Automating SolidWorks 2004 using Macros

A Visual Basic for Applications tutorial for SolidWorks users

By Mike Spens

Schroff Development Corporation

www.schroff.com
www.schroff-europe.com
Introduction .................................................................................. 46
Part 1: Basic Material Properties ................................................. 46
User Forms ..................................................................................... 47
Arrays .......................................................................................... 51
Part 2: Working with Assemblies ................................................... 55
Selection Manager ......................................................................... 56
Verification and Error Handling ..................................................... 60
Conclusion ...................................................................................... 62
Material Properties

Introduction

This exercise is designed to go through the process of changing document properties such as density. It will also review several standard VB programming methods. This exercise could easily be extended to copying document properties to other SolidWorks files. It will also examine a method for parsing through an assembly tree to change some or all of the parts in the assembly.

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 density to the active part. The user should be able to cancel the operation without applying a material density.

Macro Limitations

One of the limitations of recorded macros in SolidWorks is that some dialog box actions are not recorded. At the time of publication, material option settings were not recorded. To automate selections in dialogs you will need to find the appropriate method or property from the API help. This can take some searching, so I will point out several throughout the example.

Initial Code

1. Create a new macro by selecting Tools, Macro, New or clicking .

2. Save it as materials.swp.

3. Declare swApp and Part publicly so they can be used in other modules and add the code to capture the active document.

```vbnet
Option Explicit
Public swApp As SldWorks.SldWorks
Public Part As SldWorks.ModelDoc2
```
User Forms

Through the first exercises, 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 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 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 pull-down list or ComboBox and two command buttons.

4. Add a user form to your macro by clicking or select Insert, UserForm.

A new user form window will show on the screen along with the same Controls Toolbox we saw in Excel. We need to add a ComboBox control and two CommandButton controls to our form.

5. Drag and drop each control from the Control Toolbox onto your form.

After adding the controls your form should look something like the following. The controls can be moved around and resized for
Material Properties

visibility as desired. A good 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 a list of properties that you can change. It would be impractical to have to remember that CommandButton1 is for OK and that CommandButton2 is for Close. So you can change their caption property to be OK and Close respectively.

6. Select **CommandButton1**. The Property list will show all of the object’s properties. This is the same idea as the SolidWorks Property Manager.

7. Change the text next to **Caption** to **OK** and change the **Name** to **cmdOK**. Change the **Default** property to **True**.
The Default property will make the OK button the default of the available command buttons. If the user selects a material from the list and hits the enter key, it will be the same as clicking the button.

8. Select CommandButton2 and change its Caption to Close and its Name to cmdClose.

9. Select the form itself and change its Name to frmMaterials, its Caption to Materials and ShowModal to False.

Setting the ShowModal property to False allows the SolidWorks user to interact with SolidWorks while the dialog box displays. This can be helpful if the user forgot to select some parts before running the macro.

10. Select the ComboBox and set its Name to cboMaterials, its ColumnCount property to 2 and its ColumnWidths property to 80 pt.

The ColumnCount property will allow us to see both the material name and its density in the same combo box. The ColumnWidths property sets the width of each column in our combo box control.

**Populating and Showing the Form**

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 only attach to the SolidWorks application and active document. But it would never show the form.

11. Change the code in your main procedure as follows.

```vbs
Sub main()
  Set swApp = Application.SldWorks
  Set Part = swApp.ActiveDoc

  'Set up materials list in the form: (column, row)
  Dim MyProps(1, 2) As String
```
Material Properties

MyProps(0, 0) = "Steel"
MyProps(1, 0) = "0.284"
MyProps(0, 1) = "Aluminum"
MyProps(1, 1) = "0.100"
MyProps(0, 2) = "Plastic"
MyProps(1, 2) = "0.040"

'Populate pull-down list with values
frmMaterials.cboMaterials.Column = MyProps

frmMaterials.Show

End Sub

12. Go ahead and run the main procedure as a test.

You should see your new dialog box show up. This is a result of the frmMaterials.Show. Every user form object has a show method that makes the form visible to the user.

If you click on the combo box, you should see Steel, Aluminum and Plastic listed with their corresponding densities in lbs/in^3. The OK and Close buttons are not yet defined however.

13. To close the running macro, click the close button in the upper-right corner or click the stop button in the VBA editor. Do not close the VBA interface yet.

It was determined that the combo box control would have two columns for data when its properties were set. These properties were set at design time since they were set before the code was running. These two columns were filled with an array of values at run time or while the program was running.
Arrays
An array is simply an m by n matrix of values. To declare an array you use the form `Dim variablename(m,n) As type`.

`MyProps(1, 2)` was declared as a string type variable. In other words, you made room for two columns and three rows of string information in that one variable. “Wait! I thought you declared one column and two rows”! If you haven’t learned this already, programmers count from zero. If you stick to this practice, you will avoid confusion in most cases.

ComboBox.Column Property
To populate the combo box with the array, you must tell the macro where to put things. By typing `frmMaterials.cboMaterials.Column = MyProps` you have told the procedure that you want to populate the column of the cboMaterials object in the frmMaterials object with the values in the MyProps array. The combo box control automatically creates a row for each row in the array. There is no need to assign the combo box the number of rows required, just the number of columns.

Command Button Code
In the Model Parameters exercise you learned how to add code to a command button. You will use the same method for the OK and Close buttons. You can start with the easy one.

14. In the VBA editor, switch to the user form window and double-click on the Close button. This will open a code module window and will create a name for the procedure.

The name of the procedure is directly connected to the name of the object. For example, the Close button is named cmdClose. So the procedure is as follows.
Material Properties

Private Sub cmdClose_Click()
End Sub

The _Click portion of the name is related to the action performed on the button that causes this code to run. The default action for command buttons is click. That is what most people do with buttons isn’t it? In the code module window you will see a list of actions that can be performed on each object. This is a good way to learn how to make your user interfaces more complicated. You can create code for any action in the list.

End Statement

All the Close button should do is end the macro without applying any material properties. That is easy.

15. Add the End statement as shown.

Private Sub cmdClose_Click()
End
End Sub

Now you will get your macro to do something real!

16. Switch back to the user form window and double-click on the OK button.
17. Add the code in the newly created procedure as follows.

Private Sub cmdOK_Click()
    Dim density As Double

    'The required density is in kg/m^3
    'so multiply by a scale factor

End
End Sub

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

You first declared a new variable density. This variable is declared as a Double since it will be used in the call SetUserPreferenceDoubleValue.

Working with ComboBoxes
The procedure also sets the variable density to the value stored in the combo box object. You used a couple properties of the combo box object to get the density value that corresponds to the material selected. The goal is to extract the value from the combo box’s second column and the current row.

```vba
comboBox.Column(column, row) [= Variant]
comboBox.ListIndex [= Variant]
```
Material Properties

The Column property should be 1 (remember to count from 0) and the current row number is represented by the ListIndex property of the same combo box.

The Column property requires \((column, row)\) input, so you have listed column 1 and then the current row. Adding a multiplier of 27680 converts the combo box value from lbs/in^3 to the required kg/m^3 (specified in the API help).

The last thing added to the code was the End statement prior to End Sub. This automatically terminates the program so the user does not have to click OK and then Close to finish.

Optional Additions

You could now go in several different directions from your base macro. You can add additional columns to your combo box control for hatch type, shaded color and lighting options. This can be useful if you wish to select a material from the list and have the density, hatch, color, texture and lighting set accordingly as is done in SolidWorks 2004.

18. Add the following code changes to set the hatch pattern.

```vba
Sub main()
    Set swApp = Application.SldWorks
    Set Part = swApp.ActiveDoc

    'Set up materials list in the form: (column, row)
    Dim MyProps(2, 2) As String

    MyProps(0, 0) = "Steel"
    MyProps(1, 0) = "0.284"
    MyProps(2, 0) = "ANSI32 (Steel)"
    MyProps(0, 1) = "Aluminum"
    MyProps(1, 1) = "0.100"
    MyProps(2, 1) = "ANSI38 (Aluminum)"
    MyProps(0, 2) = "Plastic"
    MyProps(1, 2) = "0.040"
    MyProps(2, 2) = "ANSI34 (Plastic Rubber)"
```
'Populate pull-down list with values
frmMaterials.cboMaterials.Column = MyProps
Materials.Show
End Sub

---------------------------------------------

Private Sub cmdOK_Click()
Dim density As Double
Dim hatch As String

'The required density is in kg/m^3
'so multiply by a scale factor
density = frmMaterials.cboMaterials.Column(1, _
 frmMaterials.cboMaterials.ListIndex) * 27680

hatch = frmMaterials.cboMaterials.Column(2, _
 frmMaterials.cboMaterials.ListIndex)

Part.SetUserPreferenceDoubleValue _
 swMaterialPropertyDensity, density
Part.SetUserPreferenceStringValue _
 swMaterialPropertyCrosshatchPattern, hatch

End
End Sub

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. And who wants to open up every part and set the properties individually?
Selection Manager

In the Model Parameters exercise we discussed selections using the SelectByID method. However, the code had to be specific to the item being selected. We had to pass the name of the component. Instead, 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 object will be the key tool for this part of the macro.

SelectionManager Object

Connecting to the SelectionManager object is similar to setting the ModelDoc2 object (called Part). The SelectionManager object is a child of the ModelDoc2 object.

19. Add the following code to your main procedure to declare the selection manager and to attach to it.

Option Explicit
Public swApp As SldWorks.SldWorks
Public Part As SldWorks.ModelDoc2
Public SelMgr As SldWorks.SelectionMgr
Sub main()

Set swApp = Application.SldWorks
Set Part = swApp.ActiveDoc
Set SelMgr = Part.SelectionManager

...
Material Properties

you must get the ModelDoc2 object and then set its material settings.

20. Add the following code under the OK button’s Click event as shown.

Private Sub cmdOK_Click()
Dim density As Double
Dim hatch As String
Dim Component As SldWorks.Component2
Dim Model As SldWorks.ModelDoc2
Dim i as Integer

' The required density is in kg/m^3 so we're multiplying _
by a scale factor

density = frmMaterials.cboMaterials.Column(1, _
    frmMaterials.cboMaterials.ListIndex) * 27680

hatch = frmMaterials.cboMaterials.Column(2, _
    frmMaterials.cboMaterials.ListIndex)

For i = 1 To SelMgr.GetSelectedObjectCount
    Set Component = SelMgr.GetSelectedObjectsComponent(i)
    Set Model = Component.GetModelDoc
    Model.SetUserPreferenceDoubleValue _
        swMaterialPropertyDensity, density
    Model.SetUserPreferenceStringValue _
        swMaterialPropertyCrosshatchPattern, hatch

Next i

Part.ClearSelection2 True
' End
End Sub

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 = 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 VB 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.

Using Selection Manager Methods
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
GetSelectedObjectCount method.

For i = 1 To SelMgr.GetSelectedObjectCount
    (loop code here)
Next i
Material Properties

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.

**GetSelectedObjectsComponent**

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 SelectionManager’s `GetSelectedObjectsComponent(item number)` method. You must use the Component object to access its underlying ModelDoc2 object. Notice the declarations for `Model` and `Component`. They are specific to the type of SolidWorks object we are accessing (early binding).

**GetModelDoc**

The `Component.GetModelDoc` method allows you to access the underlying ModelDoc object.

Now that you have the ModelDoc2 object assigned to variable `Model`, you can use the same code to set the material properties.

**Component vs. ModelDoc Objects**

If you have been wondering why we have to take the extra time to dig down to the ModelDoc2 object of the Component object, 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 object 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 object. 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
Material Properties

properties, then you must take the extra step of getting to the actual ModelDoc2 object.

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 GetSelectedObjectsComponent 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 error or crash because there is no ModelDoc2 object.

1. Add the following additions to check for any of these possible user errors.

```vba
Private Sub cmdOK_Click()
    Dim density As Double
    Dim hatch As String
    Dim Component As SldWorks.Component2
    Dim Model As SldWorks.ModelDoc2
    Dim i as Integer

    'The required density is in kg/m^3
    'so multiply by a scale factor
    density = frmMaterials.cboMaterials.Column(1, -
                                          frmMaterials.cboMaterials.ListIndex) * 27680
```
Material Properties

hatch = frmMaterials.cboMaterials.Column(2, _
frmMaterials.cboMaterials.ListIndex)

If Part.GetType = swDocASSEMBLY Then
    'check for selected components
    If SelMgr.GetSelectedObjectCount < 1 Then
        MsgBox "Please select one or more " _
        & "components first.", vbCritical
    End
End If

For i = 1 To SelMgr.GetSelectedObjectCount
    Set Component = _
        SelMgr.GetSelectedObjectsComponent(i)
    Set Model = Component.GetModelDoc
    'if the model is a part
    If Model.GetType = swDocPART Then
        Model.SetUserPreferenceDoubleValue _
            swMaterialPropertyDensity, density
        Model.SetUserPreferenceStringValue _
            swMaterialPropertyCrosshatchPattern, hatch
    End If
Next i

ElseIf Part.GetType = swDocPART Then
    Part.SetUserPreferenceDoubleValue _
        swMaterialPropertyDensity, density
    Part.SetUserPreferenceStringValue _
        swMaterialPropertyCrosshatchPattern, hatch
Else
    MsgBox "You can only use this tool " _
        & "in parts or assemblies.", vbExclamation
End If
'End
End Sub

GetType Method

The first If statement is used to determine the active document type. This could be either a part or an assembly. If the type is the SolidWorks constant swDocPart then the selected object is a part and we can apply its material properties. The GetType method allows us to find the type of model we have accessed. The constant swDocAssembly refers to an assembly (in case you had not guessed).
Material Properties

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 GetSelectedObjectCount property of the Selection Manager object 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 VB 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 trapping into your macros. Users tend to quickly get frustrated when a tool crashes or generates undesired results.

Conclusion
This macro now adds some additional value to the SolidWorks interface. In a simple format you have given the user a quick way to change material settings in any selected part. This functionality can easily be extended to changing or listing any setting related to a model.