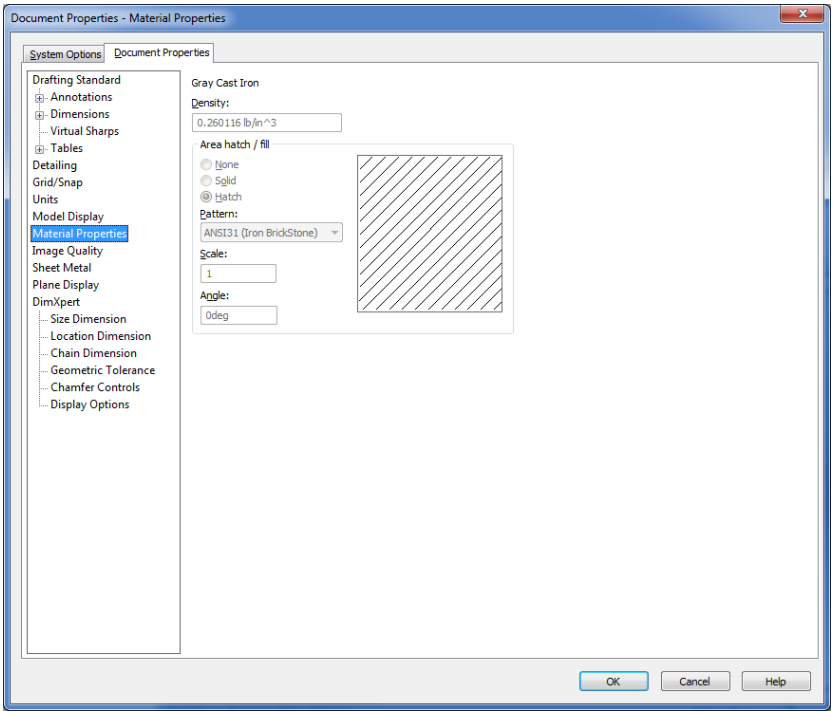
Automating SolidWorks 2011 Using Macros

A Visual Studio Tools for Applications tutorial for SolidWorks users

Using the Visual Basic.NET Language

Written by Mike Spens

Material Properties



- **Basic Material Properties**
- **Adding Forms**
- **Arrays**
- **Working with Assemblies**
- **Selection Manager**
- **Verification and Error Handling**

Introduction

This exercise is designed to go through the process of changing settings for materials. It will also review several standard Visual Basic.NET programming methods. It will also examine a method for parsing through an assembly tree to change some or all of the parts in the assembly.

As an additional preface to this chapter, remember that SolidWorks sometimes does things better than your macros might. For example, since SolidWorks 2009, you can select multiple parts at the assembly level and set their materials in one shot. You used to have to edit materials one part at a time. This chapter was originally written before you could set materials so easily. As a result, you should use this chapter as a means to better understanding some of the tools available through Visual Basic.NET and the SolidWorks API rather than as a handy tool that SolidWorks does not provide already in the software. No matter how clever you get with your macros, at some point someone else might come up with the same idea. In the perfect world, you could obsolete your macros as SolidWorks adds the functionality to the core software!

Part 1: Basic Material Properties

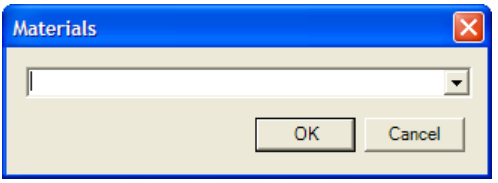
The initial goal will be to make a tool that allows the user to select a material from a pull-down list or combo box. When he clicks an OK button, the macro should apply the appropriate material to the active part. The user should be able to cancel the operation without applying a material.

We could take the approach of recording the initial macro, but the code for changing materials is simple enough that we will build it from scratch in this example.

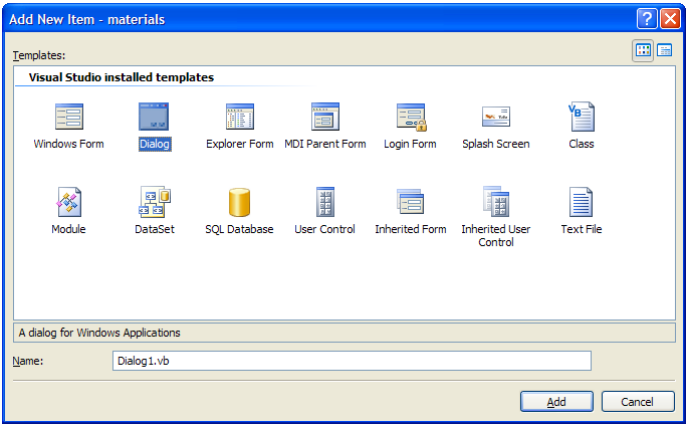
User Forms

Through the first chapters, two different methods for gathering user input have been introduced. One method was the input box.

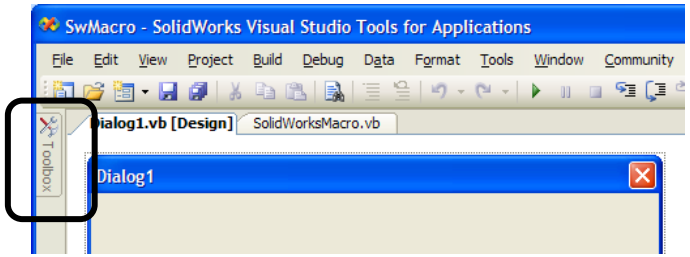
That is fine for simple text input. The other method was Microsoft Excel. This improved things by adding command buttons and multiple input values. Many times it is better to be able to organize user input in a custom dialog box or form. This is a standard with practically any Windows software. These allow users to input text, click buttons to activate procedures, and select options. After this example you will have created a form that looks like the one shown using a drop down list or ComboBox and two buttons.



1. Add a form to your macro by selecting **Project, Add Windows Form**.
2. Choose the **Dialog** template and click **Add**.



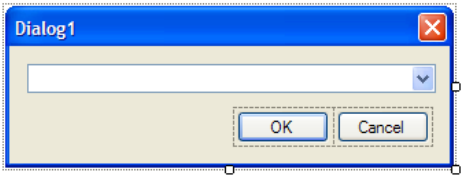
A new form will be added to your project named *Dialog1.vb* and will be opened for editing. The Dialog template has two Button controls for OK and Cancel already pre-defined. In order to add additional controls to the form you will need to access the controls Toolbox from the left side of the VSTA interface. It is a collapsed tab found immediately to the left of the newly created dialog form.



A **ComboBox** control must be added to the form.

3. Click on the **Toolbox** and optionally select the pushpin to keep it visible as you build your form.
4. Drag and drop the **ComboBox** control from the Toolbox onto your form.

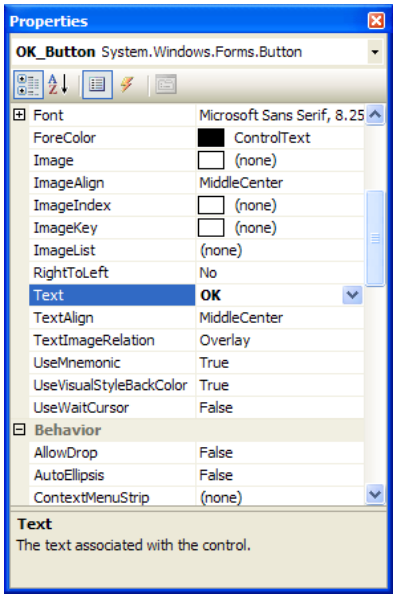
After adding the ComboBox and resizing, your form should look something like the following. An effective form is one that is compact enough to not be intrusive while still being easy to read and use.



Object Properties

Each of these controls has properties that you can change to affect its visual display as well as its behavior.

5. Select the **OK** button. The Properties window will show all of the control's properties. This is the same general idea as the SolidWorks Property Manager.



6. **Review** the following properties that were defined by the use of the Dialog template.

   - Text = OK. This is the text that is visible to the user. Use an ampersand (&) before a character to assign the Alt-key shortcut for the control.

   - (Name) = OK_Button. This name is what your code must reference to respond to the button or to change its properties while your macro is running.

7. Select the Cancel button and review its Text and Name properties as well.

8. Select the dialog itself from the designer, not the Project Explorer, and change its Text property to “Materials”.

9. Review the following properties of the dialog form that were set by using the Dialog template.

- AcceptButton = OK_Button
- CancelButton = Cancel_Button

10. Select the **ComboBox** and set its Name to "Materials_Combo".

Showing the Dialog

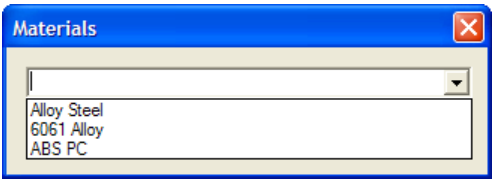
The macro needs a line of code that will display the form at the right time. If you run the macro right now, it will not do anything since the main procedure is empty.

11. Add the following code in your main procedure as follows.

```
Sub main()

    'Initialize the dialog
    Dim MyMaterials As New Dialog1
    Dim MyCombo As Windows.Forms.ComboBox
    MyCombo = MyMaterials.Materials_Combo

    'Set up materials list
    Dim MyProps(2) As String

    MyProps(0) = "Alloy Steel"
    MyProps(1) = "6061 Alloy"
    MyProps(2) = "ABS PC"

    MyCombo.Items.AddRange(MyProps)

    MyMaterials.ShowDialog()
End Sub
```

12. **Run** the macro as a test.

You should see your new dialog box show up. This is a result of **MyMaterials.ShowDialog()** at the bottom of the code. Every user form has a ShowDialog method that makes the form visible to the user and returns the user's action. We will make use of the return value shortly.



If you click on the combo box, you should see Alloy Steel, 6061 Alloy and ABS PC listed.

13. **Close** the running macro and return to the VSTA interface.

There are a few steps required to make a form or dialog visible. The first of which is to declare a variable named `MyDialog` as a new instance of the `Dialog1` class. Even though you have created a dialog in the project, it is not created or used at run time until you use it. It is important to note that the name of the class does not always match the name of the file as it does in this example.

14. **Review** the code behind Dialog1.vb by **right-clicking** on it in the Project Explorer and selecting **View Code**.

```
Imports System.Windows.Forms

Public Class Dialog1

    Private Sub OK_Button_Click(ByVal sender …
        [Additional code here]

    End Sub

End Class
```

Notice that the code in the form itself is declared as a public class named `Dialog1`. You could change the name of the class without changing the name of the vb code file itself. In fact, a single code file can contain as many classes as you want, although it makes it a little more difficult to manage in the long run.

15. Switch back to the **SolidWorksMacro.vb tab** to return to the main procedure.

The first time you use any class, whether it be a separate code module or a dialog, you must typically use the `New` keyword before you can reference it. This is distinctly different than Visual Basic 6. It essentially created new instances of forms and dialogs if you ever referenced them. It may seem that the .NET method of referencing forms is a little more verbose, but it has some real benefits as we will see a little further on.

Windows.Forms Namespace

A variable named `MyCombo` was also declared as `Windows.Forms.ComboBox` and was set to the `Materials_Combo` control from the instance of the form named `MyMaterials`. This reference should make it apparent as to what namespace or library a control comes from. The ComboBox class is a child of the Forms namespace which is a child of the Windows namespace. To add another level of complexity, the Windows namespace is a member of the System namespace which has already been referenced by the Imports statement at the top of the code window. I personally still like to think of namespaces as libraries. I am sure someone had a good reason for naming them namespaces, but I still do not think I completely understand the reason. If you think of a namespace as a library of classes, as I do, it may be easier to think of what they are all about.

Since the macro will reference several components from the Windows.Forms namespace, it will make the code less wordy to import that namespace.

16. **Add** the following Imports statement to the top of the code window to reference the necessary namespace. Notice that this is the same imports statement as was used in the *Dialog1.vb* code window.

```
Imports SolidWorks.Interop.sldworks
Imports SolidWorks.Interop.swconst
Imports System
Imports System.Windows.Forms
```

Now the declaration of `MyCombo` can be simplified as follows.

```
Dim MyCombo As ComboBox
```

Now that there is a reference to the ComboBox control, it is populated with an array of values.

Arrays

An array is simply list of values. To declare an array you use the form **Dim variablename(x) As type**.

`MyProps(2)` was declared as a string type variable. In other words, you made room three rows of text in that one variable. "Wait! I thought you declared two rows"! If you have not learned this already, arrays count from zero. If you stick to this practice, you will avoid confusion in most cases.

ComboBox.Items.AddRange Method

To populate the combo box with the array, you must tell the macro where to put things. By typing `MyCombo.Items.AddRange(MyProps)` you have told the procedure that you want to populate the items (or list) of the MyCombo control with the values in the MyProps array by using the AddRange method. The ComboBox control automatically creates a row for each row in the array. If you wanted to add items one at a time rather than en masse, you could use the Add method of the Items property.

DialogResult

Once a user has selected the desired material from the drop down, this material should be applied to the active part if he/she clicks OK. However, if the user clicks Cancel, we would expect the macro to close without doing anything. At this point, either button simply continues running the remaining code in the main procedure – which is nothing.

17. **Modify** the main procedure as follows to add processing of the **DialogResult**.

```
...
MyProps(0) = "Alloy Steel"
MyProps(1) = "6061 Alloy"
MyProps(2) = "ABS PC"
MyCombo.Items.AddRange(MyProps)

Dim Result As DialogResult
Result = MyMaterials.ShowDialog()

If Result = DialogResult.OK Then
    'Assign the material to the part

End If
End Sub
```

The ShowDialog method of a form will return a value from the System.Windows.Forms.DialogResult enumeration. Since we have used the `Imports System.Windos.Forms` statement in this code window, the code can be simplified by declaring `Result` as `DialogResult`. You probably noticed that when you typed "If Result = ", IntelliSense immediately gave you the logical choices for all typical dialog results.

As a result of the If statement, if the user chooses Cancel, the main procedure will simply end.

Setting Part Materials

Now you will finally get your macro to do something with SolidWorks. The next step will be to set the material based on the material name chosen in the drop down.

18. Add the code inside the If statement to set material properties as follows.

```
If Result = DialogResult.OK Then
    'Assign the material to the part
    Dim Part As PartDoc = Nothing
    Part = swApp.ActiveDoc
```

```
    Part.SetMaterialPropertyName2("Default", _
    "SolidWorks Materials.sldmat", MyCombo.Text)
End If
```

IPartDoc Interface

The first thing you might have noticed is the different way that Part was declared. It was declared as `PartDoc` rather than `ModelDoc2` as was done in the previous macros. This part gets a little more complicated to explain. Think of ModelDoc2 as a container that can be used for general SolidWorks file references. It can be a part, an assembly or a drawing. There are many operations that are standard across all file types in SolidWorks such as adding a sketch, printing and saving. However, there are some operations that are specific to a file type. Material settings, for example, are only applied at the part level. Mates are only added at the assembly level. Views are only added to drawings. Since we are accessing a function of a part, the PartDoc interface is the appropriate reference. The challenging part is that the ActiveDoc method returns a ModelDoc2 object which can be a PartDoc, an AssemblyDoc or a DrawingDoc. They are somewhat interchangeable. However, it is good practice to be explicit when you are trying to call a function that is unique to the file type. Being explicit also enables the correct IntelliSense information so it is easier to code.

IPartDoc.SetMaterialPropertyName2 Method

The simplest way to set material property settings is using the SolidWorks materials. SetMaterialPropertyName2 is a method of the IPartDoc interface and sets the material by name based on configuration and the specified database.

Dim instance As IPartDoc
Dim ConfigName As String
Dim Database As String
Dim Name As String

instance.SetMaterialPropertyName2(ConfigName, Database, Name)

- **ConfigName** is the name of the configuration for which to set the material properties. Pass the name of a specific configuration as a string or use "" (an empty string) if you wish to set the material for the active configuration.

- **Database** is the path to the material database to use, such as *SolidWorks Materials.sldmat*. If you enter "" (an empty string), it uses the default SolidWorks material library.

- **Name** is the name of the material as it displays in the material library. If you misspell the material, it will not apply any material.

At this point the macro is fully functional. Try it out on any part. It will not work on assemblies at this point.

Part 2: Working with Assemblies

You can now extend the functionality of this macro to assemblies. When completed, you will have a tool that allows the user to assign material properties to selected components in an assembly. After all, it is usually when trying to figure out the mass of an assembly when you realize that you have forgotten to set the mass properties for your parts.

Is the Active Document an Assembly?

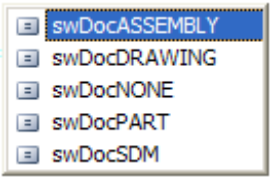
To make this code universal for parts or assemblies, we need to filter what needs to happen. If the active document is an assembly, we need to do something to the selected components. If it is a part, we simply run the code we already have.

19. Add the following If statement structure to check the type of active document. The previous material settings are now inside this If statement (not bold).

```
If Result = DialogResult.OK Then
  Dim Model As ModelDoc2 = swApp.ActiveDoc
  If Model.GetType = swDocumentTypes_e.swDocPART Then
      'Assign the material to the part
      Dim Part As PartDoc = Model
      'Part = swApp.ActiveDoc
      Part.SetMaterialPropertyName2("Default", _
         "SolidWorks Materials.sldmat", MyCombo.Text)
  ElseIf Model.GetType = swDocumentTypes_e._
    swDocASSEMBLY Then
      Dim Assy As AssemblyDoc = Model
    'set materials on selected components

  End If
End If
```

It is important to pay attention to the interchange between ModelDoc2, PartDoc and AssemblyDoc. Notice the simplification of the declaration of `Model`. Rather than initializing the variable to `Nothing` as was done in previous examples, it is initialized directly to `swApp.ActiveDoc`. This is simply a shorthand way to accomplish a declaration and the variable's initial value. When `Part` and `Assy` are declared, they are initialized to Model which is still a reference to the active document. However, since they are declared explicitly as PartDoc and AssemblyDoc, they inherit the document type specific capabilities of parts and assemblies.

Also, notice the use of the **ModelDoc.GetType** method. GetType is used to return the type of ModelDoc that is currently active. This test is important before attempting to deal with specific PartDoc and AssemblyDoc methods. For example, if you use the general ModelDoc2 declaration and attach to the active document, and it is a part, any attempt to call an assembly API like AddMate will cause an exception or crash. There is an enumeration, named **swDocumentTypes_e**, of document types that you have used when testing GetType. When you typed in the code, you should have noticed the different document types show up in the IntelliSense pop-up.



Selection Manager

In the Model Parameters exercise we discussed selections using the SelectByID2 method. However, the code had to be specific to the item being selected. We had to pass the name of the component or a selection location. To get around those limitations you can employ a method that is similar to several functions in SolidWorks. You can require the user to pre-select the components he wishes to change prior to running the macro. The only trick is to program your macro to change the settings for each item the user selects. The selection manager interface will be the key tool for this part of the macro.

ISelectionMgr Interface

Connecting to the selection manager is similar to getting the PartDoc interface (called `Part`). The selection manager is a child of the ModelDoc2 interface.

20. Add the following code inside the assembly section of the If statement to declare the selection manager and to attach to it.

```
ElseIf Model.GetType = swDocumentTypes_e.Then Then
   Dim Assy As AssemblyDoc = Model
   'set materials on selected components
   Dim SelMgr As SelectionMgr
   SelMgr = Model.SelectionManager

End If
...
```

Some of the things you can access from the selection manager interface are the selected object **count**, **type**, or even the **xyz point** where the object was selected in space. In this macro you will

need to access the selected object count or number of items selected, and get access to the components that were selected. Remember that components in SolidWorks can be either parts or assemblies. Since we can only set density for parts, we will need to make sure the item selected is a part. For each item in the selection manager, you must get the ModelDoc2 interface and then set its material settings. You cannot set material settings at the component level in the API.

21. Add the following code to set the material to all selected components.

```
ElseIf Model.GetType = swDocumentTypes_e.swDocASSEMBLY Then
  Dim Assy As AssemblyDoc = Model
  'set materials on selected components
  Dim SelMgr As SelectionMgr
  SelMgr = Model.SelectionManager

  Dim Comp As Component2
  Dim compModel As ModelDoc2
  For i As Integer = 1 To _
  SelMgr.GetSelectedObjectCount2(-1)
    Comp = SelMgr.GetSelectedObjectsComponent3(i, -1)
    compModel = Comp.GetModelDoc2
    If compModel.GetType = swDocumentTypes_e.swDocPART Then
      compModel.SetMaterialPropertyName2("Default", _
        "SolidWorks Materials.sldmat", MyCombo.Text)
    End If
  Next
End If
...
```

For … Next Statements and Loops

As was mentioned earlier, you want to set the material properties for each part that was selected by the user. What if the user has selected 500 parts? You certainly do not want to write 500 lines of code for each item selected. In many cases you will want to apply the same action to a variable number of items.

For … Next statements allow you to repeat a section of code over as many iterations as you want. Computers are great at this! You

just have to know how many times to loop through the code if you use a For … Next statement.

```
For I As Integer = 0 To 10
    MsgBox("You have clicked OK " & I & " times!")
Next I
```

Add this sample code to a procedure and then run the procedure. You get a MessageBox stating exactly how many times you have clicked OK. That is great if you know exactly how many times the loop needs to process. In the macro, you do not know how many times to repeat the loop because you do not know how many parts the user might select. So you can make the procedure figure it out for you. You can use the SelectionManager object to help.

ISelectionMgr.GetSelectedObjectCount2

You need to determine how many times to run through the loop. This should correspond to the number of items selected. This number can be extracted from the selection manager through its **GetSelectedObjectCount2** method. The argument passed is related to a selection Mark. A value of -1 indicates all selections will be counted, regardless of Mark. See the API Help for more information on marks.

```
For i = 1 To SelMgr.GetSelectedObjectCount2(-1)
    '(loop code here)
Next i
```

The For loop starts with an initial value of 1 so that it will only loop if the number of selected items is greater than zero.

GetSelectedObjectsComponent3

The next thing you need to do is get the ModelDoc2 object for each of the selected items. You will need this object so you can set its material properties. This requires a two-step process. The first thing you need to access will be the Component object. To get the component object you use the selection manager's **GetSelectedObjectsComponent3(*item number, Mark*)** method. The Mark argument is again -1 to get the component regardless of selection Mark. You must use the Component interface to access its underlying ModelDoc2 interface. Notice the declarations for `compModel` and `Comp`. They are specific to the type of SolidWorks object we are accessing.

GetModelDoc2

The **Component.GetModelDoc2** method allows you to access the underlying ModelDoc interface of the component.

Now that you have the model, you can use the same code from the part section to set the material properties.

Component vs. ModelDoc

If you have been wondering why we have to take the extra time to dig down to the ModelDoc2 interface of the Component interface, this discussion is for you. If you have not and it all makes perfect sense, move on to the next subject.

Think of it this way – a Component interface understands specific information about the ModelDoc2 it contains. It knows which configuration is showing, which instance it is, if it is hidden or suppressed and even the component's rotational and translational position in the assembly. These are all things that can be changed in the Component interface. However, if you want to change something specific to the underlying part such as its material density, or to the underlying assembly such as its custom properties, then you must take the extra step of getting to the actual ModelDoc2 interface.

Verification and Error Handling

At times you may want to check the user's interactions to make sure they have done what you expected. After all, your macro may not have a user's guide. And even if it does, how many people really read that stuff? If you're reading this, you probably would. What about the other 90% of the population?

You can make sure the user is doing what you think they should. First, define some criteria.

- Is the user in an assembly? The user must be in an assembly to use the **GetSelectedObjectsComponent3** method.

- If the active document is an assembly, has the user pre-selected at least one part? If not, they will think they are applying material properties while nothing happens.

- Has the user selected items other than parts? If they select a plane, the macro may generate an exception or crash because there is no ModelDoc2 interface.

- Does the user even have a file open in SolidWorks?

The only conditions we have not yet added error handling for are the number of selections and if there is an active document.

21. Add the following to check for an active document.

```
Dim Model As ModelDoc2 = swApp.ActiveDoc
If Model Is Nothing Then
  MsgBox("You must first open a file.", _
    MsgBoxStyle.Exclamation)
  Exit Sub
End If
```

22. Add the following to verify that the user has selected something in an assembly.

```
...
If SelMgr.GetSelectedObjectCount2(-1) < 1 Then
  MsgBox("You must select at least one component.", _
    MsgBoxStyle.Exclamation)
  Exit Sub
End If
For i As Integer = 1 To SelMgr.GetSelectedObjectCount2(-1)
  Comp = SelMgr.GetSelectedObjectsComponent3(i, -1)
...
```

If … Then…Else Statements

If the active document is an assembly, you should check if the user has selected at least one component before continuing. Checking the GetSelectedObjectCount2 method of the selection manager for a value less than one will accomplish this. If it is less than one the user has failed to select anything.

MessageBox

You can make use of the Visual Basic **MessageBox** function to give the user feedback. The MessageBox function allows you to tell the user anything in a small dialog box. This dialog box has an OK button by default, but it can have Yes and No buttons, OK and Cancel, or other combinations. If you use anything besides the default you can use the return value to determine which button the user selected.

The macro will now provide feedback to the user to make sure they are using the macro correctly. It makes good programming sense to build good error handling into your macros. Users tend to quickly get frustrated when a tool crashes or generates undesired results.

Conclusion

Even though this tool itself is redundant to SolidWorks capabilities, this procedure can easily be extended to changing or listing any setting related to a model or parts in an assembly.