Paper 012-2011

Tips and Techniques for Automating the SAS® Add-In for Microsoft Office with Visual Basic for Applications

Tim Beese, SAS Institute Inc., Cary, NC

ABSTRACT

Do you want to run SAS® Stored Processes in Microsoft Excel and set the prompt values from your worksheet? Do you want to filter your SAS data sets based on the values of specific cells? Do you want to customize your Microsoft Office content with buttons and other controls that allow you to open data, run SAS code and refresh results? This is now possible!

The SAS® Add-In 4.3 for Microsoft Office adds many new features that allow users to interact with their content through Visual Basic for Applications (VBA). Users can insert and refresh data, stored processes, and reports using VBA. It is also possible to provide prompt values to stored processes, filter and sort strings for data, and control where results are displayed. Combining these features with the existing functionality provided by VBA in Microsoft Office, integration of SAS within Microsoft Office will become easier and more powerful!

WHAT IS SCRIPTING?

Scripting is the ability to write programmatic steps to reproduce a series of operations on an application. Microsoft Office is scriptable, featuring a rich interface that can be accessed from VBA. The SAS Add-In for Microsoft Office also has an interface that users can access from their VBA scripts in Microsoft Excel, Word, and PowerPoint.

The SAS Add-In for Microsoft Office has had a scriptable interface for several releases, but the functionality was limited. The “Create Schedule” feature of the SAS Add-In for Microsoft Office creates a script and schedules it using the Microsoft Windows Task Scheduler. When the task runs, the script executes the SAS Add-In for Microsoft Office code and the content is refreshed.

WHY IS SCRIPTING USEFUL?

The purpose of scripting is to make your job easier. If you have a series of steps that you do over and over that involve manipulating SAS content in a Microsoft Office document, then writing a script can make that process easier.

Suppose you have a process in place to import sales data each morning into your Microsoft Excel workbook. Once you have that data available, you copy it to your SAS server so that it is available to everyone. Then you have a series of three stored processes that you run. These stored processes analyze the data and provide specific information about it. Reproducing these steps each day is not difficult, but it can be time consuming. With a VBA script, you can automate all of these steps instead of having to manually perform each step.

Scripting can also make other people’s work easier. Users who understand VBA very well and who know which SAS Add-In for Microsoft Office code must run in order to perform some specific analysis can define macros in their Microsoft Office document. These macros connect to SAS and interact with the SAS Add-In for Microsoft Office. Users can also create buttons or menu items in Microsoft Office that invoke their macros. Then, the end users who receive the Microsoft Office document simply click the right buttons, and everything will be started for them!

HOW DO YOU SCRIPT THE SAS ADD-IN FOR MICROSOFT OFFICE

To script the SAS Add-In for Microsoft Office, you must first go to the Visual Basic Editor in Microsoft Office. In Microsoft Office 2003, select **Tools > Macros > Visual Basic Editor**. In Microsoft Office 2007 and 2010, from the **Developer** tab, click **Visual Basic**.

The Developer tab does not appear by default. To display this tab in Microsoft Office 2007, you go to the Excel Options dialog and select the Popular page. From there you select “Show Developer tab in the Ribbon”. In Microsoft Office 2010 you go to the Excel Options dialog and choose the Customize Ribbon page. On this page you can select the Developer tab as one that should be shown. You may also use Alt+F11 to bring up the Visual Basic Editor directly.

Once you have the Visual Basic Editor open, you will want to insert a new module. Select the VBAProject node in the Explorer view, and then choose **Insert > Module**. A new Module node is added to the tree under your project. This is where you can insert your VBA code.

In order to use Intellisense within the VBA editor, you must add a reference to the SAS Add-In for Microsoft Office. To do this, select **Tools > References**. Find SAS Add-In 4.3 for Microsoft Office in the list, and select the check box next to it. Now you have a reference to the SAS Add-In for Microsoft Office and you are ready to begin writing VBA code.

Now is a good time to test things out and make sure that your connection to the API is working. You can write a simple VBA script to verify that you are successfully interacting with the SAS Add-In for Microsoft Office. Enter the following code into the VBA Editor, and then run the macro (by pressing F5):

```
01      Sub TestConnectionToSas()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          sas.HelloWorld
05      End Sub
```

When you run this macro, you will tell the SAS Add-In for Microsoft Office to display a message box that simply reads, “Hello World”. If you can see this message , then you are successfully calling into the SAS Add-In for Microsoft Office.

Let’s look at this simple example a little closer. You will very likely be using the syntax on lines 2 and 3 very often. This is how you get your connection to the API in the SAS Add-In for Microsoft Office. On line 2 you are defining a variable of type SASExcelAddIn. This is the main object that you will be making your calls against while automating the SAS Add-In for Microsoft Office.

Line 3 is where we get our reference to the object. To get this, we must access the list of all the COM Add-Ins that are currently loaded in Microsoft Office. We know that ours has a progId of “SAS.ExcelAddIn”, so we can search for that COMAddIn. The search returns us a COMAddIn object, which is the Microsoft Office wrapper for the SAS Add-In for Microsoft Office. To get directly to the SAS Add-In for Microsoft Office object behind the wrapper, you must add the “.Object” property to the call so that you can get directly to the SAS API.

When you are working with Microsoft Word you will define a SASWordAddIn object, and in Microsoft PowerPoint you will define a SASPowerPointAddIn object. Similarly, when you look for the item in the list of COM Add-Ins, you will look for SAS.WordAddIn and SAS.PowerPointAddIn.

SCRIPTING A REFRESH

Let’s do something a little more involved. In the past several releases of the SAS Add-In for Microsoft Office, you have been able to script a refresh of SAS content, though you might not have been aware of this ability. You can call refresh for any of the SAS content in your Microsoft Office document (except for content displayed in the OLAP Viewer). There are different ways to do this. Let’s look at an example that shows many ways to refresh different content in the Microsoft Office document.

```
01      Sub RefreshSasContent()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          sas.Refresh
05          sas.Refresh ThisWorkbook
06          sas.Refresh ThisWorkbook.ActiveSheet
07          sas.Refresh Sheet1.Range("A1")
08          sas.Refresh "Bar_Chart"
09      End Sub
```

The preceding code will refresh the same content five different ways. Before you can run this macro and refresh the content, you must create some content in your Excel workbook. For this example I ran a Bar Chart task and placed the contents in cell A1 of Sheet1. As I run this macro, I can look at the footnote and see the time get updated each time I call refresh.

Lines 2 and 3 should look familiar; they are getting a reference to the SAS Add-In for Microsoft Office API from Excel. Line 4 simply calls Refresh with no parameters. This will refresh everything that the SAS Add-In for Microsoft Office can find, in every workbook. If you have four workbooks open and each one has SAS content, everything will get refreshed.

Line 5 refreshes everything in the workbook where this macro is defined. The "ThisWorkbook" syntax returns an Excel workbook object corresponding to the workbook where your macro was added. With this call, all the SAS content on any sheet in this workbook will be refreshed.

Line 6 refreshes everything on the active worksheet. Again, we use the ThisWorkbook syntax to get an Excel workbook object, and then access the ActiveSheet property to get an Excel worksheet object. All of the SAS content that appears on the worksheet will be refreshed.

Line 7 refreshes the SAS content that exists in the given range. This uses the "Sheet1" syntax to get a reference to the Excel Worksheet object defined as Sheet1, and then uses the Range method to get an Excel Range object for cell A1. This is the equivalent to selecting the given cell and pressing the Refresh button on the SAS Ribbon in Microsoft Office.

Line 8 refreshes a specific SAS result. Each result that the SAS Add-In for Microsoft Office renders in the Microsoft Office document has an object name, and you can pass this object name to the Refresh method. To find out what the object name is, simply place the selection inside the SAS content, and then select **Properties** from the SAS Ribbon. The "Object Name" will appear on the General tab of the Properties dialog box. If you are not sure where your content is, you can select **Manage Content** from the SAS Ribbon to see all of the content in the active Microsoft Office document, and you can view the properties from there.

The Refresh functionality has been available for the past several releases. However, with the SAS Add-In 4.3 for Microsoft Office, you can also call Modify instead of Refresh. All of the same options are available for which content to modify. The difference is that instead of simply refreshing the content, the script will display any prompts for the content, just as if you had selected **Modify** from the SAS user interface.

USING VBA TO INSERT SAS DATA INTO YOUR WORKBOOK

Now let's look at some more of the new functionality in the SAS Add-In 4.3 for Microsoft Office that enables you to insert new content into your Microsoft Office document. The different types of content that you can open directly into your Microsoft Office document from VBA are data sets (in both tabular or PivotTable view), cubes, stored processes, and SAS reports. Once this content is in your Microsoft Office document, you can still refresh it as shown in the preceding section.

Note: You cannot open information maps or tasks via scripting with this release. Also, opening data sets and cubes is only supported in Microsoft Excel.

Let's start with a simple example of opening a data set into an Excel workbook:

```
01     Sub InsertSasDataSet()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         Dim data As SASDataView
05         Set data = sas.InsertDataFromLibrary("SASApp", "SASHELP", _
06                     "ZIPCODE", Sheet1.Range("A1"), 25, _
07                     "STATECODE = 'NV'", "City ASC")
08         MsgBox data.DisplayName
09     End Sub
```

Running this macro will insert a data set into the worksheet. Let's break this down line by line and see what each one does, starting with line 4. This defines a SASDataView object. When we insert the data, we will get back a SASDataView object that will enable us to continue to work with this data. On lines 5 through 7, we have the call that inserts the data. The first three parameters, SASApp, SASHELP, and ZIPCODE, define the server, library, and data set that we want to open. We also must provide a location for where to put the data. In this example we use a familiar syntax to tell the SAS Add-In for Microsoft Office to put it in cell A1 on Sheet1.

The rest of the parameters are optional, and if they are not provided, the SAS Add-In for Microsoft Office will use the default values for them. The next parameter is the page size. This determines the number of records to display at once. By choosing 25, I will get only 25 rows of data when I open the data set. The next parameter is the filter to apply to the data set. This is the filter that will be used when determining which rows to open. To see examples of the filter format, simply open any data set through the SAS Add-In for Microsoft Office user interface and use the **Filter/Sort** button to apply a filter through the dialog box provided. Then bring up the Properties dialog box for the resulting data view, and on the **Data** tab you will see a "Filter description" field that will show the filter that was

applied. For this filter, we only want records where the STATECODE column is equal to NV. The last parameter is the Sort that should be applied to the data. For this, you simply put the name of the column to sort by, and then either ASC or DESC to designate ascending or descending. If you do not provide a sort order, it will default to ascending order. To apply multiple sorts, separate them with a comma.

Finally on line 8, we display a basic message to let the user know that the job is done. We can use the "DisplayName" property of the SASDataView object to show the name of the object that was just rendered.

MANIPULATING EXISTING SAS DATA IN YOUR WORKBOOK

Now that we have inserted the content, we can manipulate it through the user interface by refreshing, navigating, or changing the filter or sort on it. We can also do some of this through the API. Here is an example where we find the data set that already exists, change how we display it, and then refresh it:

```
01     Sub ChangeFilterAndPageSize()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         Dim data As SASDataView
05         Set data = sas.GetDataView("SASApp_SASHELP_ZIPCODE")
06         data.PageSize = 10
07         data.Filter = "STATECODE = 'NC'"
08         data.Refresh
09     End Sub
```

As before, we have to declare a new variable of type SASDataView on line 4, but instead of creating a new data view, we are going to get one that already exists in the workbook. There are a couple of different ways to do this. The easiest is to call "GetDataView" on the sas object and pass in the name of the content, as seen on line 5. This requires that you know the object name, which you can get from the Properties dialog box for those results. When this returns, you will have a SASDataView that you can work with.

On line 6, we are changing the page size to 10, and on line 7, we change the filter. When we change these properties we are replacing the existing value of that property, not adding to it. In order to see the results displayed in the worksheet, you will need to call refresh on it, as seen in line 8. When the Refresh method is called, the changes you have made to the SASDataView object are applied to the contents in the workbook, and then refreshed so your changes can take effect.

Let's look at another example using multiple data views in the same worksheet. Here is a macro that will insert two data sets, side by side, in an Excel worksheet:

```
01     Sub InsertTwoSasDataSets()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         sas.InsertDataFromLibrary "SASApp", "SASHELP", "CLASS", Sheet1.Range("A1")
05         sas.InsertDataFromLibrary "SASApp", "SASHELP", "CLASS", Sheet1.Range("H1")
06     End Sub
```

With this example, you will notice that we do not declare any SASDataView instances. This is fine since we are not going to use the object that is returned from the "InsertDataFromLibrary" call. Also, notice that the PageSize, Filter, and Sort parameters are omitted. When doing this, the default page size will be used, and the filter and sort will be left empty. After we run this macro, the two data views will both be in Sheet1.

Now we want to get references to these data views and change them. In the same workbook where we just added the two data sets side by side, we can run the following macro and change the page size as well as put a filter on the data so that one table shows only males and the other shows only females:

```
01     Sub UpdateDataViews()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         Dim list As SASDataViews
05         Set list = sas.GetDataViews(Sheet1)
06         For i = 1 To list.Count
07             Dim data As SASDataView
08             Set data = list.Item(i)
```

```
09              data.PageSize = 5
10              If i = 1 Then
11                  data.Filter = "Sex = 'M'"
12              Else
13                  data.Filter = "Sex = 'F'"
14              End If
15              data.Refresh
16          Next i
17      End Sub
```

Notice on line 4 we are declaring a variable called 'list' that is of type 'SASDataViews'. SASDataViews is a collection of SASDataView objects. It has a Count property to tell you how many items are in the collection, and an Item method that you can use to access an item from the collection. To get your SASDataViews object, you call GetDataViews on the sas object. This method takes a parameter for the location. This can be a Workbook, Worksheet, or Range object. In this case, we know that our content is on Sheet1, so we can use that for the parameter. If the data views were on different worksheets, we could use the 'ThisWorkbook' object to get everything in the workbook.

Now that we have the SASDataViews collection, we can do a for loop to perform some operation on each SASDataView in the collection. On lines 7 and 8, we define a new SASDataView object and get it from the collection, and then on line 9, we change the PageSize.

Lines 10 through 14 are an IF/ELSE block that sets the filter on the data view. This example applies one filter to the first data view, and a different filter to all other data views. Then on line 15, we call refresh for each data view so that the content in the worksheet will be updated.

In addition to being able to open data from a SAS library, the SAS Add-In for Microsoft Office is also able to open data from a SAS folder in metadata or from your local machine. These calls are almost the same as inserting from the library, but with some different parameters.

When inserting data from a SAS folder, instead of passing in the server, library, and data set names, you pass in the full metadata path to the data set. You can also choose to pass in the page size, filter, and sort parameters as well.

When inserting data from your local machine, instead of passing in the server, library, and data set names, you pass in the path on your machine to the data set. You can also choose to pass in the page size, filter, and sort parameters as well.

INSERTING PIVOT TABLES INTO YOUR EXCEL WORKBOOK

PivotTables work very much the same as data views. We are actually still opening a data set into the workbook, but instead of putting it in tabular form, we are putting it into a PivotTable. As a result of this view, you are not able to provide a Sort value or PageSize value to the PivotTable, because these options do not apply to a PivotTable. Here is an example of inserting a pivot table into your workbook:

```
01      Sub InsertPivotTable()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          sas.InsertPivotTableFromLibrary "SASApp", "SASHELP", "CLASS",_
05               Sheet1.Range("A1")
06      End Sub
```

Now suppose you want to add a filter to this PivotTable. Interacting with the pivot tables works very much like interacting with the data, but instead you will use SASPivotTable objects instead of SASDataView objects. Here is an example of providing a filter for this PivotTable:

```
01      Sub AddFilterToPivotTable()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          Dim pivot As SASPivotTable
05          Set pivot = sas.GetPivotTables(Sheet1).Item(1)
06          pivot.Filter = "Sex = 'F'"
07          pivot.Refresh
08      End Sub
```

With this macro, notice the difference in how we can get a reference to our SASPivotTable object. We could have called sas.GetPivotTable(*string*) and passed in the object name of the content we wanted as a *string*. But in this case I knew that there was just one pivot table in the worksheet, so we can call GetPivotTables and pass in the worksheet. This gives us back a list of one item, which we can then use to access the first item in the list. Once we have the SASPivotTable object we can set the filter and call refresh on it.

PivotTables can be opened from a library, a SAS folder, or from your local machine, just like data can be opened. In addition, you can open an OLAP cube into your PivotTable. When opening an OLAP cube, you are not allowed to specify the filter or sort, since OLAP cubes do not provide a way to apply those.

```
01      Sub InsertPivotTable()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          sas.InsertPivotTableFromOlap "SASApp", "Foundation", "CustomerSummary",
Sheet1.Range("A1")
05      End Sub
```

When inserting the PivotTable from an OLAP cube, you must provide the name of the OLAP server, the catalog, and the name of the cube that you want to open, as well as the location for where to put the PivotTable.

INSERTING SAS STORED PROCESSES

Another type of content that you can add to your Microsoft Office documents is SAS Stored Processes. These are usable in Microsoft Excel, Word, or PowerPoint, whereas data and pivot tables are accessible only in Excel. Here is a simple example of adding a stored process to your document:

```
01      Sub InsertStoredProcess()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          sas.InsertStoredProcess "BIP Tree/Tim/Battle", Sheet1.Range("A1")
05      End Sub
```

When inserting the stored process, all that you need to provide is the full metadata path to where the stored process lives, and the location where you want to put the results. When you are running in Word or PowerPoint, you cannot specify a location for the results. The results will be inserted at your current selection point or slide. To make sure that you put your results where you want them to go, you can simply use VBA to activate the correct slide, or place the cursor at the position you want.

Many stored processes also have prompts associated with them. Using the API provided by the SAS Add-In for Microsoft Office, you can provide the prompts for a stored process. To do this you will need to understand how the SASPrompts object works. Consider this example:

```
01      Sub InsertStoredProcessWithPrompts()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          Dim prompts As SASPrompts
05          Set prompts = New SASPrompts
06          prompts.Add "division", "AFC West"
07          Dim a1 As Range
08          Set a1 = Sheet1.Range("A1")
09          sas.InsertStoredProcess "BIP Tree/Tim/NFL 2009", a1, prompts
10      End Sub
```

When inserting a stored process, you can provide a SASPrompts object to pass along to it. As you can see on lines 4 and 5, we define and then create a new SASPrompts object. Then, on line 6 we add a new item to the list. When adding prompts through the scripting interface, you must know the name of the prompt (which you can find through SAS® Enterprise Guide®) and the value you want to set it to. The SAS Add-In for Microsoft Office supports setting prompt values to a single value only through the automation interface, so if the stored process requires a list of prompt values, then it might not be a prompt that you can automate. Any prompts that are not provided by the macro will use their default values when executed. The prompt value is set on line 6. The name of the prompt is “division”, and we give it the value of “AFC West”.

Lines 7 and 8 show another way of choosing where to put the results. In this example, we define an Excel Range object, set its value, and then pass that object to the stored process. In the previous examples this was all done on one line.

You will see on line 9 that we pass the prompts object as the third parameter to the InsertStoredProcess method so that the prompts will be used initially.

CHANGING STORED PROCESS PROMPT VALUES

Once your stored process has been run and you want to refresh it, it is easier to change the prompt values. You no longer need to use the SASPrompts object, because you can change them on the SASStoredProcess object itself. See the following example:

```
01     Sub ChangeStoredProcessPrompts()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         Dim stp As SASStoredProcess
05         Set stp = sas.GetStoredProcesses(Sheet1).Item(1)
06         stp.SetParameter "division", "NFC South"
07         stp.Refresh
08     End Sub
```

On line 6, we call SetParameter on the SASStoredProcess object, and pass in the same name and value pair that we would have added to the SASPrompts object. Now, when we refresh the stored process, the new prompt value will be used.

STORED PROCESSES WITH OUTPUT PARAMETERS

Similar to providing prompts, another option when inserting a stored process is the ability to supply any output parameters and where they should be rendered. This is supported only in Microsoft Excel. If the stored process defines an output parameter, you can use the SASRanges object to tell the stored process where to put the value from the output parameter. Here is an example of a stored process that has both prompts and output parameters:

```
01     Sub InsertStoredProcessWithOutputParamters()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         Dim prompts As SASPrompts
05         Set prompts = New SASPrompts
06         prompts.Add "team", "Kansas City Chiefs"
07         prompts.Add "team2", "New Orleans Saints"
08         Dim outputParams As SASRanges
09         Set outputParams = New SASRanges
10         outputParams.Add "courtesyOf", Sheet1.Range("A18")
11         outputParams.Add "dateCreated", Sheet1.Range("A19")
12         Dim a1 As Range
13         Set a1 = Sheet1.Range("A1")
14         sas.InsertStoredProcess "BIP Tree/Tim/NFL Team Statistics", a1,
               prompts, outputParams
15     End Sub
```

In this example, you can see the creation of the SASPrompts on lines 4 through 7. The SASRanges object is much the same. We define the SASRanges object on line 8, and then create a new instance of it on line 9. We add items to it the same way as the SASPrompts, but instead of associating the name of the prompt with the value of the prompt, we are associating the name of an output parameter with an Excel Range object for where to render it.

On line 10, we are defining a range at A18 and associating it with an output parameter named “courtesyOf”. On line 11, we do the same with A19 and associate it with “dateCreated”. When we insert the stored process on line 14 and the results are rendered, the SAS Add-In for Microsoft Office will look through all of the output parameters. If we know where to render the output parameters, we will put the value there. If the stored process defines output parameters, but the script does not define a location for them, they will not be rendered. However, the rest of the stored process will execute as normal.

Similar to prompts, you can change the location where the output parameter is rendered when you refresh it. Say we want to move the output parameters from column A to column D. Here is the sample macro:

```
01      Sub MoveOutputParameters()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          Dim stp As SASStoredProcess
05          Set stp = sas.GetStoredProcesses(Sheet1).Item(1)
06          stp.SetExcelOutputParameterLocation "courtesyOf",
                Sheet1.Range("D18")
07          stp.SetExcelOutputParameterLocation "dateCreated",
                Sheet1.Range("D19")
08          stp.Refresh
09      End Sub
```

Because this is the only stored process on this worksheet, we can easily get to the SASStoredProcess object by getting the first item from the list of all the stored processes. Then, we simply call the SetExcelOutputParameterLocation method, which will reassign the location for the given output parameter.

STORED PROCESSES WITH INPUT STREAMS

Another feature that was added for stored processes in the SAS Add-In 4.3 for Microsoft Office is the ability to provide an input stream to a stored process. This is a good way for users to get their Excel data directly into the stored process. Consider the following data in Excel:

	A	B
1	Name	Age
2	Andrew	23
3	Bruce	21
4	Catherine	24
5	David	25
6	Ernie	25
7	Frank	22
8	Gertrude	24
9	Harry	22

Display 1. Sample Excel Data

If you want to send this content to a stored process that accepts input streams, you can do that through the user interface. You can also do that through the scriptable API on the SAS Add-In for Microsoft Office. Here is a sample of a macro that defines a SASRanges object to pass in the range to use as the input stream:

```
01      Sub InsertStoredProcessWithInputStreams()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          Dim streams As SASRanges
05          Set streams = New SASRanges
06          streams.Add "instream", Sheet1.Range("A1:B27")
07          sas.InsertStoredProcess "BIP Tree/Tim/InputStream",
                Sheet1.Range("D1"), , , streams
08      End Sub
```

We use the same SASRanges object for the input streams that we used for the output parameters. It maps the name of the input stream to the range of cells where the data is contained. With input streams, you need to reference the entire range, not just a single cell. As you can see on line 6, we use everything from A1 to B27 by using a ':' character to indicate a range of cells from top left to bottom right.

On line 7 when we insert the stored process, you will see that we have some empty parameters after we reference cell D1 for where to put the output. This is where the prompts and output parameters would go. However, there is nothing to pass for those. Therefore, we can leave those blank because they are optional.

Once you have the content in the workbook, you can change the input stream location by getting a reference to the SASStoredProcess object, and then calling the SetExcelInputStreamLocation method. This works just like the SetExcelOutputParameterLocation method, only it applies to input streams instead of output parameters.

INSERTING SAS REPORTS

The last type of content that I will show you how to create is SAS Reports. We can open SAS Reports from SAS® Web Report Studio or from your local machine. These are by far the simplest of all the content to insert. You only have to provide the full path to the report (whether it is in metadata or on your local machine) and indicate where to put the results (available in Excel only). Here is an example:

```
01      Sub InsertReport()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          sas.InsertReportFromSasFolder "BIP Tree/Tim/FirstReport.srx",
                Sheet1.Range("A1")
05      End Sub
```

The SAS Add-In for Microsoft Office does not provide the ability to script the setting of prompts for SAS Reports. If the SAS Report includes prompts, the user will be prompted for them.

CHANGING YOUR METADATA PROFILE THROUGH VBA

There are several other things that you can do with the SAS Add-In for Microsoft Office API. One of these things is to change which metadata profile you are connected to. This could be useful if you want to make sure that you are connected to the proper server before you try to insert your stored processes. Here is an example that shows how you can manipulate your metadata profile:

```
01      Sub SetProfile()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          MsgBox sas.ActiveProfile.Name
05          Dim profiles As SASProfiles
06          Set profiles = sas.profiles
07          For i = 1 To profiles.Count
08              Dim profile As SASProfile
09              Set profile = profiles.Item(i)
10              If (profile.Name = "labsrv01") Then
11                  Set sas.ActiveProfile = profile
12              End If
13          Next
14          MsgBox sas.ActiveProfile.Name
15      End Sub
```

First, notice that there is an ActiveProfile property on the sas object. This gives you back a SASProfile object, which can tell you the name, port, and host name properties of the active profile. On line 4, we simply display the name in a message box. On lines 5 and 6, we create a new SASProfiles object, which works very much like all of our other collection objects. It contains a list of SASProfile objects that you can iterate through.

In this example, we loop through all of the profiles, and on line 10, we compare the name of the profile to “labsrv01”. This is the metadata profile that we want to use for whatever content we are going to insert. If we find a match, then we will set the ActiveProfile property of the sas object to this profile.

Finally, on line 14 we show another message box that shows the name of the profile that is active. If you write a script that changes the active profile, you will be able to see the change in the two message boxes. When the active profile is changed from the automation interface, the change will not carry over to the next invocation of Microsoft Office. The change will show only for the duration of this Microsoft Office session.

SAVING DOCUMENTS TO SAS FOLDERS

The SAS Add-In for Microsoft Office enables you to save the Microsoft Office document to SAS folders. If you create an Microsoft Office document and want to make it available through the SAS folders, you can automate that process. Here is the code to save the document:

```
01     Sub SaveToSasFolders()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         sas.PublishDocumentToRepository ThisWorkbook,
               "BIP Tree/Tim/SugiPaper"
05     End Sub
```

As you can see, this is a very simple call, simply passing the document to save. In Excel this will be a workbook, in Word this will be a document, and in PowerPoint this will be a presentation. You must also provide the path to where you want to save the file. The correct extension will be placed on the document during the publish operation. When saving a document that contains macros to SAS folders, you will want to save the document to your local machine before doing the publish operation. By doing this, you force Excel to decide whether this file should be saved as an xlsx or xlsm file type. The default file extension is xlsx. If the document has macros, then it needs to be formatted such that the macros are retained; doing a local save achieves this.

SETTING YOUR SAS OPTIONS THROUGH VBA

All of the SAS Add-In for Microsoft Office options are settable from the automation interface as well. They are listed in the help, along with an example that shows how to set each of the values. Here is a small example that sets a few of the options:

```
01     Sub SetOptions()
02         Dim sas As SASExcelAddIn
03         Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04         sas.Options.ResetAll
05         sas.Options.Excel.ApplySASStyle = False
06         sas.Options.Excel.OpenOutputDataSets = True
07         sas.Options.Excel.UseRawValuesForSasReports = True
08         sas.Options.ImageFormat = ImageFormat_ActiveX
09         sas.Options.AutomaticallyDeleteTempFiles = False
10         sas.Options.SaveOnExit = True
11     End Sub
```

As you can see on lines 4 through 10, we are working with the Options property of the sas object. This is where all of our interaction with the options is done. On line 4, we tell the options to ResetAll. This is the same as clicking the **Reset All** button on the Options dialog box.

On lines 5 through 7, we are accessing options that are specific to Excel. When options are specific to one of the Microsoft Office clients, they will be grouped in the options. If the options are general and shared across all of the Microsoft Office clients, then they are accessed directly off of the Options property, as shown on lines 8 and 9.

Looking more closely at line 8, we are setting the ImageFormat to an enumeration value that specifies we should use ActiveX. There are several options in the SAS Add-In for Microsoft Office that use enumerations for the value. When these are set through VBA, the value must start with the enumeration type, in this case “ImageFormat”, then an underscore, and then the name of the value, which in this case is ActiveX. Putting all of those together gives us “ImageFormat_ActiveX”. The easiest way to set these is to use the Intellisense that is provided by the Visual Basic Editor in Microsoft Office, and it will complete this for you.

Finally, looking at line 10 we can specify whether to save the options when Microsoft Office exits. If you set this value to true, then these options will be automatically saved. If you set this value to false, then these options will not be saved. However, if the user brings up the Options dialog box manually and chooses OK, then all of the options that have previously been set with the macro will be saved along with the user’s manual changes to the options.

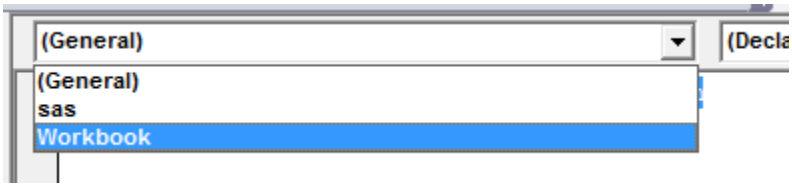
HANDLING THE ITEMUPDATED EVENT FROM THE SAS ADD-IN FOR MICROSOFT OFFICE

Whenever the SAS Add-In for Microsoft Office updates an item in the workbook, it will fire off an event. Therefore, any VBA macro listening to that event will get called and can do some processing of its own. This takes a little bit more work to set up, but is not much code.

To do this, you first open your Microsoft Office document and go to the Visual Basic Editor as usual. However, instead of adding a new module, go to the VBAProject tree and double-click **ThisWorkbook**. This will bring up an empty editor area to work with. Here you can define a SASExcelAddIn object as a member of the workbook, and write some VBA code so that when the workbook is opened, we will initialize a connection to it. First, create your member variable:

```
01      Private WithEvents sas As SASExcelAddIn
```

Next, you can use the menu to add an event handler for the WorkbookOpen event that Excel will fire when the workbook is opened.



Display 2. Choosing the Workbook Events

When you select **Workbook**, by default the Workbook_Open event handler will be added. You can also add it manually by typing it in yourself. Once you have the event added, you can initialize the sas variable so that can call into the SAS Add-In for Microsoft Office API.

```
01      Private WithEvents sas As SASExcelAddIn
02      Private Sub Workbook_Open()
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04      End Sub
```

Line 3 should look very familiar because it is the same way that we always initialize our sas variable. However, this time we have defined the object at the document level instead of making it local to whichever macro we are running.

Now we can add the event handler for the ItemUpdated event, which is fired from the SAS Add-In for Microsoft Office. To do so, return to the menu shown in the preceding display (where we selected Workbook), and select **sas**. Because SAS provides only one event, when you click **sas** the event handler is added. You can also add it manually.

At this point you can add a simple MsgBox call to the event handler just to make sure that everything is wired correctly. Before you can run this, you must save (as .xlsm in Microsoft Office 2007 and 2010), close, and reopen your document in order to make sure that the document recognizes the new variable that has been added to it, and to have the Workbook_Open event fire off to initialize it. Once you do that, you can run any job from the user interface, and when it finishes, it will call your event handler. Let's look at what we have completed so far:

```
01      Private WithEvents sas As SASExcelAddIn
02      Private Sub Workbook_Open()
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04      End Sub
05      Private Sub sas_ItemUpdated(ByVal item As Variant)
06          MsgBox "Event Received!"
07      End Sub
```

You will see the ItemUpdated event handler defined on line 5, and if you run a SAS job using the SAS Add-In for Microsoft Office, the MsgBox will be displayed. The 'item' parameter that gets sent to the ItemUpdated event handler is the object name of the item that was just updated.

Using this, we can begin to interact with the event a little better. Here is an example of what you could do in the event handler:

```
01      Private Sub sas_ItemUpdated(ByVal item As Variant)
02          Dim data As SASDataView
03          Set data = sas.GetDataView(item)
04          If data Is Nothing Then
05              Exit Sub
06          End If
07          Dim sheet As Worksheet
08          Set sheet = Application.ActiveSheet
09          sheet.Name = data.Dataset
10          MsgBox "Opened " + data.Dataset + " with Filter: " + data.Filter
11      End Sub
```

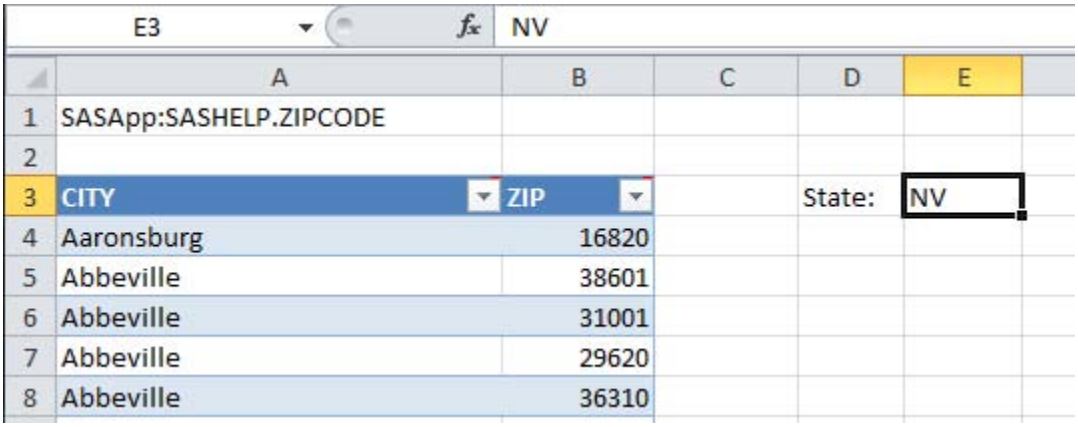
In this event handler, we are now determining whether the object name that is returned matches a SASDataView with that name, as seen on line 3. Lines 4 through 6 verify that we have a valid object to work with. On line 7, we determine what the active sheet is, and then set the name of that worksheet to the data set name. Finally on line 10, we display a message box that specifies which data set was opened and what the filter was.

If you go through the user interface to insert a data set, then you will see this in action. Or, if you want to insert a module in your VBAProject now, you can write a macro to insert new content, or refresh existing content.

INSERTING MICROSOFT OFFICE CONTROLS TO CALL YOUR VBA MACROS

As a last example, I will show you how to include Microsoft Office controls in your document that will help you call your macros and interact with Microsoft Office. Let's say that you have a worksheet that displays all the cities and their corresponding ZIP code.

You can change which cities you are looking at based on a particular state. You can use the **Filter & Sort** button to change your filter through the user interface, but with a quick macro, you could make the filter value come directly from your worksheet. You can add a cell to your workbook that contains the filter value that you want to use for your data.



E3 | fx | NV

	A	B	C	D	E
1	SASApp:SASHELP.ZIPCODE				
2					
3	CITY	ZIP		State:	NV
4	Aaronsburg	16820			
5	Abbeville	38601			
6	Abbeville	31001			
7	Abbeville	29620			
8	Abbeville	36310			

Display 3. Sample Data Set with Filter

The next step is to write a macro that takes the value from cell E3 and puts it into a filter for this data set. Here's what this macro would look like:

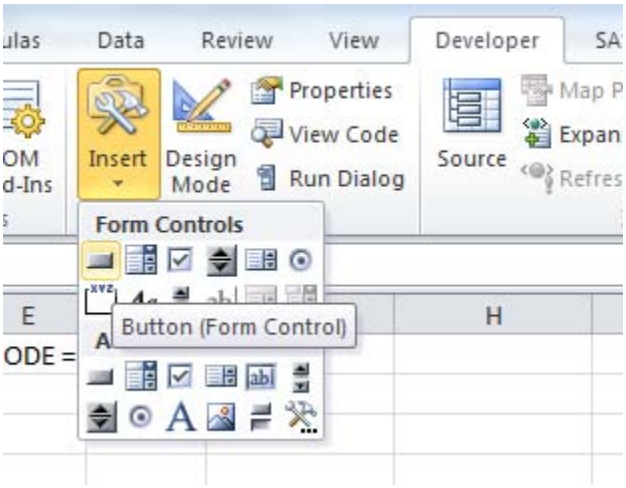
```
01      Sub UpdateDataWithCellFilter()
02          Dim sas As SASExcelAddIn
03          Set sas = Application.COMAddIns.Item("SAS.ExcelAddIn").Object
04          Dim data As SASDataView
05          Set data = sas.GetDataViews(Sheet1.Range("A3")).Item(1)
06          Dim state As String
07          state = Sheet1.Range("E3").Value
08          Dim filter As String
09          filter = "STATECODE = '" + state + "'"
10          data.filter = filter
```

```
11          data.Refresh
12      End Sub
```

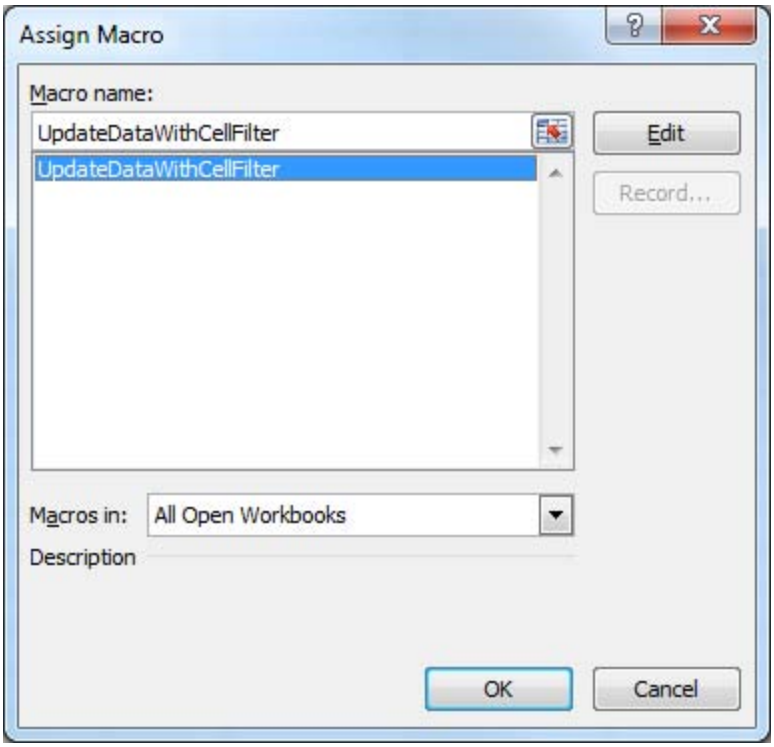
In this macro, we get a list of DataViews that exist in A3. There will be only one, so we know that we can access the first item in the list. This gives us our SASDataView object. To determine what we need to put in our filter, we define a string on line 6 and assign it on line 7. This reads the value from the cell and stores it in the state variable. On line 8 we define another string, which we will use to build our filter.

We want our filter to be STATECODE = 'NV'. We must include the single quotation marks around the string value that we are passing in as the filter. On line 9, we build this string, inserting the state code that was entered in E3 in the filter appropriately. Now, on line 11 we can call refresh, and the new filter will be applied. Then, the data will update in our workbook.

To make this really cool, we need an easy way to launch the macro. From the **Developer** tab in Excel you can add a button to the worksheet, and assign the button so that when it is clicked, it will call your macro. From the **Developer** tab, select **Insert**, and then choose a button. You can choose where to display the button and how large to make it. In the ensuing box, choose your macro from the list, and click **OK**. Now you have a button that will fire off your macro when you click on it.



Display 4. Choosing a Button from the Microsoft Excel Ribbon



Display 5. Wiring Your Macro to Your Button

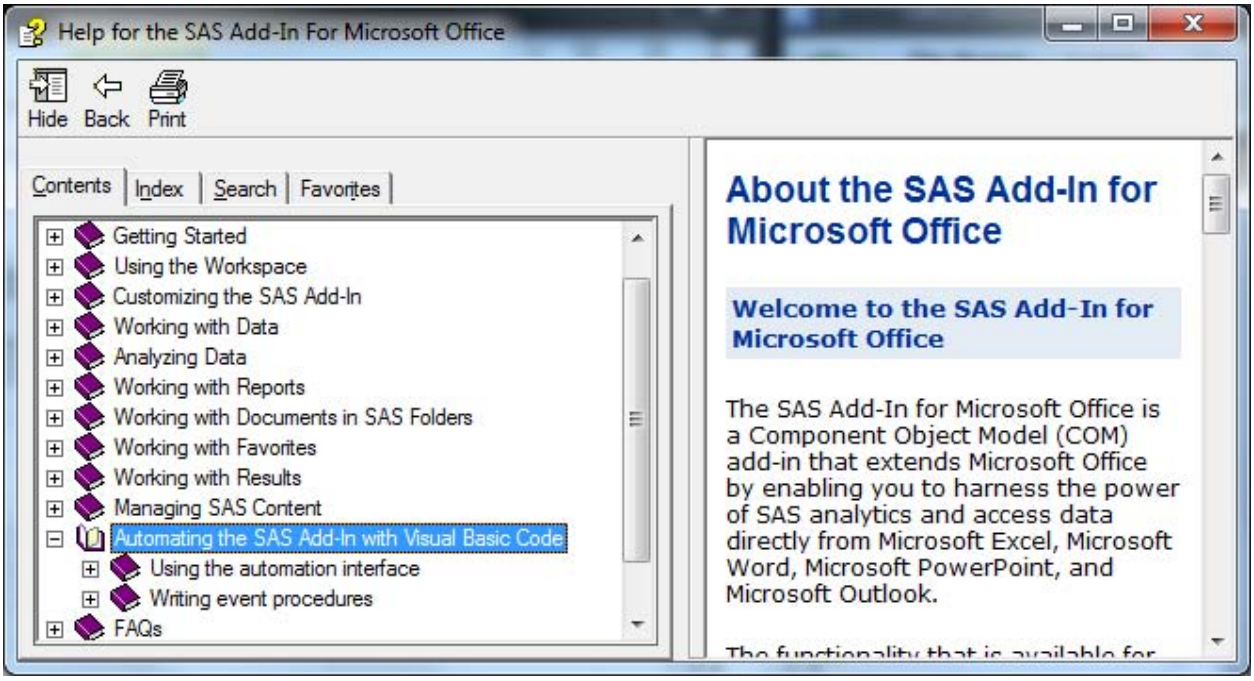
	A	B	C	D	E	F
1	SASApp:SASHELP.ZIPCODE filtered by STATECODE = 'NC'					
2						
3	CITY	ZIP		State:	NC	
4	Aberdeen	28315				
5	Advance	27006				
6	Ahoskie	27910				
7	Alamance	27201				
8	Albemarle	28001				

Update Filter

Display 6. Sample Worksheet with Filter and Button

LOCATING THE HELP FOR AUTOMATION IN THE SAS ADD-IN FOR MICROSOFT OFFICE

For more examples on automating the SAS Add-In 4.3 for Microsoft Office with Visual Basic for Applications, see the help that is delivered with the SAS Add-In for Microsoft Office. From the **SAS** tab on the Ribbon, select **Help**, and choose **Help for the SAS Add-In for Microsoft Office**. When the help is launched, see the section called "Automating the SAS Add-In with Visual Basic Code". This section of the help contains examples that show how to interact with all of the different objects that are defined by the SAS Add-In for Microsoft Office, as well as descriptions of all the properties and methods that are available.



Display 7. Where to Find Help for Automation

You may also use the Object Browser from the Visual Basic Editor to view the API. To view the Object Browser from the Visual Basic Editor, select **View** from the menu and then select **Object Browser**. The SAS Add-In for Microsoft Office objects will appear in the list of Classes.

Another location to find help for the SAS Add-In for Microsoft Office is on the SAS Web site. The following Web page has several resources that you can use. http://support.sas.com/documentation/onlinedoc/addin/index.html

By automating the SAS Add-In for Microsoft Office, you can customize and enhance your work environment by further integrating the power of SAS with your Microsoft Office documents.

CONTACT INFORMATION

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

Tim Beese
SAS Institute Inc.
SAS Campus Drive
E-mail: Tim.Beese@sas.com