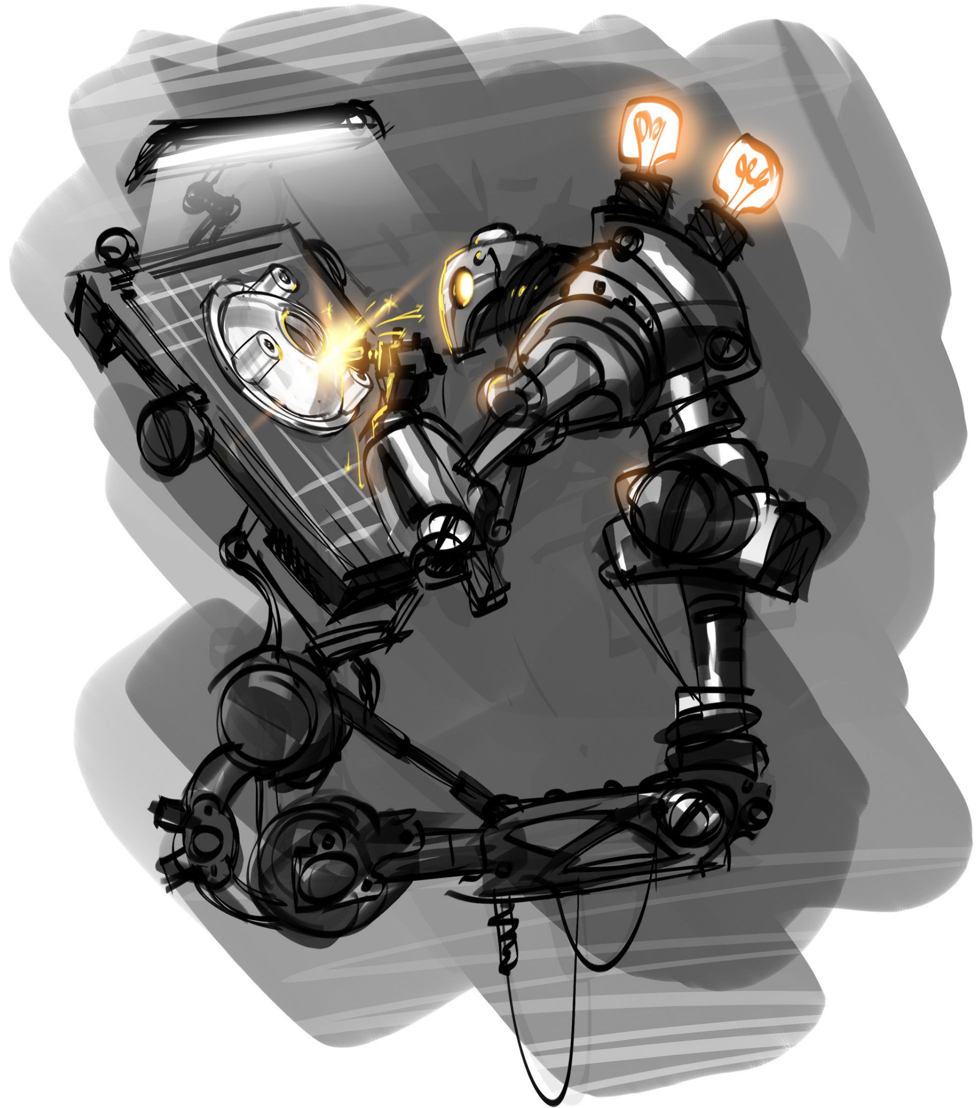
# Automating SOLIDWORKS® 2017 Using Macros

A guide to creating VSTA macros using the Visual Basic.NET Language



Mike Spens

SDC PUBLICATIONS
Better Textbooks. Lower Prices.
www.SDCpublications.com

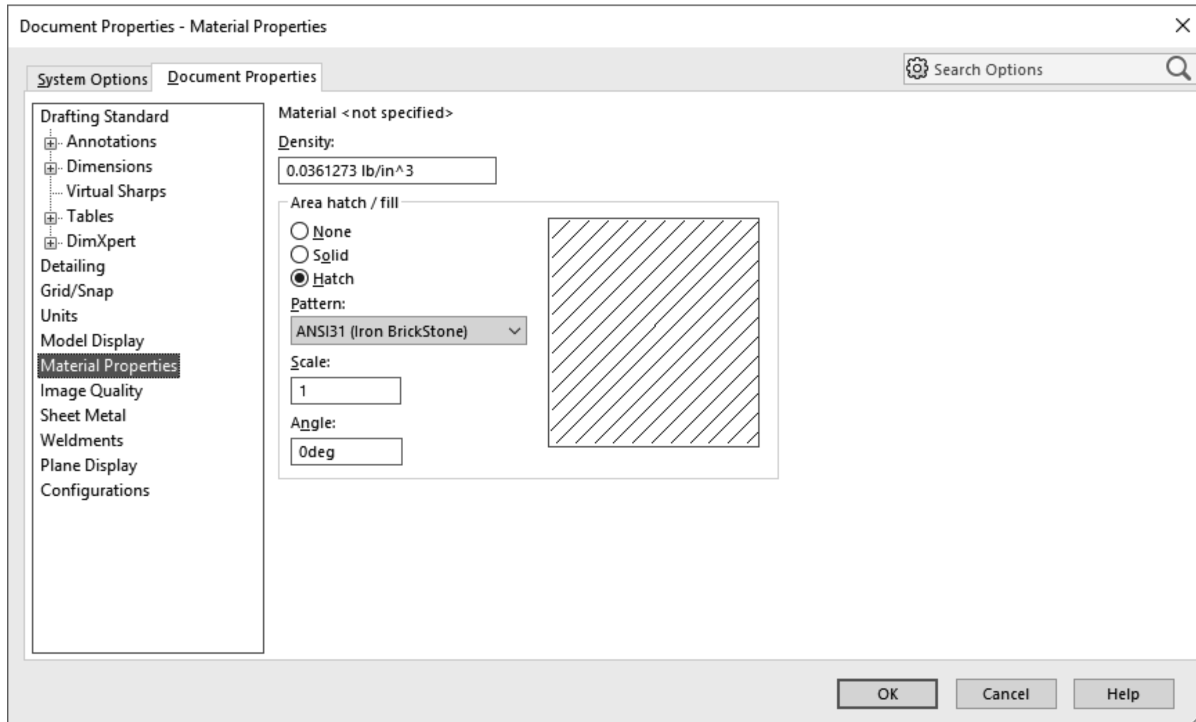
Visit the following websites to learn more about this book:

amazon.com

Google books

BARNES & NOBLE

# 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, you can select multiple parts at the assembly level and set their materials in one shot. 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. 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 should obsolete your macros as SOLIDWORKS adds the functionality to the core software.

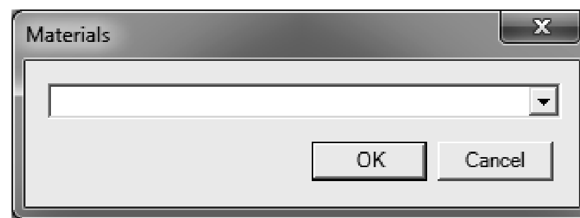
## Part 1: Basic Material Properties

The first step 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 selected material to the active part. The user should be able to cancel the operation as well.

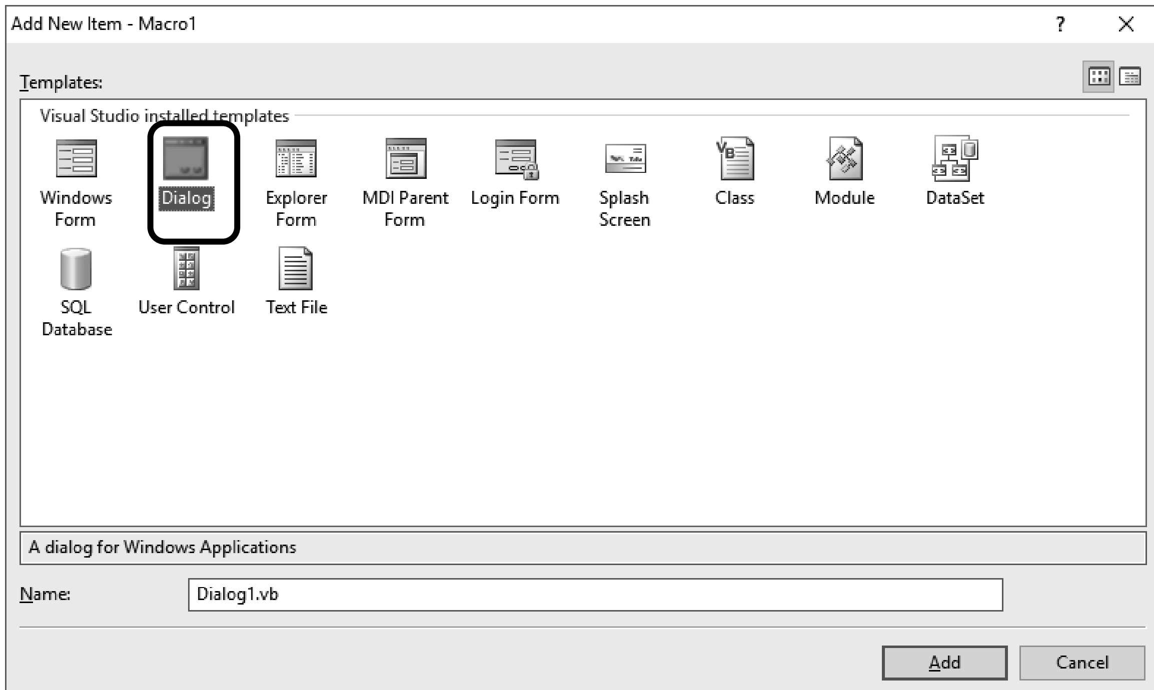
We could take the approach of recording the initial macro, but the code for changing materials is simple enough that we will build it from scratch in this example.

## User Forms

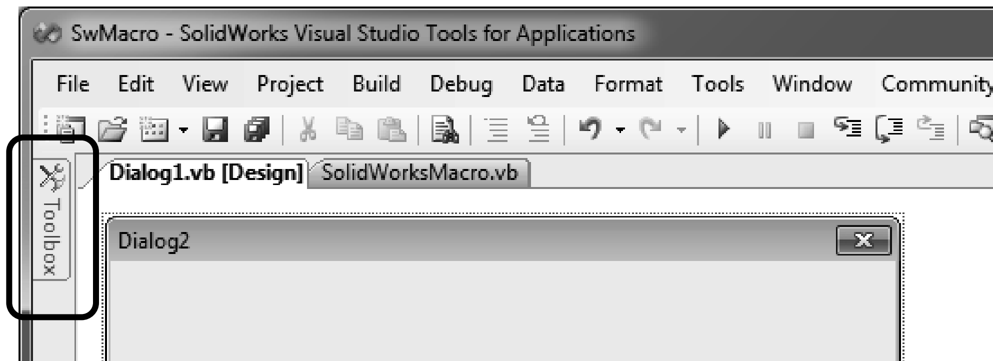
Two different methods for gathering user input have already been introduced. The first was the input box. That's fine for simple text input. The second was using Microsoft Excel, enabling creation of command buttons and multiple input values in cells. More commonly, developers organize user input in a custom dialog box or form. These allow users to input text, click buttons to activate procedures, and select options. This example builds a form that looks like the one shown using a drop down list or ComboBox and two buttons.



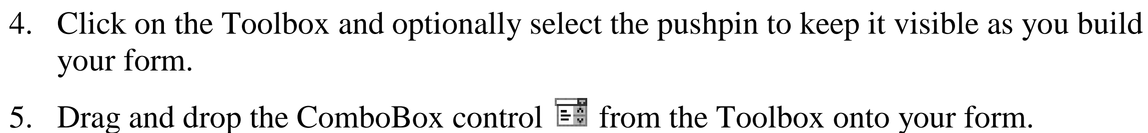

1. Start a new VSTA macro and save it as *materials.vbproj*.
2. Add a form to your macro by selecting Project, Add Windows Form.
3. Choose the Visual Studio Dialog template and click Add.




A new form will be added to your project named *Dialog1.vb* and will be opened for editing. The Dialog template has two pre-defined Button controls for OK and Cancel. 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.

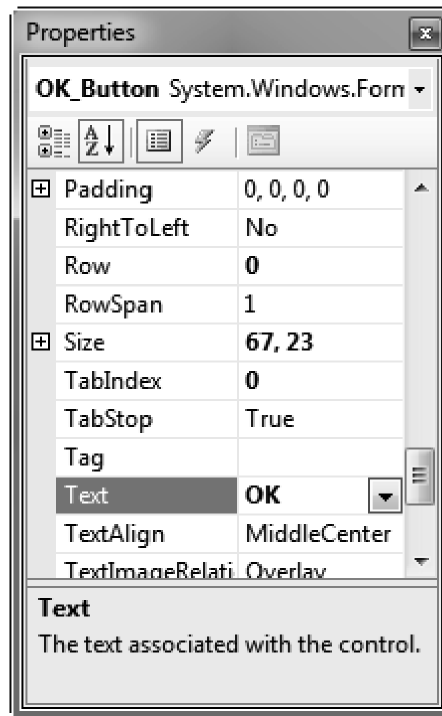
4. Click on the Toolbox and optionally select the pushpin to keep it visible as you build your form.
5. Drag and drop the ComboBox control  from the Toolbox onto your form.

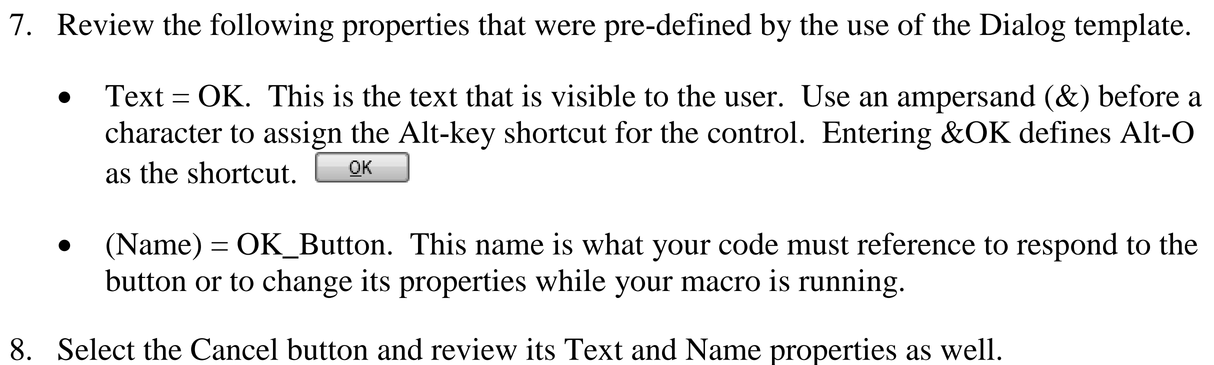
After adding the ComboBox and resizing, your form should look something like the following. An effective form is one that is compact enough to not be intrusive while still being easy to read and use.

## Object Properties

Each form control has properties you can change to affect its visual display as well as its behavior. The properties panel is visible on the right side of VSTA under the Project Explorer.

6. Select the OK button on the form. The Properties window will list the control's properties.



7. Review the following properties that were pre-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. Entering &OK defines Alt-O as the shortcut.

   - (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.

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

9. Click anywhere inside the form, not on a control, and change its Text property to “Materials.”

10. Review the following additional properties of the form.

    - AcceptButton = OK_Button
    - CancelButton = Cancel_Button

| Misc | |
|---|---|
| AcceptButton | **OK_Button** |
| CancelButton | **Cancel_Button** |
| KeyPreview | False |

11. Select the ComboBox and set its Name to “Materials_Combo.”

## Show the Dialog

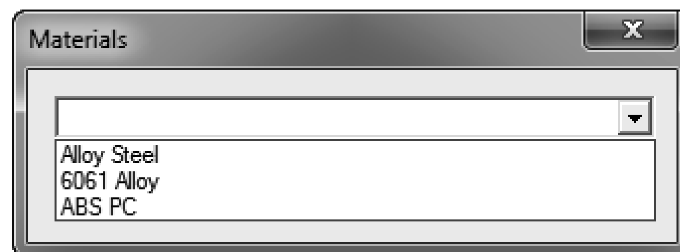
The macro needs a line of code that will display the form at the right time. If you run the macro right now, it will not do anything since the main procedure is empty.

12. Switch back to the SolidWorksMacro.vb tab and add the following code in your main procedure.

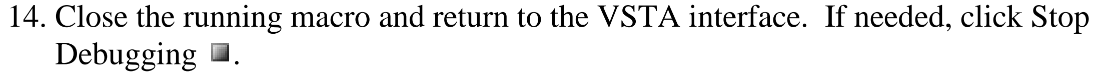
```
Sub main()

    '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
```

13. Run the macro and test.

You should see your new dialog box. 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.



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

14. Close the running macro and return to the VSTA interface. If needed, click Stop Debugging .

There are a few steps required to make a form or dialog visible. The first 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 reference it. It's worth mentioning that the name of the class does not always match the name of the file as it does in this example.

15. 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 can change the name of the class without changing the name of the vb code file. In fact, a single code file can contain as many classes as you want, though it makes it more difficult to manage and reuse.

16. Switch back to the SolidWorksMacro.vb tab to return to the main procedure.

The first time you use any class, whether it be code or a dialog, you must use the `New` keyword before you can reference it. This is distinctly different than Visual Basic 6 (VBA). VBA creates new instances of forms and dialogs as soon as they're referenced. It may seem that the .NET method of referencing forms is a little more verbose, but it has real benefits we'll see later.

## Windows.Forms Namespace

A variable named `MyCombo` was declared as `Windows.Forms.ComboBox` and was set to the `Materials_Combo` control from the instance of the form named `MyMaterials`. The declaration defines the 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. Think of namespaces as libraries of pre-build elements.

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

17. Add the following Imports statement to the top of the code window to reference the namespace. Notice that this same imports statement was automatically added to the *Dialog1.vb* code.

```
Imports SOLIDWORKS.Interop.sldworks
Imports SOLIDWORKS.Interop.swconst
Imports System.Runtime.InteropServices
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 an ordered list of values. The general syntax to declare an array is `Dim variablename(x) As type`.

`MyProps(2)` was declared as a string data type. In other words, you made room for three text elements in that one variable. "Wait! I thought you declared two?" Arrays count from a zero element, so a size of 2 gives room for 3.

## ComboBox.Items.AddRange Method

To populate the combo box with the array, you must tell the macro where to put them. Typing `MyCombo.Items.AddRange (MyProps)` tells 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 element 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

After a user has selected the desired material, it should be applied to the active part after clicking OK. 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.

18. 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 once the form is closed or dismissed. Any code after ShowDialog will then be run. Since we have added the `Imports System.Windows.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 find the If statement False, skip the inner code, and run to the end.

## Setting Part Materials

Let's get the 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.

19. 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

Notice the new declaration of the Part variable as IPartDoc rather than ModelDoc2. Think of IModelDoc2 as a container that can be used for all SOLIDWORKS files. It could be a part, an assembly or a drawing. There are many operations that are common 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 calling a function specific to a part, the IPartDoc interface is the appropriate reference. The challenge is that the ActiveDoc method returns an IModelDoc2 interface which could be a IPartDoc, an IAssemblyDoc or a IDrawingDoc. 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. Explicit declaration also enables the correct IntelliSense information, making coding easier.

## IPartDoc.SetMaterialPropertyName2 Method

The simplest way to set material properties is using the SOLIDWORKS materials library. SetMaterialPropertyName2 is a method of the IPartDoc interface and sets the material based on its configuration and a specific database.

**IPartDoc.SetMaterialPropertyName2 (ConfigName, Database, Name)**

- **ConfigName** is the name of the configuration to be used. Pass the name of a specific configuration as a string or use "" (an empty string) 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. Use a fully qualified path if the library isn't working as expected.

- **Name** is the name of the material as it displays in the material library. If you misspell the material, nothing will be applied.

The macro is now fully operational. Try it out on any part.

## *Part 2: Working with Assemblies*

You can now extend the functionality of this macro to assemblies.

### Is the Active Document an Assembly?

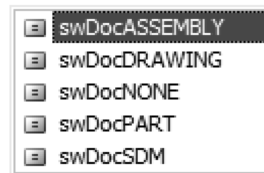
To make this code universal for parts and assemblies, we need to know what's active. If the active document is an assembly, we need to do something to the selected components. If it is a part, we run the code we already have.

20. Add the following If statement to check the active document type. The previous code has been moved 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
```

Notice the interchange between ModelDoc2, IPartDoc and IAssemblyDoc. The declaration of `Model` has also been simplified. Rather than initializing the variable to `Nothing` like the previous examples, it is initialized directly to `swApp.ActiveDoc`. This is a shorthand way to declare the variable and set its 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 IPartDoc and IAssemblyDoc, they inherit the document type specific capabilities of parts and assemblies.

Also, notice the use of the IModelDoc2.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 IPartDoc and IAssemblyDoc 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. The enumeration swDocumentTypes_e lists the possible types. 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 the SelectByID2 method. However, the code had to be specific. We had to pass the name of the component or a selection location, but that gets restrictive if you expect a user to interact with your macro. To get around those limitations you can employ a pre-selection method that is similar to most SOLIDWORKS features. You can require the user to pre-select the components he wishes to change prior to running the macro. Then write your macro to operate on each item the user selects. The Selection Manager interface makes this easy.

Connecting to the ISelectionMgr interface is similar to getting the IPartDoc (called `Part`). The Selection Manager is a child of IModelDoc2.

21. 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.swDocASSEMBLY Then
   Dim Assy As AssemblyDoc = Model
   'set materials on selected components
   Dim SelMgr As SelectionMgr
   SelMgr = Model.SelectionManager

End If
...
```

From the Selection Manager, you can get to the selected object count, type, or even the xyz point in space where the object was selected. In this macro you will need to access the selected object count (number of items selected), and get 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, get to IModelDoc2 and then set its material if it is a part.

22. 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.GetSelectedObjectsComponent4 (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. 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. You get a message box stating how many times you have clicked OK. That is great if you know 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. You can use ISelectionManager to help.

## ISelectionMgr.GetSelectedObjectCount2

The number of selected items is retrieved by using GetSelectedObjectCount2. 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. They're critical for features that require several distinct selection sets.

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

The For loop starts with an initial `i` value of 1 so that it will only loop if the number of selected items is greater than zero. If nothing is selected, the method returns 0.

### GetSelectedObjectsComponent4

The next step is to get the ModelDoc for each of the selected items. It requires a two-step process. The first gets the IComponent2 interface through the selection manager's GetSelectedObjectsComponent4 (*item number, Mark*) method. The Mark argument is again -1 to get the component regardless of selection Mark. The underlying ModelDoc is retrieved from IComponent2. Notice the declarations for `compModel` and `Comp`. They are specific to the type of SOLIDWORKS object we are accessing.

### GetModelDoc2

IComponent2.GetModelDoc2 gives access the underlying IModelDoc2. No arguments are required.

Now that you have the ModelDoc, you can use the same code from the part section to set the material properties after checking its type for parts.

### Component vs. ModelDoc

If you have been wondering why we have to take the time to dig down to the IModelDoc2 interface of the Component, this discussion is for you. If not, and it all makes perfect sense, move on to the next subject.

Think of it this way – an IComponent2 interface understands information about the IModelDoc2 it references. 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. All of these can be accessed and changed using the IComponent2 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 IModelDoc2 interface.

## *Verification and Error Handling*

It's always a good practice 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 95% of the population?

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

- Is the user in an assembly? The user must be in an assembly in our example to use the GetSelectedObjectsComponent4 method.
- If the active document is an assembly, has the user pre-selected at least one part? If not, they may assume they are applying material properties while nothing happens.
- Has the user selected items other than parts? If they select a plane or sketch, the macro may generate an exception or crash because there is no IModelDoc2 interface.
- Does the user even have a file open?

The only conditions left untested are the number of selections and if there is an active document.

23. Add the following immediately following the declaration of `Model` 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
...
```

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

```
...
Dim Comp As Component2
Dim compModel As ModelDoc2
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. Check GetSelectedObjectCount2 for a value less than one. If it is less than one the user has failed to select anything.

## MsgBox

Make use of the Visual Basic MsgBox to give the user feedback. This pre-defined dialog 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, similar to using ShowDialog on the form.

The macro now gives the user better feedback. It makes good programming sense and is worth the extra effort to build good error handling into your macros. Users tend to quickly get frustrated when a tool crashes or generates undesired results.

## Conclusion

Windows Forms are easy to design and are highly customizable. Get creative and ask for user feedback as you develop your own tools. The Selection Manager will also help you process most user selections. Finally, explore and experiment with IComponent2, IModelDoc2, IPartDoc, IAssemblyDoc and IDrawingDoc and their unique methods and properties, as well as their relationship to each other.