working with the templates we have created previously, make a new macro or project and connect to SolidWorks. Call the SolidWorks object variable **swApp**.

Saving Drawing Sheets as DXF

One of the most common and popular requests, and often one of the driving factors for getting into SolidWorks API programming is to do things automatically that the user would normally have to do over and over manually. The following code will allow us to save every sheet of the current drawing to separate DXF files. We will also get the save as location from the user in the .Net examples.

To get a handle to the current open drawing in SolidWorks we do the following:

C#

```
swModel = (ModelDoc2)swApp.ActiveDoc;

if (swModel == null)
{
    MessageBox.Show("Failed to get active document");
    return;
}
if (swModel.GetType() != (int)swDocumentTypes_e.swDocDRAWING)
{
    MessageBox.Show("Active document is not a drawing");
    return;
}
```

VBA

```
Set swModel = swApp.ActiveDoc
If swModel Is Nothing Then
MsgBox "Failed to get active document"
Exit Sub
End If

If swModel.GetType() <> swDocumentTypes_e.swDocDRAWING Then
MsgBox "Active document is not a drawing"
Exit Sub
End If
```

** Find VB.Net example on CD*

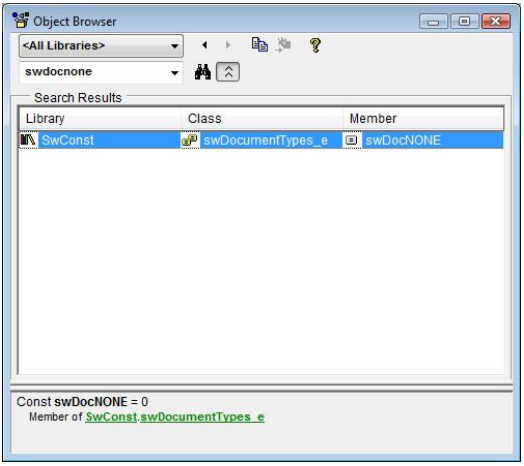
Notice we get the **ActiveDoc** object from **swApp,** which will return either nothing or the handle to the document that is visible in SolidWorks. We then check that we have a document, and if we do we check that it is a drawing, else we stop. We do this by calling the **ModelDoc2** function **GetType,** which returns an **integer** value representing the type of either **None, Part, Assembly** or **Drawing**. If you read the API help and type in **ModelDoc2::GetType** as the search you will find that it doesn't tell us much regarding this, but at the bottom it does give us the possible return values:

> ***Remarks***
> *The retval argument might be one of the following values:*
>
> *swDocNONE - no document*
> *swDocPART - part document*
> *swDocASSEMBLY- assembly document*
> *swDocDRAWING - drawing document*

Object Browser

Here is how I found out that the different type values I needed were in the enumerator **swDocumentType_e**. If you are in VBA, go to **View->Object Browser** or press **F2**, and if you are in .Net then go to **View->Other Windows->Object Browser**. This will bring up a new window that's a bit like a search tool. From here you can look at all of the properties and methods (functions) that any library object has. It comes in handy when you simply need to search for a function as no other help or documentation tells you what you need.

In the search box, type in any one of the values that the help file told us was a possible return value, let's use "swDocNONE" without quotes.



Once you press enter you will get the results. Notice that only one object contains this exact text, and that object is within the **SwConst** library, under **swDocumentType_e**. And that is how we find out where the values are stored. Using this we just compare the type returned from the **GetType** function, with the drawing enumerator value.

With a handle to the active drawing sheet, we get a list of all the drawing sheets, activate each one and then save them as a DXF.

C#

```
DrawingDoc swDrwDoc = (DrawingDoc)swModel;
string[] sheetNames = (string[])swDrwDoc.GetSheetNames();
string saveAsLocation = GetLocationFromUser();

foreach (string sheetname in sheetNames)
{
    // Activate and save drawing sheet here
}
```

VBA

```
Dim sheetNames As Variant
sheetNames = swModel.GetSheetNames()
Dim saveAsLocation As String

Dim sheetname As Variant
For Each sheetname In sheetNames
' Activate and save drawing sheet here
Next sheetname
```

Firstly if we use .Net, because our active document variable **swModel** is of type **ModelDoc2**, we have to cast it to the specific document type of **DrawingDoc**. This is done automatically in VBA.

With the **DrawingDoc** object, we get all of the sheet names of this drawing by calling **GetSheetNames**. In .Net we cast this to an array (collection) of strings (text values), which is basically a list of the sheet names. In VBA we use the variable equivalent **Variant**.

Next we call a function that will ask the user to specify a save location for the DXF files that will be created. We will write this function later.

Then, we use a **For Each** statement, which will loop all of the code within its block for each item it finds within an array. So in this case we loop through every sheet name in the collection of sheet names, so that we can process each one.

Within the **for each** statement block, we write the code to firstly activate the sheet, and secondly, save it as DXF.

C#

```
int lErrors = 0, lWarnings = 0;
bool bRet;
int eVersion = (int)swSaveAsVersion_e.swSaveAsCurrentVersion;
int eOptions = (int)swSaveAsOptions_e.swSaveAsOptions_Silent;

foreach (string sheetname in sheetNames)
{
   swDrwDoc.ActivateSheet(sheetname);
   bRet = swModel.Extension.SaveAs(saveAsLocation + sheetname +
".dxf", eVersion, eOptions, null, ref lErrors, ref lWarnings);
   if (!bRet)
      MessageBox.Show("Failed to save " + sheetname + " as DXF");
}
```

VBA

```
Dim lErrors As Long, lWarnings As Long
lErrors = lWarnings = 0
```

```
Dim bRet As Boolean
Dim eVersion As Integer
Dim eOptions As Integer
eVersion = swSaveAsCurrentVersion
eOptions = swSaveAsOptions_Silent

Dim sheetname As Variant
For Each sheetname In sheetNames

    swModel.ActivateSheet sheetname

    bRet = swModel.Extension.SaveAs(sheetname & ".dxf", eVersion,
eOptions, Nothing, lErrors, lWarnings)

    If bRet = False Then MsgBox "Failed to save " & sheetname & " as
DXF."
Next sheetname
```

Within the **for each** loop we start by activating the current sheet we are iterating through by calling the **ActivateSheet** function of the **DrawingDoc** object, and passing the current sheet name as the parameter.

Then, we call the **SaveAs** function just as we did in the previous chapter, only this time instead of specifying a name ending in ".sldprt", we end it with ".dxf". SolidWorks will automatically save the file as a DXF without you needing to do anything else.

As for the save location; once we get our path where the user would like to save the file, we add the current sheet name and then the dxf extension.

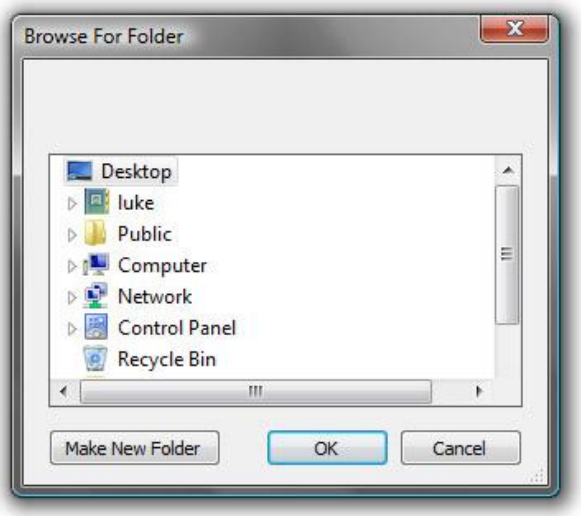
Get input from the user

The last piece of the puzzle is to write the **GetLocationFromUser** function we called in the previous step. Remember to place functions outside of other functions, but inside the main class. You may notice I have not wrote a function for VBA as it is quite long winded to explain at this stage.

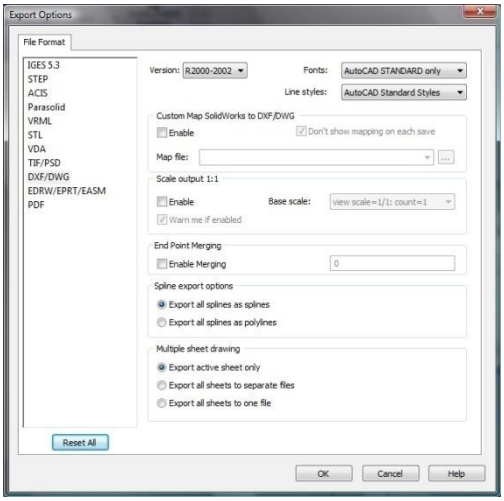
In .Net, we can use a **System** library function to ask the user to select a folder, and then return the selected location. We place the terminating backslash to the path as the function we use leaves it off.

```
public string GetLocationFromUser()
{
    FolderBrowserDialog folderBrowser = new FolderBrowserDialog();
    folderBrowser.ShowDialog();
    return folderBrowser.SelectedPath + @"\";
}
```

Creating a new instance of the **FolderBrowserDialog** class, we then call the **ShowDialog** function which shows the standard **Folder Browser** to the user for them to select a folder. Our code will not continue until the user has closed this browser. Once it has returned, we return the path that the user selected. If they clicked **Cancel** it will return blank and so save in the same location as the drawing.



Run your code and watch as the drawing sheet flicks between each drawing sheet and saves as a DXF.



Note: At the time of writing, SolidWorks still has a bug in setting the DXF properties to save only the active sheet and not all when we run the **SaveAs** command, so I left out the code for setting up the DXF options to save just the active sheet. If your output DXF files have all the sheets within them, just go to manually save a drawing, select DXF, and then click the **Options** button and make sure the last setting is selected as **Active Sheet Only**.

Getting Document Information

Some of the most important things you need to know before you can create any sophisticated or complex code are filenames and locations, titles, configuration name(s), the type of file, if we are in read-only/view-only/large assembly mode, the material and/or summary information.

Start with the usual template connecting into SolidWorks, and then get a handle to the active model document using **swApp.ActiveDoc** setting it to the **swModel** variable like in the last example. It is this model handle we will be performing all our inspection on.

Filename and Location

Let's start with getting the complete location and filename of the active document, and then we will split this information into **Path**, **Filename** and **Extension**.

If you are using C# or VB.Net, first add the following line to the **using/Imports** section respectively to allow us to use functions from the Input/Output library.

```
Using System.IO;  or  Imports System.IO
```

C#

```
string fullpath = swModel.GetPathName();
string filelocation = Path.GetDirectoryName(fullpath);
string filename = Path.GetFileNameWithoutExtension(fullpath);
string extension = Path.GetExtension(fullpath);
```

Using VBA is quite a lot more work to achieve the same goal. Firstly, we must create a function that will return the last position of string within another string. Explaining this function is beyond the scope of this book so if you wish to understand this function you can discuss it on the AngelSix forum, but basically the concept is to find the last occurrence of a certain character such as a period (.) to find the extension, or a backslash (\) to find the directory.

VBA

```
Function LastIndexOf(stringToSearch As String, searchFor As String) As
Integer

Dim iPos As Integer
Dim iTemp As Integer
iPos = -1
iTemp = 0

Do Until iPos = 0
If iPos = -1 Then iPos = 0
iPos = InStr(iPos + 1, stringToSearch, searchFor)
If iPos <> 0 Then iTemp = iPos
Loop

iPos = iTemp - 1

LastIndexOf = iPos

End Function
```

Now with this function at our disposal, it is still a bit messier than the .Net version:

VBA

```
Dim fullname As String
Dim filelocation As String
Dim filename As String
Dim extension As String

fullname = swModel.GetPathName()
filelocation = Left(fullname, LastIndexOf(fullname, "\"))

Dim filenamewithext As String
filenamewithext = Right(fullname, Len(fullname) - Len(filelocation) - 1)
filename = Left(filenamewithext, LastIndexOf(filenamewithext, "."))
extension = Right(fullname, Len(fullname) - LastIndexOf(fullname, "."))
```

Title & Type of File

Getting the title is simple enough; this is the name that is displayed in the windows explorer when you select to open it, so if you have the windows option of displaying known file extensions (such as .doc, .txt etc...) then the title will have the extension also.

You have already seen how to get the type of file from the previous chapter, but not how to change the numeric value we got back into a text or descriptive name for the user.

C#

```
string title = swModel.GetTitle();
```

```
string filetype = ((swDocumentTypes_e)swModel.GetType()).ToString();
```

The title is self-explanatory. As for the file type; we start by calling **swModel.GetType()**, which returns an integer value. From there we cast it to the enumerator type **swDocumentTypes_e**, which will give us an enumerator object, and then we simply convert that to a string representation.

VBA

```
Dim title As String
Dim filetype As Integer
title = swModel.GetTitle()
filetype = swModel.GetType()
```

Due to its very limited powers VBA has no easy way to get a string representation of an enumerator, so we simply get a number. Obviously not much use for the user to see, but if you came to that level anyway I would recommend using C# or VB.Net.

File/Model Modes

As well as titles and paths, we may need to know if the file we are working on is capable of having itself altered and saved, or if we can edit it or not. This is where file modes come in to play.

We will get all 3 file modes at once and store them in **Boolean** variables:

C#

```
bool bReadOnly = swModel.IsOpenedReadOnly();
bool bViewOnly = swModel.IsOpenedViewOnly();
bool bLargeMode = swModel.LargeAssemblyMode;
```

VBA

```
Dim bReadOnly As Boolean
Dim bViewOnly As Boolean
Dim bLargeMode As Boolean

bReadOnly = swModel.IsOpenedReadOnly()
bViewOnly = swModel.IsOpenedViewOnly()
bLargeMode = swModel.LargeAssemblyMode
```

Documents Material

Another piece of potentially useful information is the current material that is set for the part. The material has several values; the visible user-friendly name, and the internal ID.

C#

```
string materialName = swModel.MaterialUserName;
string materialID = swModel.MaterialIdName;
```

VBA

```
Dim materialName As String
Dim materialID As String
materialName = swModel.MaterialUserName
materialID = swModel.MaterialIdName
```

That was easy enough wasn't it? If you retrieve these values from an assembly or drawing, or a part without a material set, you simply get a blank string returned.

Configuration Names

Sometimes you may need the names of all configurations within a part or assembly, for things such as retrieving preview images, settings custom properties or printing.

C#

```
string[] configNames = (string[])swModel.GetConfigurationNames();
foreach (string configname in configNames)
{
    // Do anything with configname here
}
```

Here in the first line we create a new array of string values, and then cast the object returned from the **swModel** function **GetConfigurationNames** to the correct type.

Then, to access each configuration name one by one, we use a **foreach** loop.

And here is the VBA version:

VBA

```
Dim configNames As Variant
Dim configname As Variant
configNames = swModel.GetConfigurationNames()

For Each configname In configNames
   ' Do anything with configname here
Next
```

Summary Information

Mainly useful for tracking, searching or archiving, the summary information is a powerful piece of information to have at your disposal. The **Summary Information** is the information set in the **File->Properties** form such as title, subject, author, keywords, comments, saved by, creation date and save date.

The .Net versions may seem more complicated, but this is purely because we are using an automated method, that will update itself whenever the SolidWorks enumerator values change, whereas the VBA version is simply a static code that may well become invalid:

C#

```
string[] summmaryNames =
Enum.GetNames(typeof(swSummInfoField_e));
foreach (string summaryName in summaryNames)
{
   int sumId = (int)Enum.Parse(typeof(swSummInfoField_e),
summaryName);
   string summaryValue = swModel.get_SummaryInfo(sumID);
}
```

Firstly we get all of the names in the **swSummaryInfoField_e** enumerator, which contains all of the ID values we need to retrieve the values of the summary fields.

With the summary names (which will be Title, Subject, Author etc…), we then go through them one at a time using another **foreach** loop and call the **swModel** function **get_SummaryInfo**, which takes a single parameter of an ID, which we retrieve by parsing the name of the enumerator back to an actual enumerator, and then casting it into an integer. This seems a bit messy but it is in fact about the only way to automatically iterate through a set of enumerator values. Now we access all the information we require through the **summaryName**, **sumID** and **summaryValue** each time we enter the loop.

In VBA, we cannot automatically get the names of an enumerator set so we have to manually type each one in. Be warned this is not good programming practise:

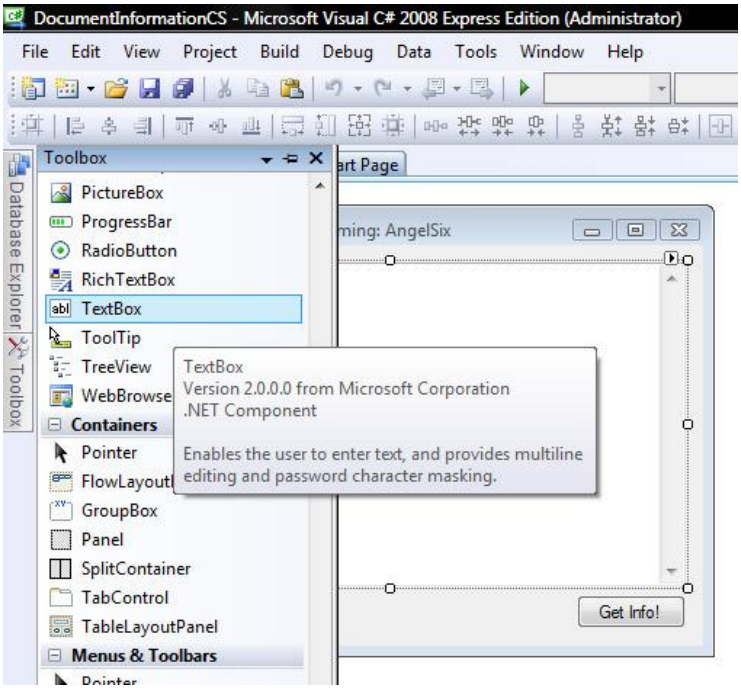
VBA

```
Dim sumInfoTitle As String
Dim sumInfoSubject As String
Dim sumInfoAuthor As String
Dim sumInfoKeywords As String
Dim sumInfoComment As String
Dim sumInfoSavedBy As String
Dim sumInfoCreateDate As String
Dim sumInfoSaveDate As String
Dim sumInfoCreateDate2 As String
Dim sumInfoSaveDate2 As String

sumInfoTitle = swModel.summaryinfo(swSumInfoTitle)
sumInfoSubject = swModel.summaryinfo(swSumInfoSubject)
sumInfoAuthor = swModel.summaryinfo(swSumInfoAuthor)
sumInfoKeywords = swModel.summaryinfo(swSumInfoKeywords)
sumInfoComment = swModel.summaryinfo(swSumInfoComment)
sumInfoSavedBy = swModel.summaryinfo(swSumInfoSavedBy)
sumInfoCreateDate = swModel.summaryinfo(swSumInfoCreateDate)
sumInfoSaveDate = swModel.summaryinfo(swSumInfoSaveDate)
sumInfoCreateDate2 = swModel.summaryinfo(swSumInfoCreateDate2)
sumInfoSaveDate2 = swModel.summaryinfo(swSumInfoSaveDate2)
```

Display Document Information

Now we have gathered all this useful information, it would be nice to actually see it. We start by creating a multiline textbox to put the information into, and then simply add all of the information separated by a new line for each item.

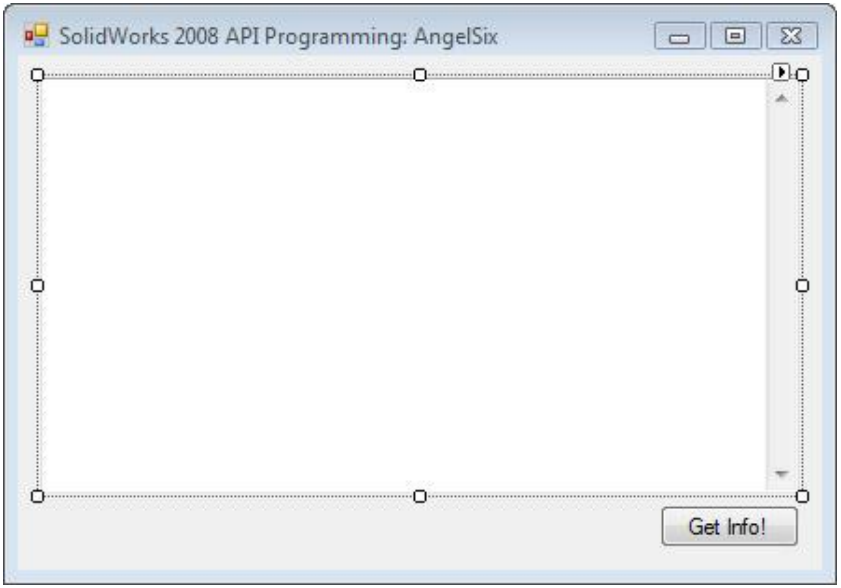
In the .Net languages we already have a form created automatically for us, and if you are using the template from the first chapter to run this code, we already have a button, so all that is left is to add a textbox. Go to the **Form Designer** by double-clicking the **Form1.cs** file in the **Solution Explorer**. Now go to the **Toolbox** to the left and this time instead of dragging a **Button** object on to our form we are going to drag a **TextBox** object on.



Now drag the textbox to any size you would like and position it as such. Then making sure it is still selected, go to the **Properties Window** to the right, and set the following properties:

Multiline	True
ScrollBars	Vertical

And here is what my form looks like:



Now switch back to the coding of the form. Within the buttons event handler function, we write the code to connect to SolidWorks and to get the active document code, and then all of the code we have just been writing above. It should all make sense if you have been following along, and the only thing different is that instead of storing the information in separate variables, we have simply made one string variable that stores all of the values, separated by a new line.

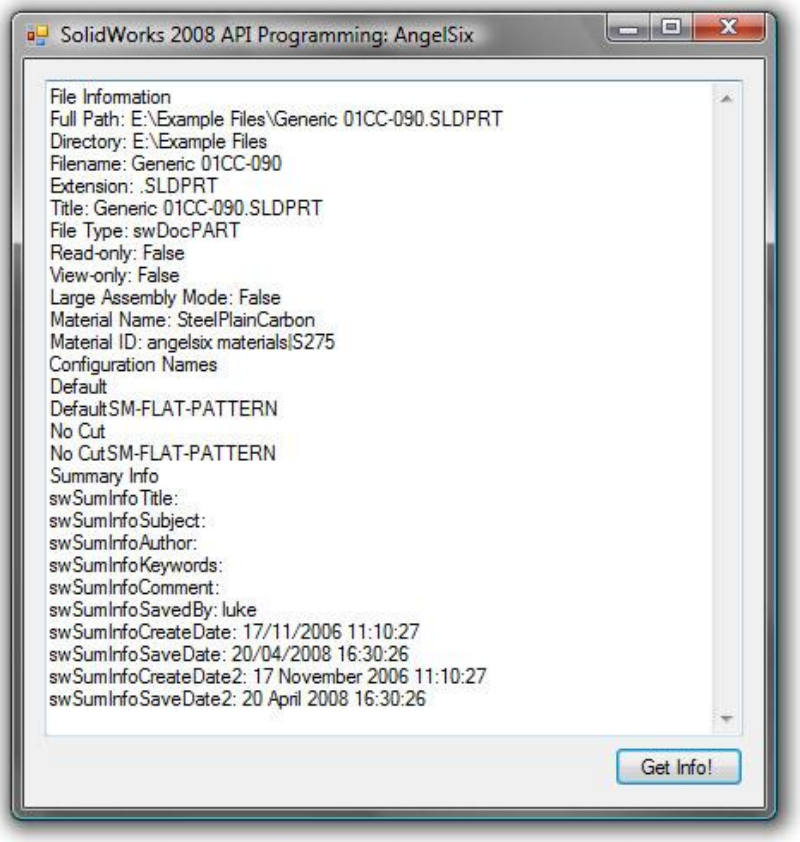
C#

```
try
{
    swApp =
(SldWorks.SldWorks)Marshal.GetActiveObject("SldWorks.Application");
}
catch
{
    MessageBox.Show("Error getting SolidWorks Handle");
    return;
}

swModel = (ModelDoc2)swApp.ActiveDoc;

if (swModel == null)
{
    MessageBox.Show("Failed to get active document");
    return;
}

string nl = System.Environment.NewLine;
string strInfo = "File Information" + nl;

string fullpath = swModel.GetPathName();
strInfo += "Full Path: " + fullpath + nl;
if (fullpath != string.Empty)
{
    strInfo += "Directory: " + Path.GetDirectoryName(fullpath) + nl;
    strInfo += "Filename: " + Path.GetFileNameWithoutExtension(fullpath)
+ nl;
```

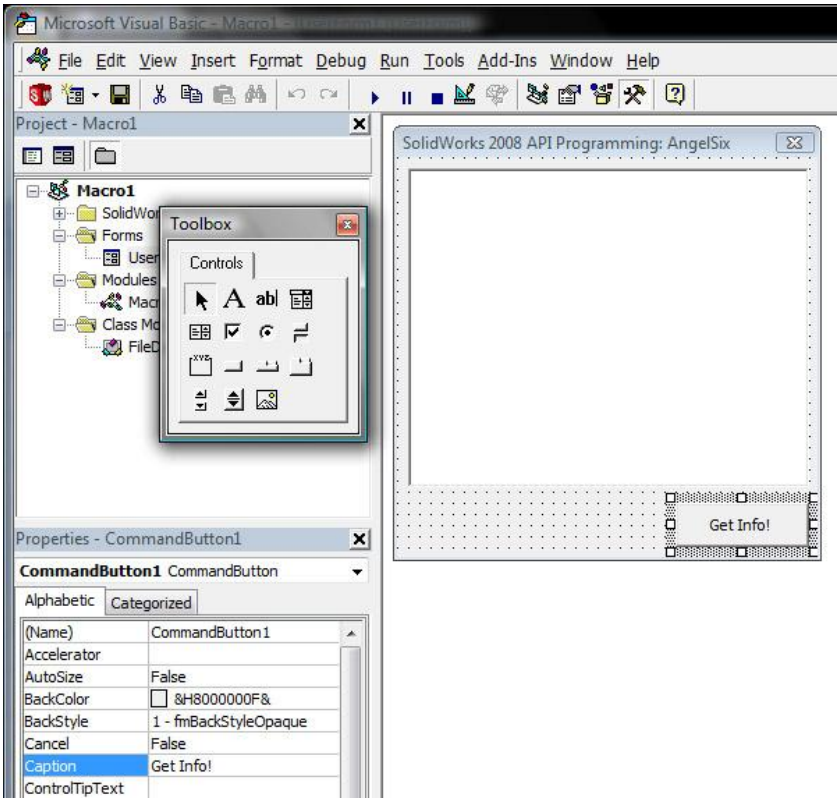
```
    strInfo += "Extension: " + Path.GetExtension(fullpath) + nl;
}

strInfo += "Title: " + swModel.GetTitle() + nl;
strInfo += "File Type: " +
((swDocumentTypes_e)swModel.GetType()).ToString() + nl;

strInfo += "Read-only: " + swModel.IsOpenedReadOnly().ToString() + nl;
strInfo += "View-only: " + swModel.IsOpenedViewOnly().ToString() + nl;
strInfo += "Large Assembly Mode: " +
swModel.LargeAssemblyMode.ToString() + nl;

strInfo += "Material Name: " + swModel.MaterialUserName + nl;
strInfo += "Material ID: " + swModel.MaterialIdName + nl;

strInfo += "Configuration Names" + nl;
string[] configNames = (string[])swModel.GetConfigurationNames();
foreach (string configname in configNames)
{
    strInfo += configname + nl;
}

strInfo += "Summary Info" + nl;
string[] summaryNames =
Enum.GetNames(typeof(swSummInfoField_e));
foreach (string summaryName in summaryNames)
{
    int sumID = (int)Enum.Parse(typeof(swSummInfoField_e),
summaryName);
    string summaryValue = swModel.get_SummaryInfo(sumID);
```

```
    strInfo += summaryName + ": " + summaryValue + nl;
}

textBox1.Clear();
textBox1.Text = strInfo;
```

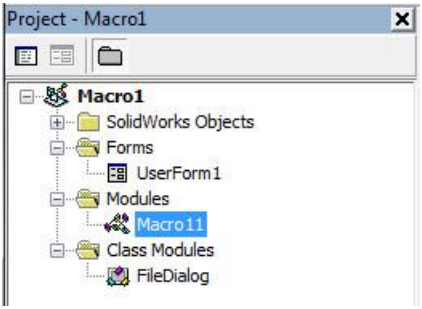
Any here is the finished result when you run the code:



For VBA we do much the same thing; start by going to **Insert->User Form** within the VBE. Then much like VS just drag a **CommandButton** and a **TextBox** onto the form and from the properties on the left, set the **TextBox** property **MultiLine** to **True**, and change the **Caption** property of the button to whatever you would like the button to say.



Now double-click the button to create the event handler for the click. Like VS we will place all of our code in here, and modify the module containing the **main()** function to simply open our form.



Start by editing the **Macro1** file module by double-clicking it in the **Project Explorer**.

Within the **main()** function place the following code to

open the form upon running of the macro.

```
Sub main()
UserForm1.Show
End Sub
```

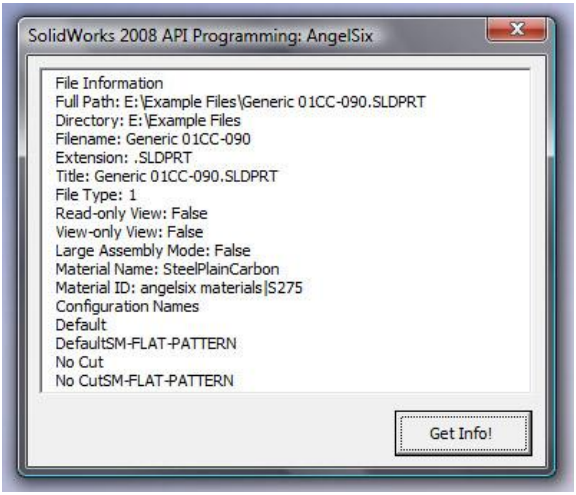
If you left the form **Name** property as default, then it will be called **UserForm1**. If you do not get the **IntelliSense** menu popping up when you press the period key after its name and before typing **Show,** then you have the wrong name, so check the **name** property of the form in the properties of the form designer.

Below this main subroutine we can place the **LastIndexOf** function we created earlier, or we can put this in the form code. I have placed it below the main for clarity.

```
Function LastIndexOf(stringToSearch As String, searchFor As String) As
Integer

Dim iPos As Integer
Dim iTemp As Integer
iPos = -1
iTemp = 0

Do Until iPos = 0
If iPos = -1 Then iPos = 0
iPos = InStr(iPos + 1, stringToSearch, searchFor)
If iPos <> 0 Then iTemp = iPos
Loop
```

```
iPos = iTemp - 1

LastIndexOf = iPos

End Function
```

Now it s time to place all of our code for getting the information that we wrote above into the event handler function of the button that was created earlier, so that the information will be shown in the textbox when the user clicks the button. To get back to the coding window with the button event handler just right-click the **UserForm1** item in the **Project Explorer** and select **View Code**.



VBA

```
Dim strInfo As String

Private Sub CommandButton1_Click()

Set swApp = GetObject("", "SldWorks.Application")
If swApp Is Nothing Then
```

```
MsgBox "Error gettings SolidWorks Handle"
Exit Sub
End If

Set swModel = swApp.ActiveDoc
If swModel Is Nothing Then
MsgBox "Failed to get active document"
Exit Sub
End If

Dim fullname As String
Dim filenamewithext As String
Dim filelocation As String

AddToStr "File Information"

fullname = swModel.GetPathName()
AddToStr "Full Path: " & fullname
If fullname <> "" Then
filelocation = Left(fullname, LastIndexOf(fullname, "\"))
AddToStr "Directory: " & filelocation

filenamewithext = Right(fullname, Len(fullname) - Len(filelocation) - 1)
AddToStr "Filename: " & Left(filenamewithext,
LastIndexOf(filenamewithext, "."))
AddToStr "Extension: " & Right(fullname, Len(fullname) -
LastIndexOf(fullname, "."))
End If

AddToStr "Title: " & swModel.GetTitle()
```

```
AddToStr "File Type: " & swModel.GetType()

AddToStr "Read-only View: " & swModel.IsOpenedReadOnly()
AddToStr "View-only View: " & swModel.IsOpenedViewOnly()
AddToStr "Large Assembly Mode: " & swModel.LargeAssemblyMode

AddToStr "Material Name: " & swModel.MaterialUserName
AddToStr "Material ID: " & swModel.MaterialIdName

AddToStr "Configuration Names"
configNames = swModel.GetConfigurationNames()

Dim configname As Variant
For Each configname In configNames
   AddToStr CStr(configname)
Next

AddToStr "Summary Information"
AddToStr "Title: " & swModel.summaryinfo(swSumInfoTitle)
AddToStr "Subject: " & swModel.summaryinfo(swSumInfoSubject)
AddToStr "Author: " & swModel.summaryinfo(swSumInfoAuthor)
AddToStr "Keywords: " & swModel.summaryinfo(swSumInfoKeywords)
AddToStr "Comment: " & swModel.summaryinfo(swSumInfoComment)
AddToStr "Saved By: " & swModel.summaryinfo(swSumInfoSavedBy)
AddToStr "Creation Date: " &
swModel.summaryinfo(swSumInfoCreateDate)
AddToStr "Save Date: " & swModel.summaryinfo(swSumInfoSaveDate)
AddToStr "Creation Date 2: " &
swModel.summaryinfo(swSumInfoCreateDate2)
```

```
AddToStr "Save Date 2: " &
swModel.summaryinfo(swSumInfoSaveDate2)

TextBox1.Text = ""
TextBox1.Text = strInfo

End Sub

Sub AddToStr(info As String)
strInfo = strInfo & info & vbCr
End Sub
```

* Find the complete C#, VBA and VB.Net examples on the CD if you have any troubles.

Working with Selected Objects

Identifying Selected Objects

Mating Selected Objects

Setting Material of Selected Objects

Manipulate Dimension

Selecting Objects

Another major need for any good SolidWorks programmer is the ability to work with and manipulate the selected objects. And the first task once you obtain the selection is to identify it, so you know how to proceed.

Indentifying Selected Objects

When you run a macro or program and the user has previously selected objects, you may wish to either clear them all so it doesn't affect your program, or to use the selection. If you wish to clear the selected objects you call:

C#

```
swModel.ClearSelection2(true);
```

VBA

```
swModel.ClearSelection2 True
```

Now, say you wanted to actually use the selected objects that the user has picked to do something. We must identify each of these selections and make sure that they are of the correct type, and in some cases, even in the correct order. I am going to show you how to identify the selected objects of an assembly, add a coincident mate and then error check.

Start with the usual template or whichever one you please, and connect to SolidWorks and get the active document as usual.

If we are to work with selections we must add another variable to our program. Please this in the same location as the **swApp** and

swModel variables and call it **swSelMgr** (short for Selection Manager), and make it of type **SelectionMgr**.

C#

```
SelectionMgr swSelMgr;
```

VBA

```
Dim swSelMgr As SelectionMgr
```

Next, after we have successfully acquired the active document, we then get the **SelectionManager** of that document. We use this object for getting all our information about and handles to the selected objects.

C#

```
swSelMgr = (SelectionMgr)swModel.SelectionManager;
```

VBA

```
Set swSelMgr = swModel.SelectionManager
```

Now we must check that the active document is an assembly, and if so, get the selected objects and check that they are of the correct type. For this example we are only going to allow mating of **Faces**, **Edges** and **Vertices**. We will create a better selection process later using the **Property Pages**.

C#

```
if (swModel.GetType() != (int)swDocumentTypes_e.swDocASSEMBLY)
{
    MessageBox.Show("Can only add mate in an assembly");
    return;
}

if (swSelMgr.GetSelectedObjectCount2(-1) != 2)
{
    MessageBox.Show("You must select 2 entities, no more, no less");
    return;
}
else
{
    int selID1 = swSelMgr.GetSelectedObjectType3(1, -1);
    int selID2 = swSelMgr.GetSelectedObjectType3(2, -1);
    if ((selID1 != (int)swSelectType_e.swSelEDGES && selID1 !=
(int)swSelectType_e.swSelFACES && selID1 !=
(int)swSelectType_e.swSelVERTICES)
        || (selID2 != (int)swSelectType_e.swSelEDGES && selID2 !=
(int)swSelectType_e.swSelFACES && selID2 !=
(int)swSelectType_e.swSelVERTICES))
    {
        MessageBox.Show("You must select only edges, faces or
vertices");
        return;
    }
}
```

Here we first check the type of document that is active. You have seen this before so if you need an explanation return to the previous chapter. Then we use a function of the acquired selection manager called **GetSelectionObjectCount2**; this returns an integer value representing the number of objects that are selected by the user. Because we are doing a coincident mate we want exactly two.

Next we call another selection manager function called **GetSelectedObjectType3**, which accepts one parameter which is the selected object to get the type for. We get the type value for each of the 2 selected items, and then check them against the enumerator values for **Faces**, **Edges** and **Vertices**. If either of the two objects is not one of the allowed selected type then we stop and warn the user.

```
&& = AND
|| = OR
```

If we pass all of the above criteria we know we are in SolidWorks, with an assembly open, and 2 entities selected. We could take this further and check that the selected objects components are different so we are not trying to mate an object to itself, but for now this will do us.

Mating Selected Objects

All that is left to do now is attempt to apply a coincident mate between the 2 selected entities and check for success.

If the operation succeeds, SolidWorks by default when programmatically adding a mate does not deselect the selected entities after, but the user operation does. So, in an attempt to act more like an actual user mate procedure, we will deselect the entities once the mate has been made.

```
int iErrors;
((AssemblyDoc)swModel).AddMate3(
    (int)swMateType_e.swMateCOINCIDENT,
    (int)swMateAlign_e.swMateAlignCLOSEST,
    false, 0, 0, 0, 0, 0, 0, 0, 0, false,
    out IErrors);

if (iErrors != (int)swAddMateError_e.swAddMateError_NoError)
{
    MessageBox.Show("Error mating components: " +
((swAddMateError_e)iErrors).ToString());
}
else
{
    swModel.ClearSelection2(true);
}
```

If you read the SolidWorks API help file for **AssemblyDoc::AddMate3** you will see that the only parameters that

matter for us are the first 3 and the last 2. After we cast our **ModelDoc2 swModel** variable to an **AssemblyDoc** variable we can use this function.

```
Mate Type, Mate Alignment, Flip, Positioning Only,
Error variable
```

We select the correct type and alignment from the enumerator options the help file tells us about, tell it not to flip the components, not to create a positioning-only mate, and we pass a variable for storing our errors.

Once it has run we check for errors by comparing the resultant variable with the **swAddMateError_NoError** value from the errors enumerator, and if we have an error, convert it to a readable string and display it.

Once you have done, test your program.

Doing this in VBA is exactly the same, only we would have to manually state each error again for each enumerator value like before as VBA cannot easily, if at all, convert enum to strings at runtime.

VBA

```
Set swSelMgr = swModel.SelectionManager

If swModel.GetType() <> swDocumentTypes_e.swDocASSEMBLY Then
MsgBox "Can only add mate in an assembly"
Exit Sub
End If
If swSelMgr.GetSelectedObjectCount2(-1) <> 2 Then
```

```
    MsgBox "You must select 2 entities, no more, no less"
    Exit Sub
Else
    Dim selID1 As Integer
    Dim selID2 As Integer
    selID1 = swSelMgr.GetSelectedObjectType3(1, -1)
    selID2 = swSelMgr.GetSelectedObjectType3(2, -1)
    If (selID1 <> swSelEDGES And selID1 <> swSelFACES And selID1
<> swSelVERTICES) Or (selID2 <> swSelEDGES And selID2 <>
swSelFACES And selID2 <> swSelVERTICES) Then
        MsgBox "You must select only edges, faces or vertices"
        Exit Sub
    End If
End If
Dim iErrors As Long
swModel.AddMate3 swMateCOINCIDENT, swMateAlignCLOSEST,
False, 0, 0, 0, 0, 0, 0, 0, 0, False, iErrors

If iErrors <> swAddMateError_NoError Then
    Dim errortext As String
    errortext = Switch(iErrors = 0, "Error Unknown", iErrors = 2, "Incorrect
Mate Type", iErrors = 3, "Incorrect Alignment", iErrors = 4, "Incorrect
Selection", iErrors = 5, "Over Defined Assembly", iErrors = 6, "Incorrect
Gear Ratios")
    MsgBox "Error mating component: " & errortext
Else
    swModel.ClearSelection2 True
End If
```

In VBA, we have had to specify each error value and its corresponding string value. For this we use a **Switch** statement; it works by returning the variable after the first value that returns true. So in this case, we check if "iErrors = 0", if it does it returns the variable after that, which is "Error Unknown", if it is false, it carries on until it finds one.

Settings Material of Selected Objects

As well as mating, you can do almost anything with the selected objects. This quick example will show you that by selecting any edge, face, vertex, plane, sketch or part itself, that we can find its associated component, and set its material.

This code presumes you have already connected to SolidWorks, got the active document and obtained the **SelectionManager** object, and SolidWorks has an assembly already open and you have checked that the user has pre-selected at least one object.

You know how to check for the number of selected objects and their types now so I will just get down to the raw code of getting a handle to the **Component** of the selected object, and setting its material.

C#

```
string materialDB = "solidworks materials.sldmat";
string material = "Ductile Iron";

for (int i = 1; i <= swSelMgr.GetSelectedObjectCount2(-1); i++)
{
   Component2 comp =
(Component2)swSelMgr.GetSelectedObjectsComponent3(i, -1);

   if (comp != null)
   {
      ModelDoc2 model = (ModelDoc2)comp.GetModelDoc();
      if (model.GetType() == (int)swDocumentTypes_e.swDocPART)
         ((PartDoc)model).SetMaterialPropertyName2("", materialDB,
material);
```

```
    }
}

swModel.EditRebuild3();
```

Firstly we specify the materials database to use; either a custom database or the default SolidWorks database. Secondly we specify the material from that library that we wish to set all selected parts to.

Now we create a loop for every selected object. Within that loop we firstly get the component that is associated with this selected object. The **GetSelectedObjectsComponent3** function accepts the selection index, which we pass each time. All the function does is simply jump up the feature tree until hitting a **Component** object, so no matter what item or feature of a part you select, this function will get its component.

Before we continue, we check the function actually returned one, and then if did we get the **ModelDoc2** object associated with that component, as we cannot set a material value to a component, it has to be to the document of the component (i.e. the part/assembly).

With a **ModelDoc2** object to work with, it is just a case of checking its type to make sure it is a part, and then setting its material properties using **SetMaterialPropertyName2**. This function takes in the configuration name, material database and material name. In this example we pass a blank configuration name so it sets it in the active configuration. And then finally with all selected items processed we do a rebuild of the assembly to update the parts.

Doing this procedure in VBA is no different and due to the looseness of the language it actually looks a bit simpler in comparison as we do not need any casting.

VBA

```
Dim materialDB As String
Dim material As String

materialDB = "solidworks materials.sldmat"
material = "Ductile Iron"

Dim i As Integer
For i = 1 To swSelMgr.GetSelectedObjectCount2(-1)
   Dim comp As Component2
   Set comp = swSelMgr.GetSelectedObjectsComponent3(i, -1)

   If Not comp Is Nothing Then
      Dim model As ModelDoc2
      Set model = comp.GetModelDoc()

      If model.GetType() = swDocPART Then
         model.SetMaterialPropertyName2 "", materialDB, material
      End If
   End If
Next
swModel.EditRebuild3
```

Manipulating Dimensions

One of the major features of any 3D software package is dimensions, and even through altering them through code it as simple as 1-2-3, here is a quick code snippet of how to double the value of the selected dimension. Again, presuming you are up to the point of having a working selection manager and the user has selected a dimension.

C#

```
if (swSelMgr.GetSelectedObjectCount2(-1) == 0)
{
    MessageBox.Show("You must select one or more objects");
    return;
}

if (swSelMgr.GetSelectedObjectType3(1, -1) !=
(int)swSelectType_e.swSelDIMENSIONS)
{
    MessageBox.Show("You must select a dimension");
    return;
}

DisplayDimension dispDim =
(DisplayDimension)swSelMgr.GetSelectedObject6(1, -1);
Dimension dim = (Dimension)dispDim.GetDimension2(0);

if (dim.DrivenState !=
(int)swDimensionDrivenState_e.swDimensionDriving || dim.ReadOnly)
{
```

```
    MessageBox.Show("Dimension cannot be altered");
    return;
}

double[] oldVals =
(double[])dim.GetSystemValue3((int)swInConfigurationOpts_e.swThisCo
nfiguration, null);
oldVals[0] *= 2;
int retVal = dim.SetSystemValue3(oldVals[0],
(int)swInConfigurationOpts_e.swThisConfiguration, null);

if (retVal != (int)swSetValueReturnStatus_e.swSetValue_Successful)
{
    MessageBox.Show("Error setting dimension: " +
((swSetValueReturnStatus_e)retVal).ToString());
}

swModel.EditRebuild3();
```

We start by checking that the user has selected an item. If they have we check that the selection is of type **swSelDimensions** before proceeding to attempt to alter it.

Once we know we have a dimension selected, we get it using **GetSelectedObject6** and pass **1** as the index to retrieve the first selected object, and just ignore any more after that.

In order to get and set dimension values we must get the actual **Dimension** object, not the **DisplayDimension** that is retrieved by the user selection. We use the **DisplayDimension** method

GetDimension2 to achieve this; passing **0** into this function gets the first chamfer dimension and **1** gets the second if you have multiple dimension display. In this example we are working with single dimensions, so just pass **0**.

Next, we check that the dimension is not read-only or driven; else we will not be able to alter it.

With everything OK up to this point, we now want to perform our operation; double the original size. Firstly we start by getting the system value, which is the actual fully evaluated value of the dimension. If you have tolerances then you will retrieve multiple dimensions, so you must loop them and handle them how you like, but for now this example will only alter the first dimension.

To get the system value we use the **Dimension** function **GetSystemValue3**, which takes two parameters; the first is the configuration to get the value from, which is an enumerator, and the second is the actual configuration names if we use the enumerator value for specific configurations. In our example we do not need this so we pass **null**. The return type is a **double array** (a variable containing zero or more numbers) regardless of retrieving a single value or multiple, so we set our double array variable to the result.

Now with the original values in an array, we access the first value in that array and multiply it using the statement:

```
oldVals[0] *= 2;
```

Which could be written out the long way like this:

```
oldVals[0] = oldVals[0] * 2;
```

With the value doubled we must now set this new value back to the dimension. We do this with the equivalent function **SetSystemValue3**. This however accepts a single double value not an array, so we pass the first value of our array as the first parameter, and then the enum for this configuration again, and **null** for the configuration names. This function returns an error checking enumerator value, so we retrieve this like any other function we have checked before and then check it against the successful enum value. If it is not successful something went wrong so we display the error by casting the integer return value back to an enumerator value, and then to a readable string.

Finally, with the dimension successfully altered or not, we do a rebuild of the document.

This code will work on assemblies, parts *and* drawings, provided the dimension is alterable.

The VBA is exactly the same except the usual lack of enum names, and no need to cast any objects.

VBA

```
If swSelMgr.GetSelectedObjectCount2(-1) = 0 Then
   MsgBox "You must select one or more objects"
   Exit Sub
End If
If swSelMgr.GetSelectedObjectType3(1, -1) <> swSelDIMENSIONS
Then
   MsgBox "You must select a dimension"
   Exit Sub
End If
```

```
Dim dispDim As DisplayDimension
Dim ddim As Dimension

Set dispDim = swSelMgr.GetSelectedObject6(1, -1)
Set ddim = dispDim.GetDimension2(0)

If ddim.DrivenState <> swDimensionDriving Or ddim.ReadOnly Then
   MsgBox "Dimension cannot be altered"
   Exit Sub
End If

Dim oldVals As Variant
oldVals = ddim.GetSystemValue3(swThisConfiguration, Nothing)
oldVals(0) = oldVals(0) * 2

Dim retVal As Integer
retVal = ddim.SetSystemValue3(oldVals(0), swThisConfiguration,
Nothing)

If retVal <> swSetValue_Successful Then
   MsgBox "Error setting dimension"
End If

swModel.EditRebuild3
```

And there you have it! Hopefully by now you are getting the hang of things. Once you know the basics such as casting objects to the correct value in .Net languages, and retrieving enumerator values

and checking them for errors and the likes, the procedure is pretty much the same for all functions and properties.

The main problem with the SolidWorks API documentation is its lack of explanations in regards to these topics, but now you know how to convert enumerators, integers, double arrays, **ModelDoc2** to any of the 3 children **AssemblyDoc**, **PartDoc** and **DrawingDoc**, as well as converting errors into messages and other techniques, you should be well on your way.

Selecting Objects

One more thing I would like to cover before we move on is how you actually select objects within your code. Because there are many different ways of doing this and it all comes down to your situation, I will just list the single lines of code you use to select specific objects. We will encounter some of them later in the book, but they are simple enough to understand.

ModelDocExtension::SelectByID2

In order to select almost any object from any location, you can use the universal **SelectByID2** of **ModelDocExtension** of **ModelDoc2**, where you pass the fully qualified name of the entity you would like to select. This name is based on the location of the entity within the **ModelDoc2** object that you are calling the function from. The best way to find your required ID name is to record a macro selecting the part you would like.

The definition is as follows:

```
retval = ModelDocExtension.SelectByID2 ( Name, Type,
X, Y, Z, Append, Mark, Callout, SelectOption )
```

Name	*String*
Type	*String*
X, Y, Z	*Double*
Append	*Boolean*
Mark	*Integer*
Callout	*Pointer*
SelectOption	*Enumerator (Integer)*

The **Name** is the fully qualified name of the object to select that I will explain further in a second.

The **Type** is the string name of the type of entity you are going to select. If you take a look at the help it will give you all of the types.

The **X**, **Y** and **Z** positions are used if you wish to select by position instead of the name.

The **Append** is used to tell SolidWorks whether you intend to add this selection to the currently already selected objects or not, and if this exact object is already selected, whether to deselect it.

The **Mark** is used if we are setting marks to selections, which is beyond the scope of this book.

The **Callout** is used mainly in add-ins and again is beyond this book.

Finally, the **SelectOption** is an enumerator of type **swSelectOption_e**; you can basically pick either **Default**, or **Extensive**.

In order to select anything by its ID, in most cases you need the fully qualified name. In order to work out this name you must follow the following format:

```
 [Name]@[Feature Name]@[Part]-[InstanceID]@[Top-
Level Assembly]/[SubAssembly]-[InstanceID]@[Bottom-
Level Assembly]
```

Omitting any field that you don't need. Here are a few examples:

From an assembly, selecting a parts thickness dimension

```
Part.Extension.SelectByID2("Thickness@Sheet-Metal1@0Bracket-
1@MachineContainer", "DIMENSION", ...
```

Where

```
Thickness          = Dimension name
Sheet-Metal1       = Feature Name
Bracket-1          = Part Name & Instance ID
MachineContainer   = Assembly Name
```

The model used above was an assembly called **MachineContainer** with a part within it called **Bracket**. That bracket was a sheet metal part (hence the **Sheet-Metal1** feature), with a dimension of the sheet metal feature called **Thickness**. The **-1** after the bracket is because in assemblies you need a way to know which part to select if there are multiple instances. The instance in SolidWorks is the number after the part within the angled brackets (**<>**).

From an assembly, selecting a top-level planes offset dimension from the top level

```
Part.Extension.SelectByID2("Web Thickness@Web Thickness
Offside@MachineContainer", "DIMENSION", ...
```

Where

```
WebThickness = Dimension name
Web Thickness Offset = Plane Name
MachineContainer = Assembly Name
```

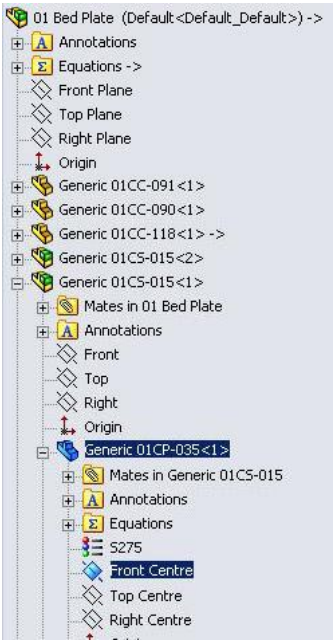
From an assembly, selecting a sub-assemblies parts plane

Assembly->SubAssembly->SubAssembliesPart->PartsPlane

```
Part.Extension.SelectByID2("Front Centre@Generic 01CS-015-1@01
Bed Plate/Generic 01CP-035-1@Generic 01CS-015", "PLANE", ...
```

Where

```
Front Centre              = Feature Name (Plane)
Generic 01CS-015-1        = Part Name & Instance ID
01 Bedplate               = Top-Level Assembly
/Generic 01CP-035-1       = Sub-Assembly & Instance ID
Generic 01CS-015          = Bottom-Level Assembly
```



The **Part** variable in all of these examples is the **ModelDoc2** object acquired for the model that the title states, such as "From an assembly...."

The best way to learn about this function is trial and error, and to record macros to select what you want and see the results.

Selecting specific objects

The alternate method to using the universal function is to use specific functions for the object you wish to select. This could be an annotation, a body, a component, an edge, a feature, a document (assembly/part/drawing), a sketch etc…

Find below a list of methods to use for each type of object to select. You should by now be able to figure out from the API reference, how to pass the correct parameters. As for actually getting a handle to the object in the first place; we will cover some of them in the next chapter.

`Annotation`	`Annotation::Select3`
`Body`	`Body2::Select2`
`BreakLine`	`BreakLine::Select`
`Component`	`Component2::Select3`
`Configuration`	`Configuration::Select2`
`Edge Point`	`EdgePoint::Select`
`Entity`	`Entity::Select4`
`Feature`	`Featur::Select2`
`Sketch Contour`	`SketchContour::Select2`
`Sketch Hatch`	`SketchHatch::Select4`
`Sketch Point`	`SketchPoint::Select4`
`Sketch Segment`	`SketchSegment::Select4`

Setting a Selection Filter

As you saw in the previous example we allowed the user to select anything first, and then check there selection after. A much better way is to limit the users' selection beforehand, and then tell them to select their objects, and then run the code. Unfortunately this is not easy to demonstrate the natural flow of setting the selection filter, getting the selection, and then once they have selected, carrying on with your code. The following function is usually used in add-ins or full-blown applications with a user interface. I intend to cover this subject in the future perhaps on a dedicated add-ins book.

However, just for knowledge, in order to limit the users' selection you call the following function:

```
void SldWorks.SetSelectionFilter ( selType, state )
```

selType	*Enumerator (swSelectType_e)*
state	*Boolean*

The **state** variable determines whether or not to enable/disable that selection, to allow or deny the user from selecting it. The **selType** is an enumerator that can be stacked to select multiple options. An example to enable the user to select bodies and edges would be:

C#

```
swApp.SetSelectionFilter(swSelectType_e.swSelFACES |
swSelectType_e.swSelEDGES);
```

VBA

```
swApp.SetSelectionFilter swSelFACES Or swSelEDGES
```

Property Manager Pages

Deriving the base class

Adding items to the Page

Responding to events

Many of you who have already attempted to create Property Manager Pages from the provided documentation of SolidWorks have probably not understood 90% of the code, but rather done what the documentation has told you, accepted it as *the* way to create a page, and modified the bits you want to work with in the hopes that it will work. Am I correct?

Well now, prepare to go through step-by-step every stage of creating a PMP in VBA and the .Net languages. I will take you through each stage, explaining along the way the reason for every part of the code, and how to do many things with a property manager page.

You can treat PMP's more like forms in the way that you can have your code working from events instead of loops and straight through until completion. You can run some initialisation code, wait for the user to select components and click a button, and then react to that. Sound exciting? Let's get started.

Deriving the base class

In order to create a PMP we must first create a class. This class is created based on the template of a PMP called an **abstract class**; this means that before we can create a page we must create an object (class) that has all the required properties and functions required by a PMP. By "deriving" from this base PMP class, it ensures that our new class has all of these required properties and methods. Think of it as if you want to create a new car, called the **SuperFly**, you cannot start by creating a new car with 3 square wheels, and no steering wheel. You have got to follow certain rules such as having 3-4 wheels, containing doors to get in, and having certain safety factors. These rules would be put inside a base abstract class called **Car** for

example, to ensure that any car created will follow these basic rules. We would derive our **SuperFly** car class from the base **Car** class, so that it inherited, and was forced to follow, these base rules.

In the case of our Property Manager Page, we are going to create a new page, called **MyFirstPMP** that is derived from the base PMP class, which is actually called **PropertyManagerPage2Handler5**.

When you derive from a base class, your new class first needs to state that it is deriving from that class, and secondly you must create all of the functions that the base class asks you too. So, we are going to create a new class, derive from the PMP base class, and specify all of the required blank functions.

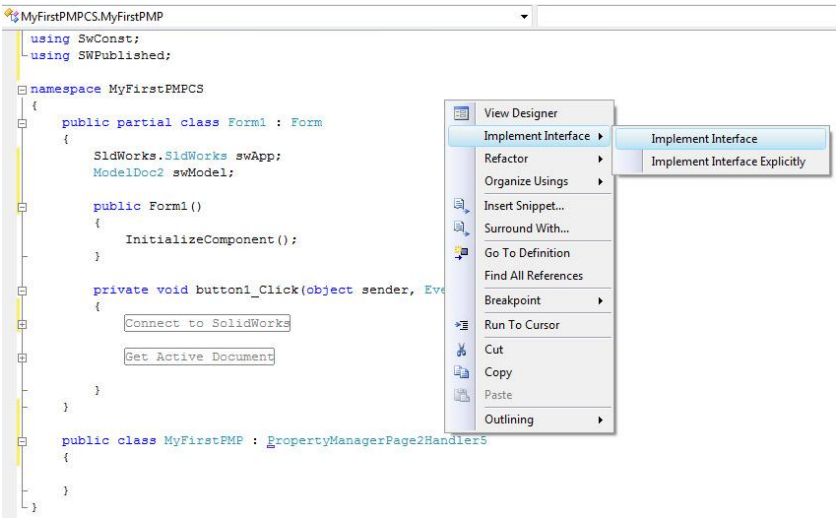
Start by creating a new project in.Net from the normal template with the button to start the code, and go to the coding view. Add the following **using** code:

```
using SWPublished;
```

This will allow us to access the base PMP class.

Now create a new class outside of the main form class but inside the namespace, and call it **MyFirstPMP**. Now for the important part; after the class name, but *before* the opening curly braces, place a colon (:), followed by the name of the base class to derive from, in this case **PropertyManagerPage2Handler5**.

Now we must implement all of the required functions of this base class, luckily .Net provides a method to do this for us; right-click on the base class name, and from the menu select **Implement Interface -> Implement Interface**:



This will automatically generate you all of the required functions of the base class. Trust me this is a life saver in cases like this with over 30 functions!

You will also notice that within each function there is one line of code throwing an error to the system. This is inserted by default and when the function is called it will throw a system error stating that the function is not yet implemented. This is not the functionality we want as we don't really care if some functions are not implemented. So remove that line of code to be left with empty functions. The end result looks like this:

C#

```
public class MyFirstPMP : PropertyManagerPage2Handler5
{
   #region IPropertyManagerPage2Handler5 Members
```

```
    public void AfterActivation(){}
    public void AfterClose(){}
    public int OnActiveXControlCreated(int Id, bool Status) { return
(int)swHandleActiveXCreationFailure_e.swHandleActiveXCreationFailure
_Cancel; }
    public void OnButtonPress(int Id){}
    public void OnCheckboxCheck(int Id, bool Checked){}
    public void OnClose(int Reason){}
    public void OnComboboxEditChanged(int Id, string Text){}
    public void OnComboboxSelectionChanged(int Id, int Item){}
    public void OnGroupCheck(int Id, bool Checked){}
    public void OnGroupExpand(int Id, bool Expanded){}
    public bool OnHelp(){}
    public bool OnKeystroke(int Wparam, int Message, int Lparam, int
Id){}
    public void OnListboxSelectionChanged(int Id, int Item){}
    public bool OnNextPage(){}
    public void OnNumberboxChanged(int Id, double Value){}
    public void OnOptionCheck(int Id) { }
    public void OnPopupMenuItem(int Id) { }
    public void OnPopupMenuItemUpdate(int Id, ref int retval) { }
    public bool OnPreview() { }
    public bool OnPreviousPage() { }
    public void OnSelectionboxCalloutCreated(int Id) { }
    public void OnSelectionboxCalloutDestroyed(int Id) { }
    public void OnSelectionboxFocusChanged(int Id) { }
    public void OnSelectionboxListChanged(int Id, int Count) { }
    public void OnSliderPositionChanged(int Id, double Value) { }
    public void OnSliderTrackingCompleted(int Id, double Value) { }
```

```
    public bool OnSubmitSelection(int Id, object Selection, int SelType, ref
string ItemText){}
    public bool OnTabClicked(int Id){}
    public void OnTextboxChanged(int Id, string Text) { }
    public void OnUndo(){}
    public void OnWhatsNew(){}

    #endregion
}
```

I have moved the curly braces up to the same line instead of on new lines to save space in the coding; it just makes for easier reading.

Some functions you have to add a return statement for as they require feedback, but since we won't be using them in this book just type what you see. I will explain some functions later on and explain the return values and what they mean, but for now just accept what is written and write your code the same.

The **#region** and **#endregion** are Visual Studio styling tags, they are nothing to do with the program code and do not get compiled when you create your program. Their function is to tidy up your code. Anything within the **#region/#endregion** block can be expanded and collapsed using the **+** and **–** box to the left of the **#region** tag. This is very useful when your code gets long in order to keep it tidy.

Before we move on, let's create the same coding but in VBA. Start by creating the usual VBA template that connects to SolidWorks, and acquires the active document. Then leave the main function like that for now.

Go to **Insert->Class Module** to insert a new class coding file. Open up this class in the coding window if not already, by expanding the **Class Module** folder and double-clicking the **Class1** file. This should display a blank file in the coding window.

Before we start writing our code in here let me explain exactly where we are now; if this were C#, we would effectively be within the following code block:

```
public class MyFirstPMP :
PropertyManagerPage2Handler5
{ // We are here! }
```

The slight difference in VBA is that although we are effectively within the class when typing our code in the coding window, we do not specify the base class outside of the braces (because there aren't any), but inside the class. So, inside our class file code we now want to start by letting it know we are deriving from the **PropertyManagerPage2Handler5** base class. Place this code at the first line of your class file:

VBA

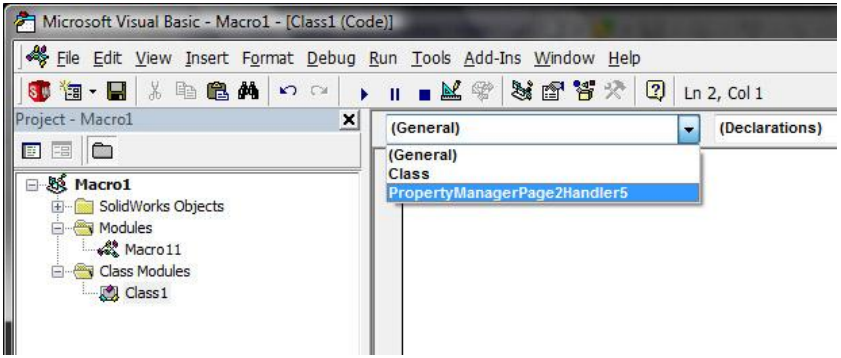
```
Implements PropertyManagerPage2Handler5
```

Now, unlike .Net languages VBA does not have the power to automatically create all of the required base functions for us, but don't panic! It does have a semi-automated method. In the coding window, notice the 2 drop-down boxes right above the first line of code. The one on the left is the **Object Box**, and the one on the right is the **Procedures/Events Box**.

Procedures is yet another word for a function or a method or a subroutine; each language uses its own term but they all mean exactly the same, unless you want to get pedantic and state that a subroutine should not return a value whereas all others can, but I wasn't going to say anything.

Creating the base functions in VBA

OK, so how do we semi-automatically create all of the required functions of our base class? Fairly easily; drop down the **Object Box** above your code to list all of the objects that are related to this class. Select the base class **PropertyManagerPage2Handler5**.

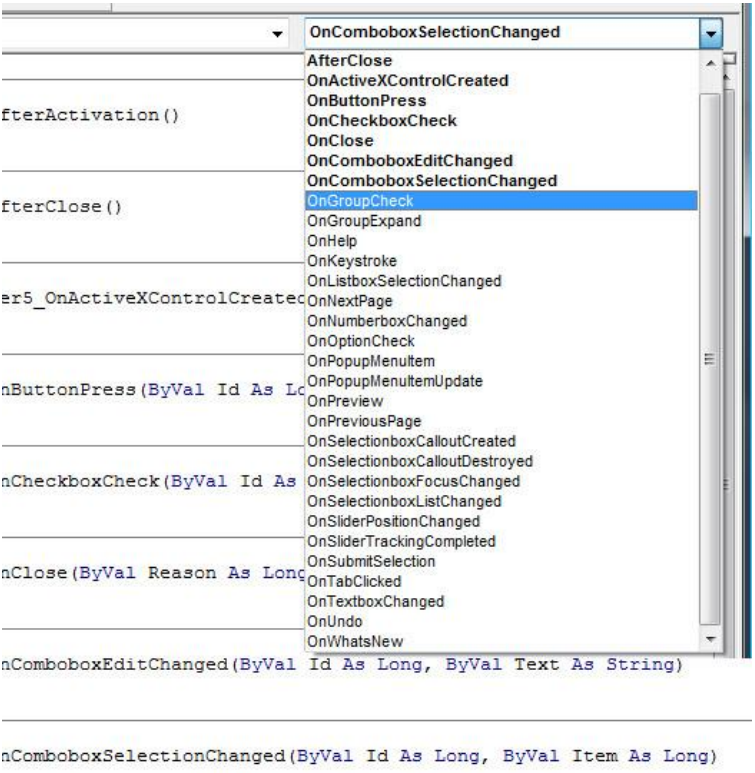


This will populate the **Procedures Box** to the right, and have inserted a single function for you.

```
Private Sub
PropertyManagerPage2Handler5_OnClose(ByVal Reason As
Long)

End Sub
```

What has happened here is that whenever you select an item from the **Procedure Box** to the right, VBA will create an empty function for that item if it doesn't exist, and because by selecting the base class from the **Object Box** populates the **Procedure Box** to the right, it has selected the first item in its new list by default, and so create an empty function for it.

Now we need an empty procedure for *all* of the base class functions for this to work so from the **Procedure Box** just go through one by one and select each and every item in the list.



Once you have selected an item it will turn **bold** indicating there is already a function for it, so selecting it again will just take you to it. Select all items so they are all bold and then you will have the following code:

VBA

```
Implements PropertyManagerPage2Handler5

Private Sub PropertyManagerPage2Handler5_AfterActivation()
End Sub

Private Sub PropertyManagerPage2Handler5_AfterClose()
End Sub

Private Function
PropertyManagerPage2Handler5_OnActiveXControlCreated(ByVal Id As
Long, ByVal Status As Boolean) As Long
End Function

Private Sub PropertyManagerPage2Handler5_OnButtonPress(ByVal Id
As Long)
End Sub

Private Sub PropertyManagerPage2Handler5_OnCheckboxCheck(ByVal
Id As Long, ByVal Checked As Boolean)
End Sub

Private Sub PropertyManagerPage2Handler5_OnClose(ByVal Reason
As Long)
End Sub

Private Sub
PropertyManagerPage2Handler5_OnComboboxEditChanged(ByVal Id
As Long, ByVal Text As String)
End Sub
```

```
Private Sub
PropertyManagerPage2Handler5_OnComboboxSelectionChanged(ByVa
l Id As Long, ByVal Item As Long)
End Sub

Private Sub PropertyManagerPage2Handler5_OnGroupExpand(ByVal Id
As Long, ByVal Expanded As Boolean)
End Sub

.....
.....

Private Function
PropertyManagerPage2Handler5_OnSubmitSelection(ByVal Id As Long,
ByVal Selection As Object, ByVal SelType As Long, ItemText As String)
As Boolean
End Function

Private Function PropertyManagerPage2Handler5_OnTabClicked(ByVal
Id As Long) As Boolean
End Function

Private Sub
PropertyManagerPage2Handler5_OnTextboxChanged(ByVal Id As
Long, ByVal Text As String)
End Sub
```

I have cut out the middle code as I am sure you get the point. With our new class created we are ready to create a new instance of it (a variable of it), and then call a function of the class to tell it to show.

In .Net we must first add another function to our PMP class; add the following function before all of the base class functions:

C#

```
public void Show(SldWorks.SldWorks swApp)
{

}
```

Now we can call this function from our main function that connects to SolidWorks. Go back to the main function of our code (the place where we connected to SolidWorks, in .Net this is usually the button click event). In our main function, after we have got the active document, we want to create a new instance of our class and call a method to tell it to initialise and show the PMP.

C#

```
MyFirstPMP myPMP = new MyFirstPMP();
myPMP.Show(swApp);
```

What this does is create a new variable of type **MyFirstPMP**, which is the name of our PMP class we created previously. The instance of this class (the variable) is called **myPMP**, call it whatever you want. We then assign the variable as a new instance of our class. Finally we call the instances function that we just declared in the class called

Show; this will run whatever code we place within the **Show** function of our class and pass in the active SolidWorks Application variable.

As far as our main code is concerned we are now done, all that remains is to do something with our **Property Manager Page**. But before we do that, let's run over the VBA coding too.

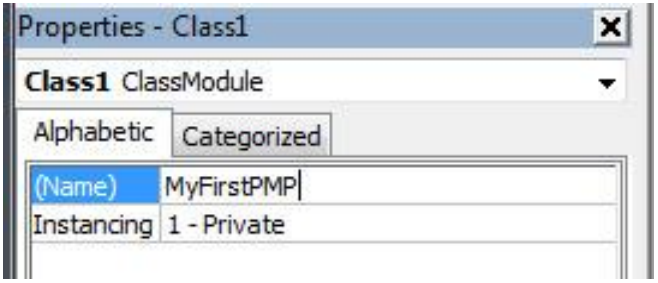
Open the class module code. After the first line:

VBA

```
Sub Show(Dim swApp As SldWorks.SldWorks)

End Sub
```

We can call this function from our main function that connects to SolidWorks. Back to the main function in our main module, after we have acquired the active document, we want to create a new instance of our class and call the **Show** function.

By default our class is called **Class1**. To find out the name of the class, single-click the class module file in the **Project Explorer** and look at the **Property Window** below it. The field **(Name)** is the name of the class; alter this to **MyFirstPMP**, and press enter.



Now go back to our main module and after we acquired the active document, place the following code:

VBA

```
Dim myPMP As MyFirstPMP
Set myPMP = New MyFirstPMP
myPMP.Show  swApp
```

What this does is create a new variable of type **MyFirstPMP,** which is the name of our PMP class we created previously. The instance of this class (the variable) is called **myPMP,** call it whatever you want. We then assign the variable as a new instance of our class. Finally we call the instances function that we just declared in the class called **Show**; this will run whatever code we place within the **Show** function of our class.

We are now ready to run the project, however this will not do anything yet as we have not created anything to display.

Adding items to the Page

So what we have so far is a macro or program that will be run on start or at the press of a button, that will create a new instance of our PMP class, and call a function within it called **Show**. What we need to do is focus our attention on the actual PMP class, in correctly initialising it, adding items to the page, and making sure it all works as expected.

To start with, let me first explain something; although I have been describing the class as deriving from a Property Manager Page base class for ease of understanding, in truth it is actually a Property Manager Page **Handler** class. The difference is that our PMP handler class *handles* Property Manager Pages, it is not one itself. So our first task is to create an actual instance of a PMP. The reason we have had to create a handler class is to actually handle events that come back from our pages, such as clicking a button, changing a dropdown box, opening and closing etc... We could have just created a PMP directly and shown it, but we would not have been able to handle anything it does, so it would be as useful as a candle in a rainstorm.

By creating a new **Page** within our handler class, we are automatically fed back all events from the page to our base class functions we created earlier by passing the actual class we call the creation function from into that function as a parameter. You will see this later.

An example of the event-driven feedback would be that when the Page is first created we get informed of this through the **AfterActivation** function, and when they close the PMP we are again informed by the **OnClose** and **AfterClose** functions. You will see the benefits of this event-driven class soon.

Let's start by creating the variables that we require such as the actual **Page,** a single **Group** and a **Label** so that we can see something on the page to begin with. Once we have done this and we know that our page is working we will go on to create some more advanced and useful pages.

Add the following variables to the **MyFirstPMP** class, at the top of the class:

C#

```
PropertyManagerPage2        pmPage;
PropertyManagerPageGroup    pmGroup;
PropertyManagerPageLabel    pmLabel;

int idGroup = 0;
int idLabel = 1;
```

The **pmPage** is a variable for a Page. The **pmGroup** is a group box object of a page, and the **pmLabel** is a simple text label on the form. You will see these later. The **idGroup** and **idLabel** variables are created and set to unique numbers, any you decide, as they are simply used for the page to identify its controls as unique.

Now in the **Show** function we will create a few more variables ready for use in a second:

C#

```
int pageoptions =
(int)(swPropertyManagerPageOptions_e.swPropertyManagerOptions_Ok
ayButton |
```

```
swPropertyManagerPageOptions_e.swPropertyManagerOptions_Cancel
Button |
swPropertyManagerPageOptions_e.swPropertyManagerOptions_Locked
Page);
int groupoptions =
(int)(swAddGroupBoxOptions_e.swGroupBoxOptions_Expanded |
swAddGroupBoxOptions_e.swGroupBoxOptions_Visible);

int iErrors = 0;
int controloptions;
short controltype;
short controlalign;
```

The first variable **pageoptions** is the combination of enumerator values we are going to use when creating our page. These enumerators specify the options and type of page we create. The complete list of page options is as follows:

swPropertyManagerOptions_OkayButton

This adds an OK button to the top of the page

swPropertyManagerOptions_CancelButton

This adds a cancel button to the top of the page

swPropertyManagerOptions_LockedPage

This prevents the page from automatically closing if the user tries to edit the part, or select an object, or change to another document etc...

swPropertyManagerOptions_CloseDialogButton

This has no use for us.

swPropertyManagerOptions_MultiplePages

This property is set to show the previous/next page buttons if you create multiple pages.

swPropertyManagerOptions_PushpinButton

This shows the pushpin button.

swPropertyManagerOptions_PreviewButton

This shows a default Preview button. The benefit of this is that a LockedPage option will treat this button specially, and won't close the page on click.

swPropertyManagerOptions_DisableSelection

This prevents the user from selecting any models in the document.

swPropertyManagerOptions_WhatsNew

This shows a "What's New" button.

swPropertyManagerOptions_AbortCommand

This forces any current command in progress such as mating to be aborted when the page gets displayed.

swPropertyManagerOptions_UndoButton

This shows the user an undo button, and calls the OnUndo function when clicked.

swPropertyManagerOptions_CanEscapeCancel

This allows the user to exit the Page by pressing escape. This is not work if your page has a selection box.

swPropertyManagerOptions_HandleKeystrokes

This sets the page up to handle all keystrokes, which get sent to the OnKeystroke function for you to handle.

swPropertyManagerOptions_IsDragDropCmd

Allows drag-dropping on page.

For our purpose we just want to show the OK and Cancel Buttons, and lock the page.

The **groupoptions** variable stores the options for our group box we will create later. Again this is an enumerator like the page options, but with far fewer options; whether the group is visible or not, if it is expanded, and if it has a checkbox and whether that checkbox is checked or not. For our example we simply show the group box and make it expanded.

The other variables will come into play in a minute. Now it is time to create the actual page:

C#

```
pmPage =
(PropertyManagerPage2)swApp.CreatePropertyManagerPage("My First
PMP", pageoptions, null, ref iErrors);
```

CreatePropertyManagerPage

In order to create and setup a new **PropertyManagerPage** object we must call the **CreatePropertyManagerPage** function of the SolidWorks Application object, in our case the **swApp** object. You can create PMP's without any documents open, but you must have a document open before you can display one.

The **CreatePropertyManagerPage** function looks like this:

```
retval = SldWorks.CreatePropertyManagerPage ( title, Options, handler, errors )
```

*The **title** is a string value that is shown in the title of the page; this is the white writing that appears on the blue-background for windows such as the Mate window, or the Dimension Property Window.*

*The **Options** are the options we have just been over regarding what buttons and styles to apply.*

*The **handler** is a handle (think of it as a link) to the class that should receive the feedback events, but in order to receive these feedbacks our program has to be run in the same memory space as SolidWorks, and for .Net that means we must create a add-in; stand-alone applications cannot receive these feedbacks, so we pass null instead.*

*What this means is that our .Net examples here will not actually receive any feedback to any of the functions we have declared in our class such as **AfterActivation** or **OnClose** etc… Advanced add-ins is a dedicated topic that could take a whole book to explain in itself.*

For now we will stick to creating the pages with .Net just to show you how it is done, and then move onto VBA (which is run in the same memory space as SolidWorks) to handle feedbacks.

*The **errors** is the usual error variable we pass in as a reference and read back after the check for errors:*

C#

```
if (iErrors !=
(int)swPropertyManagerPageStatus_e.swPropertyManagerPage_Okay)
{
   MessageBox.Show("Failed to create page: " +
((swPropertyManagerPageStatus_e)iErrors).ToString());
   return;
}
```

You should recognise and understand this error checking routine by now.

With the page successfully created, we will go ahead and set a message to be displayed in yellow at the top of the page, and add a group, and then finally add a label inside the group and show the page.

C#

```
pmPage.SetMessage3("Welcome to my first PMP",
(int)swPropertyManagerPageMessageVisibility.swImportantMessageBox
,
(int)swPropertyManagerPageMessageExpanded.swMessageBoxMaintai
nExpandState, "My Caption");
```

```
pmGroup =
(PropertyManagerPageGroup)pmPage.AddGroupBox(idGroup, "My
groupbox", groupoptions);

controloptions =
(int)(swAddControlOptions_e.swControlOptions_Enabled |
swAddControlOptions_e.swControlOptions_Visible);
controltype =
(int)swPropertyManagerPageControlType_e.swControlType_Label;
controlalign =
(int)swPropertyManagerPageControlLeftAlign_e.swControlAlign_Indent;
pmLabel = (PropertyManagerPageLabel)pmGroup.AddControl(idLabel,
controltype, "My label", controlalign, controloptions, "My tip");

pmPage.Show();
```

We start by setting the page message. This will appear at the top of the page and is usually used for a brief description of what is going on. The function to do this is called **SetMessage3,** of the **PropertyManagerPage** object. You do not have to call this function and if you chose not to then the PMP will simply not display this standard message box.

In this example we are going to set it to show you what it looks like:

SetMessage3

We use this function to set a top message of a page.

```
retval = PropertyManagerPage2.SetMessage3 ( Message,
Visibility, Expanded, Caption )
```

*The **Message** variable is literally the message to display.*

*The next two options, **Visibility** and **Expanded** are again enumerators for the styling and visibility of the message box. Play around with changing these values to see the results.*

*The **Caption** is the title of the message box.*

AddGroupBox

Once we set this message, we then create a new group, adding it to the **pmPage** page. The function is as follows:

```
retval = PropertyManagerPage2.AddGroupBox ( Id,
Caption, Options )
```

*The **Id** is a unique number to identify the control by.*

*The **Caption** is the title of the group box.*

*The **Options** is an enumerator specifying the style and visibility of the control.*

We pass in the id value we specified previously, which is simply a unique number to identify the control by, this can be anything. The ID will not denote the position on the page of the control; the order in which you add the controls will be the order in which they are laid out.

We then pass in the **groupoptions** variable we specified earlier for the group box so that it is visible and by default expanded not collapsed.

With the group box create we then add a label inside the group box be calling the general **AddControl** function. This function is available from any **PropertyManagerPage** object and any **PropertyManagerPageGroup** object.

In this example, we do not add the label to the **pmPage** object, but instead we add it to the group we just created. Because every other control other than a page or a group box is added using a unique **AddControl** function.

AddControl

The AddControl function looks like this:

```
retval = PropertyManagerPage2.AddControl ( Id,
ControlType, Caption, LeftAlign, Options, Tip )
```

*The **ID** is the id we specified early.*

*The **ControlType** is the enumerator option for the type of control we are adding.*

*The **Caption** is the text that will be displayed in the control.*

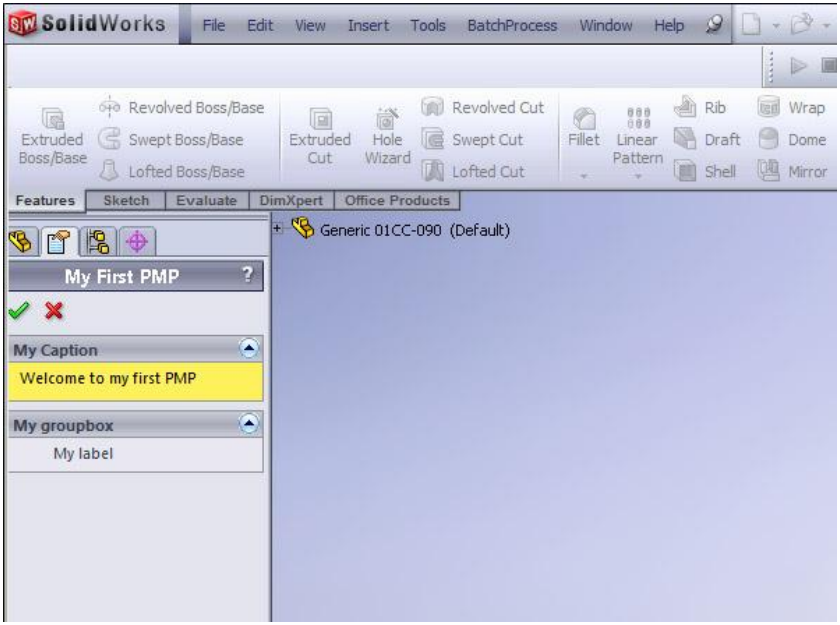
*The **LeftAlign** and **Options** are again enumerators for styling, state and visibility.*

*And, the **Tip** is the text that will be displayed in a tooltip if the user hovers over the control. This is usually a brief description or help.*

We pass in our unique ID for the **ID**, and the enumerator for the **Label** type. The other two enumerators we specify are for the alignment of the control, and the visibility and state; these options are self explanatory so just play around with them if you wish to see the difference.

By calling the **pmGroup**'s **AddControl** function instead of the **pmPage**'s **AddControl** function we add the label to the group not the page.

Finally, we call the pages **Show** function to display the page to the user. Compile your code and give it a go. You should see the following results.



You can continue to add other controls and play around with the PMP's in .Net, but unless you plan on creating an add-in then you cannot really do much with them. Personally, I find very little use in PMP's when using .Net as you have the entire power of the .Net framework and a programming language at your disposal to get user information or display to them whatever you like. That's not to say PMP's don't have their place.

Let's carry on exploring them but using VBA, as this will allow us to quickly and easily handle the events back from the page. I will not explain the coding that has already been explained in the .Net version as the principles are all the same.

First, let's add the items to the page in VBA just like we have done in .Net.

In the **MyFirstPMP** class, before all of the functions but after the **Implements PropertyManagerPage2Handler5** line, place the following variables:

VBA

```
Dim pmPage As PropertyManagerPage2
Dim pmGroup As PropertyManagerPageGroup
Dim pmLabel As PropertyManagerPageLabel

Dim idGroup As Integer
Dim idLabel As Integer

Dim lErrors As Long
```

```
Dim pageoptions As Integer

Dim groupoptions As Integer

Dim controloptions As Integer
Dim controltype As Integer
Dim controlalign As Integer
```

Those are the only variables we need for now to get our example up and running.

Go to the **Show** function we created and add the following code. This code will create a new page and set it to the variable **pmPage,** then give the page a message and add a group and label.

VBA

```
Sub Show()

IErrors = 0
idGroup = 0
idLabel = 1

pageoptions = swPropertyManager_CancelButton +
swPropertyManager_OkayButton +
swPropertyManagerOptions_LockedPage
groupoptions = swGroupBoxOptions_Expanded +
swGroupBoxOptions_Visible
```

```
Set pmPage = swApp.CreatePropertyManagerPage("My First PMP",
pageoptions, Me, lErrors)

If lErrors = swPropertyManagerPage_Okay Then

    pmPage.SetMessage3 "Hello World! This is my first PMP",
swImportantMessageBox, swMessageBoxMaintainExpandState,
"Important Message"
    Set pmGroup = pmPage.AddGroupBox(idGroup, "My Group",
groupoptions)

    controltype = swControlType_Label
    controlalign = swControlAlign_Indent
    controloptions = swControlOptions_Enabled +
swControlOptions_Visible
    Set pmLabel = pmGroup.AddControl(idLabel, controltype, "My Label",
controlalign, controloptions, "My Tip")

    pmPage.Show

Else
    MsgBox "Error creating Property Manager Page"
End If

End Sub
```

As you can see this is identical to the .Net version, just with the syntax of VBA. The advantage we have here however is that notice the call we have made to **CreatePropertyManagerPage**:

```
Set pmPage = swApp.CreatePropertyManagerPage("My
First PMP", pageoptions, Me, lErrors)
```

Now compare that with the .Net version, and see the difference:

```
pmPage =
(PropertyManagerPage2)swApp.CreatePropertyManagerPag
e("My First PMP", pageoptions, null, ref iErrors);
```

In .Net, remember I said we have to be running in-process of SolidWorks to handle event feedbacks, and for that reason we passed null as the handle to the class to handle these events? Probably not, but anyway that is what I said.

Now if you look at the VBA function, we have passed a variable called **Me**; this is a special variable that can be used to reference the current object that the code resides in. Because we are calling this **CreatePropertyManagerPage** function from within the **MyFirstPMP** class, whatever instance of the class we are within, that is what the **Me** variable refers to, so in this case we are actually passing a reference to our **myPMP** instance of the class we created, and called this **Show** function from:

```
Dim myPMP As MyFirstPMP
Set myPMP = New MyFirstPMP
myPMP.Show  swApp
```

Because we have passed an instance of our class as the handle to this page, it means that any events triggered by this page will be fed back and handled by our **myPMP** class, which contains all of those functions we created at the start, such as **OnClose**, **AfterActivation** and **OnKeystroke** etc... So now, to test that we are handling these

events, we will start by placing a simple message box in the **OnClose**, **AfterClose**, and **AfterActivation** event functions:

VBA

```
Private Sub PropertyManagerPage2Handler5_AfterActivation()
MsgBox "Activated"
End Sub

Private Sub PropertyManagerPage2Handler5_AfterClose()
MsgBox "Closed"
End Sub

Private Sub PropertyManagerPage2Handler5_OnClose(ByVal Reason
As Long)
MsgBox "Closing"
End Sub
```

Run your macro and you will notice that the **AfterActivation** event fires before the PMP is visible; this is because it receives activation control as soon as the message process starts for that control, not when it gets drawn (displayed).

Now close the PMP using the **X** button, and you will see that the **OnClose** message box appears before the PMP is closed, and the **AfterClose** event fires once it is destroyed.

Good, so let's create something useful and start handling those events!

Responding to Events

In order to best demonstrate working with events and responding to them we are going to create a new PMP that has a selection tool in it that limits the user to selecting faces. From there we are then going to change the colour of those selected faces to Red, Blue or Green based on the button the user clicks.

We will begin by creating a new macro, connecting to SolidWorks, getting the Active Document, and creating a new instance of a class that implements **PropertyManagerPage2Handler5**; so basically we are creating an identical copy of the macro we have just created.

The first modification to add a check in the main function that the active document is not a drawing; after the check for an active document place the following:

VBA

```
If swModel.GetType() = swDocDRAWING Then
MsgBox "Cannot set faces in a drawing"
Exit Sub
End If
```

Next we need to change the items that we create; instead of creating a group and a label, we create a selection box and 3 buttons. We start by specifying the usual variables and the new ones for our buttons and selection control:

VBA

```
Dim pmPage As PropertyManagerPage2
Dim pmGroup As PropertyManagerPageGroup
```

```
Dim pmSelection As PropertyManagerPageSelectionbox
Dim pmButtonRed As PropertyManagerPageButton
Dim pmButtonGreen As PropertyManagerPageButton
Dim pmButtonBlue As PropertyManagerPageButton

Dim idGroup As Integer
Dim idSel As Integer
Dim idButtonRed As Integer
Dim idButtonGreen As Integer
Dim idButtonBlue As Integer

Dim lErrors As Long
Dim pageoptions As Integer
Dim groupoptions As Integer
Dim controloptions As Integer
Dim controltype As Integer
Dim controlalign As Integer

Dim filters(0) As Long
```

You will recognise all of these variables, except two new types for the **Selectionbox** and the **Buttons**. There is nothing really to explain, there are no different options for the buttons than there was for the label control, and for *creating* the selection box. You will see how to set the **Selectionbox** options up in a minute.

The **filters** variable is used to store selection filter data, it will be explained later.

Now inside the **Show** function we want to create the usual page, settings the initial variables, and checking for success before attempting to add our controls. Here is the code we should have so far:

VBA

```
lErrors = 0
idGroup = 1
idSel = 2
idButtonRed = 3
idButtonGreen = 4
idButtonBlue = 5

pageoptions = swPropertyManager_CancelButton +
swPropertyManager_OkayButton +
swPropertyManagerOptions_LockedPage
groupoptions = swGroupBoxOptions_Expanded +
swGroupBoxOptions_Visible

Set pmPage = swApp.CreatePropertyManagerPage("FunFaces",
pageoptions, Me, lErrors)

If lErrors = swPropertyManagerPage_Okay Then

 ‘ ADD OUR ITEMS TO THE PAGE HERE

Else
    MsgBox "Error creating Property Manager Page"
End If
```

Where I have placed the comment we will add our new controls. We start by setting the pages message up to tell the user what our macro is going to do:

VBA

```
    pmPage.SetMessage3 "Select any faces you would like and then
click a color button to change the face colors!",
swImportantMessageBox, swMessageBoxMaintainExpandState,
"Important Message"
```

Adding a Selection Box & Buttons

Next we want to add our group and selection box:

VBA

```
    Set pmGroup = pmPage.AddGroupBox(idGroup, "Select faces",
swGroupBoxOptions_Expanded + swGroupBoxOptions_Visible)

    Set pmSelection = pmGroup.AddControl(idSel,
swControlType_Selectionbox, "Select faces", swControlAlign_Indent,
swControlOptions_Visible + swControlOptions_Enabled, "")
```

Simple enough wasn't it? Finally we add 3 buttons with a name of Red, Green and Blue respectively:

VBA

```
    controltype = swControlType_Button
    controlalign = swControlAlign_Indent
```

```
    controloptions = swControlOptions_Enabled +
swControlOptions_Visible

    Set pmButtonRed = pmPage.AddControl(idButtonRed, controltype,
"Red", controlalign, controloptions, "")
    Set pmButtonGreen = pmPage.AddControl(idButtonGreen,
controltype, "Green", controlalign, controloptions, "")
    Set pmButtonBlue = pmPage.AddControl(idButtonBlue, controltype,
"Blue", controlalign, controloptions, "")
```

And as usual, once we have created our controls, we want to display our PMP:

VBA

```
    pmPage.Show
```

Setting up Selection Filters

Before we try our macro, let's first setup the selection filters; anywhere after the **pmSelection** object has been created, place the following code:

VBA

```
    filters(0) = swSelFACES

    pmSelection.SingleEntityOnly = True
    pmSelection.Height = 50
    pmSelection.SetSelectionFilters filters
```

```
pmSelection.SetStandardPictureLabel swBitmapLabel_SelectFace
```

The first line sets our **Long** array variable **filters** first and only item to **swSelFACES**, which means we are only allowing faces to be selected. This value is from the **swSelType_e** enumerator.

Next, we set up the selection mode to single-select, the height of the selection box to 50 pixels, and the filter we just defines to the selection box.

The last line sets the icon to use for this control. This function can be called from any control you add such as a label, button, drop-down list and all others. There are many icons:

```
swBitmapLabel_LinearDistance
swBitmapLabel_AngularDistance
swBitmapLabel_SelectEdgeFaceVertex
swBitmapLabel_SelectFaceSurface
swBitmapLabel_SelectVertex
swBitmapLabel_SelectFace
swBitmapLabel_SelectEdge
swBitmapLabel_SelectFaceEdge
swBitmapLabel_SelectComponent
swBitmapLabel_Diameter
swBitmapLabel_Radius
swBitmapLabel_LinearDistance1
swBitmapLabel_LinearDistance2
swBitmapLabel_Thickness1
swBitmapLabel_Thickness2
swBitmapLabel_LinearPattern
swBitmapLabel_CircularPattern
swBitmapLabel_Width
swBitmapLabel_Depth
swBitmapLabel_KFactor
```

```
swBitmapLabel_BendAllowance
swBitmapLabel_BendDeduction
swBitmapLabel_RipGap
swBitmapLabel_SelectProfile
```

With using a selection box, we can place custom rules on whether to allow certain selections, such as don't allow fillets over 5mm or don't allow more than 5 items etc… This is pretty cool power to have, but for our example we simply accept any selection that passes the normal selection filter we set up of faces. So, in order to bypass the custom rules we must pass **True** every time a selection is made in the **OnSubmitSelection** function:

VBA

```
Private Function
PropertyManagerPage2Handler5_OnSubmitSelection(ByVal Id As Long,
ByVal Selection As Object, ByVal SelType As Long, ItemText As String)
As Boolean
PropertyManagerPage2Handler5_OnSubmitSelection = True
End Function
```

As you can see this function does one thing; return true all the time, allowing any selection that passes our filter to be selected and added to the selection box.

Now you are ready to run the macro, give it a go and select a face, then close the page.

You will notice that clicking the buttons does nothing at the moment. That is because we haven't handled the event of clicking the button. So let's do that now!

Getting Selection & Settings Face Colours

When the user clicks any of the 3 buttons we want to get the current selection, and then set the face colour to the respective buttons colour, and finally clear the selection so that they can carry on.

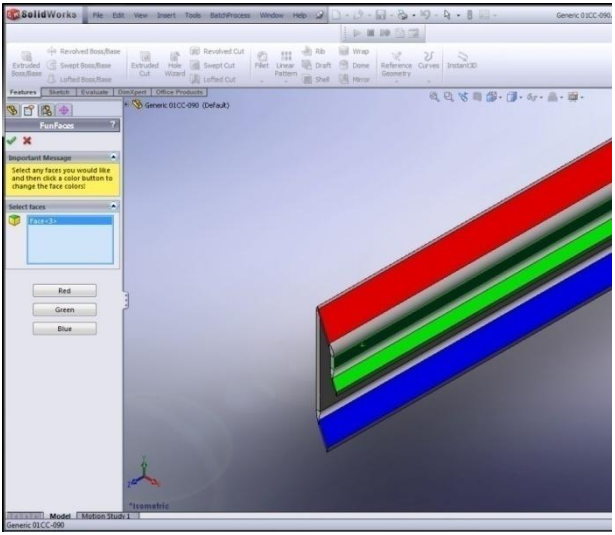
It all happens in the **OnButtonPress** event as you can probably figure out. Within the **OnButtonPress** function we firstly want to identify which button was pressed and create a variable for the colour. Once we have the colour variable everything else is the same regardless of what button was pressed:

VBA

```
Private Sub PropertyManagerPage2Handler5_OnButtonPress(ByVal Id
As Long)

Dim r As Integer, g As Integer, b As Integer
r = g = b = 0
If Id = idButtonRed Then r = 255
If Id = idButtonGreen Then g = 255
If Id = idButtonBlue Then b = 255
```

```
Dim v As Variant
v = swModel.MaterialPropertyValues

swModel.SelectedFaceProperties RGB(r, g, b), v(3), v(4), v(5), v(6), v(7),
v(8), False, ""

swModel.ClearSelection2 (True)
End Sub
```

All we have done here is to create 3 variables for the red, green and blue colours, set them all to **0**, and then depending on which button was pressed, set that specific colour value. With the RGB values set we get the default part material properties then we set the selected face properties to the colour we defined above, and finally clear the selection so we can see the results.



Traversing

Traversing through an Assembly

Traversing through a Component

Displaying the results

Playing with Components and Features

Traversing is the art of looping through every item in a list. By traversing we are going to loop through every single component in an assembly, later we will take it a step further and traverse though all features of all of those components, and then after we are done we will have some fun with .Net by creating a tree-view program and adding some functionality to work with features and components.

Traversing through an Assembly

The basics of traversing an assembly are:-

- Get the root component of the assembly
- Get the children of that root component
- For each child, check if it has any children
- For each child of the child, repeat the loop

The way we create this loop so that the coding will traverse all children of all children so we get every component in the assembly is to create a function that accepts a **Component2** object as a variable.

Within this function we check if the **Component2** object passed in has children, and if it has any, pass each child's **Component2** back into the function, so that child gets checked for children also. This loop will continue for every child so long as there are children. This loop will effectively pass over every single component in an assembly. For now we will just display a message box each time we pass over a component.

Later on we will actually mimic the SolidWorks feature tree list that is displayed on the left.

Start by creating the usual template and getting the active document. Now add the following variables next to the **swApp** and **swModel** variables:

C#

```
Configuration     swConf;
Component2        swRootComponent;
```

After the code where you get the active document, place the following code to get the root component of the active document:

C#

```
swConf = (Configuration)swModel.GetActiveConfiguration();
swRootComponent = (Component2)swConf.GetRootComponent();
```

Before we can get components, we need to get a configuration, as each configuration can have different components. We use the **ModelDoc2**'s function **GetActiveConfiguration** function to get the current configuration, casting it to **Configuration**. With the active configuration we then call the **Configuration** function **GetRootComponent** to get our root component. Job Done!

Now it is time to loop through this component looking for children, and then to loop through those children. After we have acquired the root component, pass this root component into a function that will then check that component for children:

C#

```
TraverseComponent(swRootComponent);
```

Now let us create the actual function to simply check if the component has children, and if it does pass each child back into this function:

C#

```
private void TraverseComponent(Component2 component)
{
    object[] children = (object[])component.GetChildren();

    if (children.Length > 0)
    {
        foreach (Component2 comp in children)
        {
            TraverseComponent(comp, tn);
        }
    }
}
```

This function is simple enough; we first attempt to get all, if any, of the components children. Then we check if it does have any children, and if it does we pass each child back into this loop and repeat. We have to cast the array return from **GetChildren** as an **Object** array not a **Component2** array, and then cast each object to a **Component2** within the **foreach** loop. If you attempt to cast the returned array straight to a **Component2** array it will fail.

To show that we have passed over each component we will add a message box showing the component title before the call to **TraverseComponent**:

C#

```
MessageBox.Show(((ModelDoc2)comp.GetModelDoc()).GetTitle());
```

All we have done here is got the **ModelDoc2** document associated with the component, and then got its title.

Now for the same code in VBA; start with the usual template, adding the following additional variables;

VBA

```
Dim swConf As Configuration
Dim swRootComponent As Component2
```

Then after getting the active document place the following:

VBA

```
Set swConf = swModel.GetActiveConfiguration()
Set swRootComponent = swConf.GetRootComponent()
```

This will get the root component as explain in the .Net version. Now we call the **TraverseComponent** function:

VBA

```
TraverseComponent swRootComponent
```

And here is the **TraverseComponent** function to do just the same as the .Net version:

VBA

```
Sub TraverseComponent(component As Component2)

Dim children As Variant
children = component.GetChildren()

If UBound(children) > 0 Then
   Dim i As Integer
   For i = 1 To UBound(children)
      MsgBox children(i).GetModelDoc().GetTitle()
      TraverseComponent children(i)
   Next
End If

End Sub
```

And that is all there is to it really. Using this form of loop we can get access to all of the **Component2** objects in an assembly and do what we want with them. You will see some useful things to do later.

Traversing through a Component

Now we have access to all of these components, the next thing that would be handy is to then traverse these components for all of their features.

The following code will show how to traverse a component. You can place this code within the assembly traversal loop if you wish. I will take you through creating a complete tree-view program next to see this in use.

Within a component are features, and within those features can be sub-features.

We will start by calling a function that accepts a **Feature** as a parameter, a bit like the assembly component feature, that will loop through all features:

C#

```
TraverseFeatures((Feature)comp.FirstFeature());
```

The **comp** object is a component we have acquired. If you place this code within the assembly loop we made just we will loop every component in an assembly. And here is the **TraverseFeatures** function to loop all features:

C#

```
private void TraverseFeatures(Feature firstfeature, TreeNode tn)
{
    Feature feat = firstfeature;
```

```
    while (feat != null)
    {
        MessageBox.Show(feat.Name);
        feat = (Feature)feat.GetNextFeature();
    }
}
```

As you can see this is much similar to the previous looping function, however we only run through the top-level features. As I mentioned earlier, features can have sub-features themselves, so before the message box line, place the following line to call another function that will loop all sub-features:

C#

```
TraverseSubFeatures(feat);
```

And the function to loop all sub-features will also loop itself for any sub-features of sub-features, just like the assembly components functions did, this way we get every single feature there is:

C#

```
private void TraverseSubFeatures(Feature feature, TreeNode node)
{
    Feature subfeat = (Feature)feature.GetFirstSubFeature();
    while (subfeat != null)
    {
        TraverseSubFeatures(subfeat, subtn);
        subfeat = (Feature)subfeat.GetNextSubFeature();
```

```
    }
}
```

This looks identical to the previous function except instead of calling **GetNextFeature** we call **GetNextSubFeatue**. Again within this **TraverseSubFeatures** function you can place a message box call to display the feature name.

Now for the VBA version; wherever you have a component object you wish to traverse place the following function call:

VBA

```
TraverseFeatures (children(i).FirstFeature())
```

The **children** variable is an array of components from the previous example, and we are calling the **Component2** function **FirstFeature** and passing it into the function **TraverseFeatures**. And here is the **TraverseFeatures** function:

VBA

```
Sub TraverseFeatures(feature As feature)

Dim feat As feature
Set feat = feature

While Not feat Is Nothing

   MsgBox feat.Name
   TraverseSubFeatures feat
```

```
    Set feat = feat.GetNextFeature()

Wend

End Sub
```

This function loops through all features, starting with the feature passed in. It shows a message box to the user with the name of the current feature within the loop. It then calls the **TraverseSubFeatures** function to loop through all sub-features:

VBA

```
Sub TraverseSubFeatures(feature As feature)

Dim feat As feature
Set feat = feature.GetFirstSubFeature()

While Not feat Is Nothing

    MsgBox feat.Name
    TraverseSubFeatures feat
    Set feat = feat.GetNextSubFeature()

Wend

End Sub
```

This function is again identical to the .Net version so if you need an explanation on what's going on here read over the .Net section.

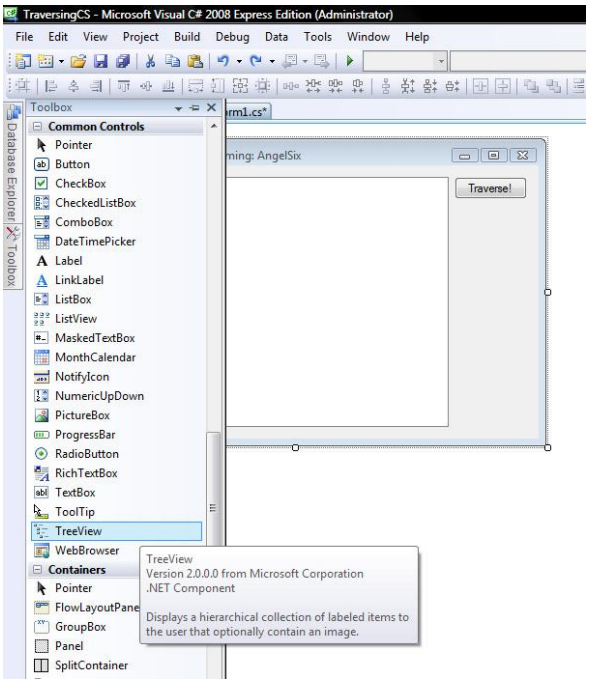
And that covers that. We have now looped through every component and every feature within an assembly. However, there is a much better way to view all of these components and features other than a message box to the user; through an actual tree-view list just like in SolidWorks. That is what we are going to do next.

Displaying the results

Taking what we have just learned and applying it to a new .Net program, we are going to create a tree-view to display an assembly/part tree-view list.

Begin by creating a new project from the normal template in your desired .Net language. As well as the button in the form, we want to add a new control; a tree-view.

Start by going into the **Form Designer** view by double-clicking the **Form1.cs** file from the **Solution Explorer**, and then from the **Toolbox**, drag a new **TreeView** control onto your form and size it how you like.



Into the coding view; add the following variables below the typical **swApp** and **swModel** variables:

C#

```
Configuration      swConf;
Component2         swRootComponent;
```

These will be used in a second. Now inside the button click event function, after you have acquired the active document, you want to do a check that we aren't working with a drawing:

C#

```
if (swModel.GetType() == (int)swDocumentTypes_e.swDocDRAWING)
{
    MessageBox.Show("Active document cannot be a drawing");
    return;
}
```

Now we have an active document that is either a part or an assembly we are ready to start traversing as usual, only this time instead of a message box we want to add items to the **TreeView** control we just created.

Before we do anything, we want to clear the current tree view list before we start adding a new list:

C#

```
treeView1.Nodes.Clear();
```

Next get the root component:

C#

```
swConf = (Configuration)swModel.GetActiveConfiguration();
swRootComponent = (Component2)swConf.GetRootComponent();
```

We will start our tree view with the top item being the actual active model, like it is in SolidWorks:

C#

```
TreeNode tn = treeView1.Nodes.Add(swModel.GetTitle());
```

The Nodes **Add** function adds a node to the parent node or top-level **TreeView** control, and it returns the actual node added. Because we will be using this as a reference to add our child components and features to we store it in a variable called **tn**.

Next we want to call the **TraverseFeatures** function we created in the last section, and to call the **TraverseComponent** function too. The addition in this instance is we are passing in a **TreeNode** variable as the second parameter. We will use this **TreeNode** variable and add nodes to it.

C#

```
TraverseFeatures((Feature)swModel.FirstFeature(), tn);
if (swRootComponent != null)
    TraverseComponent(swRootComponent, tn);
```

We do a check that the root component was acquired before we traverse it. If we are in a part, then there is no root component.

Let's take a look at the modified **TraverseComponent** function where we add each component to its parent tree node item:

C#

```
private void TraverseComponent(Component2 component, TreeNode
treeNode)
{
    object[] children = (object[])component.GetChildren();

    if (children.Length > 0)
    {
        foreach (Component2 comp in children)
        {
            TreeNode tn =
treeNode.Nodes.Add(((ModelDoc2)comp.GetModelDoc()).GetTitle());
            TraverseFeatures((Feature)comp.FirstFeature(), tn);
            TraverseComponent(comp, tn);
        }
    }
}
```

The only modification we have made here is removed the message box code and instead add each child components title to the **TreeNode** passed in. Then for each child, we loop its features and child components, but instead passing it the components **TreeNode** as the node to add any results too. This creates the list we are after.

Next is to place this same modification into the **TraverseFeatures** function too, and to add another small check to prevent adding the assemblies components as features as well as components:

C#

```
private void TraverseFeatures(Feature firstfeature, TreeNode tn)
{
   Feature feat = firstfeature;

   while (feat != null)
   {
      if (string.Compare(feat.GetTypeName2(), "Reference", true) == 0)
         return;

      TreeNode subtn = tn.Nodes.Add(":: " + feat.Name);
      TraverseSubFeatures(feat, subtn);
      feat = (Feature)feat.GetNextFeature();
   }
}
```

The first thing we do is check that the features type is not a reference. In assemblies the components within it report as features as well as components; as a feature, components types return the name "Reference", so we check for this and ignore the features that are references. If we removed this line of code we would get all of the assemblies components added as single features, as well as components.

If the feature is not a reference we then add it to the node passed in to the function. For each feature we then loop through all sub-

features, again passing in the features **TreeNode** object as the node to add results too. Finally here is the modified **TraverseSubFeatures** function:

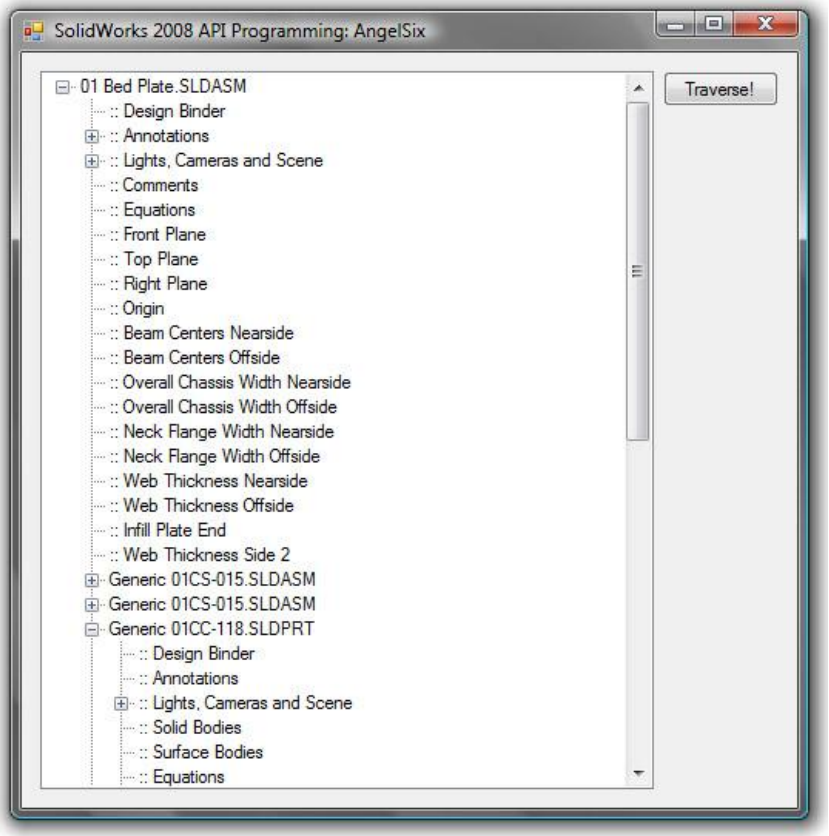
C#

```
private void TraverseSubFeatures(Feature feature, TreeNode node)
{
   Feature subfeat = (Feature)feature.GetFirstSubFeature();

   while (subfeat != null)
   {
      TreeNode subtn = node.Nodes.Add("::" + subfeat.Name);

      TraverseSubFeatures(subfeat, subtn);

      subfeat = (Feature)subfeat.GetNextSubFeature();
   }
}
```

Nothing much to explain here really, other than we add a child **TreeNode** to the **TreeNode** object passed into the function for each sub-feature, and loop the function for all sub-features.

Try running your program. Click the button once you have an assembly or part open and see as your results in the tree view. Once you click the button it may take a few seconds to complete depending on the size of your assembly.

Find the complete code listing below. The VB.Net example is also on the accompanying CD as per-usual.


SolidWorks 2008 API Programming: AngelSix
Traverse!
01 Bed Plate.SLDASM
:: Design Binder
:: Annotations
:: Lights, Cameras and Scene
:: Comments
:: Equations
:: Front Plane
:: Top Plane
:: Right Plane
:: Origin
:: Beam Centers Nearside
:: Beam Centers Offside
:: Overall Chassis Width Nearside
:: Overall Chassis Width Offside
:: Neck Flange Width Nearside
:: Neck Flange Width Offside
:: Web Thickness Nearside
:: Web Thickness Offside
:: Infill Plate End
:: Web Thickness Side 2
Generic 01CS-015.SLDASM
Generic 01CS-015.SLDASM
Generic 01CC-118.SLDPRT
:: Design Binder
:: Annotations
:: Lights, Cameras and Scene
:: Solid Bodies
:: Surface Bodies
:: Equations

Playing with Components and Features

Now we have created a tree-view much like the SolidWorks one, it is a brilliant time to have a play with these components and features.

We will start my making a small modification to the program we just made so that we can reference any item in the tree view back to the actual SolidWorks components and features. Add the following to the using section:

C#

```
using System.Collections;
```

Add the following to the variables section:

C#

```
Hashtable        hashStore;
```

We will use this variable to store one reference to the **TreeNode** object, and one to the corresponding SolidWorks **Component, ModelDoc2** or **Feature**.

After the code line where we clear the tree view list we also want to initialise a new hash table, effectively clearing it at the same time, ready for a fresh start:

C#

```
hashStore = new Hashtable();
```

A few more lines down after we add the first item to the tree view, before traversing the root components, place the following code:

C#

```
hashStore.Add(tn, swModel);
```

What this does is add an item to our **Hashtable**; each item stores a **key**, and a **value**. The **key** is used to lookup or find the **value**. Here the **tn** variable is the **TreeNode** item we just added, and the **value** is the actual **swModel** variable linked to our model. You will see how we look the **swModel** back up later.

Inside the **TraverseComponent** function, right below the line where we add the **TreeNode** again, add the same line as before, but this time adding the component:

C#

```
hashStore.Add(tn, comp);
```

Inside the **TraverseFeatures** function, below the **TreeNode** code line, just like before, add the following:

C#

```
hashStore.Add(subtn, feat);
```

And finally, in the **TraverseSubFeatures** function below the **TreeNode** code add the following:

C#

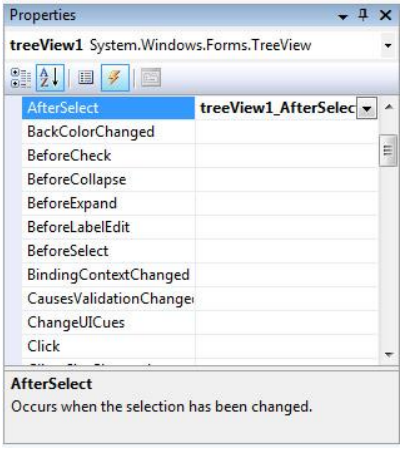
```
hashStore.Add(subtn, subfeat);
```

All we have done here is every time we add any **TreeNode** to the **TreeView** control we add a reference in the **Hashtable** to the tree node item and the associated model, component or feature.

Linking the list with SolidWorks

The simplest task to perform on any component or feature is to select it. What we will do is that every time the selection in our tree view list is changed, we will select the corresponding object in SolidWorks.

Start by going to the **Form Designer**, selecting the **TreeView** object and going to its **Events** in the **Property Window**. Scroll down to the **AfterSelect** entry and double-click it to add an event function for every time the selection is changed.



This function will get called every time the user selected an item from our list. So we want to get the currently selected item from the tree view, and use that as the key in our hash table to lookup whatever component or feature that item links back to. From there we will have access to the **Component2** or **Feature** object to do with as we please.

Inside the **AfterSelect** function place the following code:

C#

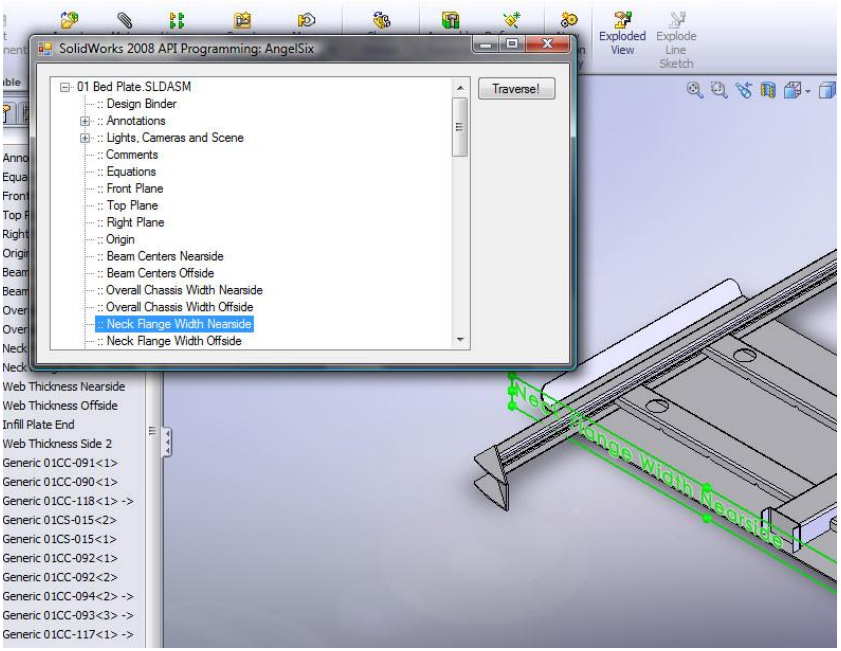
```
if (treeView1.SelectedNode == null)
    return;

if (hashStore.Contains(treeView1.SelectedNode))
{
    object link = hashStore[treeView1.SelectedNode];

    if (treeView1.SelectedNode.Index != 0)
    {
        try
        {
            Component2 comp = (Component2)link;
            comp.Select3(false, null);
        }
        catch
        {
            Feature feat = (Feature)link;
            feat.Select2(false, -1);
        }
    }
}
```

We first check that we have an item in our list selected, and if so that our hash table actually contains a link from the selected node (although from our current code it always will). Then we acquire the associated SolidWorks object be retrieving it from the hash table passing in the selected node as the key. We then check if the

selected node is the first item, in which case it is the model we add in the beginning. Because we cannot select the model itself we skip the code if it is.

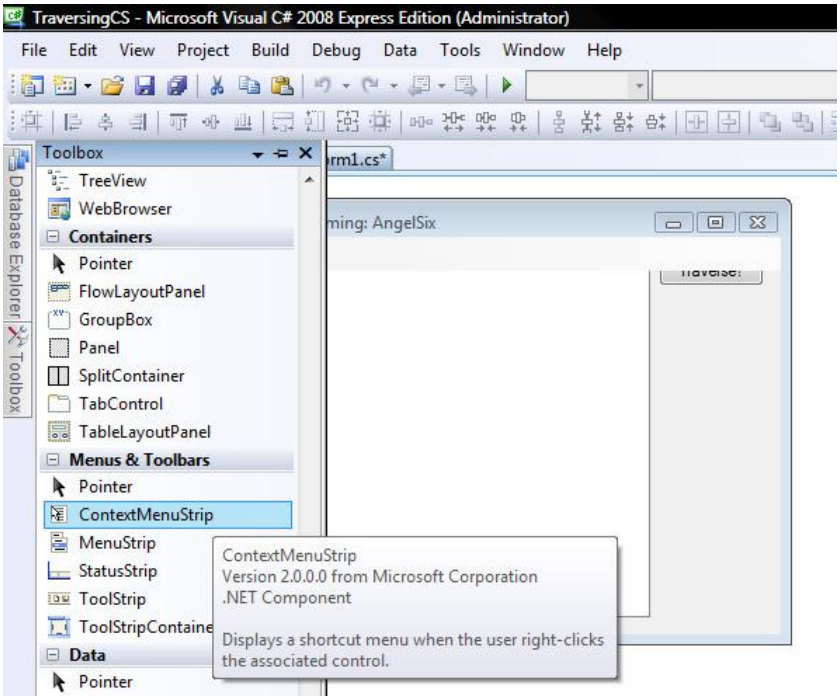
If the selected node is not the first node, then it will be either a **Component2** object or a **Feature** object, but because the object is a COM object, we cannot simply test what type it is, so we use a **Try/Catch** block. If casting the object to a **Component2** object fails, we know it is a **Feature**.

Once we have the desired object, we call its respective **Select** function to select it. Run our program and click the button to fill the list, then select some items from the list and watch in SolidWorks as the object gets selected.



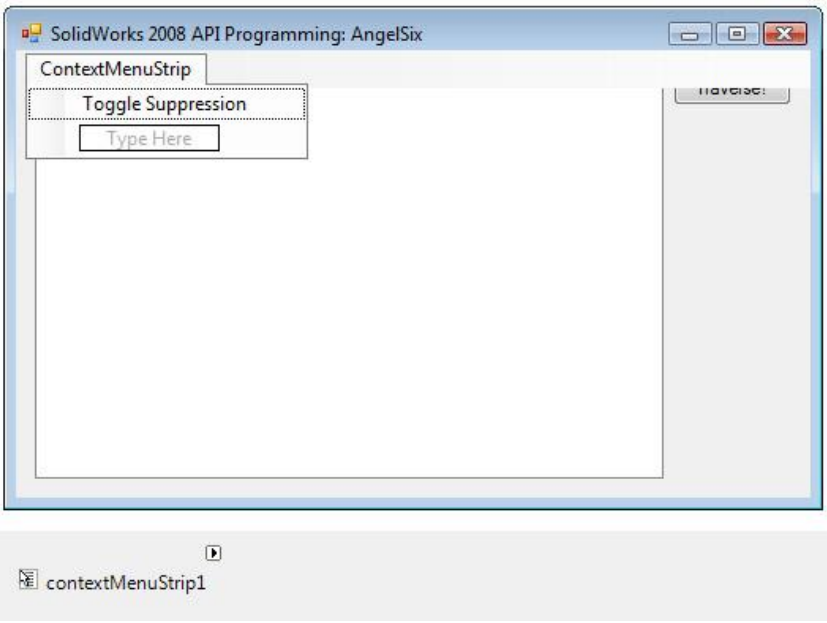
Pretty good hu? That's not all. We are now going to add the ability to toggle the suppression state of the components and features using a right-click menu to our list. This can then be easily expanded to call any function or do any task you would like on any component or feature within an assembly or part.

Go back to the **Form Designer** and this time we are going to drag a new **ContextMenuStrip** from the toolbar to the form.

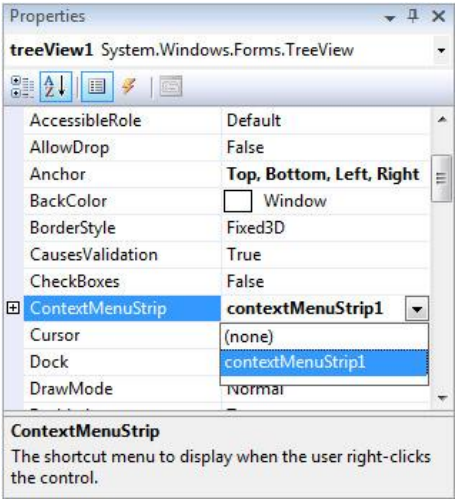


You will then notice at the bottom of the **Form Designer** that you have your newly created menu called **ContextMenuStrip1**. If you single-click on this the menu designer will appear at the top of your form. To add a new menu item you single-click the box where it states "Type Here", type the name of the menu, and press enter.

Add a new menu item called "Toggle Suppression", and in order to add an event function for **OnClick**, just double-click the menu item.



Before we continue, go back to the **Form Designer** and select the **TreeView** control. We must associate our newly create menu with this control so that when the user right-clicks within it the menu displays. In the **Property Window** with the **TreeView** control selected, for the **ContextMenuStrip** property, select your menu.



Now it's time to implement our suppression function. Within the event function for the menu click that we created a minute ago place the following code. This is mostly identical with the previous code but where the **Select** coding is, we replace it with the suppression toggle code:

C#

```
if (treeView1.SelectedNode == null)
   return;

if (hashStore.Contains(treeView1.SelectedNode))
{
   object link = hashStore[treeView1.SelectedNode];

   if (treeView1.SelectedNode.Index != 0)
   {
      try
      {
         Component2 comp = (Component2)link;
         int state = comp.GetSuppression();
         if (state ==
(int)swComponentSuppressionState_e.swComponentSuppressed)
            state =
(int)swComponentSuppressionState_e.swComponentFullyResolved;
         else
            state =
(int)swComponentSuppressionState_e.swComponentSuppressed;

         comp.SetSuppression2(state);
      }
```

```
        catch
        {
            Feature feat = (Feature)link;
            Boolean[] suppression =
(Boolean[])feat.IsSuppressed2((int)swInConfigurationOpts_e.swThisConf
iguration, null);
            int state;
            if (suppression[0])
                state =
(int)swComponentSuppressionState_e.swComponentFullyResolved;
            else
                state =
(int)swComponentSuppressionState_e.swComponentSuppressed;

            feat.SetSuppression2(state,
(int)swInConfigurationOpts_e.swThisConfiguration, null);
        }
    }
}
```

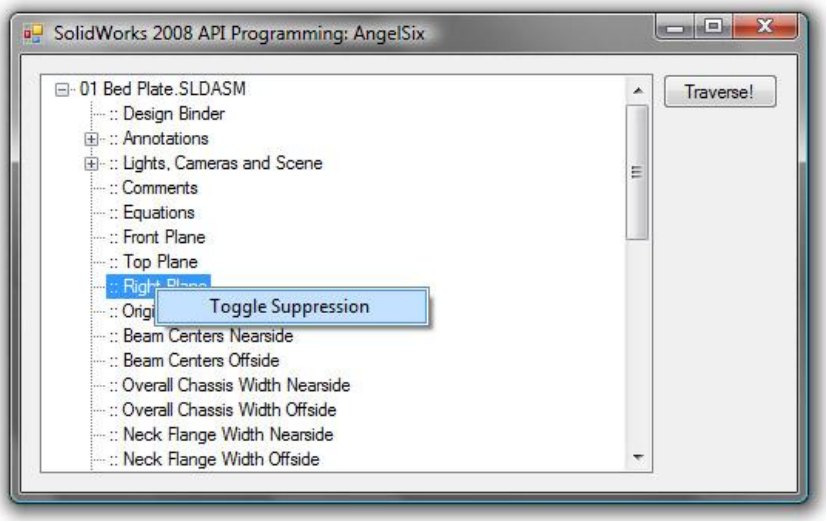
If the associated object is a component we call the **GetSuppression** function to retrieve the current state of the component. Next, we check if it is already suppressed; if it is, we set the new state to unsuppressed, if it isn't we set it to suppressed.

With the state defined, we set it using the function **SetSuppression2** function.

If the object is a feature, we call the Feature function **IsSuppressed2**, which returns an array of suppression states for the feature in all

configurations. Since we are only interested in the current configuration, we just check the first item in the returns array. Again, if it is already suppressed we unsuppress it, and if it is unsuppressed we suppress it. With the state defined, we set it using the function **SetSuppression2** just like the component.

Test the program, and this time you can select an object first by left-clicking, and then right-click to show the context menu, and click **Toggle Suppression** to toggle the suppression state of the item.



Here endith the lesson. You should now have a good understanding of the traversing process, as well as some of the functions and methods you can use upon components and features. You should be able to easily expand this program to do anything you please.

Custom Property Manager

Acquiring a Custom Property Manager

Adding Custom Properties

Deleting Custom Properties

Check Custom Property Existence

Updating Custom Properties

The ConfigSearcher program

Using the **Custom Property Manager** is fairly simple; once you have the CPM, you can either **Add**, **Delete** or **Update** a custom property in either the **Custom Property** or the **Configuration Specific Property**. Once you have these basics down, we will create a simple user interface for letting the user modify properties in the active document.

Acquiring a Custom Property Manager

By now you should be more than familiar with ways of getting a handle to a **ModelDoc2** object so I will not bore you with the finer details.

In order to get a **CustomPropertyManager** object to allow us to work with the custom properties of the associated model, we have to get the **Extension** object of the **ModelDoc2** object, and call its function **get_CustomPropertyManager** in C#, and **CustomPropertyManager** in VB.Net and VBA. This function accepts one parameter:

```
Retval = ModelDocExtension.CustomPropertyManager ( ConfigName )
```

> ***ConfigName*** *is the name of the configuration to get the manager for.If you pass an empty string in you will get the custom property manager, not the configuration specific manager.*

So, firstly create a new variable for our manager:

C#

```
CustomPropertyManager cpm;
```

VBA

```
Dim cpm As CustomPropertyManager
```

Then all that is left is to call the function to get the manager from any **ModelDoc2** object:

Custom Property Managers

C#

```
cpm = swModel.Extension.get_CustomPropertyManager("");
```

VBA

```
Set cpm = swModel.Extension. CustomPropertyManager("")
```

You now have a working **CustomPropertyManager** object to access and modify the custom properties with.

But what if you want to get a manager to access the configuration specific properties?

Configuration Specific Managers

C#

```
cpm = swModel.Extension.get_CustomPropertyManager("Default");
```

VBA

```
Set cpm = swModel.Extension. CustomPropertyManager("Default")
```

This will get a manager for accessing the properties of the configuration called "Default". But what if you do not know the names of the configurations? This is how you get get all configuration names. You can then loop through them and use them however you like:

C#

```
string[] modelconfigs = (string[])swModel.GetConfigurationNames();
```

VBA

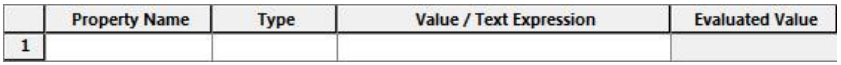
```
Dim modelconfigs As Variant
modelconfigs = swModel.GetConfigurationNames
```

Adding Custom Properties

With access to a working **CustomPropertyManager** you can now start to use it how you wish. Let's begin with showing you how to add, delete and update properties, as well as check for success.

To add a custom property we call the following method from the CPM:

```
Retval = CustomPropertyManager.Add2 ( FieldName,
FieldType, FieldValue)
```



	Property Name	Type	Value / Text Expression	Evaluated Value
1				

*The **FieldName** parameter is the name of the custom property field that you want to add. In the image above it is the equivalent of the **PropertyName** field.*

*The **FieldType** is an enumerator of type **swCustomInfoType_e**, with the following options:*

- *swCustomInfoUnknown*
- *swCustomInfoText*
- *swCustomInfoDate*
- *swCustomInfoNumber*
- *swCustomInfoYesOrNo*
- *swCustomInfoDouble*

*In the user interface for the custom properties the user can only select 4 options (Text, Date, Number, Yes/No). The **Double***

*option is simple a non-whole number, and the **Unknown** option is for errors.*

*The FieldValue is the **Value / Text Expression** field that the user sees, not the **Evaluated Value**.*

*The return value is **0** if it fails and **1** if it succeeds.*

So, to add a custom property called "Description" with a value of "My Description", of type **Text** we do this:

C#

```
int iRet = cpm.Add2("Description",
SwConst.swCustomInfoType_e.swCustomInfoText, "My Description");
```

VBA

```
Dim iRet As Long
iRet = cpm.Add2("Description", swCustomInfoText, "My Description")
```

We then just check this **iRet** value for true (**1**) or false (**0**):

C#

```
if (iRet == 0)
    // Failed
else
    // Succeeded
```

VBA

```
If iRet = 0 Then
‘ Failed
Else
‘ Succeeded
End If
```

Another quick note: the **Add2** function will return a failure if the property already exists in the first place.

Deleting Custom Properties

The **Delete** function is much shorter to explain than the **Add2** function as it only has 1 parameter.

```
retval = CustomPropertyManager.Delete ( FieldName)
```

	Property Name	Type	Value / Text Expression	Evaluated Value
1				

*The **FieldName** is the name of the property you wish to delete. This is the **Property Name** field in the image above.*

*The return value is **0** if it fails and **1** if it succeeds.*

To delete the custom property field we theoretically just created, called "Description", we call the method **Delete** of the **CustomPropertyManager** like so:

C#

```
int iRet = cpm.Delete("Description");
```

VBA

```
Dim iRet As Long
iRet = cpm.Delete("Description")
```

And once again we check for a success for failure:

C#

```
if (iRet == 0)
      // Failed
```

```
else
    // Succeeded
```

VBA

```
If iRet = 0 Then
‘ Failed
Else
‘ Succeeded
End If
```

Note: This function will return a failure if the custom property did not exist in the first place, so before we move on to updating custom properties let’s take a look at how to check if one already exists.

Check Custom Property Existence

The quick and simple way to get a custom property value is to use the function **Get2** of the CPM object:

C#

```
string theval, thevalres;

cpm.Get2(fieldName, out theval, out thevalres);
```

VBA

```
Dim theval As String
Dim thevalres As String
Cpm.Get2 fieldName, theval, thevalres
```

The **fieldName** is the name such as "Description" that we used earlier, and the **theval** and **thevalres** variables are the **Value** and **Evaluated Value** of the property. After the function call we can then check if the **theval** variable is a blank string. This usually indicates that the property does not exist because the user cannot set a property value as nothing. The problem is macros and programs can programmatically set a property value as a blank string, so this is not always an accurate way to test if a property actually exists.

The better way

The longer-winded but fool-proof way to check whether a custom property already exists is to get a list of all of the names of the properties using **GetNames** function, and then checking every name until you get a match.

C#

```
string theval, theval2;

// Get all property names
List<string> propnames = new List<string>();
string[] getnames;

 getnames = (string[])cpm.GetNames();

 if (getnames != null)
     foreach (string prop in getnames)
         propnames.Add(prop.ToUpper());

// Check whether to overwrite if exists
if (propnames.Contains(fieldName.ToUpper()))
{
    // FOUND
}
else
    // NOT FOUND
```

We start by creating a new string **List** called **propnames** and a string array called **getnames**. We fill **getnames** with all the property names in our CPM, then we then loop through every name and add it to our **propnames** List. The only reason we do this is so that after, we can run a **List** function called **Contains**, which checks for us whether the list contains a certain element.

We check whether it contains the field we are looking for, and go from there.

You may notice that I add and check the names with a function after them called **ToUpper**. This function converts all letters to uppercase because the **Contains** function is case sensitive, but the CPM is not, so by them both uppercase it removes any case mismatch.

Now for the VBA version:

VBA

```
Dim theval As String
Dim thevalres As String
Dim bFound As Boolean

Dim getnames As Variant

getnames = cpm.getnames()
bFound = False

If UBound(getnames) > 0 Then

  Dim prop As Variant

  For Each prop In getnames
    If UCase(prop) = UCase(fieldName) Then
      bFound = True
    End If
  Next
End If
```

Here we have done things slightly different; instead of using a list, we have simply checked the results as we go, and the end result is in the **bFound** variable.

You can bundle this code up into a nice function to return true or false, you will do this later.

Then to properly check for a deleted property you can first check if it exists or not first.

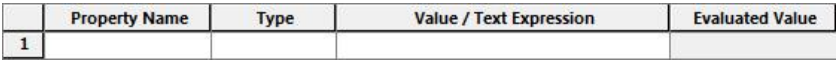
Updating Custom Properties

The final step in this little journey is to see how to update an already existing custom property.

Like the **Delete** function, updating a custom property will return a failure if it does not already exist, so before you try to update a property check that it exists first.

To update a custom property we use the following function:

```
retval = CustomPropertyManager.Set ( FieldName, FieldValue)
```



	Property Name	Type	Value / Text Expression	Evaluated Value
1				

*The **FieldName** is the **Property Name** in the image above like other other functions, of the field you want to update the value for.*

*The **FieldValue** is the new value you want to give the property. This is the **Value / Text Expression** field in the image.*

*The return value is **1** if it fails and **0** if it succeeds.*

We update our property like this:

C#

```
cpm.Set(fieldName, "My New Value");
```

VBA

```
cpm.Set fieldName, “My New Value”
```

Where **fieldName** is the usual value for the field**,** such as “Description” in this running example and we are setting its value to “My New Value”.

If you want to get the original value before overwriting it use the **Get2** function we went over previously.

The ConfigSearcher Program

We are going to create a program that will search custom properties in all configurations of parts and assemblies, and display its results in a **TreeView** control. This should help demonstrate using the **Custom Property Manager**, and you should easily be able to expand this tool to add, delete and update properties.

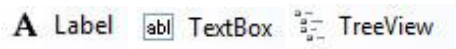
This tool will connect to the active SolidWorks running, get the active document, and if it is an **Assembly** or **Part**, do the following:-

- Ask user for **Property Name**
- Ask user for **Value** to search for

Then, with that information, when the user clicks the button, the tool will go through every configuration specific CPM object, check *all* of the custom properties looking for the **Property Name** specified for the user, and if its value contains the **Value** that the user entered, add the results to the **TreeView** control.

Start with a new project in C# or VB.Net, using the usual template. Go to the **Form Designer**. You should already have the default button there; we are going to add a few more controls.

From the toolbar to the left drag in 2 **Label** controls, 2 **TextBox** Controls, and 1 **TreeView** control.



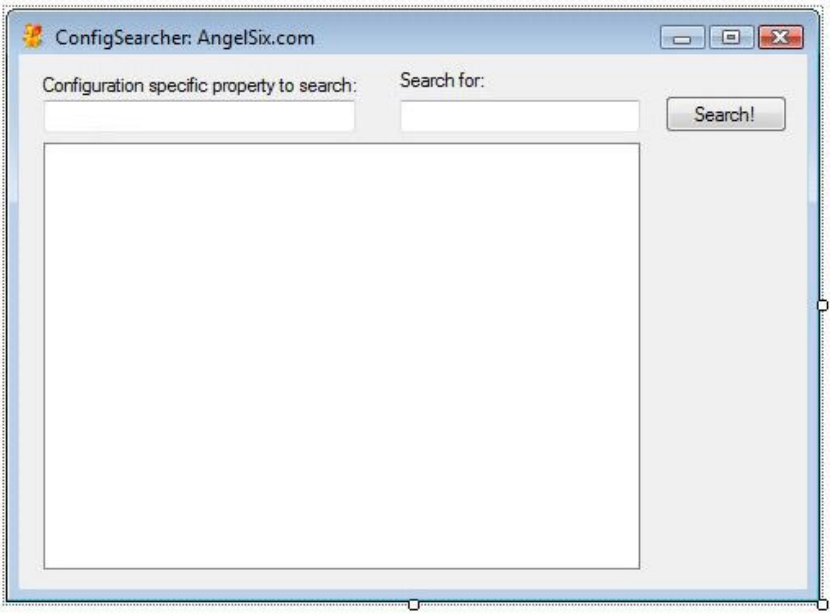
Change the **Text** property of one **Label** control to:

```
"Configuration specific property to search:"
```

Change the other to:

```
"Search for:"
```

Then position the items something like this:



Set the **(Name)** property of the left-hand **TextBox** control to:

```
tbConfigProperty
```

Set the **(Name)** property of the right-hand **TextBox** control to:

```
tbFind
```

And finally we want to change the name of the **TreeView** control (the big control at the bottom):

Set the **(Name)** property of the **TreeView** control to:

```
tvList
```

Searching the properties

If you haven't already, add a **Click** event handle to the button like usual. Within this buttons event function place the code for connecting to SolidWorks, getting the active documents, and checking that the document is not a drawing (as drawings cannot have configuration specific properties):

C#

```
try
{
   swApp =
(SldWorks.SldWorks)Marshal.GetActiveObject("SldWorks.Application");
}
catch
{
   MessageBox.Show("Error getting SolidWorks Handle");
   return;
}

swModel = (ModelDoc2)swApp.ActiveDoc;

if (swModel == null)
{
   MessageBox.Show("Failed to get active document");
   return;
```

```
}
if (swModel.GetType() == (int)swDocumentTypes_e.swDocDRAWING)
{
    MessageBox.Show("Active document cannot be a drawing");
    return;
}
```

Now we are ready to start our search. We begin by clearing the **TreeView** control of any items from previous searches, and getting the variables entered by the user ready for use:

C#

```
tvList.Nodes.Clear();

string configProp = tbConfigProperty.Text;
string searchFor = tbFind.Text;
string value;
```

As you can see the first line is a call to our **TreeView** control that we called **tvList**. By accessing the **Nodes** object we can call the function **Clear,** to remove any nodes in the list.

We then gather the information from the **TextBox** controls using the **Text** property of them which returns the text that the user has typed in.

The third variable **value** will be used to store the original value of the custom property we retrieve later, before comparing it the what the user wants to find.

We continue with a few more variables and retrieving all of the configuration names for this document, so that we can check each one later:

C#

```
Configuration config;
CustomPropertyManager cpm;
string[] modelconfigs = (string[])swModel.GetConfigurationNames();
```

No explanation needed here. Now for the main loop; don't try to understand it all straight away I will take you through it:

C#

```
if (modelconfigs != null)
   foreach (string s in modelconfigs)
   {
      config = (Configuration)swModel.GetConfigurationByName(s);
      cpm = config.CustomPropertyManager;

      value = GetCustomProperty(cpm, configProp);

      if (value.ToUpper().Contains(searchFor.ToUpper()))
         AddConfig(cpm, s);
   }
```

Firstly, we check that we managed to actually get the configuration names; normally this wouldn't fail but just in case.

Then we loop through every name in the list, and for each name we acquire its **Configuration** using the **ModelDoc2** function **GetConfigurationByName**. This takes the name of the configuration as a parameter:

```
retval = ModelDoc2.GetConfigurationByName ( name )
```

The function returns **Nothing** in VBA and VB.Net, and **null** in C# so we could check if we managed to get it. The reason we don't is because the only time this returns failure is when you pass an invalid name in, and since we used the **GetConfigurationNames** function this will never fail.

With the **Configuration** object we can then acquire the CPM we have been long waiting for by accessing it through the **Configuration** object.

The only thing left to do is get the current custom property value for the field we are after, and to check whether the value we are searching for is contained within it. As you can see we get the value using a function called **GetCustomProperty** which accepts a **CustomPropertyManager** object and a **string**. This is not a built-in function it's one you will create in a moment.

With the value of the field the user was after, we check whether it contains the string the user is searching for. To do this we must cast both the original string and the one we are looking for to uppercase to remove any case-sensitivity (such as 'A' = 'a' being false), unless you want case-sensitivity. With uppercase values we then call the string function **Contains**, which returns true or false if a match is found.

Finally, if the property value the user specified does contain the value they were searching for, we want to add this configuration to our list to show that it was found. Again we do this with a function called **AddConfig** that takes a **CustomPropertyManager** object, and a **string**. You will create this function as well as the **GetCustomProperty** function now.

GetCustomProperty function

You have actually written this function before! If you go back several pages to near the beginning of this chapter to the section **Check Custom Property Existence**, you will find the following:

C#

```
string theval, theval2;

// Get all property names
List<string> propnames = new List<string>();
string[] getnames;

 getnames = (string[])cpm.GetNames();

 if (getnames != null)
    foreach (string prop in getnames)
       propnames.Add(prop.ToUpper());

// Check whether to overwrite if exists
if (propnames.Contains(fieldName.ToUpper()))
{
   // FOUND
```

```
}
else
   // NOT FOUND
```

This searched for the existence of a property. All we do now is add 2 lines of code replacing the 2 comments, and add a return value to get the function we need, which retrieves the value of the property that is passed in as a **string,** using the **CustomPropertyManager** object given:

C#

```
private string GetCustomProperty(CustomPropertyManager cpm, string
fieldName)
{
   string theval, thevalres;

   // Get all property names
   List<string> propnames = new List<string>();
   string[] getnames;

   if (cpm != null)
   {
      getnames = (string[])cpm.GetNames();
      if (getnames != null)
         foreach (string prop in getnames)
            propnames.Add(prop.ToUpper());
   }

   // Check whether to overwrite if exists
```

```
    if (propnames.Contains(fieldName.ToUpper()))
    {
        cpm.Get2(fieldName, out theval, out thevalres);
    }
    else
        theval = "";

    return theval;
}
```

If the property we are looking for doesn't exist, we just return a blank string; else it calls the **Get2** function I described in the previous section to get the value of the property. The function returns the **theval** variable to the caller.

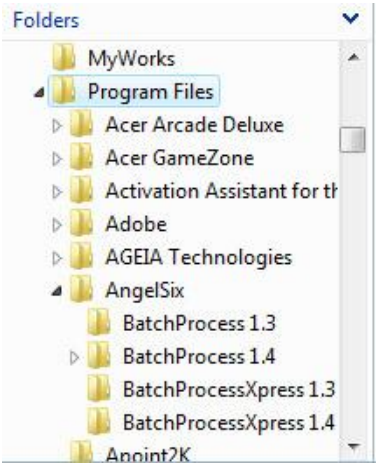
AddConfig function

The last step is to actually display the results once they have been found. For this we want to show each configuration that matches the search criteria, and as a bonus we will then add every configuration specific property to the tree view node of that configuration:

C#

```
private void AddConfig(CustomPropertyManager cpm, string name)
{
    TreeNode tn = tvList.Nodes.Add(name);
    string[] getnames;
    if (cpm != null)
    {
```

```
        getnames = (string[])cpm.GetNames();
        if (getnames != null)
            foreach (string prop in getnames)
                tn.Nodes.Add(prop + ": " + GetCustomProperty(cpm, prop));
    }

}
```



We start by adding a new node to our **TreeView** control using the **Add** function. This function returns a handle to the **TreeNode** object it just created, and adds the node to the tree. A node is an entry like in a **ListBox** control, but it can have child nodes within it, which display under it with the **+/-** symbols, like the Windows Explorer.

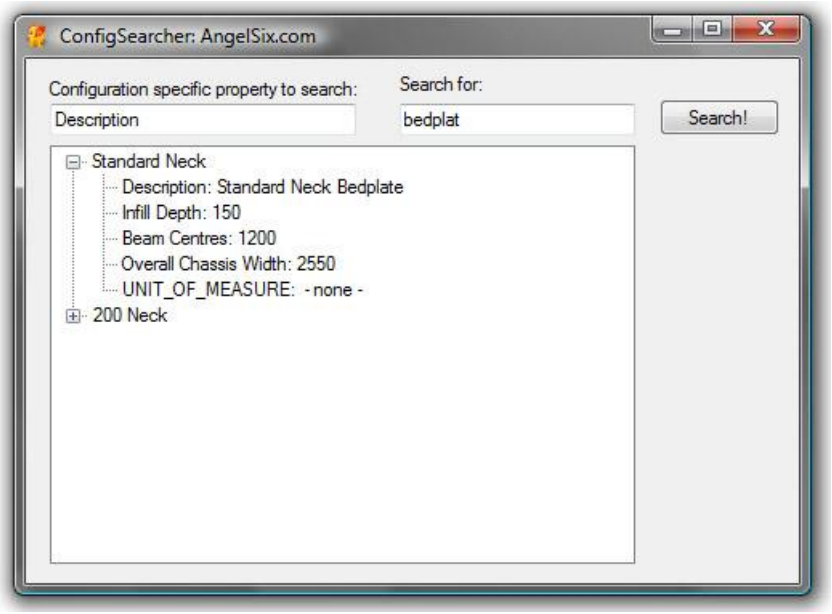
We store this in a new **TreeNode** variable for use later. Passing a **string** into the **Add** function sets the text of the node to that string, so it makes sense that we want our item to appear with the name of the configuration, so we passed that into this function in our code above.

With the **TreeNode** added, we then check whether the CPM object is valid or not, and if it is we get all of the property names from the CPM (all of the configuration specific properties), and then add a child node for each property showing its name and value.

And that is pretty much it in a nutshell! Compile and run your program and with SolidWorks open and an assembly or part open test it out.

Make sure the model has some configuration specific properties so that you can test the searching function and then run it to see the results.

As usual find the complete source code on the CD. This tool is only made it the .Net languages.



Working with Drawings

Automatically create Drawing Sheet

Counting Views

Printing Drawing Sheets

We have focused mostly on parts and assemblies so far, so let's take a look at some things to do with drawings. We will start by creating a macro/program that automatically creates drawing sheets from parts and assemblies, and move on to analysing features of the drawing and more.

Automatically create Drawing Sheet

This is simple enough to do; you know the routine by name, usual template with button or macro with it connecting to SolidWorks, but this time do not bother checking for an active document as we do not need one in this example.

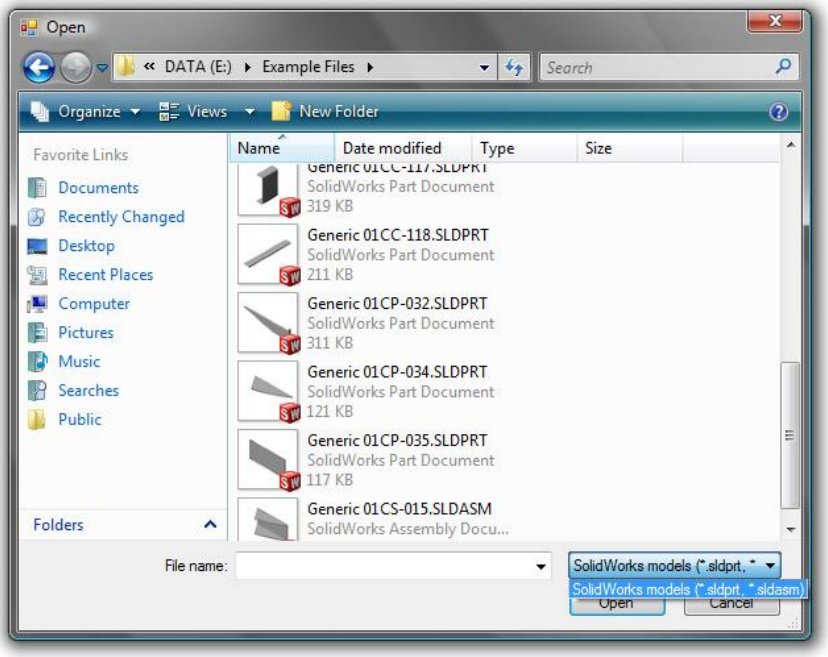
The first thing we want to do when the user clicks the button or runs the macro is to ask the user for the part or assembly to create a new drawing from. In .Net we can do this using an **OpenFileDialog,** but in VBA we must use a much more basic technique of having the user type in. We could import Windows API calls from the **comdlg32.dll,** but that's too much to explain and would veer off the SolidWorks programming topics.

C#

```
OpenFileDialog ofd = new OpenFileDialog();
ofd.CheckFileExists = true;
ofd.Filter = "SolidWorks models (*.sldprt, *.sldasm)|*.sldprt;*.sldasm";
ofd.ShowDialog();
```

We create a new instance of the **OpenFileDialog** object and then show it to the user by calling the **ShowDialog** function.

Before that we set the **Filter** property, which will limit the user to selecting only the file formats we want them too. This filter follows the format of starting with the name which will be displayed to the user, separated by a pipe (**|**) then the actual regex string to match file types, separated by semicolons (**;**).



Notice the filter list that we typed in and how it has limited the files we can see to assemblies and parts.

Once the **ShowDialog** function has returned, the property **FileName** of the **OpenFileDialog** will either be empty if the user clicked cancel, or the full location of the file they selected, so we check this out.

If the user clicked cancel we will just ignore this and end our function, but if they selected a file we will create a new variable to store this value:

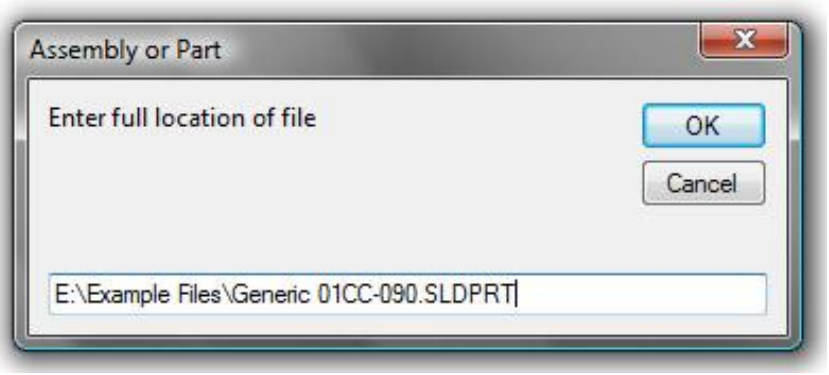
C#

```
string filename = ofd.FileName;

if (filename == "")
    return;
```

And here is the VBA coding so far:

VBA

```
Dim filename As String
filename = InputBox(Prompt:="Enter full location of file",
Title:="Assembly or Part")

If filename = "" Then
Exit Sub
End If
```



The InputBox function of VBA creates this little input box for us, so it will do for now.

Unlike .Net, VBA has not prevented the user from selecting specific files, nor whether the file even exists, so we must do this now:

VBA

```
Private Function FileExists(ByVal sPathName As String, Optional
Directory As Boolean) As Boolean

On Error Resume Next
If sPathName <> "" Then

If IsMissing(Directory) Or Directory = False Then
FileExists = (Len(Dir(sPathName)) <> 0)
Else
FileExists = (Len(Dir(sPathName, vbDirectory)) <> 0)
End If
End If
```

```
End Function
```

This function will return **True** if a file or directory exists, and **False** if not, so now we do this:

VBA

```
If Len(filename) < 8 Then
MsgBox "Invalid filename"
Exit Sub
End If

Dim ext As String
ext = UCase(Right(filename, 7))
If Not FileExists(filename) Or (ext <> ".SLDASM" And ext <> ".SLDPRT")
Then
MsgBox "Invalid filename"
Exit Sub
End If
```

Here we check the filename is long enough to be valid, and if so get the last 7 characters of the name, and check whether the case-insensitive version matches that of a part or assembly, and that the file exists, else it exits.

You can see the clear advantages .Net has here.

Creating the Drawing Sheet

With the user input gathered for the file location all that is now left is to create a drawing from this file. We are going to add the 3 common views to the sheet for this example.

When creating a drawing we must firstly select a drawing template to use, so let's start by getting that using from the SolidWorks **System Settings** for this user.

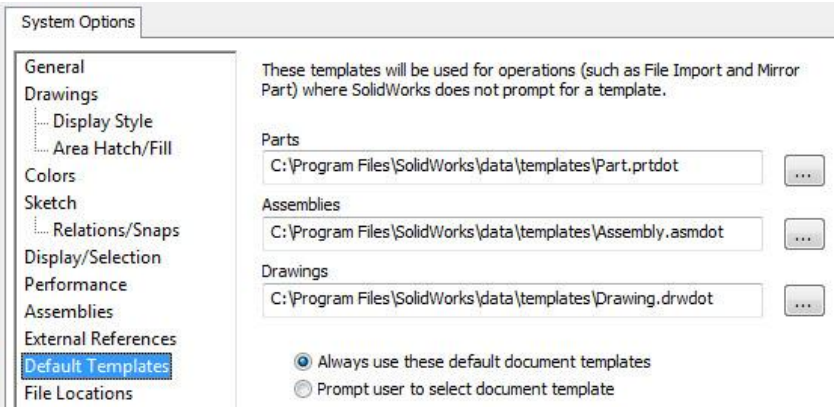
C#

```
string drwTemplate =
swApp.GetUserPreferenceStringValue((int)swUserPreferenceStringValu
e_e.swDefaultTemplateDrawing);
```

VBA

```
Dim drwTemplate As String
drwTemplate =
swApp.GetUserPreferenceStringValue(swDefaultTemplateDrawing)
```

The **SldWorks** functions **GetUserPreference*** and **SetUserPreference*** are used to get and set most of the options you see in the **System** and **Document** options of SolidWorks. In this case we are pulling in the default drawing template location:



With the template name acquired we can now create a new drawing using the following function of the **SldWorks** object. We actually used this function back at the very beginning:

```
SldWorks.NewDocument ( templateName, paperSize,
width, height )
```

Only this time we will be using the **paperSize, width** and **height** parameters. We will acquire all of these from the template name we just got using another function:

```
retval = SldWorks.GetTemplateSizes ( filename )
```

> *The **filename** is the name of the template to retrieve the values from.*
>
> *The return value is an array of **Double** numbers, or in VBA a **Variant** object, containing, in order, the paper size, width and height.*
>
> *The **papersize** value can be cast to an enumerator of type **swDwgPaperSizes_e** if necessary.*

Let's get the sizes to complete the equation:

C#

```
double[] sizes = swApp.GetTemplateSizes(drwTemplate);

if (sizes == null)
{
    MessageBox.Show("Failed to get valid template sizes.");
    return;
}
```

VBA

```
Dim sizes As Variant
sizes = swApp.GetTemplateSizes(drwTemplate)

If IsEmpty(sizes) Then
MsgBox "Failed to get valid template sizes."
Exit Sub
End If
```

We also check that we managed to get the sizes before the next step, as this will fail if the template is corrupt or has invalid data, or does not exist.

With the template name, paper size, width and height, we call the function **NewDocument** to create the new drawing sheet we are after, and store the returned handle to a new variable to access later for adding views to.

C#

```
DrawingDoc swDrawing =
(DrawingDoc)swApp.NewDocument(drwTemplate, (int)sizes[0], sizes[1],
sizes[2]);
```

VBA

```
Dim swDrawing As DrawingDoc
Set swDrawing = swApp.NewDocument(drwTemplate, sizes(0), sizes(1),
sizes(2))
```

In C# we have to cast the first size entry to an **int** as the function requires an integer. We also cast the returned **Object** to a **DrawingDoc,** usual procedure. VBA does the casting for us so it looks a bit simpler.

At this stage SolidWorks should now have a new drawing document in its active window**,** and our handle should point to it. Do the usual check just in case however:

C#

```
if (swDrawing == null)
{
    MessageBox.Show("Failed to create new drawing document.");
    return;
}
```

VBA

```
If swDrawing Is Nothing Then
MsgBox "Failed to create new drawing sheet."
Exit Sub
End If
```

Adding the drawing views

With our drawing sheet successfully create the next step is to add the standard 3-views of our selected model to our new drawing document using this function:

```
retval = DrawingDoc.Create3rdAngleViews2 (
modelName)
```

> *The **modelName** is the name of the assembly or part that you want to create the views from.*
>
> *The return value is **true** if success, and **false** if failure.*

There is also a **Create1stAngleViews2** function if you would like to create 1st angle views instead, its left up to you. You could even have a checkbox or ask the user which they would like to create very simply.

Here is our call:

C#

```
bool bRet = swDrawing.Create3rdAngleViews2(filename);

if (!bRet)
```

```
{
    MessageBox.Show("Failed to add 3 common views.");
    return;
}
```

VBA

```
Dim bRet As Boolean
bRet = swDrawing.Create3rdAngleViews2(filename)

If bRet = False Then
MsgBox "Failed to add 3 common views."
Exit Sub
End If
```

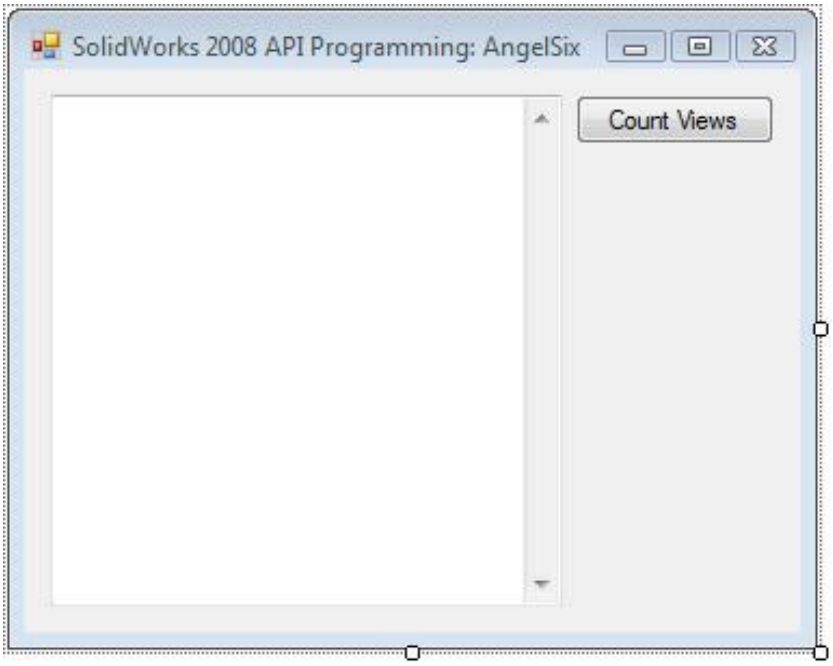
Notice the variable we pass in (**filename**) is the name of the model we asked the user to select earlier. If all has gone well we have now created the 3-views in the drawing document of the user selected model.

Counting Views

Another off the shelf random exercise for you now; this time we are going to find out how many drawing views a drawing has in each sheet. This may not seem of much use, but this demonstrates how to access handles to drawing views to do what you please with. We will be displaying the drawing view types and names in this example.

Start with the usual template, and get the active document, check that it is a drawing and proceed.

Starting with .Net as usual, in the **Form Designer** drag a new **TextBox** control to your form; we are going to use this to display the information. Set its **Multiline** property to **True** and its **ScrollBars** property to **Both**.



If you remember some chapters back where we created the program/macro to loop all drawing sheets and save them as DXF, we will be using the same coding here to loop through each sheet, and then within that loop get our information:

C#

```
DrawingDoc swDrwDoc = (DrawingDoc)swModel;
string[] sheetNames = (string[])swDrwDoc.GetSheetNames();
Sheet swDrwSheet;
textBox1.Text = "";

foreach (string sheetname in sheetNames)
{
   swDrwDoc.ActivateSheet(sheetname);
   swDrwSheet = (Sheet)swDrwDoc.GetCurrentSheet();

   GetSheetInformation(swDrwSheet);
}
```

The only difference here is that we clear our **TextBox** control of any text before we begin so that any text already in is removed, and we have added an extra variable called **swDrwSheet**, which is a **Sheet** object. Once each sheet is activated, we get a handle to it using the function **GetCurrentSheet**. With this variable we can then do what we please with the sheet, so to keep things clean and tidy we pass this **Sheet** object to a new function that will do our dirty work, called **GetSheetInformation**. Place your function below the button event function:

C#

```
private void button1_Click(object sender, EventArgs e)
{
  ...
  ...

   DrawingDoc swDrwDoc = (DrawingDoc)swModel;
   string[] sheetNames = (string[])swDrwDoc.GetSheetNames();
   Sheet swDrwSheet;

   foreach (string sheetname in sheetNames)
   {
      swDrwDoc.ActivateSheet(sheetname);
      swDrwSheet = (Sheet)swDrwDoc.GetCurrentSheet();

      GetSheetInformation(swDrwSheet,
(SldWorks.View)swDrwDoc.GetFirstView());
   }
}

private void GetSheetInformation(Sheet sheet, SldWorks.View firstView)
{

}
```

I have omitted some code; this is just to show you where to place your function.

The function takes the **Sheet** object for obvious reasons, and a **View** object that will be the first view of the sheet so that we can loop

through them all within the function, gather the information, and add that info to our **TextBox** control.

C#

```
int iCount = 0;
string nl = System.Environment.NewLine;
string tab = " -  ";

textBox1.Text += "Sheet Name: " + sheet.GetName() + nl;

while (firstView != null)
{
   string viewType =
((swDrawingViewTypes_e)firstView.Type).ToString();
   textBox1.Text += tab + firstView.GetName2() + " (" + viewType + ")" +
nl;

   iCount++;
   firstView = (SldWorks.View)firstView.GetNextView();
}

textBox1.Text += tab + "Total views: " + iCount.ToString() + nl + nl;
```

The first thing we do is to create a new variable to store the number of views we encounter called **iCount**. Then we create a few **string** variables for a new line (like pressing **Enter** on the keyboard), and for a tab to show that an item is indented. Then we add the sheet name to our **TextBox** text as well as a new line.

Next we loop every view within this sheet by checking if the current view is **null** or not, and if it isn't process it and then call the View function **GetNextView**; this will return the next view of its parent **Sheet** object, or **null** if it has reached the end.

For each view we then get the type of view we are dealing with by converting the integer returned from the **Type** property of the **View** object back to an enumerator variable of type **swDrawingViewTypes_e**, and then back into a readable string like we have done in previous examples.

With a readable type value, we then add the views' name and type to the text. Notice we have added the tab variable to the start; this creates the effect of indenting our entries to show it is part of the sheet.

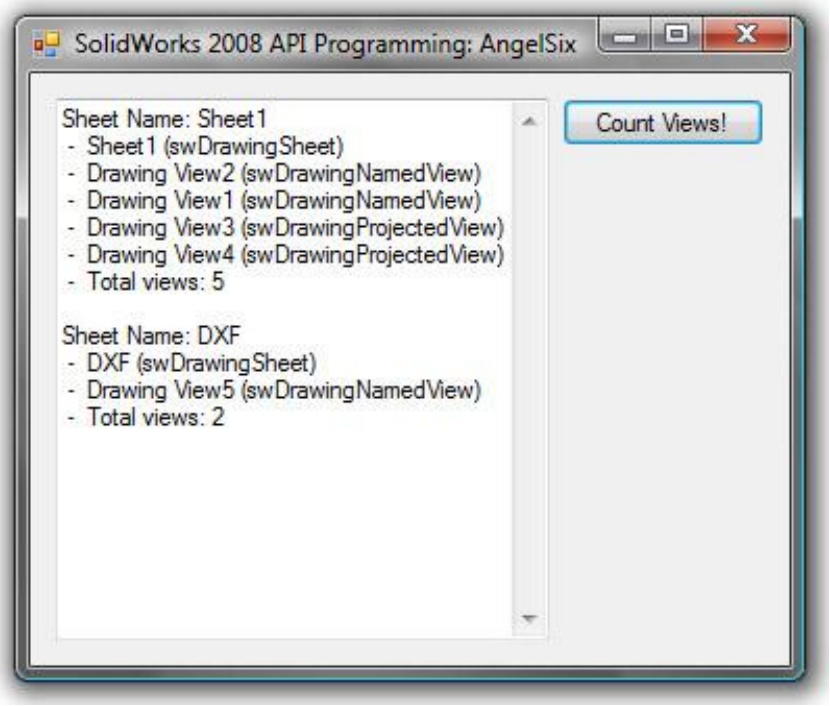
Once the details have been added we increment the view count and loop.

Finally when all views have been processed we want to tell the user how many views there were on this particular sheet. All we need to do for this is to show the user the **iCount** value:

```
textBox1.Text += tab + "Total views: " + iCount.ToString() + nl + nl;
```

With this being the final line of this sheet that we are adding to the **TextBox**, we add two new lines instead of one to give it a clean break from the next sheet.

Compile and test your program and take a look at the results:

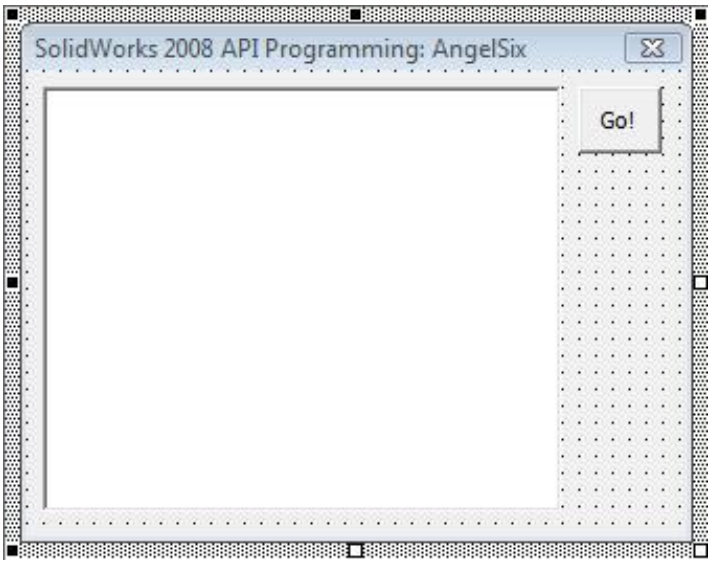


You will notice here that the first view of all sheets is actually the sheet itself; this is because the structure of the SolidWorks sheet is basically just a bunch of **View** objects embedded inside other **View** objects. If you want to exclude this item from the list just check the **Type** value, and if it is the type **swDrawingSheet**, skip it.

Now onto the VBA version of this little tool; start with the usual template. Insert a new **User Form** from the **Insert** menu and add a **CommandButton** and a **TextBox** to the form.

Set the **MultiLine** property of the **TextBox** to **True**, and the **ScrollBars** property to **Both**.

Position them so they look something like this:



Double-click the button to add the button Click event function. Move the usual code (for connecting to SolidWorks, getting the active document and checking if it is a drawing) from the **main** function to this Click event function, and within the **main** function place this code to show the form:

VBA

```
Sub main()
UserForm1.Show
End Sub
```

Back to the event function below the code we just pasted in, we add the same code I described above in the .Net version.

VBA

```
Dim swDrwDoc As DrawingDoc
Set swDrwDoc = swModel
Dim sheetNames As Variant
sheetNames = swModel.GetSheetNames()
Dim swDrwSheet As sheet

TextBox1.Text = ""

Dim sheetname As Variant
For Each sheetname In sheetNames

   swDrwDoc.ActivateSheet sheetname
   Set swDrwSheet = swDrwDoc.GetCurrentSheet()

   GetSheetInformation swDrwSheet, swDrwDoc.GetFirstView()

Next sheetname
```

If you need an explanation of this code just take a look at the C# version above as all of the theory is explained there.

All that is left now is to create the **GetSheetInformation** function. This function is slightly different than the other version as explained many times now VBA lacks the ability to convert and enumerator to a string directly.

I will leave out the code line of acquiring the type value and show that separately.

VBA

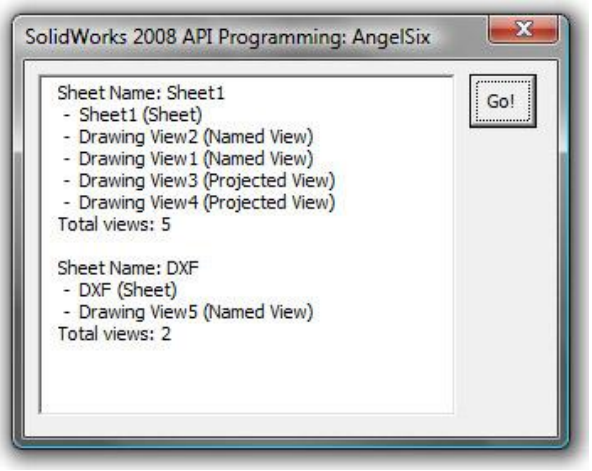
```
Private Sub GetSheetInformation(ByVal sheet As sheet, firstView As
View)

Dim iCount As Integer
iCount = 0
Dim sTab As String
sTab = " -  "

TextBox1.Text = TextBox1.Text & "Sheet Name: " & sheet.GetName() &
vbCr

While Not firstView Is Nothing
    Dim iType As Integer
    iType = firstView.Type
    Dim viewType As String
    viewType = TODO

    TextBox1.Text = TextBox1.Text & sTab & firstView.GetName2() & " ("
& viewType & ")" & vbCr

    iCount = iCount + 1
    Set firstView = firstView.GetNextView()

Wend

TextBox1.Text = TextBox1.Text & "Total views: " & iCount & vbCr & vbCr

End Sub
```

Notice the **TODO** note within the code. This is the like where we statically convert the integer value of the **Type** property of the **View** object to a user-friendly readable string variable.

VBA

```
viewType = Switch(
iType = swDrawingViewTypes_e.swDrawingAlternatePositionView,
"Alternate Position View",
iType = swDrawingViewTypes_e.swDrawingAuxiliaryView, "Auxiliary
View",
iType = swDrawingViewTypes_e.swDrawingDetachedView, "Detached
View",
iType = swDrawingViewTypes_e.swDrawingDetailView, "Detail View",
iType = swDrawingViewTypes_e.swDrawingNamedView, "Named View",
iType = swDrawingViewTypes_e.swDrawingProjectedView, "Projected
View",
iType = swDrawingViewTypes_e.swDrawingRelativeView, "Relative
View",
iType = swDrawingViewTypes_e.swDrawingSectionView, "Section
View",
iType = swDrawingViewTypes_e.swDrawingSheet, "Sheet",
iType = swDrawingViewTypes_e.swDrawingStandardView, "Standard
View")
```

All we have done here is done a **Switch** function to select the correct value from the list. This function was explained in a previous chapter, but basically it works by returning the variable after the first value that returns **True**; so in this case, we check if "iType = swDrawingDetailView", if it does it returns the variable after that, which is "Detail View", if it is false, it carries on until it finds one.



You may notice that the **TextBox** object can actually be edited by the user simply by typing into it. This is not really a problem but technically it is not good practise for a program to allow the user to do anything they are not meant to be doing. If you want to fix this all you need to do is go back to the **Form Designer** and alter the **Locked** property to **True**. In .Net the property is called **Readonly**.

Printing Drawing Sheets

Another popular function commonly asked for by SolidWorks programmers is the ability to print a document. This section will show you how to print any document, by it a drawing, part or assembly, using the 2 possibly functions (one of which includes the ability to specify your printer name and options), and then give an example of printing each drawing sheet.

I will start by describing the functions we are going to use, and then we will go ahead and implement them into a nice little package.

The two options we have here are to print the document with one line of code, that prints the document to the last printer it was printed to, or the option to specify all details such as printer name. Starting with the simplest method:

PrintDirect function

This function of the **ModelDoc2** object takes no parameters and returns no value, so it couldn't be simpler. Once you have a **ModelDoc2** handle (or an **AssemblyDoc**, **PartDoc** or **DrawingDoc** to cast back to), you can call this function:

```
void ModelDoc2.PrintDirect ()
```

Although this is nice and simple, that is pretty much the last thing you want when you are printing something; it provides no options for printing multiple copies, printing sheet ranges or printing to a specific printer.

PrintOut2 function

If you require more control over your printing then this is the function for you.

```
void = ModelDocExtension.PrintOut2 ( PageArray,
Copies, Collate, Printer, PrintFileName)
```

*The **PageArray** parameter is an array of integer values grouped in pairs. Each pair is a range from the first value to the last such as 1 and 5, which would print pages 1, 2, 3, 4 and 5. If you had an array containing 1, 5, 7 and 8 then pages 1, 2, 3, 4, 5, 7 and 8 would be printed.*

*The **Copies** parameter is the number of copies to send to the printer, ranging from 1 onwards.*

*The **Collate** parameter is a Boolean variable for whether to collate the pages or not. If you have selected to print more than one copy then collating will print pages in complete groups instead of single page groups. For example with collating on and 2 copies selected, the output with collation would be 1, 2, 3, 4, 1, 2, 3, 4. Without collating the same output would be 1, 1, 2, 2, 3, 3, 4, 4, in order of the pages coming out of the printer.*

*The **Printer** parameter is the exact matching name of the Printer of the printer to print to, from the Printers list in the Control Panel. Set to **null** or **Nothing** causes this function to print to the default printer.*

*The **PrintFileName** parameter is the name of the file to save the print output to. This is not like printing to PDF, this is actually sending the print data to a file, not the actual rendered output.*

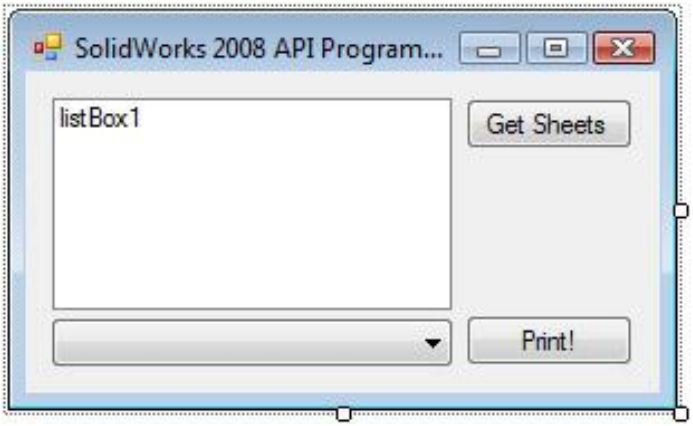
This function is part of the **ModelDocExtension** object, not the **ModelDoc2** object like the **PrintDirect** function, so in order to call it just do the following (presuming **swModel** is a **ModelDoc2** variable):

```
swModel.Extension.PrintOut2(...)
```

Although you can use this function to print any type of document, the following example will be to print drawing sheets only.

The Printing program

Starting the usual fashion with the .Net example first; create a new project with the usual template. Add 2 **Buttons**, a **ListBox**, and a **ComboBox** and place/name them like so:



Make a few changes to the properties of these controls; change the **SelectionMode** of the **ListBox** control:

This will allow the user to select multiple items from the list.

Next, change the **DropDownStyle** of the **ComboBox** control:

This will prevent the user from editing the list as we want them to be fixed to the list here. Name the buttons so there **Text** value is as shown, and add a Click event handle to each.

Getting a list of installed printers

In the **Coding View** we are going to get a list of all installed printers on the computer as soon as the form is opened; first, add the following to the **using** section so that we can access the list of printers using a **System** call:

C#

```
using System.Drawing.Printing;
```

Declare a new variable next to the usual ones:

C#

```
SldWorks.SldWorks swApp;
ModelDoc2 swModel;
string[] printers;
```

This will store our printer list. Within the constructor function add the following:

C#

```
public Form1()
{
    InitializeComponent();
    printers = new string[PrinterSettings.InstalledPrinters.Count];
    PrinterSettings.InstalledPrinters.CopyTo(printers, 0);
    comboBox1.Items.AddRange(printers);
    comboBox1.SelectedIndex = 0;
}
```

We initialise the **printers** variable to the size of the number of printers that are installed on the computer, and then we copy the list of printers to that variable by accessing the array of printers from the **PrinterSettings** object. The **CopyTo** function copies the content of the calling array (**InstalledPrinters**) to the destination array (**printers**), starting at the position in the array specified (**0**). Simplistically this just copies the **InstalledPrinters** array to the **printers** array.

With the array gathered, we load the list into our **ComboBox** control and select the first item in the list by default.

A quick note here; I have not added error checking for if there are no printers at all installed on the computer; if there are not this program will just crash. I will leave the error-checking task up to you as a simple test of what you have learned so far, so if you can manage it.

Getting sheet names

Within the **Click** function of the "Get Sheets" button we are going to place our usual code for connecting to SolidWorks and the rest, and get all drawing sheet names like we have done in previous examples. We then add the sheet names to the **ListBox** control so the user can select which sheets to print:

C#

```
private void button1_Click(object sender, EventArgs e)
{

    try
    {
        swApp =
(SldWorks.SldWorks)Marshal.GetActiveObject("SldWorks.Application");
    }
    catch
    {
        MessageBox.Show("Error getting SolidWorks Handle");
        return;
    }

    swModel = (ModelDoc2)swApp.ActiveDoc;

    if (swModel == null)
    {
        MessageBox.Show("Failed to get active document");
        return;
    }
```

```
    if (swModel.GetType() != (int)swDocumentTypes_e.swDocDRAWING)
    {
        MessageBox.Show("Active document must be a drawing");
        return;
    }

    DrawingDoc swDrwDoc = (DrawingDoc)swModel;
    string[] sheetNames = (string[])swDrwDoc.GetSheetNames();
    listBox1.Items.Clear();

    listBox1.Items.AddRange(sheetNames);
}
```

I am hoping you understand all of this code by now, as every step here has been done at some point more than once in previous chapters.

So far when we start our program (feel free to at this point), the **ComboBox** control will populate with all printer names installed on the computer for the user to select, and upon clicking the "Get Sheets" button the **ListBox** control will populate with all sheet names of the currently active drawing allowing the user to select one or more sheets which to print. So, all that is left for us to do now is to print the selected sheets from the list to the selected printer from the combo box.

Printing the selection

Start by adding a Click event function to the "Print!" button; we will then check that the user has at least selected one sheet to print from the list, and if they have, print them:

C#

```
private void button2_Click(object sender, EventArgs e)
{
   if (listBox1.SelectedItems.Count == 0)
      return;

   foreach (int position in listBox1.SelectedIndices)
   {
      int[] pages = new int[]{position + 1, position + 1};
      swModel.Extension.PrintOut2(pages, 1, false,
(string)comboBox1.SelectedItem, null);
   }
}
```

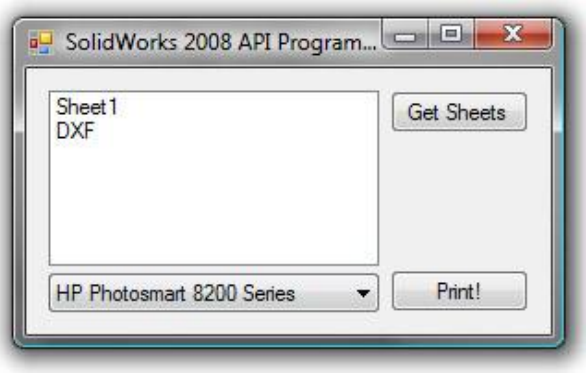
Not too much to do here. To get the user selection from the **ListBox** control we access its property **SelectedItems**. We check how many items have been selected, and if none have, exit.

Next, loop every item the user selected, but this time we use the property **SelectedIndices**. The reason for this is because as shown above using the **PrintOut2** function requires page numbers, not sheet names. The trick here is that the **GetSheetNames** function we used to retrieve the sheet names in the first place retrieves them in the page number order, so this works a treat. We add the 1 to the

position because the **SelectedIndicies** is a 0-based array, whereas page numbers as 1-based arrays (starting at 1, not 0).

Remember that we said the first parameter of the **PrintOut2** function is an array of pairs; since we want to print just this page; we pass in the same page number twice to print just that page. We tell the function to print 1 copy, not to collate, print to the printer the user selected, and ignore the **PrintToFile** name.

That is all there is to it. Run your program, start by getting the sheet names, then select one or more (using the Shift and/or Ctrl keys) and then click the "Print!" button to print the sheets!



Now onto the VBA version; everything is exactly the same as the .Net version except for small differences in acquiring the printer list and the likes.

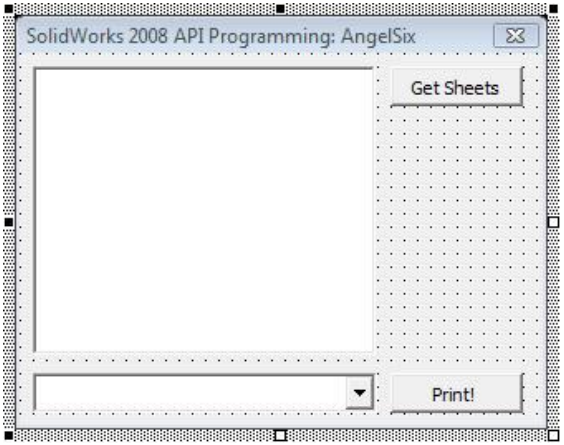
The Printing program in VBA

Start with the usual template. Insert a new form and in the **main** function place the following code to show the form:

VBA

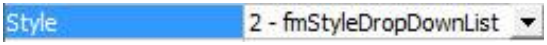
```
Sub main()
UserForm1.Show
End Sub
```

In the **Form Designer** add 2 **Buttons,** a **ListBox** control and a **ComboBox** control. Position them so they look something like this:



We need to change a few properties; select the **ListBox** control and change the property **MultiSelect** to **MultiExtended** so the user can select more than one item from the list.

Select the **ComboBox** control and change the property **DropListStyle** to **List**; this prevents the user from altering the list, which we don't want them doing.



Getting a list of installed printers

Welcome to the world of complexity! Again, due to the lack of power in VBA, getting a list of printers is somewhat more complicated than the .Net version and requires invoking functions from the Windows API libraries. Don't worry about understanding this code as you will only ever need to call the final function, not understand it. I will not explain the function in great detail because there is no need to for understanding SolidWorks API.

Either add a new Module to your project using **Insert->Module,** or use the existing Module file. At the top of the file place the following code. This in a way "imports" the functions we need from the Windows API, for lack of a simpler explanation:

VBA

```
Const PRINTER_ENUM_CONNECTIONS = &H4
Const PRINTER_ENUM_LOCAL = &H2

Private Declare Function EnumPrinters Lib "winspool.drv" Alias
"EnumPrintersA" _
        (ByVal flags As Long, ByVal name As String, ByVal Level As Long,
_
        pPrinterEnum As Long, ByVal cdBuf As Long, pcbNeeded As Long,
_
        pcReturned As Long) As Long

Private Declare Function PtrToStr Lib "kernel32" Alias "lstrcpyA" _
        (ByVal RetVal As String, ByVal Ptr As Long) As Long

Private Declare Function StrLen Lib "kernel32" Alias "lstrlenA" _
```

```
    (ByVal Ptr As Long) As Long
```

Now we can call the functions **EnumPrinters**, **PtrToStr** and **StrLen**; these are not part of VBA naturally, so that is why we had to declare them.

The final task for this is to create a function that will use these calls we declared to get a list of all printers installed on the computer and return a simple array to the user. We do this like so:

VBA

```
Public Function ListPrinters() As Variant

Dim bSuccess As Boolean
Dim iBufferRequired As Long
Dim iBufferSize As Long
Dim iBuffer() As Long
Dim iEntries As Long
Dim iIndex As Long
Dim strPrinterName As String
Dim iDummy As Long
Dim iDriverBuffer() As Long
Dim StrPrinters() As String

iBufferSize = 3072

ReDim iBuffer((iBufferSize \ 4) - 1) As Long

'EnumPrinters will return a value False if the buffer is not big enough
```

```
bSuccess = EnumPrinters(PRINTER_ENUM_CONNECTIONS Or _
        PRINTER_ENUM_LOCAL, vbNullString, _
        1, iBuffer(0), iBufferSize, iBufferRequired, iEntries)

If Not bSuccess Then
    If iBufferRequired > iBufferSize Then
        iBufferSize = iBufferRequired
        Debug.Print "iBuffer too small. Trying again with "; _
        iBufferSize & " bytes."
        ReDim iBuffer(iBufferSize \ 4) As Long
    End If
    'Try again with new buffer
    bSuccess = EnumPrinters(PRINTER_ENUM_CONNECTIONS Or _
            PRINTER_ENUM_LOCAL, vbNullString, _
            1, iBuffer(0), iBufferSize, iBufferRequired, iEntries)
End If

If Not bSuccess Then
    'Enumprinters returned False
    MsgBox "Error enumerating printers."
    Exit Function
Else
    'Enumprinters returned True, use found printers to fill the array
    ReDim StrPrinters(iEntries - 1)
    For iIndex = 0 To iEntries - 1
        'Get the printername
        strPrinterName = Space$(StrLen(iBuffer(iIndex * 4 + 2)))
        iDummy = PtrToStr(strPrinterName, iBuffer(iIndex * 4 + 2))
        StrPrinters(iIndex) = strPrinterName
    Next iIndex
```

```
End If

ListPrinters = StrPrinters

End Function
```

In brief; this function calls the **EnumPrinters** function to get a list of all printers on the system, and store them in an array **StrPrinters**, which it then returns to the caller.

Now we want to populate our **ComboBox** control with a list of printers when the form is first displayed; go to the coding view of the user form, and from the **Object Box** at the top select the **UserForm1** item from the list. Then from the **Procedures Box** select the **Activate** to add a function for when the form activates.

Within this **Activate** function add the following code to call the function we created just and fill the **ComboBox** with the printer list:

VBA

```
Private Sub UserForm_Activate()

Dim StrPrinters As Variant
Dim x As Long

StrPrinters = ListPrinters()
```

```
'Fist check whether the array is filled with anything, by calling another
function, IsBounded.
If IsBounded(StrPrinters) Then
   For x = LBound(StrPrinters) To UBound(StrPrinters)
      ComboBox1.AddItem StrPrinters(x)
   Next x
   ComboBox1.ListIndex = 0
Else
   MsgBox "Failed to get printer list"
End If

End Sub
```

All we do is create a new **Variant** variable to store our printer list and call the **ListPrinters** function. We check that the list is actually a valid array by calling a **IsBounded,** a function yet to be defined. If OK we loop through all items and add them to the **ComboBox** control.

The **IsBounded** function checks if the variable passed in is actually an array by checking that the upper bound of the variable is a numeric value. If the variable is not an array it will not return a value:

VBA

```
Public Function IsBounded(vArray As Variant) As Boolean
   On Error Resume Next
   IsBounded = IsNumeric(UBound(vArray))
End Function
```

Getting sheet names

Now for the easy part, getting the sheet names; back in the **Form Designer,** double-click your "Get Sheets" button to add a **Click** event function. Within this function we are going to place the usual connection and active document code, and then to get all the sheet names of the active document using the **GetSheetNames** function we have used before:

VBA

```
Set swApp = GetObject("", "SldWorks.Application")

If swApp Is Nothing Then
MsgBox "Error gettings SolidWorks Handle"
Exit Sub
End If

Set swModel = swApp.ActiveDoc

If swModel Is Nothing Then
MsgBox "Failed to get active document"
Exit Sub
End If

If swModel.GetType() <> swDocDRAWING Then
MsgBox "Active document must be a drawing"
Exit Sub
End If

Dim swDrwDoc As DrawingDoc
Set swDrwDoc = swModel
```

```
Dim sheetNames As Variant
sheetNames = swModel.GetSheetNames()

ListBox1.Clear

Dim sheetname As Variant
For Each sheetname In sheetNames
   ListBox1.AddItem (sheetname)
Next sheetname

End Sub
```

This should all be familiar enough for you to understand by now.

Now we are at the point where the user can get the sheet names and select a printer, and all that is left to do is to print the selected sheets.

Printing the selection

In order to print the selected sheets we follow much the same process as the .Net example, with some slight differences.

Add a **Click** event function to the "Print!" button by double-clicking it and then go to that function and add the following code to print the selected sheets:

VBA

```
Private Sub CommandButton2_Click()

Dim i As Integer

For i = 0 To ListBox1.ListCount - 1

   If ListBox1.Selected(i) Then
      Dim pages(0) As Long
      pages(0) = i + 1
      swModel.Extension.PrintOut2 pages, 1, False, ComboBox1.Text, ""
   End If
Next

End Sub
```

We start with a **For** loop to loop all of the items within the list. Within the loop we check whether that item is selected, and if it is we create a new variable for the page numbers and call the **PrintOut2** function.

The difference with the VBA call to the **PrintOut2** function is that it doesn't 2 items in the page array when printing a single page, it takes just one. Other than that everything else is the same. Give it a try!

Add-ins

The basics of an Add-in

Removing Add-in entries

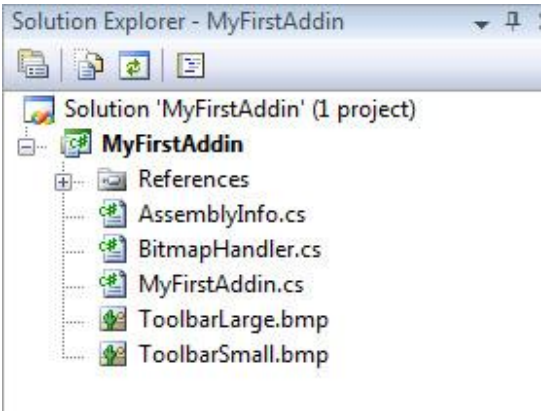
I was stuck with the decision of the last chapter either covering file importing and exporting with Excel, XML and ini files, or a quick brush over of add-ins; my choice was the latter as I feel many of you will find it harder to create an add-in on your own without a guide than you will to work with files as there are plenty of tutorials for you on file input/output. OK, so let's get straight to it.

The basics of an Add-in

As I only intend to brush over add-ins, I will not explain every nitty-gritty detail, but instead focus on the main sections of code you are most likely to need to alter and play with.

Start with the template provided on the accompanying CD in the Chapter 9 folder called **MyFirstAddin**.

When you open this solution file you will notice your project has several items in the **Solution Explorer**:

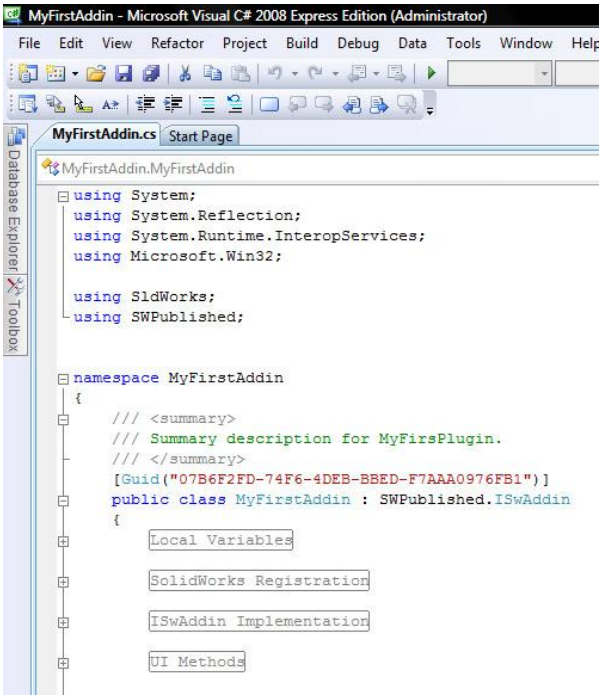


The **AssemblyInfo** is just assembly information in every .Net project, and the **BitmapHandler** is the code to generate our images for the

menus; we will not touch this coding in this example, only use its functions.

The 2 image files are used for our menu icons, and are basically 16x16 icons (small) and 24x24 icons (large) lines up horizontally to create a strip.

The only file remaining is the main add-in code file that we will be using called **MyFirstAddin**; this contains all the important code I am going to explain.



Open the **MyFirstAddin** file and you will see something similar to the image to the left.

Try not to take too much of the code in as it will likely just go right over your head and just confuse the matter, just accept the fact that it does what it does for now, which is to create a SolidWorks add-in with a few menu items. I will cover as much of the important stuff as I can in this brief section.

Start by taking a glance at the **Local Variables** section:

C#

```
ISldWorks iSwApp;
ICommandManager iCmdMgr;
int addinID;
```

These three variables will be used later.

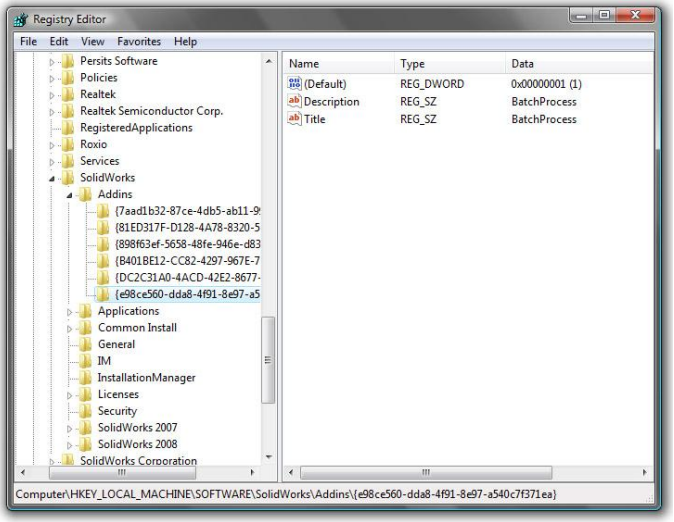
Take a look at the **SolidWorks Registration** section:

C#

```
[ComRegisterFunctionAttribute]
public static void RegisterFunction(Type t)
{
        Microsoft.Win32.RegistryKey hklm =
Microsoft.Win32.Registry.LocalMachine;
        Microsoft.Win32.RegistryKey hkcu =
Microsoft.Win32.Registry.CurrentUser;

        string keyname = "SOFTWARE\\SolidWorks\\Addins\\{" +
t.GUID.ToString() + "}";
        Microsoft.Win32.RegistryKey addinkey =
hklm.CreateSubKey(keyname);
        addinkey.SetValue(null, 0);
        addinkey.SetValue("Description", "Sample Addin ");
        addinkey.SetValue("Title", "MyFirstAddin");

        keyname = "Software\\SolidWorks\\AddInsStartup\\{" +
t.GUID.ToString() + "}";
        addinkey = hkcu.CreateSubKey(keyname);
```

```
            addinkey.SetValue(null, 0);
}

[ComUnregisterFunctionAttribute]
public static void UnregisterFunction(Type t)
{
            //Insert code here.
            Microsoft.Win32.RegistryKey hklm =
Microsoft.Win32.Registry.LocalMachine;
            Microsoft.Win32.RegistryKey hkcu =
Microsoft.Win32.Registry.CurrentUser;

            string keyname = "SOFTWARE\\SolidWorks\\Addins\\{" +
t.GUID.ToString() + "}";
            hklm.DeleteSubKey(keyname);

            keyname = "Software\\SolidWorks\\AddInsStartup\\{" +
t.GUID.ToString() + "}";
            hkcu.DeleteSubKey(keyname);
}
```

All that these two functions do is to add and delete a simple registry entry containing details of your add-in, so that SolidWorks knows where to look to load your file, that's it. An entry looks like this:



Take a look at the **ISwAddin Implementation** section. This section contains the functions that are called when SolidWorks attempts to load your add-in, and unload it once done.

C#

```
public MyFirstAddin()
{
}

public bool ConnectToSW(object ThisSW, int cookie)
{
    iSwApp = (ISldWorks)ThisSW;
    addinID = cookie;

    //Setup callbacks
    iSwApp.SetAddinCallbackInfo(0, this, addinID);
```

```
    #region Setup the Command Manager
    iCmdMgr = iSwApp.GetCommandManager(cookie);
    AddCommandMgr();
    #endregion

    return true;
}

public bool DisconnectFromSW()
{
    RemoveCommandMgr();

    iSwApp = null;
    GC.Collect();
    return true;
}
```

The first function is the class constructor, we do nothing here. The next function, **ConnectToSW**, is the function run when SolidWorks loads your add-in. In this function is where we place our start up coding.

In this example we initialise a new **ICommandManager** object by calling the **GetCommandManager** function, which accepts the add-in **cookie** variable as its parameter. With this **Command Manager** we can now add a command item here by calling the function **AddCommandItem** (defined later).

The **DisconnectFromSW** function is called when SolidWorks is closing, so we gracefully remove any items we added to the SolidWorks interface such as our command menu.

Finally, take a look at the **UI Methods** section. This is the section where our code has been placed to add the command menu:

C#

```
public void AddCommandMgr()
{
    ICommandGroup cmdGroup;
    BitmapHandler iBmp = new BitmapHandler();
    Assembly thisAssembly;

    thisAssembly =
System.Reflection.Assembly.GetAssembly(this.GetType());

    cmdGroup = iCmdMgr.CreateCommandGroup(1, "MyFirstAddin",
"This is my add-in", "", -1);

    cmdGroup.LargeIconList =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarLarge.bmp"
, thisAssembly);
    cmdGroup.SmallIconList =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarSmall.bmp"
, thisAssembly);
    cmdGroup.LargeMainIcon =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarLarge.bmp"
, thisAssembly);
```

```
    cmdGroup.SmallMainIcon =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarSmall.bmp"
, thisAssembly);

    cmdGroup.AddCommandItem2("Start my external program", 0, "Click
me", "Click me", 0, "StartExternalProgram", "", 0,
(int)(SwConst.swCommandItemType_e.swMenuItem |
SwConst.swCommandItemType_e.swToolbarItem));

    cmdGroup.Activate();
}

public void RemoveCommandMgr()
{
    iCmdMgr.RemoveCommandGroup(1);
}
```

We start by declaring 3 variables; the **iCommandGroup** is the main variable we are interested in, which is used to store a command group we create. A **Command Manager** is like a menu or a toolbar manager, only it is actually both. By creating a command manager you can tell it to create a "group", which acts as a menu and/or a toolbar within SolidWorks.

The **BitmapHandler** is a handler to convert the images we have in our **Solution Explorer** into workable icons for the menu and toolbars.

The **Assembly** is what is called a "Reflection object"; we use it to get a handle to a running .Net assembly. In this case we are getting a

handle to ourselves (the assembly itself), so that we can access the images embedded within it (i.e. the images in our **Solution Explorer**).

```
thisAssembly =
System.Reflection.Assembly.GetAssembly(this.GetType(
));
```

This line is used to get the handle to ourself by using the keyword **this**.

The end goal here is to create a few menu items, but before we can have a menu item, we need a menu to store them in. As we are using a **Command Manager** we do not add a menu, but a group. This group will act as a menu as well as a toolbar if you choose to have one.

Adding a Command Group

The function we use to add this Command Group is called **CreateCommandGroup**, of an **ICommandManager** object.

```
LpGroup = CommandManager.CreateCommandGroup (
UserID, Title, Tooltip, Hint, Position)
```

> *Like when we created a Property Manager Page control, the **UserID** is just a unique number specified by the user. This number only has to be unique for this add-in, not unique to all of the SolidWorks add-ins, so it is up to the user to make sure each call to **CreateCommandGroup** uses a different value.*

*The **Title** is the name that will appear on the menu or toolbar. For example, the **Title** values for the standard SolidWorks menus are File, View, Tools, Help.*

*The **Tooltip** is the little tip that pops up in a yellow box when the user hovers over the item.*

*The **Hint** is the same thing as the **Tooltip** only this text appears in the SolidWorks status bar at the bottom of SolidWorks when the user hovers over the item.*

The Position is the position that your menu will appear in the main menu, stating at 0 for the beginning.

To create our example group we use the following code:

```
cmdGroup = iCmdMgr.CreateCommandGroup(1,
"MyFirstAddin", "This is my add-in", "", -1);
```

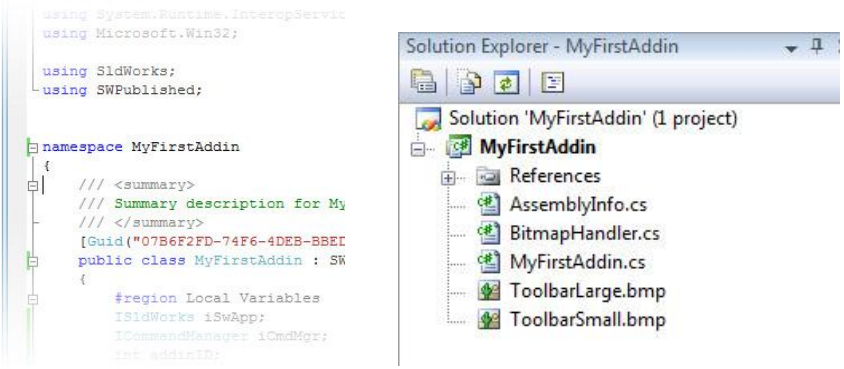
As you can see, the unique ID we assign to our group is just the number 1. We call our group "MyFirstAddin", and the tooltip as "This is my add-in", and nothing for the hint. We position it using -1 so it gets positioned by default at the end of the menu items.

Before we continue to add command items we want to add some icons ready for use with our items. To do this we use the **BitmapHandler** object and set the 4 icon properties of the **ICommandManager** object using the following lines:

C#

```
cmdGroup.LargeIconList =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarLarge.bmp"
, thisAssembly);
cmdGroup.SmallIconList =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarSmall.bmp"
, thisAssembly);
cmdGroup.LargeMainIcon =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarLarge.bmp"
, thisAssembly);
cmdGroup.SmallMainIcon =
iBmp.CreateFileFromResourceBitmap("MyFirstAddin.ToolbarSmall.bmp"
, thisAssembly);
```

The first parameter that we pass in is the exact name of the namespace that our code is within, plus the name of the image in the **Solution Explorer,** separated by a period (**.**):



Adding a Command Item

Now we have a **Command Group** and some icons to use, we are ready to add a Command Item. By adding this item it will automatically add the menu item in the add-in. For this we call the function **AddCommandItem2**:

```
*CmdIndex = CommandGroup.AddCommandItem2 ( Name,
Position, HintString, ToolTip, ImageListIndex,
CallbackFunction, EnableMethod, UserID,
MenuTBOption)
```

*The **Name** is the name that will appear in the menu or toolbar, such as the menu item of the File group for opening a file is called "Open...".*

*The **Position** is again the position of item, this time in the group; zero-based index.*

*The **HintString** is the hint that appears in the SolidWorks status bar.*

*The **ToolTip** is the tip that appears in the yellow box when the user hovers over the item.*

*The **ImageListIndex** is the position within the image list you added in the previous step, starting from 0.*

*The **CallbackFunction** is the exact case-sensitive name of the function within your add-in to actually run when the user clicks this item. You will see this in use in a moment.*

*The **EnableMethod** is again the name of a function to call, but this time this function is called before the item gets displayed. It specified whether to enable or disable the item.*

*The **UserID** is the unique identifier if this item for this group. It only has to be unique for this group and is not needed; to ignore it pass **o**.*

*The **MenuTBOption** is an enumerator value for selecting where to add the item; to the menu, to the toolbar, or both.*

The function returns the index position the menu item has been placed in the group.

For this example here is our call to this function:

```
cmdGroup.AddCommandItem2("Start my external
program", 0, "Click me", "Click me", 0,
"StartExternalProgram", "", 0,
(int)(SwConst.swCommandItemType_e.swMenuItem |
SwConst.swCommandItemType_e.swToolbarItem));
```

We call the item "Start my external program", and ignore any specific positioning by passing in o. We tell the item to run a function called **StartExternalProgram** from our add-in code, which we will create later, and we ignore the **EnableMethod** function by passing an empty string. We set the **ImageListIndex** to o so it uses the first icon from our lists that we loaded in previously. The two image strips look like this, so the index o will use the left-most icon:

We ignore the **UserID** as well by passing in **0,** and we tell the item to appear in both the toolbar and menu controls.

Now with our menu item created all that is left is to activate the group.

```
cmdGroup.Activate();
```

The final step is to create a function called **StartExternalProgram** that we told our command item to call when the item is clicked. Within this function you can place any code you like, including all of the code we have covered in this book. A good test for you here would be to place the .Net Property Manager Page code into the add-in so you can handle event call-backs properly.

For this example however I will be creating the highly requested code for running an external program or SolidWorks macro. The code is 2 short and sweet lines:

C#

```
public void StartExternalProgram()
{
    System.Diagnostics.Process.Start("notepad.exe");
    iSwApp.RunMacro(@"C:\mymacro.swp", "Macro1", "main");
}
```

The first line calls the function **Start,** from the **System.Diagnositcs.Process** class; this function is the same as writing something in the **Start->Run…** dialog box. The usual procedure is to place the full address/location of the file/program you

wish to start. Because Notepad is within a system folder it is found automatically without the need for its full location.

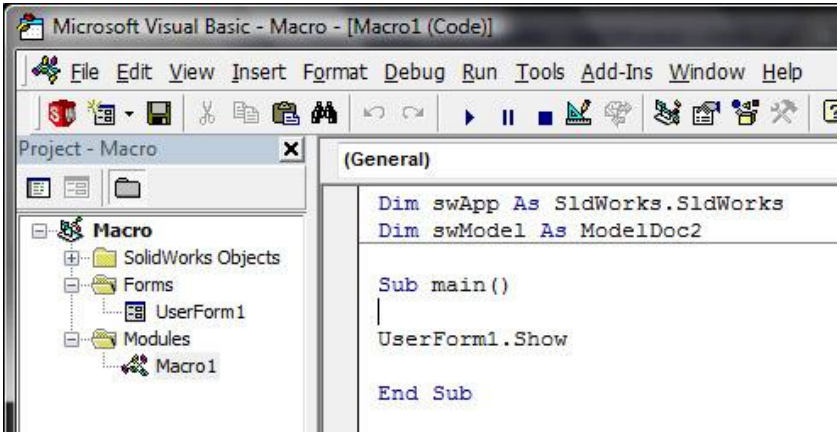
The second line runs a SolidWorks macro; the function is as follows:

```
retval = SldWorks.RunMacro ( filePathName,
moduleName, procedureName )
```

*The **filePathName** is the full location and filename of the macro to run such as C:\My Stuff\My macro.swp".*

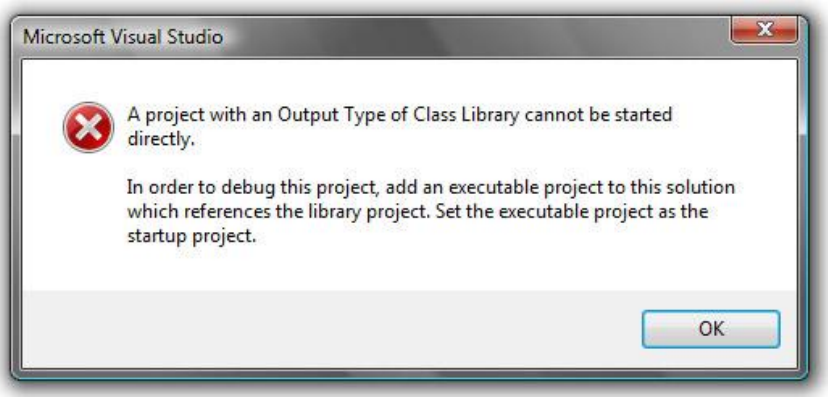
*The **moduleName** is the name of the module code file that contains the function you want to run. By default when creating macros from SolidWorks this code file is called **Macro1**.*

*The **prodecureName** is the function name within the selected module code file that you want to run. By default you would run the **main** function.*



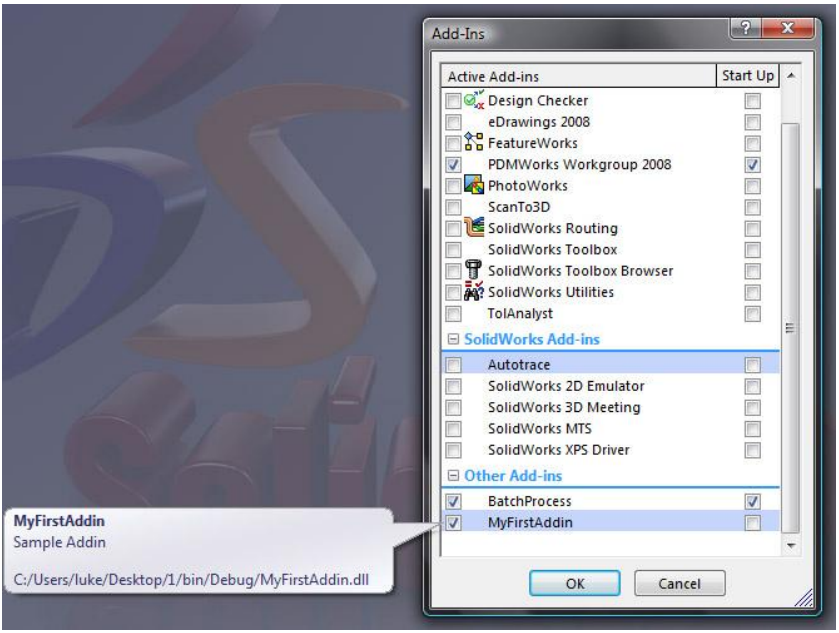
In our example we use a macro that looks like the picture above, and as you can we passed in "Macro1" as the module name, and "main" as the function.

Now compile your program making sure that SolidWorks is shut down and upon successfully building your project VS will inform you that your project cannot be run directly. Do not worry, your project has already run the **SolidWorks Registration** section code and added itself to the windows registry ready for SolidWorks to find on next load.



Open up SolidWorks and go to **Tools->Add-ins**. In there you should notice your new add-in under the name you defined in the **RegisterFunction** code:

```
public static void RegisterFunction(Type t)
{
...
   addinkey.SetValue("Description", "Sample Addin");
   addinkey.SetValue("Title", "MyFirstAddin");
...
}
```



As you can see the tooltip shows the add-in file location (which is our project output folder location), the title and description we set.

Check the box to the left to load our add-in and click OK.

You will now notice that you have a new menu group and item, named by the name that we gave it in our code, as well as a toolbar that you can select to show as well:

To get the toolbar to display you have to right-click on any area of the SolidWorks background with nothing open, or on the top or bottom toolbars if you have files open to display the menu

containing all toolbars. In that menu you will see your new add-in. Select it to show it:

Try clicking your button or menu item and see how your function **StartExternalProgram** will run, and effectively open notepad and run the macro located in the location you entered.

That is it for our brief overview of add-ins; hopefully you have learned enough to create some basic menu items, the rest of the coding is not really to do with add-ins but doing what you actually want your program to do.

One final note; every add-in you make must have a unique GUID code at the top:

```
[Guid("07B6F2FD-74F6-4DEB-BBED-F7AAA0976FB1")]
public class MyFirstAddin : SWPublished.ISwAddin
```

In order to generate a new code run the project supplied on the CD in Chapter 9, called **GUIDCreator**.

Removing Add-in entries

Once you have created an add-in if you wish to delete it all you have to do is go to **Start->Run...** and type "regedit" without quotes and press enter. This will open the Registry Editor. From there go to

```
HKEY_LOCAL_MACHINE\SOFTWARE\SolidWorks\Addins
```

Within this entry is all of the add-ins of SolidWorks, including your own. Just look for your GUID id in the list, or click each and look at the descriptions to the right. Once found, delete the entire folder.