Paper 1598-2014

Turn Your SAS® Macros into Microsoft Excel Functions with the SAS® Integrated Object Model and ADO

Chris Brooks, Melrose Analytics Ltd

ABSTRACT

As SAS® professionals, we often wish our clients would make more use of the many excellent SAS tools at their disposal. However, it remains an indisputable fact that for many business users, Microsoft Excel is still their go-to application when it comes to carrying out any form of data analysis. There have been many attempts to integrate SAS and Excel, but none of these has up to now been entirely seamless. This paper addresses that problem by showing how, with a minimum of VBA (Visual Basic for Applications) code and by using the SAS Integrated Object Model (IOM) together with Microsoft's ActiveX Data Objects (ADO), we can create an Excel User Defined Function (UDF) that can accept parameters, carry out all data manipulations in SAS, and return the result to the spreadsheet in a way that is completely invisible to the user. They can nest or link these functions together just as if they were native Excel functions. We then go on to demonstrate how, using the same techniques, we can create small Excel applications that can perform sophisticated data analyses in SAS while not forcing users out of their Excel comfort zones.

INTRODUCTION

SAS has many tools which are capable of carrying out data analysis in a much more robust and mathematically reliable way than Excel but we have to recognize that a high percentage of business users like Excel for its ease of use, the fact that they can easily share data with others and above all for its familiarity. The question then is how can we harness the power of SAS in such a way that our clients can retain all the advantages of Excel and still take advantage of all the benefits SAS has to offer? This paper attempts to answer that question by showing how the SAS Integrated Object Model can be used to leverage these technologies in a simple, non-threatening way that expands our customers' capabilities by allowing them to effectively use SAS macros as Excel functions, applications and add-ins.

WHY WOULD WE DO THIS?

There are a number of reasons we would want to do this; including:

- The ability to extend the functionality of Excel by adding SAS functions and procedures which are not natively available in Excel.
- To ensure that the same methodology is used throughout the organisation when carrying out calculations, whether they are in SAS or Excel.
- To provide easy access to Excel to data items held in SAS data sets (it would be simple, for example, to write a function called, getMonthlySales which returned the sales for a particular product for a particular month from SAS and loaded that into an Excel cell).

HOW DO WE DO THIS?

We can achieve this by writing a VBA project in Excel which will use the SAS Integrated Object Model to create and control a SAS workspace session (in effect a background SAS session). In the first example this will run a SAS macro which will perform a fairly simple calculation. This macro will be written using standard base SAS syntax. We will also see how to use ADO to retrieve the result of the calculation from a SAS data set and place the result in the cell where the function was used. In the second example we will use a VBA UserForm to provide input parameters and data, use ADO to create an input SAS data set from a worksheet range, run a single SAS Procedure and return the results of the Procedure from the output data set into a worksheet range.

This can be illustrated in Figure 1 which shows the flow of control between the different software.

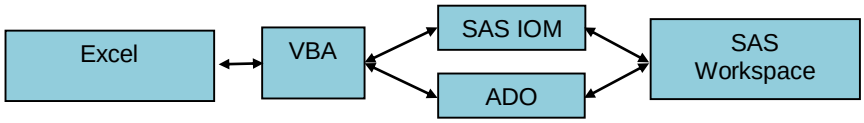


Figure 1 Flow

WHAT IS THE SAS® INTEGRATED OBJECT MODEL?

SAS Integrated Object Model software allows the developer to integrate SAS with an application written in a language such as VBA, C#, Visual Basic, or JAVA. In order to use it you must have SAS Integration Technologies licensed and installed. You may use SAS installed either on your local desktop or on a server but this paper illustrates the use of local SAS as performance is critical in using these techniques with Excel and this will prove the most performant option.

The Object Model manifests itself as a COM component allowing it to be used from a wide variety of programming environments. Additionally the IOM bridge for COM provides the capability to run the server on a platform other than Windows.

The object hierarchy consists of a number of components and the main ones we will be using are:-

- The SAS Workspace Object
- The SAS LanguageService

In addition we will be using Microsoft Activex Data Objects (ADO) to read SAS data into Excel.

GETTING READY

In order to work with the SAS Integrated Object Model and ADO in the VBA environment there are a few things we need to do.

Firstly we want to be able to access the VBA editor. There are two ways to do this in Excel

- Press the Alt-F11 key combination; or
- Make the Developer tab available on the Excel toolbar. This is done by clicking the Office button (a big circular icon in the top left corner of the Excel window), then click Excel Options, select the Popular item from the left hand listing and then locate the "Show Developer tab in the Ribbon" checkbox in the first section on the right (sub-titled "Top options for working with Excel"), put a check mark in that checkbox and the Developer's tab will appear in the Ribbon.

I would recommend the second of the above options if you intend to do a lot of VBA coding.

Once you have accessed the VBA editor you will need to add references to the project for the external class libraries we will be using. This is done by selecting Tools->References from the VBA editor menu and placing check marks against the following items:-

- SAS Integrated Object Model
- SAS ObjectManager
- SAS WorkspaceManager
- Microsoft Activex Data Objects
- Microsoft ADO Ext for DDL and Security

Figure 2 shows how this references box should look. You should note that version numbers (which I have deliberately not quoted above) may be different to those available in your environment. Also you may find more than one version of a library installed. As a general rule you should always choose the one with the highest version number. All of the references listed above should automatically be shown in the list as long as SAS Integration Technologies has been installed.

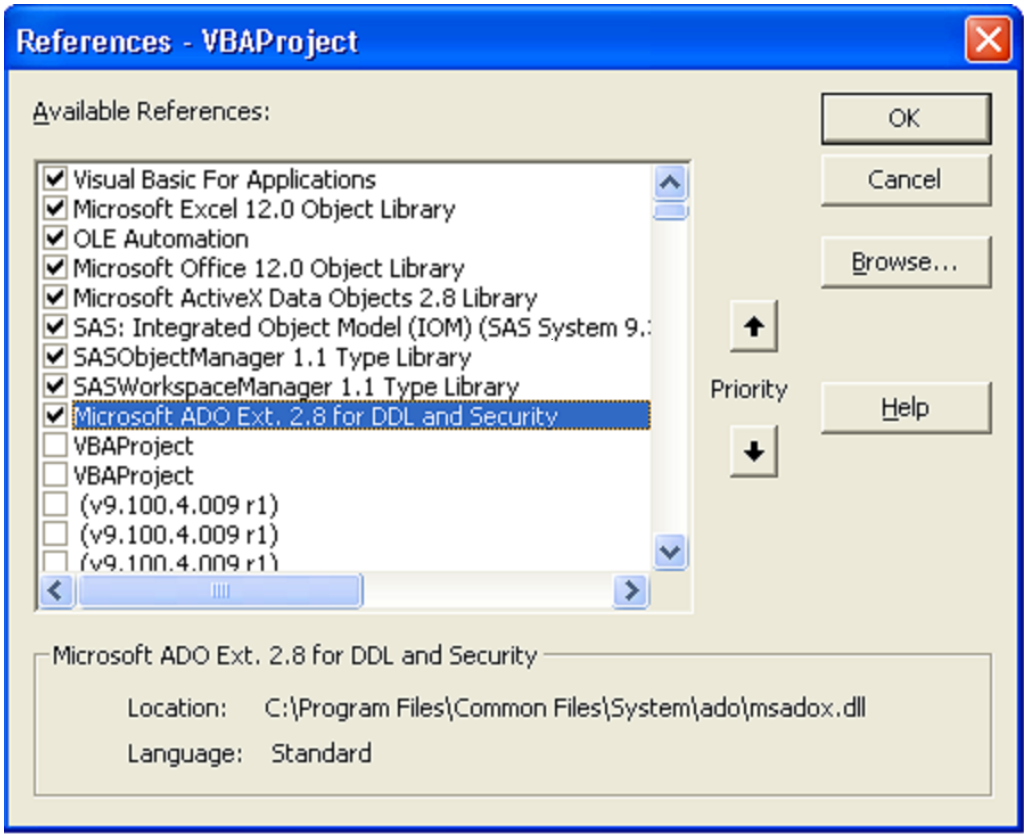


Figure 2 Required References

EXCEL FUNCTION BASICS

Creating a User Defined Function (UDF) in Excel is surprisingly easy – all you need to do is create a VBA Function that has the same name as the function you wish to create. Any parameters that need to be used should be specified, you must create a variable with the same name as the function and assign it the return value and Excel will do all the rest for you. For example a function that converts kilometres to miles would look like this

```
Option Explicit
Function kilomtomiles(kilom As Variant)
       kilomtomiles = kilom * 0.62137
End Function
```

OUR FIRST FUNCTION – THE SAS® MACRO

As an example of what can be done we will be building a function to calculate a person's Body Mass Index (BMI). The formula for this is

$$(\text{Weight(pounds)}/\text{height(inches)}^2)*703$$

If we were using metric units the formula would be very similar

$$(\text{Weight(kilos)}/\text{height(inches)}^2)$$

The SAS macro we will use to calculate BMI is shown below

```
%macro bmi(height,weight);
       data result;
          result=round(((&weight/&height**2)*703),1);
       run;
%mend bmi;
```

MAKING THE CONNECTION – STARTING SAS®!

Firstly we need to declare some variables – we will need a SAS Workspace Object variable and a Workspace Manager variable. These are declared as Public outside the function in the VBA editor.

```
Public obSAS As SAS.Workspace
Public obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager
```

Then we will declare our function and its parameters and create some private variables for use internally

```
Public Function bmi(height As Variant, weight As Variant)
    'Declare some VBA and ADO objects we will be using
    Dim obConnection As New ADODB.Connection
    Dim obRecordSet As New ADODB.Recordset
    Dim errorString As String
    Dim sourcebuffer As String
```

This is where it starts to get interesting – we will next check if we already have a SAS Workspace instance in existence. This is done by checking the obSAS variable and if it evaluates to nothing this means we do not have a SAS Workspace available and so we will create one. The CreateWorkspaceByServer method used in the way shown here will create a local SAS Workspace object (effectively a background SAS session which you can verify by checking the Processes tab on the Windows Task Manager).

```
If (obSAS Is Nothing) Then
       Set obSAS = _
obWorkspaceManager.Workspaces.CreateWorkspaceByServer("MyWorkspaceName", _
       VisibilityProcess, Nothing, "", "", errorString)
End If
```

The reason we want to check if we already have a SAS Workspace object is that we do not want to open and close one for every instance of the function in the worksheet. It may be that there are dozens of instances of the function present so for performance reasons we want to reuse the SAS Workspace we have already created. We will see how we shut down this Workspace instance later.

SETTING UP OUR SAS® SESSION AND RUNNING THE MACRO

There are a number of ways of running SAS code on this Workspace server but the simplest is to assign the code to the sourcebuffer string variable we declared earlier and then submit it using the Submit method of the SAS Language Service. Strictly speaking it is not necessary to assign the code to the sourcebuffer variable but I would recommend it as it is easier to debug because you can write its value to the VBA Immediate Window using the Debug.Print sourcebuffer statement.

The following code sets up our autocall library and runs the bmi macro passing in the two parameters passed to the Excel function as parameters to the SAS macro

```
sourcebuffer = "options sasautos=(sasautos,'d:\bmi');%bmi(" & height & "," & _
       weight & ");"
obSAS.LanguageService.Submit sourcebuffer
```

We should now have a data set in the work library holding the result of the calculation

GETTING OUR SAS® OUTPUT BACK INTO EXCEL

Now that we have the result of the calculation we must retrieve it from SAS and populate the spreadsheet cell where the function was entered. This is achieved using Microsoft's ActiveX Data Objects (ADO). This is a set of Component Object Model objects which facilitate access to data through an OLE DB provider.

To do this firstly we need to open a Microsoft Activex Data Object Connection. In the following code line sas.iomprovider.1 forces use of the latest installed version of the iomprovider and the SAS Workspace ID property is used to associate the ADO connection object with the existing SAS Workspace.

```
obConnection.Open "provider=sas.iomprovider.1; SAS Workspace ID=" + _
      obSAS.UniqueIdentifier
```

Once we have the connection we need to set up a recordset object which will actually hold the data using the recordset open method. In this example we are initializing one and :

- Specifying the data source to be the output data set from the SAS macro
- Instructing it to use a static cursor so that a copy of the data set at open time will be used
- Setting a read-only lock on the data set
- Instructing the object that the data source is to be interpreted as a table name

The code for that is

```
obRecordSet.Open "work.result", obConnection, adOpenStatic, adLockReadOnly, _
      adCmdTableDirect
```

Now that we have our recordset object we need to retrieve the result from the data set. As we only have one row (and will only ever have one row) to retrieve for the function we can use the MoveFirst method

```
obRecordSet.MoveFirst
```

All that remains is for the value to be assigned to the variable we declared earlier with the same name as the function and Excel will take care of the rest. Note that when we use obRecordset(0) the number in brackets indicates the index value of the field in the recordset fields collection. In effect it is the variable number (which starts at zero and as we only have one field in the data set this can be hard coded).

```
bmi = obRecordSet(0).Value
```

SHUTTING DOWN THE SAS® WORKSPACE

As mentioned earlier we do not want to close the SAS Workspace created for use by the function until we have finished with the Excel workbook so we need another small amount of VBA code to be placed inside a procedure called Workbook_BeforeClose which will automatically run every time the user closes the workbook.

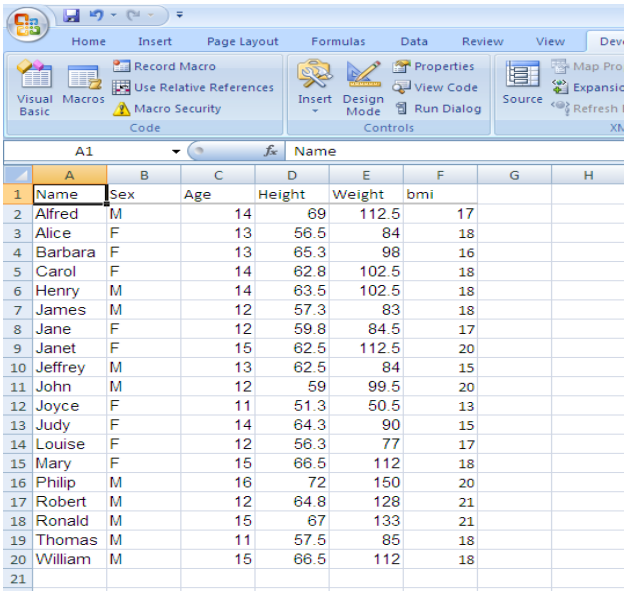
```
Private Sub Workbook_BeforeClose(Cancel as Boolean)
      If Not (obSAS Is Nothing) Then
            obWorkspaceManager.Workspaces.RemoveWorkspace obSAS
```

```
            obSAS.Close
        End If
End Sub
```

This checks if a SAS Workspace exists and if it does it uses the Workspace Manager object to remove it.

THE FUNCTION IN ACTION

Figure 3 shows the bmi function in action running against the data held in SASHELP.CLASS



	A	B	C	D	E	F	G	H
1	Name	Sex	Age	Height	Weight	bmi		
2	Alfred	M	14	69	112.5	17		
3	Alice	F	13	56.5	84	18		
4	Barbara	F	13	65.3	98	16		
5	Carol	F	14	62.8	102.5	18		
6	Henry	M	14	63.5	102.5	18		
7	James	M	12	57.3	83	18		
8	Jane	F	12	59.8	84.5	17		
9	Janet	F	15	62.5	112.5	20		
10	Jeffrey	M	13	62.5	84	15		
11	John	M	12	59	99.5	20		
12	Joyce	F	11	51.3	50.5	13		
13	Judy	F	14	64.3	90	15		
14	Louise	F	12	56.3	77	17		
15	Mary	F	15	66.5	112	18		
16	Philip	M	16	72	150	20		
17	Robert	M	12	64.8	128	21		
18	Ronald	M	15	67	133	21		
19	Thomas	M	11	57.5	85	18		
20	William	M	15	66.5	112	18		
21								

Figure 3 The Function in Action

CREATING AN ADD-IN

The previous technique is fine for relatively simple cases where you have a limited number of inputs and a single output value but if you want to create something more sophisticated some slightly different methods will need to be employed. It could be, for instance, that you wish to allow your users to run a SAS PROC such as PROC FORECAST or PROC SURVEYSELECT. These have a large number of options and parameters, some of which are incompatible with each other and also use SAS data sets for input and output. For these I recommend creating an Excel Add-in. You will need to create a UserForm for this and there are two options here

- Create a single-page form limiting the options the user can choose
- Create a wizard-type interface using the Excel multi-tab object guiding the user through the building up of selected options.

For the example we will be exploring here a single-page form is appropriate. We would use the wizard design for a procedure or macro where there were a large number of parameters and we wanted to guide the user through the process of selecting those parameters. An example of the single-page form is shown in Figure 4.

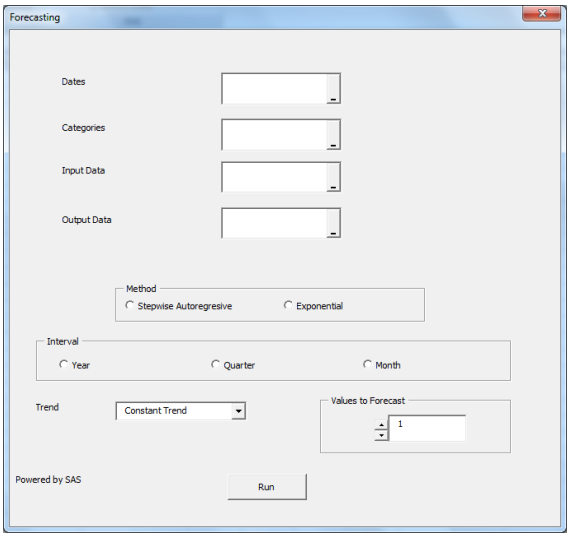


Figure 4 Sample VBA UserForm

In order to build an add-in like this we will need to use some different techniques with ADO to those shown earlier. Specifically we need to know how to

- Create a new SAS data set
- Populate it with data from an Excel Range
- Run the PROC or other piece of SAS code to produce the output
- Import the output data from a SAS data set into Excel

PRELIMINARIES

For this example we will create an add-in which will run the SAS PROC STANDARD. This standardises variables to a given mean and standard deviation. The VBA editor provides a simple drag and drop interface for creating UserForms but for reasons of brevity I will not show in detail how to create the UserForm in this example – this can be learnt from any of the excellent books on VBA available or from free resources on the Web. The form we will be using is shown in Figure 5 as seen in the VBA editor.

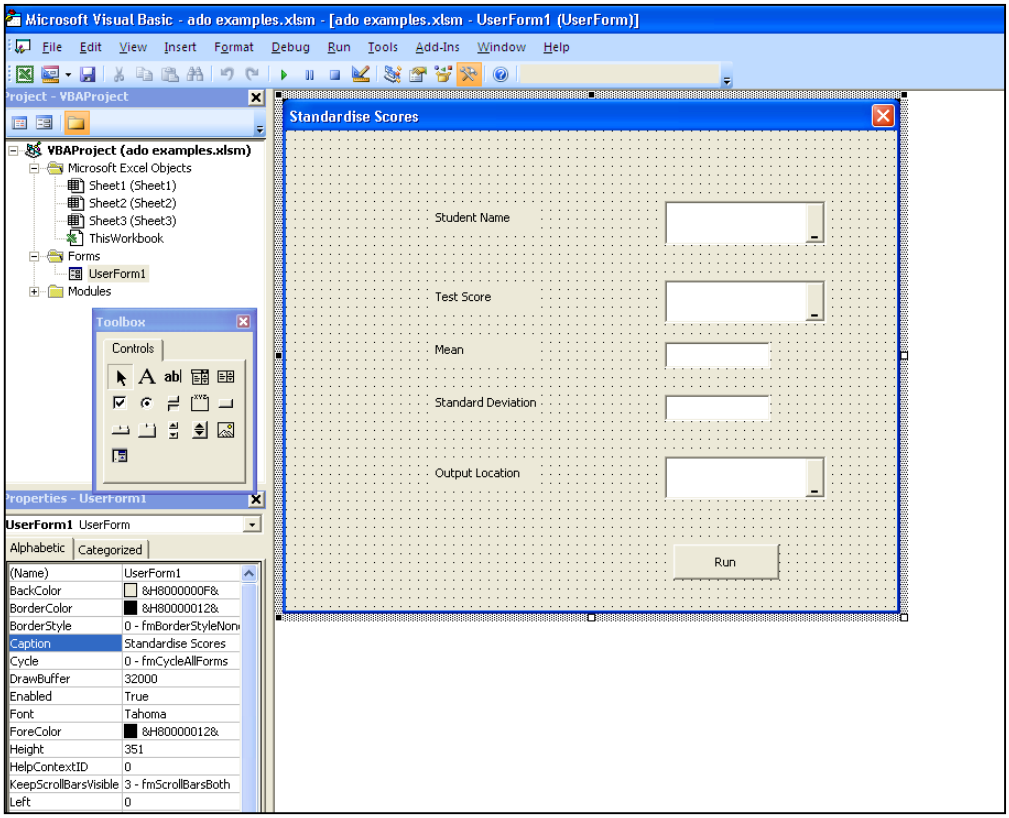


Figure 5 VBA UserForm in the VBA Editor

The relevant controls are named as followed (from top to bottom on the right hand side of the form)

- redStudent – a RefEdit control for the range holding the student name
- redTestScore – a RefEdit control for the range holding the scores to be standardised
- txtMean – a textbox control holding the mean to be used
- txtStdDev – a textbox control holding the standard deviation to be used
- redOutputLoc – a RefEdit control for the range holding the output data
- btnRun – a commandbutton control for the run button

It should be noted that an additional step is required to allow the RefEdit control to be used (this is a control which allows the user to select a range of cells). In order to make this available from the toolbox select Tools->Additional Controls from the VBA Developers menu and put a check in the box entitled “RefEdit. Ctrl”. You will, of course, also need to add the references mentioned in the previous example.

You then need to double-click the btnRun button to open the VBA editor which will automatically create a Procedure called btnRun_Click for you. This is where we will place our code.

MAKING THE CONNECTION – STARTING ADO

This is very similar to the Excel function code we saw earlier except we will be placing our code inside the btnRun_Click procedure so that it executes when the user clicks on the run button i.e.

```
' The Option Explicit statement in VBA forces us to declare variables
' by using Dim or ReDim statements rather than allow for implicit
' declarations. It is highly recommended you use this.
```

```
Option Explicit

' We will declare these next two variables as Public in scope so that
' they are available throughout the workbook

Public obSAS As SAS.Workspace
Public obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager

Private Sub btnRun_Click()
    Dim obConnection As New ADODB.Connection
    Dim errorString As String
    Dim sourcebuffer As String
    Dim obcommand As New ADODB.Command
    Dim rs As New ADODB.Recordset
    Dim scell As Range
    Dim student() As String
    Dim test() As Double
    Dim mean As Double
    Dim std As Double
    Dim i As Integer

    obConnection.Provider = "sas.IOMProvider.1"
    obConnection.Open
```

You will note that in addition to some variables we declare two arrays – student() and test(). These will hold the input data prior to it being loaded into the SAS data set. These are declared without dimensions as we do not know until run-time how many observations we will be processing. Also instead of opening the SAS Workspace session first we open the ADO connection first.

ASSIGNING A LIBNAME AND CREATING THE INPUT SAS® DATA SET

In order to assign a SAS Libname and create the SAS data set we will be using an ADO Command Object and the ADO Connection Object. The Connection object is used to create and maintain the connection with SAS while the Command Object is used to send SQL and other commands to SAS to create data sets and insert, update, delete or retrieve records. Firstly we set the Command's activeconnection property to the Connection we opened earlier

```
obcommand.ActiveConnection = obConnection
```

We then tell the command object that the command type we will be using will be text (there are a number of other types but these are not usually relevant when working with SAS in this way).

```
obcommand.CommandType = adCmdText
```

Next we set the value of the commands commandtext property to the libname statement we wish executed and call the command object's execute method, this will assign the libref in the SAS session.

```
obcommand.CommandText = "libname adotest 'd:\stnd'"
obcommand.Execute
```

Having assigned our Libref we can now create an empty data set with the correct structure. To do this we switch to the Connection object and it's execute method, passing a standard SQL command i.e.

```
obConnection.Execute "create table adotest.scores _
       (student char(15), test num);"
```

POPULATING THE SAS® DATA SET

Having created our empty data set our next task is to insert data from Excel into it. This is a multi-stage process – we have two columns of input data representing the student's names and test scores so we will first load them into the arrays declared earlier

```
' Redimension the array to the size of the number of rows in the selection
ReDim student(Range(RedStudent.Value).Rows.Count)
i=0
For Each scell In Range(RedStudent.Value)
       student(i) = scell.Text
       i = i + 1
Next

' Load the student scores into an array
i = 0

' Redimension the array to the size of the number of rows in the selection
ReDim test(Range(RedTestScore.Value).Rows.Count)

For Each scell In Range(RedTestScore.Value)
       test(i) = scell.Text
       i = i + 1
Next
```

Now we loop through the arrays building up and executing SQL insert statements thus loading the SAS data set

```
' Loop through the arrays building up and executing insert statements
For i = 0 To (UBound(student) - 1)
```

```
        obcommand.CommandText = "insert into adotest.scores values(""" & _
              student(i) & """," & test(i) & ");"

        obcommand.Execute

Next
```

EXECUTING THE SAS® PROCEDURE

Now that we have our input data we can run the SAS Procedure necessary to generate the output. In addition to the input data set we have just created we will be using the values the user entered onto the form for the mean and standard deviation to be used. We can use the same techniques we used in the first example (the bmi function) except that as there are unlikely to be multiple runs of this Procedure during a session we will not need to keep the SAS Workspace open after it has done its job.

```
Set obSAS =
obWorkspaceManager.Workspaces.CreateWorkspaceByServer("MyWorkspaceName", _
       VisibilityProcess, Nothing, "", "", errorString)

sourcebuffer = "libname adotest 'd:\stnd';"

obSAS.LanguageService.Submit sourcebuffer

sourcebuffer = "PROC STANDARD data=adotest.scores out=adotest.stndtest _

        mean=" & txtMean.Value & " std=" & txtStdDev.Value & ";run;"

obSAS.LanguageService.Submit sourcebuffer
```

LOADING THE DATA FROM SAS® INTO THE WORKSHEET

Now that we the output data from PROC STANDARD in a SAS data set all that remains is for us to retrieve that data, place it in the output range selected by the user from the UserForm and close SAS.

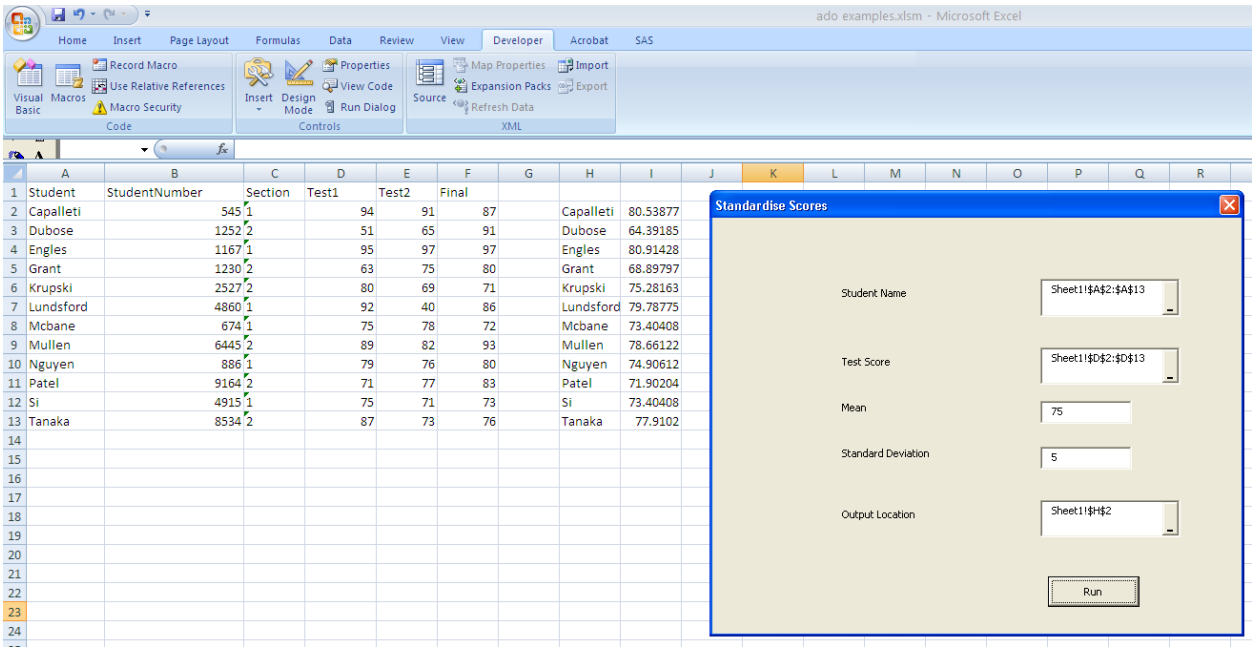
This is a very straightforward process – all we need to do is issue an SQL select command against the output SAS data set using the connection objects execute method so that the data is loaded into the recordset object which we have called rs. Then we will use the range's copyfromrecordset method to load the data from the recordset into the selected range. One thing to note is that the refedit object does not return a range object as its value or text (it holds a textual representation of the range address). Therefore in order to use the value entered in the refedit object we must wrap it inside the range function.

```
Set rs = obConnection.Execute("select * from adotest.stndtest")

Range(RedOutputLoc.Text).CopyFromRecordset rs
```

Having loaded the output into Excel we remove the Workspace object and close SAS

```
obWorkspaceManager.Workspaces.RemoveWorkspace obSAS

obSAS.Close
```

Figure 6 shows the UserForm having been completed and run and shows the output in the selected range.



Student	StudentNumber	Section	Test1	Test2	Final			
Capalleti	545	1	94	91	87	Capalleti	80.53877	
Dubose	1252	2	51	65	91	Dubose	64.39185	
Engles	1167	1	95	97	97	Engles	80.91428	
Grant	1230	2	63	75	80	Grant	68.89797	
Krupski	2527	2	80	69	71	Krupski	75.28163	
Lundsford	4860	1	92	40	86	Lundsford	79.78775	
Mcbane	674	1	75	78	72	Mcbane	73.40408	
Mullen	6445	2	89	82	93	Mullen	78.66122	
Nguyen	886	1	79	76	80	Nguyen	74.90612	
Patel	9164	2	71	77	83	Patel	71.90204	
Si	4915	1	75	71	73	Si	73.40408	
Tanaka	8534	2	87	73	76	Tanaka	77.9102	

Figure 6 The Standardise Scores Add-in in Action

DISTRIBUTING YOUR FUNCTION OR ADD-IN

Once you have completed and tested your function or add-in you have a number of ways of distributing it. For example you can

- Password protect your VBA project and send copies of the Excel file to your customers if you are distributing functions
- Password protect your project then add a button to the Excel toolbar which launches the UserForm and send copies of the file
- Create an add-in file by saving your spreadsheet as an XLAM file (to do this File->Save as>Other Formats from the main Excel menu then choose Excel Add-in from the “Save as Type” drop down box). You can then send the resultant XLAM file to your customers.

CONCLUSION

This paper has explored the construction of both a basic SAS/Excel function and add-in. However SAS IOM and ADO offer many more capabilities than can be explored here. It is hoped that the material in this paper will inspire you to explore these for yourself and create hybrid applications which offer your customers the best of both the SAS and Excel worlds without compromising on quality or ease of use.

ACKNOWLEDGMENTS

The author would like to thank Erik Tilanus for his assistance in preparing the paper under the SAS Global Forum presenter Mentoring Program.

CONTACT INFORMATION

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

Name: Chris Brooks

Company: Melrose Analytics Ltd

Address: 66 Melrose Avenue, Penylan, Cardiff, CF23 9AS, United Kingdom

Email: chrisbrooks@melroseanalytics.co.uk

Web: www.melroseanalytics.co.uk

TRADEMARKS

APPENDIX A – SOURCE CODE (EXAMPLE 1 BMI FUNCTION)

VBA CODE

```
'Option Explicit forces you to decalre all variables and is good practice

Option Explicit

'Declare SAS variable objects we will be using during the project

Public obSAS As SAS.Workspace
Public obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager

Public Function bmi(height As Variant, weight As Variant)

    'Declare some VBA and ADO objects we will be using

    Dim obConnection As New ADODB.Connection
    Dim obRecordSet As New ADODB.Recordset
    Dim errorString As String
    Dim sourcebuffer As String

    'Set up the SAS code to call - it isn't strictly necessary to use a buffer but it
can help if you need to debug the code

    sourcebuffer = "options sasautos=(sasautos,'d:\bmi');%bmi(" & height & "," _
       & weight & ");"

    'Check if an instance of the SAS object already exists. If it doesn't then create
    a local one

    If (obSAS Is Nothing) Then
        Set obSAS = _
obWorkspaceManager.Workspaces.CreateWorkspaceByServer("MyWorkspaceName", _
VisibilityProcess, Nothing, "", "", errorString)
    End If

    'Submit the SAS code previously loaded into the buffer

    obSAS.LanguageService.Submit sourcebuffer

    'Open a Microsoft Activex Data Object Connection for the current SAS session

    obConnection.Open "provider=sas.iomprovider.1; SAS Workspace ID=" + _
obSAS.UniqueIdentifier

    'Open the result data set

    obRecordSet.Open "work.result", obConnection, adOpenStatic, adLockReadOnly, _
adCmdTableDirect

    'Move the value of the result into the ADO  record set

    obRecordSet.MoveFirst

    'Return the value to the cell containing the formula

    bmi = obRecordSet(0).Value

End Function
```

```
Private Sub Workbook_BeforeClose(Cancel as Boolean)
       If Not (obSAS Is Nothing) Then
              obWorkspaceManager.Workspaces.RemoveWorkspace obSAS
              obSAS.Close
       End If
End Sub
```

BMI MACRO

```
%macro bmi(height,weight);
      data result;
            result=round(((&weight/&height**2)*703),1);
      run;
%mend bmi;
```

APPENDIX B – SOURCE CODE (EXAMPLE 2 ADD-IN)

VBA CODE

```
' The Option Explicit statement in VBA forces us to declare variables
' by using Dim or ReDim statements rather than allow for implicit
' declarations. It is highly recommended you use this.

Option Explicit

' We will declare these next two variables as Public in scope so that
' they are available throughout the workbook

Public obSAS As SAS.Workspace
Public obWorkspaceManager As New SASWorkspaceManager.WorkspaceManager

Private Sub btnRun_Click()
    Dim obConnection As New ADODB.Connection
    Dim errorString As String
    Dim sourcebuffer As String
    Dim obcommand As New ADODB.Command
    Dim rs As New ADODB.Recordset
    Dim scell As Range
    Dim student() As String
    Dim test() As Double
    Dim mean As Double
    Dim std As Double
    Dim i As Integer


    obConnection.Provider = "sas.IOMProvider.1"
    obConnection.Open


    ' Set obSAS =
obWorkspaceManager.Workspaces.CreateWorkspaceByServer("MyWorkspaceName", _
VisibilityProcess, Nothing, "", "", errorString) _
    obcommand.ActiveConnection = obConnection

    obcommand.CommandText = "libname adotest 'd:\stnd'"
    obcommand.Execute


    ' Create a new table in the Sasuser library.
    obConnection.Execute "create table adotest.scores ( student char(15), test num);"

    ' Load the student names into an array
    ' Redimension the array to the size of the number of rows in the selection
    ReDim student(Range(redStudent.Value).Rows.Count)

    i = 0
    For Each scell In Range(redStudent.Value)
        student(i) = scell.Text

        i = i + 1
    Next

    ' Load the student scores into an array
```

```
    i = 0

    ' Redimension the array to the size of the number of rows in the selection

     ReDim test(Range(redTestScore.Value).Rows.Count)

     For Each scell In Range(redTestScore.Value)
        test(i) = scell.Text
        i = i + 1
     Next


    ' Loop through the arrays building up and executing insert statements
    ' Loop through the arrays building up and executing insert statements
    For i = 0 To (UBound(student) - 1)
        obcommand.CommandText = "insert into adotest.scores values(""" & student(i) _
& """," & test(i) & ");"
        obcommand.Execute
    Next


    ' Start SAS and submit a libname statement

    Set obSAS =
obWorkspaceManager.Workspaces.CreateWorkspaceByServer("MyWorkspaceName", _
VisibilityProcess, Nothing, "", "", errorString) _
    sourcebuffer = "libname adotest 'd:\stnd';"
    obSAS.LanguageService.Submit sourcebuffer


    ' Run the required Proc passing in the required parameters

    sourcebuffer = "PROC STANDARD data=adotest.scores out=adotest.stndtest mean=" _
& txtMean.Value & " std=" & txtStdDev.Value & ";run;"

    obSAS.LanguageService.Submit sourcebuffer

    ' Get the SAS data into a recordset and copy it into the output range

    Set rs = obConnection.Execute("select * from adotest.stndtest")
    Range(redOutputLoc.Text).CopyFromRecordset rs


    ' Close down SAS

    obWorkspaceManager.Workspaces.RemoveWorkspace obSAS
    obSAS.Close

End Sub
```