Paper HOW-09

Extend the Power of SAS® to Use Callable VBS and VBA Code Files Stored in External Libraries to Control Excel Formatting Routines

William E Benjamin Jr, Owl Computer Consultancy, LLC, Phoenix AZ.

ABSTRACT

Did you ever wish you could use the power of SAS® to take control of EXCEL and make EXCEL do what you wanted WHEN YOU WANTED? Well one letter is the key to doing just that, the letter X as in the SAS X Command that opens the door to all operating system commands from SAS. The Windows operating system comes with a facility to write a series of commands called scripts. These scripts have the ability to open and reach into the internals of EXCEL. Scripts can load, execute and remove VBA macro code and control EXCEL. This level of control allows you to make EXCEL do what you want, without leaving any traces of a macro behind. This is Power.

INTRODUCTION

Operating systems need to communicate with the running programs and have methods of passing data between the different software packages. On Windows one of those methods of passing and controlling data movement between computer systems (or the operator and a computer software package) is called Visual Basic Scripting (VBS). SAS can communicate with this program through the "X" command. The Visual Basic scripting capability of the Windows Operating systems is very powerful and is included in the Microsoft operating system software. It can open, manipulate, control, and close an Excel program (along with other Microsoft products). This power permits Excel macros to be stored as individual VBA code modules (*.bas) so they can be used by any team member on a project. The ability to call these macros using VBS, allows the creation a standardized set of macros in a library for report formatting. Community level routines can be stored in a macro library for departmental use, while report specific macros can be stored in separate *.bas code directories. Many Excel formats are not available in SAS, while it is possible to generate code with the SAS PROC TEMPLATE procedure; this is not for the faint at heart. Programmers who are unfamiliar with PROC TEMPLATE may find the code difficult to update or write.

This paper will explain a method of building a system of directories, *.BAS files and *.VBS code to execute Excel macros by calling VBS from SAS to format Excel Reports. A basic working knowledge of how to build Excel Visual Basic (VBA) macros is assumed. The rest of the concept will be sketched out here to allow you to build and expand upon your project needs. The process described here stores the Excel macros in a directory as separate code modules that are only used while the macro is running to process the workbook sheets. These macros can be stored in a read-only directory with limited update access that will make them more secure and allow a wider access to the report processing when new data is available. These VBA macros will allow you to take control of EXCEL as a computer programmer and make it do your bidding. The process is simple can be setup using the following general guidelines.

Guidelines for building and Using a VBA Macro Library

1. Establish a disk directory where the VBA and VBS code modules can be stored. If multiple users need access it works best to have it available somewhere where everyone can read the directory, like a server location.

2. Publish a standard set of parameters that the VBS routine users can use to access the VBA code modules.

3. Commonly used routines can be stored in a VBA code module that everyone can use.

4. Unique code modules for individual reports can be placed into the directory and called from the VBS script using parameters setup by the SAS routine.

5. SAS can use the "X" command to start the VBS routine and assign the parameter values to process the report. (Note the SAS "X" command in not available to the SAS Enterprise Guide users due to environment restrictions that prevent SAS Enterprise Guide from knowing where it is running).

6. SAS can process the data files to be used as input to the VBS control module. The output from SAS can be any format that the VBA code modules can read.

The SAS "X" command is a powerful command tool. It allows nearly any Operating System command to be executed by starting it from within the SAS program code. Directories can be listed, other SAS jobs can be started, Excel workbooks can be opened, files can be deleted, and now we will discuss how a built-in operating system function can be used to control a Microsoft product like EXCEL (and Word). The VBS scripting language is similar to the VBA code language; but, it does have a few minor differences. You can execute the commands stored in a VBS code module (any_file_name.vbs) simply by double clicking on the filename or using the SAS "X" command to run the script. VBS scripts will also accept parameters at invocation, allowing you to control how they work internally. VBA macros can be built by recording the macro from within the EXCEL workbook. These recorded macros generally have a lot of default commands that are not required for a final macro and they also have a lot of workbook and worksheet specific cell references that may be too specific for a general purpose macro. SAS has the feature of being a top-down programming language, what that means is that code has to be defined before it can be used. Code is presented to SAS as a text stream, and each macro, data step and procedure call must be executed one group at a time. Code at the bottom of the text file is not executed until the step it is contained within is executed, usually at end of the job. However Object Oriented programming languages like VBA read in the whole set of code routines, compile them, then passes control of the program to a routine that waits for something to happen. Like a user moves a mouse pointer, clicks on a menu, or pushes a key on the keyboard. Each object has its own list of things you can make it do or do to it. These are beyond the scope of this book, but some simple things will be explored to show how to start building a set of your own VBA macros to use to create your reports.

Virtually anything that you can do using Excel you can do using VBA, after all Excel was written using the VBA language. So, you can modify cells by adding data to them, outlining them, moving them copying them, or clearing them. You can add or delete worksheets, and manipulate rows or columns of data. By being able to add or delete worksheets you have the ability to pass information to Excel that can be used in the formatting process and then be able to delete that information, This is something that a TAGSET cannot perform. Files can be read, written or converted from one format to another. By writing the VBA code yourself you have control of the order of the actions that Excel takes. The intent here is to show you how to create formatted Excel output files that are ready for delivery to your user in minutes instead of spending much longer doing it yourself. Making them high quality will be left to you. The initial setup of the first run of each report may take a little longer because you need to write VBA code, but every time you execute the report you fill save time running and getting it ready to print.

The SAS code listed below under the heading "**SAS Code to Create an Unformatted Output *.XML File**"

is a simple example. It starts with the SASHELP.CLASS file and sorts it by sex and height, then writes it to a *.xml file using the ODS Tagset EXCELXP. We will also suppress the column with the observation number to make the file a little cleaner. Additionally, the code sets up SAS macro variables to define the following:

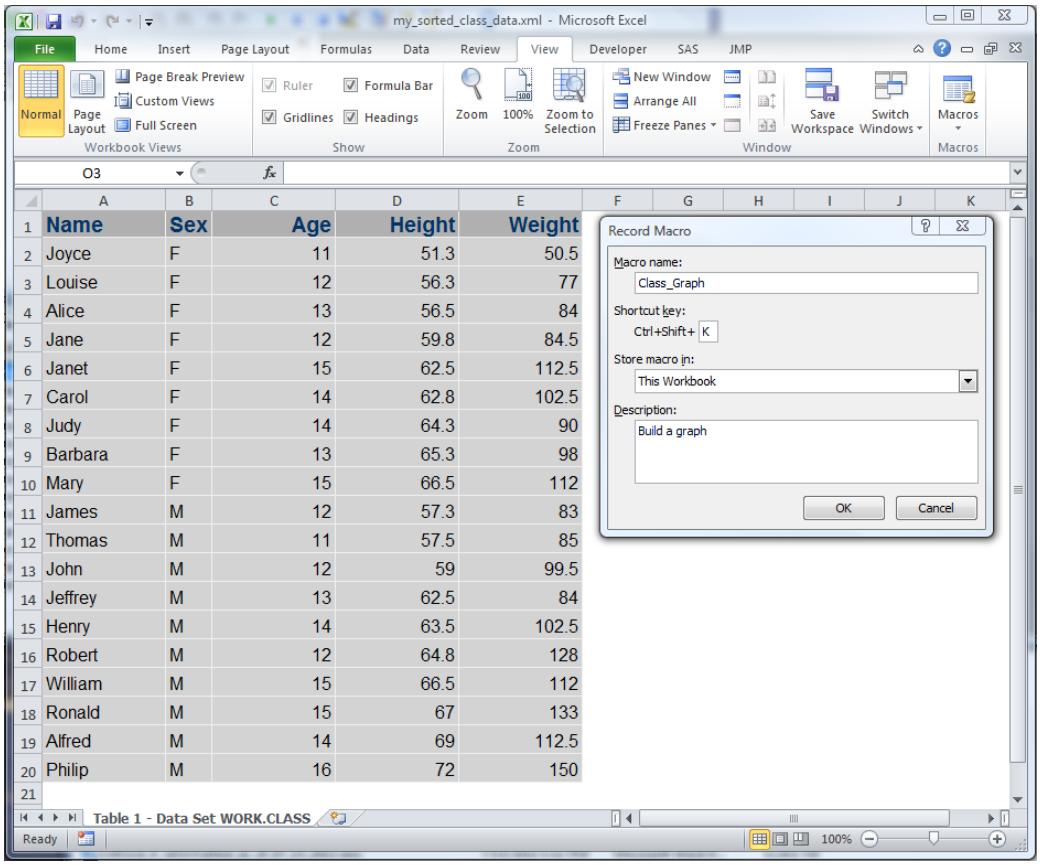
Variable	Example	Description
Vbs_code	C:\My_VBA_macros\VBS_Execute_script.vbs	VBA subroutine name to execute
Input_Excel	C:\MY_Excel_Files\my_sorted_class_data.xml	Full File path and Input file name
Output_Excel	C:\MY_Excel_Files\my_sorted_class_data.xls	Full File path and Output file name
Bas_Code_Path	C:\My_VBA_macros\	VBA Path
VBA_Module	Class_Graph	VBA module name (without the bas)
VBA_Code	Format_Class_Graph	VBA subroutine name to execute

Table 1. List of Parameters used to control the execution of the VBS script shown below.

The final line of the SAS code is the X command to run the *.vbs script to format the output *.xml file and create the requested *.xls file. In most cases you will find SAS documentation that says use the NOXSYNC and the NOXWAIT SAS options. These release the command window and allow SAS to continue running. I suggest when running SAS in a batch or background mode to use XSYNC and the XWAIT to help make sure that SAS does not finish running until the Excel formatting has finished. The VBS script code is listed below under the heading "**VBS Script to process a VBA macro**", and the SAS X command is listed below.

```
X "'&VBS_code.' ""&Input_Excel"" ""&output_excel"" ""&bas_code_path"" ""&vba_module"" ""&vba_code"" ";
```

Next, we will create a macro to do some work and save it to a disk file as a VBA code module. Here we have selected the "View" tab on the Excel ribbon and will start to record an Excel macro. We can call the macro anything but here we'll call it Class_Graph. Now we will build a graph to display the sorted data from the *.xml file we created using the EXCELXP TAGSET



	Name	Sex	Age	Height	Weight
2	Joyce	F	11	51.3	50.5
3	Louise	F	12	56.3	77
4	Alice	F	13	56.5	84
5	Jane	F	12	59.8	84.5
6	Janet	F	15	62.5	112.5
7	Carol	F	14	62.8	102.5
8	Judy	F	14	64.3	90
9	Barbara	F	13	65.3	98
10	Mary	F	15	66.5	112
11	James	M	12	57.3	83
12	Thomas	M	11	57.5	85
13	John	M	12	59	99.5
14	Jeffrey	M	13	62.5	84
15	Henry	M	14	63.5	102.5
16	Robert	M	12	64.8	128
17	William	M	15	66.5	112
18	Ronald	M	15	67	133
19	Alfred	M	14	69	112.5
20	Philip	M	16	72	150

Figure 1 Excel worksheet with the SASHELP Class data shown.

The VBA macro that was recorded selected the male class members and highlighted the cells in a light blue color. Then we built a bar graph using the name, sex, age, height, and weight columns. After the graph was built it was moved near the data table and enlarged to fill the rest of the screen, as in Figure 2 below. The key strokes that you use to build your macro may result is slightly different macro code than what is depicted here. NOTE – The macro name in the EXCEL Properties window shows Class_Graph in a VBA module called "**Module1**". We can change that by typing a new name in the Properties Window for the module, we will also call it "Class_Graph".

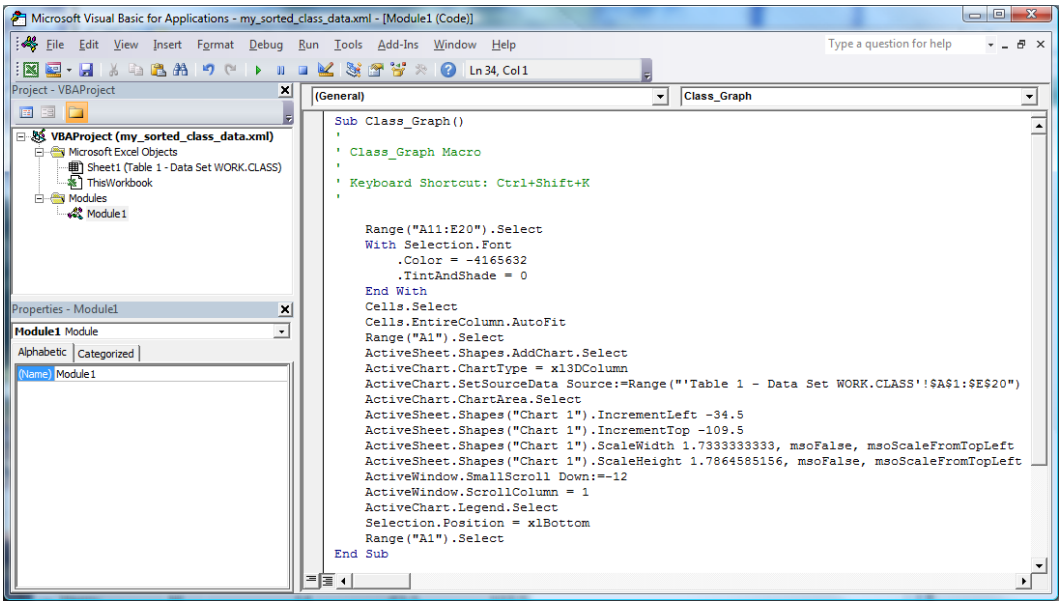


Figure 2 Excel macro code to highlight and build a graph.

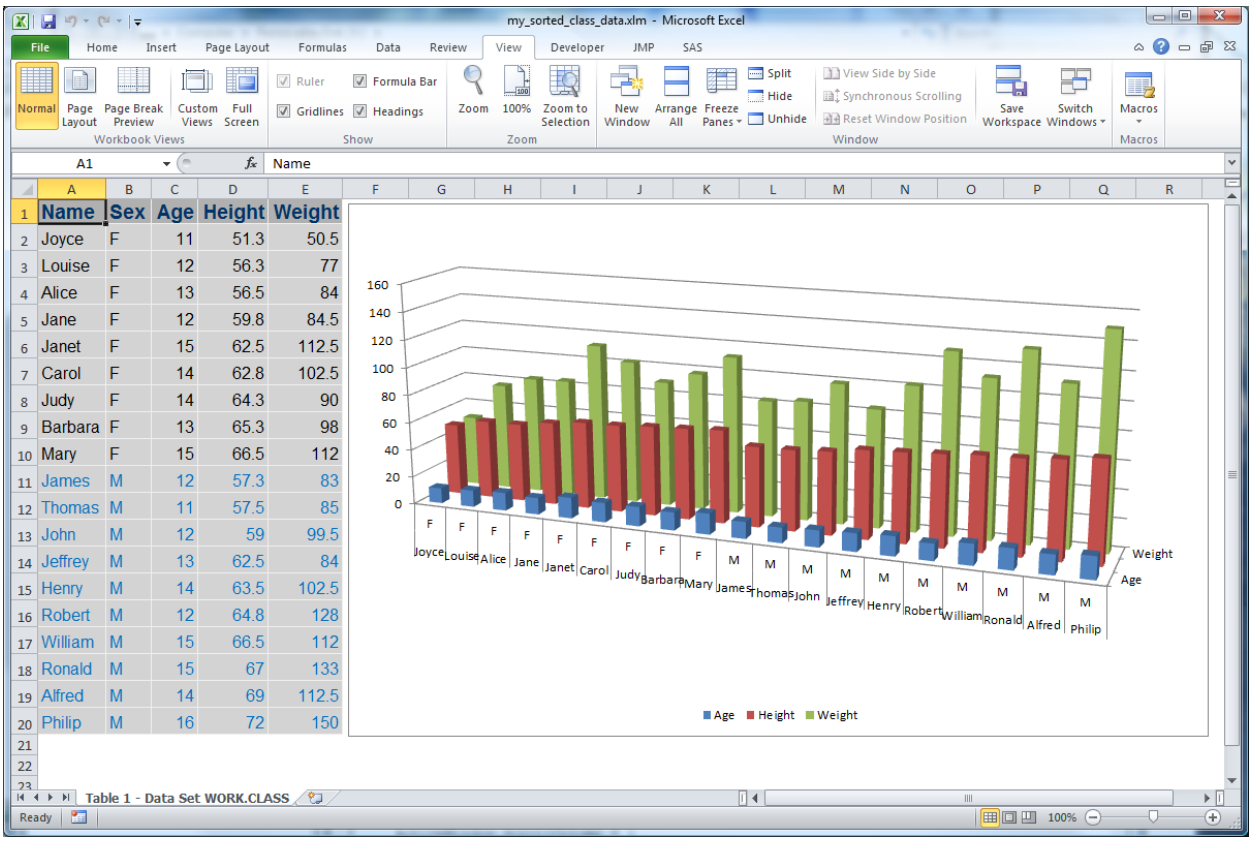


Figure 3. The sample graph that was built while recording a macro.

The intent was to have a macro always build a graph similar to Figure 3. As mentioned above minor changes to the macro may be required to make them work in a generic fashion. In this case the specific reference to “Chart 1” may

not always produce a graph; sometimes it will produce an error message and send you to the Excel Debugger. You may see a message similar to the following (Figure 4).

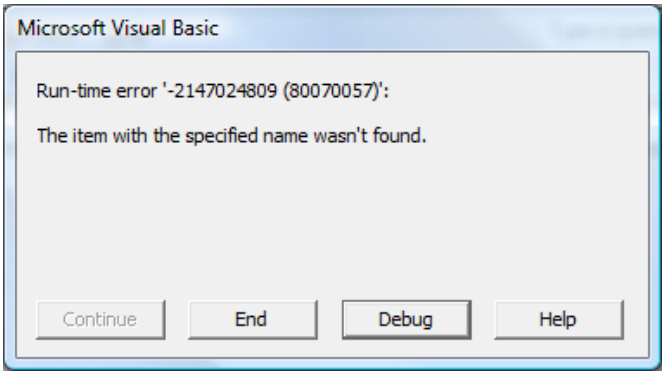


Figure 4 Error message because “Chart 1” may not exist when the macro runs.

To correct this issue we need to adjust the macro slightly to point to indexed objects rather than named objects. The next figure shows the lines that need to be changed to make the macro more generic. This will cause Excel to look for the first chart, regardless of the name of the chart. It looks for the first ActiveSheet.Shape not the one named “Chart 1”.

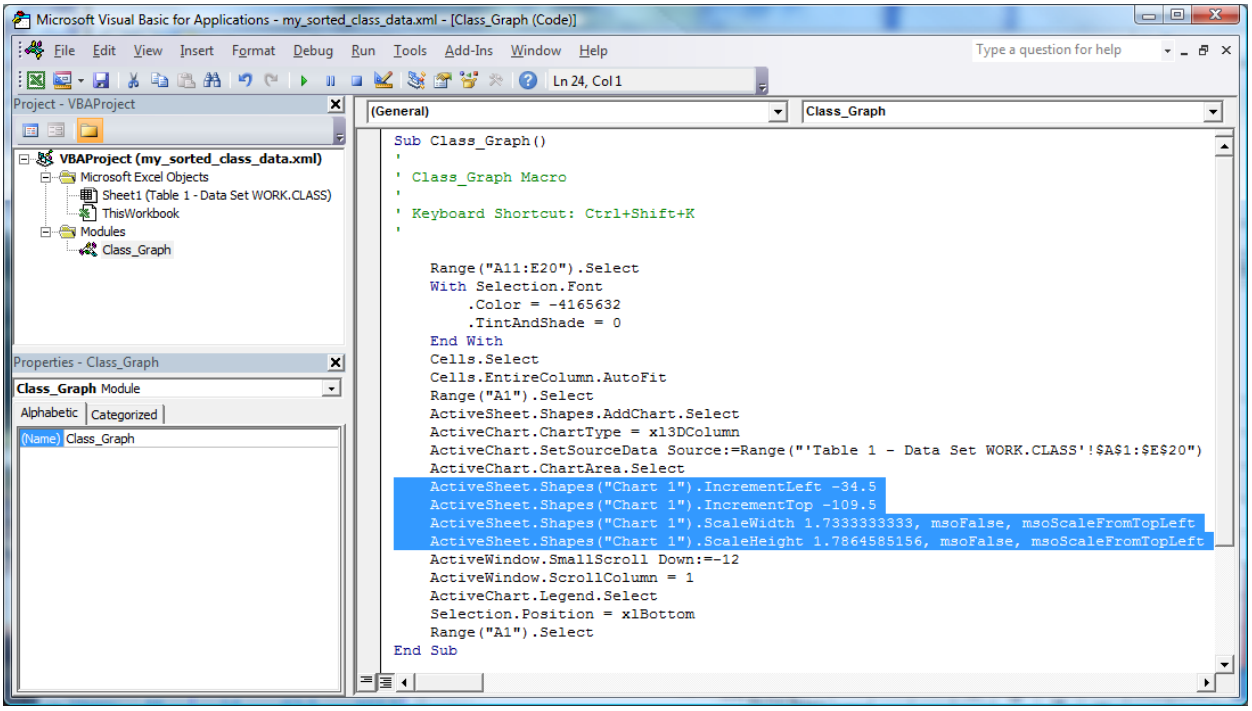


Figure 5 Changed lines of macro 1 to allow the macro to run using generic references.

Also notice that the name of the workbook and/or the worksheet ‘**Table 1 – Data Set WORK.CLASS**’ may appear embedded in the Excel macro code as part of the range of cells used as the data source for the chart. In this case the workbook name is not present but the worksheet name is shown. This will also cause the macro to force the debug mode to open and cause the macro to fail to finish building the graph if the page name ever changes. When the workbook name (and the “!”) are removed the reference defaults to the active worksheet in the current workbook. In an effort to keep things simple here we will not delete the worksheet name or the hard-coded cell range ($A$1:$E$20), just remember that it could become an issue. To make this completely generic you will need to calculate the actual size of the data range for the graph, which is not done here.

Now there is one last thing to do before we export this Excel VBA code into a macro library for later use. We will change the name to something more meaningful for the future use of the macro. The name will be changed to "Class_Graph" to make it easy to remember. This is done by editing the name property on the left side to the Visual Basic for Applications window, as shown in Figure 8. (if it is not already named correctly)

The last step here in Excel will be to record another macro, called "**make_thick_red_outside_border**" Figure 6 below has been optimized by removing default commands, consolidating instructions and removing the selection of specific cells to process. These steps make this routine reusable by calling it after selecting a range of cells to process. Additionally, the module name is changed to Common. While this VBA code module only has one VBA subroutine it could contain many subroutines. (NOTE - all VBA modules can contain many subroutines) This code is to show you how to establish a pattern of how to build these modules and subroutines for generic use. If you can do it for one module you can do it for many.

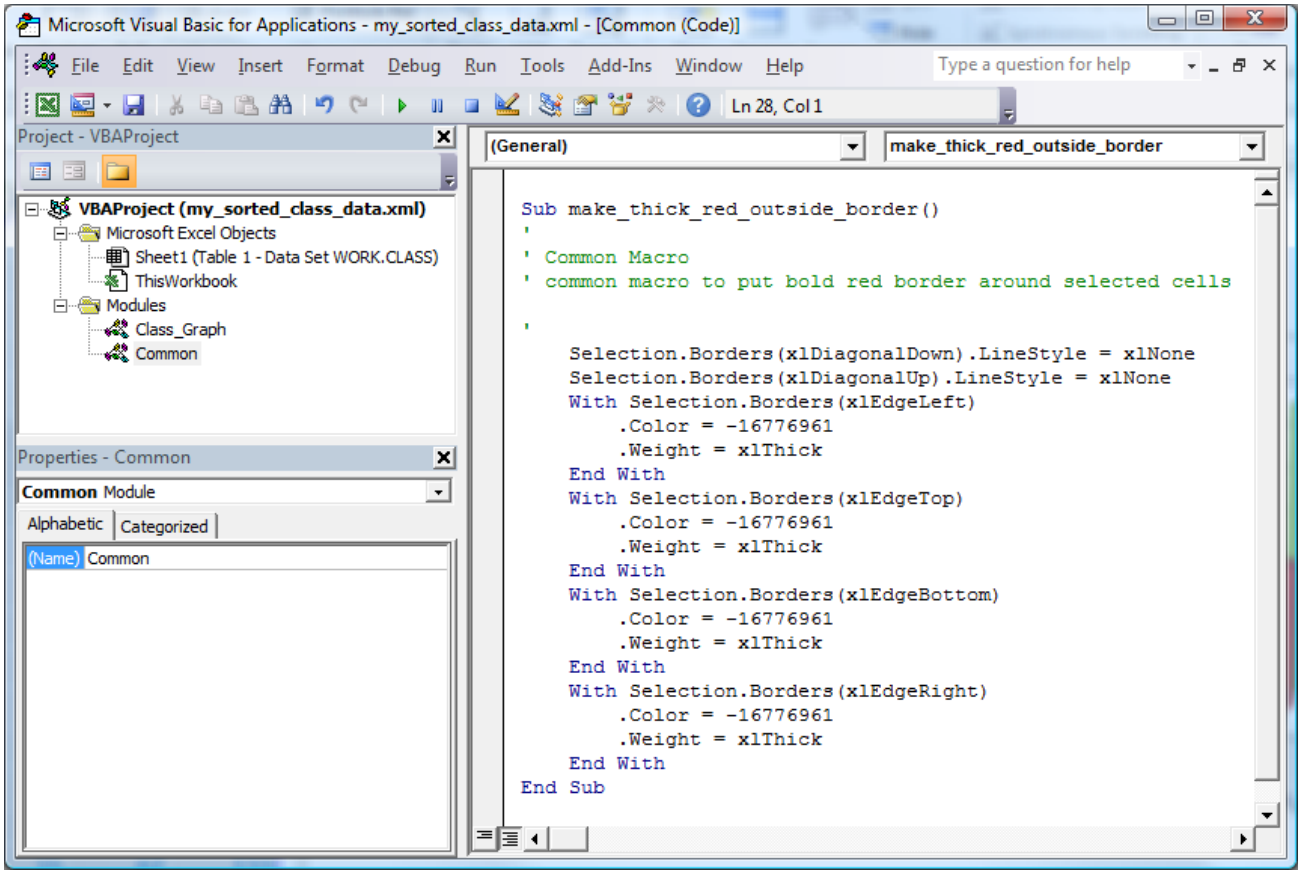


Figure 6 Macro to put a thick red border around selected cells.

Now that we have a main routine to build our graph and a common routine that will put a red border around selected cells we should write some code to use both pieces of code. The simple way to execute the subroutine to make red borders is to call it by name as in the following command:

Call make_thick_red_outside_border

All we need to do is insert this command into our original code as indicated in Figure 7.

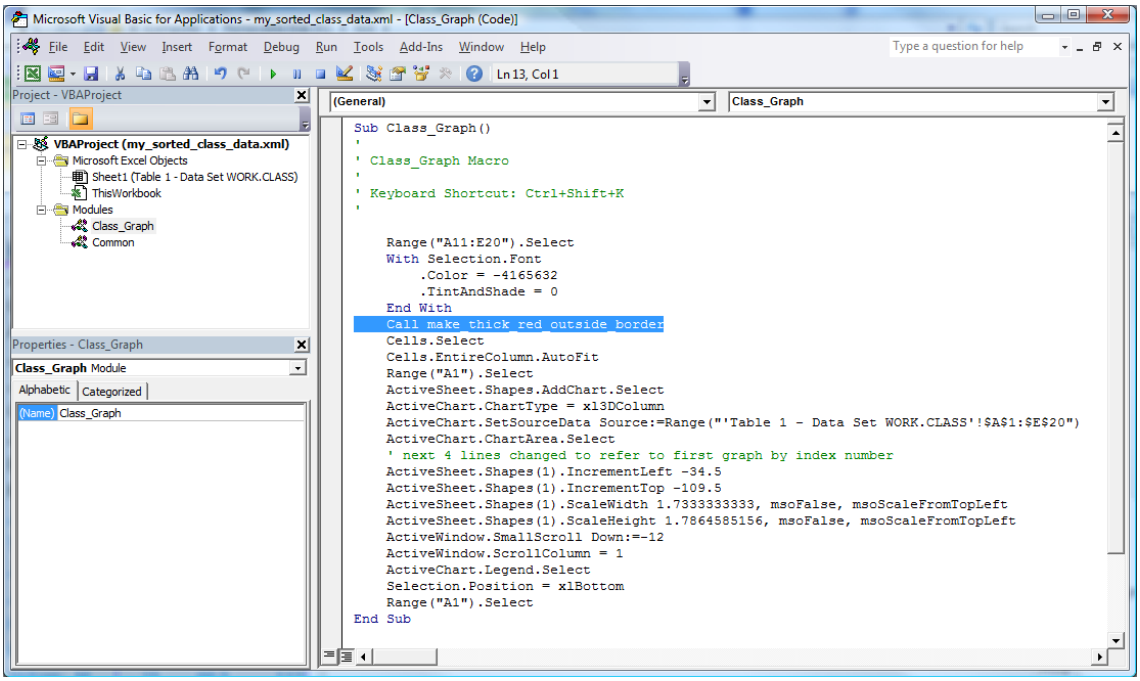


Figure 7 The command to call the subroutine “make_thick_red_outside_border” is highlighted.

See the SAS code below labeled “**SAS Code to Create an Unformatted Output *.XML File**” This code will produce a file that can be opened by Excel. It is a XML formatted file, but Excel can open the file. The VBS script labeled By using the EXPORT option on the File menu we can place the new Class_Graph macro into any directory we want to use. The one used here is the My_VBA_macros directory created for the example.

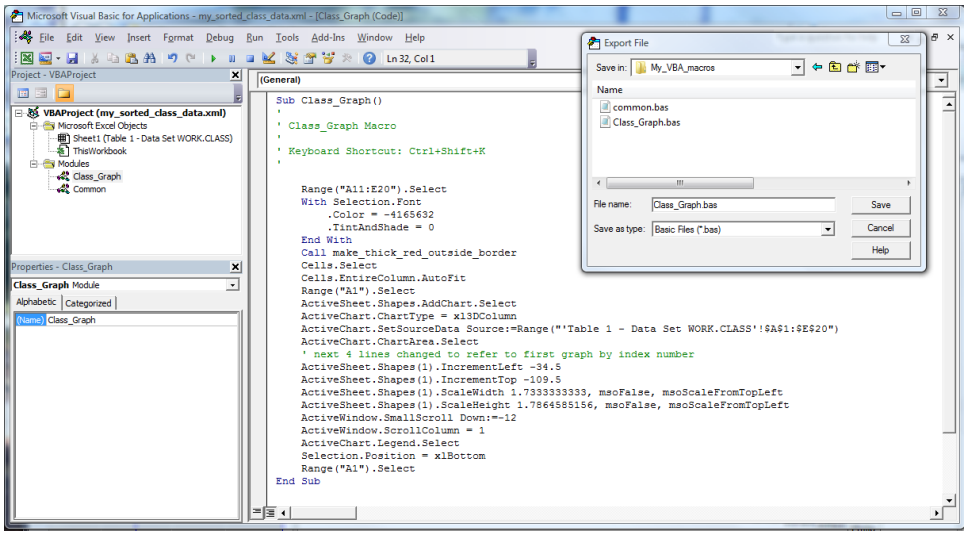


Figure 8. Saving the Class_Graph macro to the My_VBA_macros directory.

Now for the fun part, we need a VBS routine to do the work. We need it to open the XML file, load our VBA macros, build our graph, and save our data in the XLS format. The code is reproduced later as “**VBS Script to process a VBA macro**”, and has comments on nearly every line to describe the action taken by each line of code. Space here does not permit a line by line description here of the code and how it works.

Another message that may appear could be something like the one in Figure 9. that talks about trusting the Visual Basic Project code. This message will be slightly different depending upon the version of Excel you are using. Here is the message when Excel 2010 is being used.

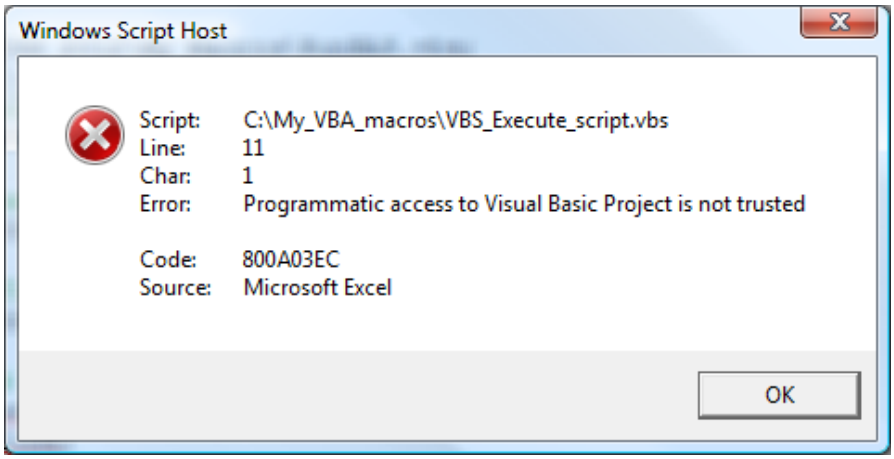


Figure 9 The Windows message about the Visual Basic Project not being trusted.

You may be able to eliminate this message by changing the Trust center options for your computer. If your System Administrator has restricted these commands, then you will need to ask them for help.

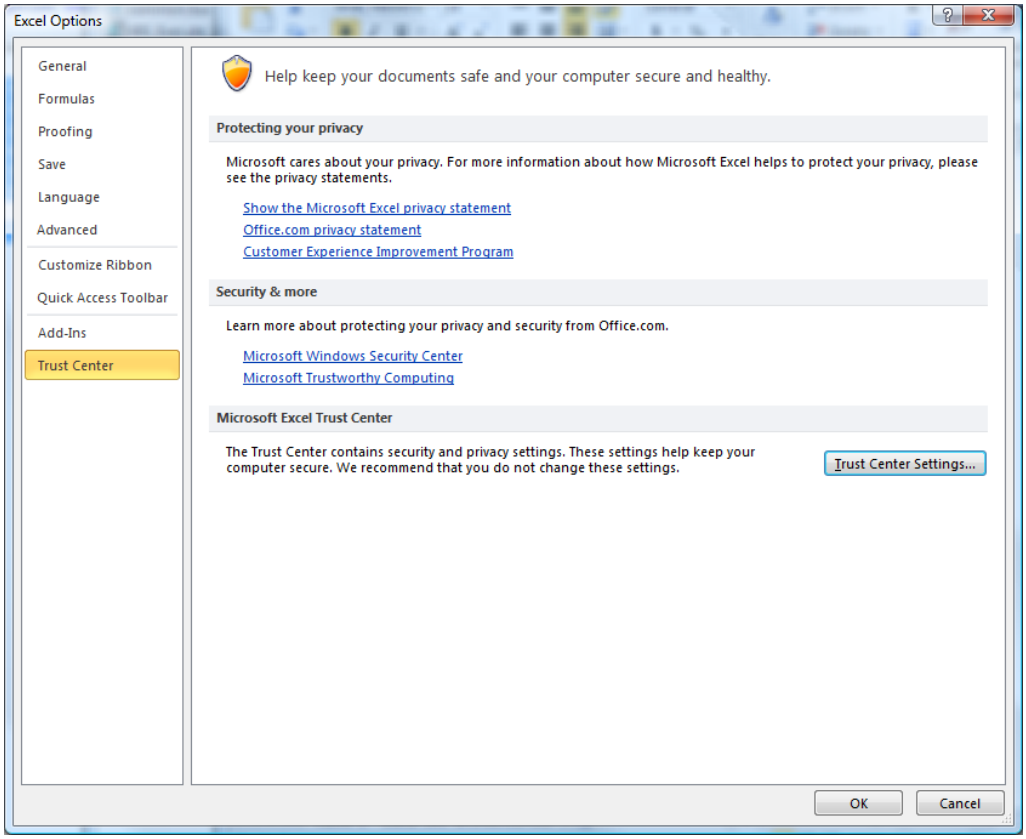


Figure 10 Excel Options window showing the “Trust Center Settings” screen.

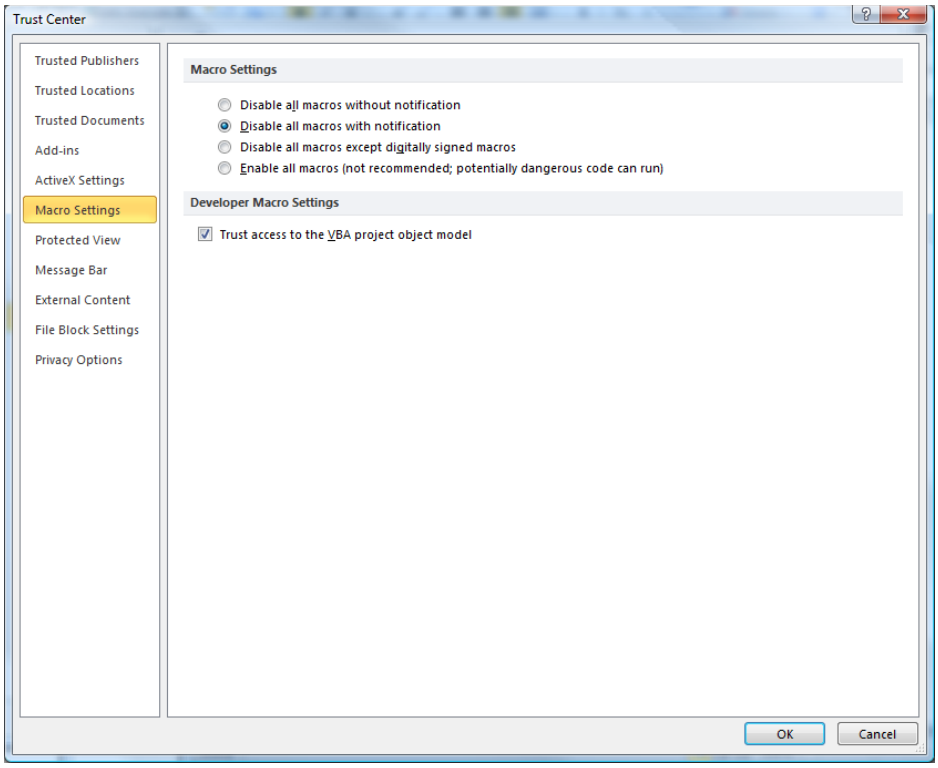


Figure 11 The "Trust Center Settings" screen showing settings to enable the Trust access to the VBA Project object model checkbox.

It is always wise to verify if you are permitted to modify these settings, Some companies have strict policies about modifying PC Settings without permission.

CONCLUSION

The ability to control how Excel formats a spreadsheet or what data or graphs are placed onto a spreadsheet is a powerful extension of skill. Then add to the process the fact that there are no VBA macros left in the Excel file to make the security tools stop the "Spread" of the file and you now have a powerful tool that will repeatedly save you time.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: William E Benjamin Jr
Enterprise: Owl Computer Consultancy, LLC
Address: P.O.Box 42434
City, State ZIP: Phoenix AZ, 85080
Work Phone: 623-337-0269
E-mail: William@owlcomputerconsultancy.com

SAS Code to Create an Unformatted Output *.XML File

```
proc sort data=sashelp.class out=class;
by sex Height;
run;

ods tagsets.ExcelXP body='C:\MY_Excel_Files\my_sorted_class_data.xml';
proc print data=class noobs;
run;
ods tagsets.ExcelXP close;

%let vbs_code      = C:\My_VBA_macros\VBS_Execute_script.vbs;    * VBS subroutine name to execute ;
%let Input_Excel   = C:\MY_Excel_Files\my_sorted_class_data.xml; * Full path and Input file name ;
%let output_excel  = C:\MY_Excel_Files\my_sorted_class_data.xls; * Full path and Output file name ;
%let bas_code_path = C:\My_VBA_macros\;                          * Full path Location of bas file ;
%let vba_module    = Class_Graph;                                * VBA Path and module name (without the bas);
%let vba_code      = Class_Graph;                                * VBA subroutine name to execute ;


X "'&VBS_code.' ""&Input_Excel"" ""&output_excel"" ""&bas_code_path"" ""&vba_module"" ""&vba_code"" ";
```

VBA Macro Code to Build a Red Border in a Common Module

```
Attribute VB_Name = "common"
Sub make_thick_red_outside_border()
'
' Common Macro
' common macro to put bold red border around selected cells
'

       Selection.Borders(xlDiagonalDown).LineStyle = xlNone
       Selection.Borders(xlDiagonalUp).LineStyle = xlNone
       With Selection.Borders(xlEdgeLeft)
              .Color = -16776961
              .Weight = xlThick
       End With
       With Selection.Borders(xlEdgeTop)
              .Color = -16776961
              .Weight = xlThick
       End With
       With Selection.Borders(xlEdgeBottom)
              .Color = -16776961
              .Weight = xlThick
       End With
       With Selection.Borders(xlEdgeRight)
              .Color = -16776961
              .Weight = xlThick
       End With
End Sub
```

* NOTE – The “Attribute” command on the first line is output by the “Export” process. It is used by the “Import” process but not part of the VBA code.

Final VBA Macro Code to Format Class_Graph output file and build a Graph

```
Attribute VB_Name = "Class_Graph"
Sub Class_Graph()
'
' Class_Graph Macro
'
' Keyboard Shortcut: Ctrl+Shift+K
'

       Range("A11:E20").Select
       With Selection.Font
              .Color = -4165632
              .TintAndShade = 0
       End With
       Call make_thick_red_outside_border
       Cells.Select
       Cells.EntireColumn.AutoFit
       Range("A1").Select
       ActiveSheet.Shapes.AddChart.Select
       ActiveChart.ChartType = xl3DColumn
       ActiveChart.SetSourceData Source:=Range("'Table 1 - Data Set
       WORK.CLASS'!$A$1:$E$20")
       ActiveChart.ChartArea.Select
       ' next 4 lines changed to refer to first graph by index number
       ActiveSheet.Shapes(1).IncrementLeft -34.5
       ActiveSheet.Shapes(1).IncrementTop -109.5
       ActiveSheet.Shapes(1).ScaleWidth 1.7333333333, msoFalse, msoScaleFromTopLeft
       ActiveSheet.Shapes(1).ScaleHeight 1.7864585156, msoFalse, msoScaleFromTopLeft
       ActiveWindow.SmallScroll Down:=-12
       ActiveWindow.ScrollColumn = 1
       ActiveChart.Legend.Select
       Selection.Position = xlBottom
       Range("A1").Select
End Sub
```

* NOTE – The “Attribute” command on the first line is output by the “Export” process. It is used by the “Import” process but not part of the VBA code.

VBS Script to process a VBA macro

```
Dim Input_Excel, output_excel, bas_code_path, vba_module, vba_code, objxl, objwk,
vbCom, myMod
Input_Excel   = WScript.Arguments(0) 'Full path and Input file name
output_excel  = WScript.Arguments(1) 'Full path and Output file name
bas_code_path = WScript.Arguments(2) 'Full path Location of .bas file
vba_module    = WScript.Arguments(3) 'VBA module (without the .bas)
vba_code      = WScript.Arguments(4) 'VBA subroutine name to execute

set objxl     = CreateObject("Excel.Application") 'Start Excel
set objwk     = objxl.Workbooks.Open(Input_Excel) 'Open the input file

'Activate special software
set vbCom     = objxl.ActiveWorkbook.VBProject.VBComponents

objxl.DisplayAlerts = wdAlertsNone 'Turn off error messages

if vba_module <> "" then
       'Import Report level VBA module
       vbCom.Import ("" & bas_code_path & vba_module & ".bas")
       'Import Common Routine VBA module
       vbCom.Import ("" & bas_code_path & "Common.bas")

       objxl.Run "" & vba_module & "." & vba_code & "" 'Run VBA routine

       'Remove VBA modules
       Set myMod = objxl.ActiveWorkbook.VBProject.VBComponents("" & vba_module & "")
       objwk.VBProject.VBComponents.Remove myMod
       Set myMod = objxl.ActiveWorkbook.VBProject.VBComponents("Common")
       objwk.VBProject.VBComponents.Remove myMod
end if

'Save as Excel workbook
if Input_Excel <> output_excel then objxl.ActiveWorkbook.SaveAs output_excel, 56
'xlExcel8
if Input_Excel = output_excel then objxl.ActiveWorkbook.Save

objxl.Workbooks.Open(output_excel).Close
objxl.Quit
set objxl = nothing
set objwk = nothing
```