

# Using VBA and BASE SAS to Get Data from SAS to Excel Without Data Integrity Issues

Timothy Adlington, AstraZeneca UK Ltd, Manchester, UK

## ABSTRACT

There have been many excellent papers detailing the transfer of data from SAS® to Microsoft Excel® using various methods such as DDE, ODS and OLE. However for some of these methods there is one aspect of the process which perhaps hasn't received enough attention. Unless you take some precautions Excel will quite happily corrupt your data without a second thought by applying it's own automatic formatting. This paper explains how to pump data into Excel using DDE & ODS with no loss of data integrity, and also how to utilise Excel's most powerful feature, it's Visual Basic for Applications programming language, to pull data from SAS. The paper concludes with the code required to create an Excel Add-In which enables a non-SAS user to access SAS datasets from within Excel without the intervention of the SAS programmer, and with no loss of data integrity.

## INTRODUCTION

There have been many articles over the years dealing with the interfacing of SAS and Excel such as . and I would urge the reader to study them if this topic is new to them. Since these papers do a very comprehensive job of covering the subject, I have no intention of going over the same ground here. I do however wish to focus on one aspect of the process which perhaps needs to be emphasized, that is taking control of Excel to ensure complete data integrity.

Working in early phase drug development where many varied and disparate information stakeholders want immediate access to the study data, it can seem that I spend more of my time dealing with Excel rather than SAS! Of primary concern is that when trying to send a SAS table to non-SAS users in the company via Excel, Excel will unfortunately apply a built-in number format to the data automatically, according to the Microsoft Knowledge Base Article ID 214233, based on the following criteria:

1. If a number contains a slash mark (/) or hyphen (-), it may be converted to a date format.
2. If a number contains a colon (:), or is followed by a space and the letter A or P, it may be converted to a time format.
3. If a number contains the letter E (in uppercase or lowercase letters; for example, 10e5), or the number contains more characters than can be displayed based on the column width and font, the number may be converted to scientific notation, or exponential, format.
4. If a number contains leading zeros, the leading zeros are dropped.

So if left to it's own devices we can expect the following output from Excel :

| Actual Intension                        | SAS Value | Excel Interpretation |
|-----------------------------------------|-----------|----------------------|
| % positively stained cells              | 1-5       | 01 – May *           |
| questionnaire responses / number issued | 60 / 90   | 2 / 3                |
| Protocol schedule time                  | 01:30     | 01:30:00             |

\* or more truly 38473 the Excel date serial value

How can the SAS programmer guard the integrity of their data once in Excel? And is using SAS to populate Excel the best approach?

The remainder of this paper deals with approaches available in Base SAS and Excel 2000 / 2003 on a windows platform. Other approaches are possible using SAS v9, the SAS Microsoft Office Add-In, SAS/ACCESS, SAS Integrated Object Model and XML (Vincent DelGobbo (2004) and Ted Conway (2005)) but these are out of scope for this discussion.

## REMOVING EXCEL'S INFLUENCE

In order to prevent Excel applying any automatic formatting routines, we need to specifically tell Excel to use it's "Text" format. This format forces Excel to treat each cell as just a string of text, hence we can be sure that no further

manipulations will be conducted on it. It's worth noting here that should you wish to apply a formula to a cell e.g. '=\$A\$1+\$B\$1' then you would NOT want to have this particular cell formatted as text since Excel won't resolve it.

The two principle methods of getting data directly into Excel from Base SAS are Dynamic Data Exchange (DDE) and the Output Delivery System (ODS) so here are a few quick examples of generating worksheets with the text format. The reader is directed to Koen Vyverman (2000), Koen Vyverman (2001), Koen Vyverman (2002) and Perry Watts (2004) for a more in depth discussion of DDE than is presented here.

### DYNAMIC DATA EXCHANGE

DDE is the direct communication between SAS and Excel using a server/client model. Excel acts as the server, and SAS as the client. Both applications need to be active for DDE to work, however this is no hardship since SAS can instigate an instance of Excel quite easily.

```
options noxwait noxsync;

x "c:\progra-1\microso-2\office\excel.exe c:\book1.xls";

data _null_;
x = sleep(10);
run;

filename xlcmds dde 'excel|system';
filename sasdata dde "excel|sheet1!r1c1:r10c2" notab;

data _null_;❶
file xlcmds;
put '[select("r1c1:r10c2", "r1c1")]';
put '[format.number("@")]';
run;

data _null_;
set sastable;
file sasdata;
put var1 '09'x var2 ;
run;
```

❶ After waking Excel, the first priority is to select the region to be populated with SAS data and set the format to "Text" using the *format.number* function and "@" as the text format identifier. After this step row 1, column 1 to row 10, column 2 have been formatted to "Text" in readiness for the population by SAS data in the next data step.

Note that DDE uses Excel version 4 macro language (X4ML) which has been superseded by Visual Basic for Applications (VBA) since the introduction of Microsoft Office 97, more of this later.

### OUTPUT DELIVERY SYSTEM

Using ODS is an alternative to DDE when dealing with tabulated output. The style option enables us to use the mso-number-format HTML style. Again, this needs to be set to "@" for the required variable. Steven Feder (2004) & Vincent DelGobbo (2003) demonstrate many useful approaches to using ODS.

```
ods html file='c:\report.xls';
proc report data=sastable nowindows;
column var1;
define var1 / display style={htmlstyle="mso-number-format:@"}; ❶
run;
quit;
ods html close;
```

❶ Use the STYLE option to set the mso-number-format to "@" as the text format identifier. This approach is variable specific, enabling all output for var1 to be correctly formatted. Other variables can have other formats defined.

### TAKING THE NEXT LOGICAL STEP – VBA

The above approaches work well, but traditionally they rely on the submission of a SAS program by the SAS programmer. How much more efficient it would be to give the information stakeholder the ability to import the data themselves without requiring any SAS knowledge, or even having SAS installed on their PC.

With a little knowledge and vision it's possible to use text files or ADO and VBA to enable a non-SAS user to access SAS datasets from *within* Excel without data corruption issues and without SAS programmer intervention. So rather than pushing data into Excel using DDE from SAS lets make Excel pull the data from SAS and unleash the full potential of VBA .

Excel VBA is a subset of Microsoft's Visual Basic object orientated programming language which references the Excel object model. Unfortunately VBA uses Object Linking and Embedding (OLE) for direct application communication and it can't be called directly through DDE. In addition each VBA statement can't be submitted in isolation but needs to be part of a function or sub procedure so we can't action VBA from within SAS.

The following examples describe three different approaches to getting data from SAS using VBA , and then there follows a bare bones "How To Create An Excel Add-in" to enable distribution of your code.

## USING VBA TO GET DATA FROM SAS

The first two techniques use ActiveX Data Objects (ADO) and SAS objects to transfer data from SAS tables into Excel. The destination of the data within Excel will be pre-formatted to ensure no data corruption issues.

### ADO AND VBA

An in-depth discussion of SAS OLE DB Providers and ADO is beyond the scope of this paper, but the reader can find very useful information at SAS Institute Inc. "ADO/OLE DB Cookbook", Ted Conway (2005) and Darren Key (2002).

A SAS data provider implements data access interfaces that conform to the OLE DB specification from Microsoft, which was built on the OLE Component Object Model (COM). ADO is a simplified programming interface to OLE DB. Referencing the Microsoft ActiveX Data Objects 2.x library within our VBA project gives us the means to access the SAS data provider.

This example uses the SAS Local data provider on a Windows client with the Microsoft VBA project referencing the Microsoft ActiveX Data Objects 2.x library.

```
' Declare variables
Dim obConnection As ADODB.Connection
Dim obRecordset As ADODB.Recordset
Dim i As Integer

Set obConnection = New ADODB.Connection ❶
obConnection.Provider = "sas.LocalProvider.1"
obConnection.Properties("Data Source") = "... folder containing SAS tables ..."
obConnection.Open

Set obRecordset = New ADODB.Recordset ❷
obRecordset.Open "...SAS table name...", obConnection, adOpenDynamic, adLockReadOnly,
ADODB.adCmdTableDirect

'format the cells to text ❸
Range(Cells(1, 1), Cells(obRecordset.RecordCount + 1,
obRecordset.Fields.Count)).NumberFormat = "@"

'add header row
Cells(1, 1).Select ❹
For i = 0 To obRecordset.Fields.Count - 1
    ActiveCell.Offset(0, i).Value = obRecordset.Fields(i).Name
Next i

' add detail rows
obRecordset.MoveFirst ❺
Cells(2, 1).Select
ActiveCell.CopyFromRecordset obRecordset ❻

obRecordset.Close ❼
Set obRecordset = Nothing
obConnection.Close
Set obConnection = Nothing
```

- ❶ ADO requires that we first establish a connection object to the data source. Other connection properties for the SAS local provider can be found in the ADO/OLE DB Cookbook which enable access rights and file format specifications. Other options are available for the other SAS data providers.
- ❷ Once the connection is open, a record set object is defined which details the SAS table to open, the active connection to use, the cursor type (how movement through the record set is defined), concurrency limit (controls the type of concurrency employed when opening the SAS table) and the command type. These last three arguments are constrained by the SAS Local Provider because it doesn't support SAS table updates and it supports only one user at a time. *adCmdTableDirect* is the only available value for the command type for the Local Provider.
- ❸ Before importing the data from the record set, the cells must first be formatted to text. The top most cell in the worksheet will contain the start of the data, and the final row and column depends on the number of records and variables in the SAS table. To get this information, use the *obRecordset.RecordCount* and *obRecordset.Fields.Count* properties and return these values into the range statement. Care should be exercised when using the *RecordCount* property since the value returned is dependent on the cursor type employed in the record set *Open* method. An alternative approach would be to set all cells to the text format using
 

```
Cells.Select
Selection.NumberFormat = "@"
```
- ❹ Add the variable names (field names) as a header row
- ❺ Make sure the record set is at the beginning otherwise not all the data will be output.
- ❻ The *CopyFromRecordset* method writes all the data into the worksheet without the need to reiterate through each field in each record
- ❼ The connection and record set objects are de-assigned to conclude the operation.

The above code directly imports SAS data into Excel while ensuring that Excel treats it as just text to preserve our data. While useful for importing an existing SAS table, there may be times we'll want to generate the SAS table on the fly. Without the use of SAS IOM we're limited to two approaches depending if SAS is installed locally or not.

### SAS OBJECTS ON WINDOWS PLATFORM

If SAS is installed locally, Brian Fairfield-Carter (2004) described a way of creating SAS objects which can submit SAS statements. First we generate a SAS object using the *CreateObject()* function and then submit standard SAS statements via the *Submit* method.

```
Dim sasobj As Object
Set sasobj = CreateObject("SAS.application.8")❶
'use sasobj.Visible = True '->to make SAS visible if you need to debug the SAS code ❷
sasobj.submit ("proc printto log='C:\temp\import.log' new;") ❸
sasobj.submit ("... SAS statement... ;")
sasobj.submit ("... SAS statement... ;")
sasobj.submit ("... SAS statement... ;")

sasobj.submit ("proc printto log=log;")
sasobj.submit ("ENDSAS;") ❹❺
```

- ❶ Create a SAS object and then use it's submit method to send SAS statements.
- ❷ Setting the SAS object to visible displays the SAS environment which is sometimes useful for debugging complicated or errant SAS code, but is distracting for production runs. The SAS object is hidden by default.
- ❸ Use PROC PRINTTO to make sure we can view the SAS log after the execution of the code.
- ❹ The SAS session is ended via the ENDSAS command.

### ACCESSING A SAS SERVER SESSION

For SAS server installations we can't generate a SAS object on the client but we can use the Shell function approach used by Ted Conway (2002). We can use the *-initstmt* option to supply SAS statements directly;

```
Call Shell("""\... server location ...\sas.exe"" -nosplash -icon -initstmt "... SAS
statements ...""", vbMinimizedNoFocus)
```

Or us the *-sysin* option to supply a SAS file to run if the SAS statements are extensive;

```
Call Shell("""\... server location ...\sas.exe"" -nosplash -icon -sysin "... SAS file name
...""", vbMinimizedNoFocus)
```

The SAS file for the *-sysin* option could be written from within VBA too so processing that is fully dynamic and data driven is still achievable. The *vbMinimizedNoFocus* argument minimizes the resultant window to the task bar and the currently active window retains focus. So to run the above SAS statements using the *-initstmt* option and Shell function we could try this approach;

```

Dim sSASCmd As String
sSASCmd = "proc printto log='C:\temp\import.log' new;"
sSASCmd = sSASCmd & "... SAS statement...;" ❶
sSASCmd = sSASCmd & "... SAS statement...;"
sSASCmd = sSASCmd & "... SAS statement...;"
sSASCmd = sSASCmd & " proc printto log = log";
sSASCmd = sSASCmd & "ENDSAS;"

Call Shell("""\... server location ...\sas.exe"" -nosplash -icon -initstmt "" & sSASCmd &
""", vbMinimizedNoFocus) ❷

```

❶ The SAS statements are concatenated together into the variable sSASCmd.

❷ Note the additional set of quotation marks and ampersand concatenation symbols to enable the sSASCmd variable to replace fixed text.

Neither of the two methods above are limited to hard coding specific information such as SAS table names or locations. Path statements, file names and data items could all be input by the user.

Unfortunately there is one draw back to using both the Shell function and the SAS object methods, and that is that both methods run asynchronously. This means that Excel won't wait until SAS has finished creating it's output (a SAS table for instance) before attempting to open it. To get around this we need to force Excel to wait. Just having an arbitrary time to wait (say 3 minutes) isn't very sophisticated or efficient and will fail if the SAS job takes longer. Excel needs to test for the presence of the output, and if it can't find it, then wait some more before testing again. For this approach to work we need to ensure the output doesn't exist prior to SAS running, so we're going to delete it, and the SAS log too for good measure just in case the code has been run previously.

```

'if previous file exists then delete it
Set fso = CreateObject("Scripting.FileSystemObject")
If fso.fileexists("C:\temp\import.sas7bdat") Then fso.deletefile
("C:\temp\import.sas7bdat")
If fso.fileexists("C:\temp\import.log") Then fso.deletefile ("C:\temp\import.log")

```

The variable fso is a File System object which enables us to manipulate the file system via the FileExists() and DeleteFile() methods. Now that we've deleted the two files, we can call SAS by either of the two above methods. Once SAS is running the following bit of code forces Excel to wait until the text file is ready from SAS.

```

' pause Excel for 3 seconds to start with
Application.Wait (Now() + TimeValue("00:00:03")) ❶

' loop until required file exists, but only wait for 3 minutes
vTimeLimit = (Now() + TimeValue("00:03:00")) ❷
Do While Not fso.fileexists("C:\temp\import.sas7bdat") ❸
    Application.Wait (Now() + TimeValue("00:00:03")) ❹
    If Now() > vTimeLimit Then ❺
        Call MsgBox(">3 minutes since SAS job started, quitting.....",
vbCritical, "Error in SAS?")
        If fso.fileexists("C:\temp\import.Log") Then
            Call Shell("notepad.exe C:\temp\import.log", vbMaximizedFocus) ❻
        End If
        Exit Sub ❼
    End If
Loop

```

❶ May as well pause Excel to start with since no matter how short the SAS program, it is still going to take some time to get SAS alive. This pauses Excel for 3 seconds.

❷ In order to prevent an infinite loop if something happens to prevent the creation of the output file, a time limit of 3 minutes is created. If the output file still doesn't exist after this time then we'll assume an error and exit the processing. Tip: Control + Break stops macro code execution. Something you might need during this loop during testing.

❸ Loop while the output file doesn't exist.

❹ Pause Excel again for another 3 seconds. This is reiterated for each loop so in effect we're testing for the output file at 3 second intervals.

❺ Check if we're over the 3 minute limit, and then warn the user via a dialog box.

❻ If it exists lets display the SAS log to the user in case this supplies some useful information. This is done using the Shell function again but this time we're calling Notepad with the path and file name of the SAS log as an argument to it. Maximizing Notepad and giving it the focus forces the user to take notice ! Note that we don't have to create another File System object, just use the one we've already created which is fso.

❼ Exit the processing now since there is nothing left to be done.

Assuming everything has gone smoothly, and the output file is now ready to be opened by Excel, it's time to remind people where the data has come from by providing them with the SAS log in Notepad. We'll use the same syntax as before.

```
If fso.fileexists("C:\temp\import.Log") Then
    Call Shell("notepad.exe C:\temp\import.log", vbMaximizedFocus)
End If
```

### **LIMITATIONS AND ENHANCEMENTS**

Needless to say the above example code is only the bare bones required to complete the task. I've not attempted to ensure SAS isn't going to exceed Excel's limits of 256 columns and 65536 rows. There is no error trapping, and the user isn't kept informed on the progress. We could remove the need for hard coding by seeking user input into things like the file name and location, and apply formatting to the worksheet.

Once your hard earned code has been written, I recommend distributing as an Excel add-in.

### **EXCEL ADD-IN**

An Excel add-in is a special kind of hidden Excel workbook (with a .XLA extension) that adds additional functionality and commands to Excel. Excel comes with its own set of standard add-ins such as the Analysis ToolPak add-in which provides functions and interfaces for financial and scientific data analysis. Creating your own add-in though is a relatively simple task. The advantages of using an add-in to distribute VBA code over a template workbook is that it is seamlessly integrated into the Excel shell, is always available once loaded and is protected from alterations or changes by users.

There are two basic tasks required of the programmer when creating an add-in. These are a) the user interface and b) the code which does something useful. There are some instances where a user interface isn't required because Excel automatically handles this aspect e.g. user defined functions, but this isn't the case in the following example.

The example add-in detailed here is aimed at non-VBA or fledgling VBA programmers to show what is possible from within Excel in relation to SAS and to encourage such a person to try to use it. More competent VBA programmers might find some interest in the VBA / SAS interface and prompt them in new directions.

### **GETTING STARTED**

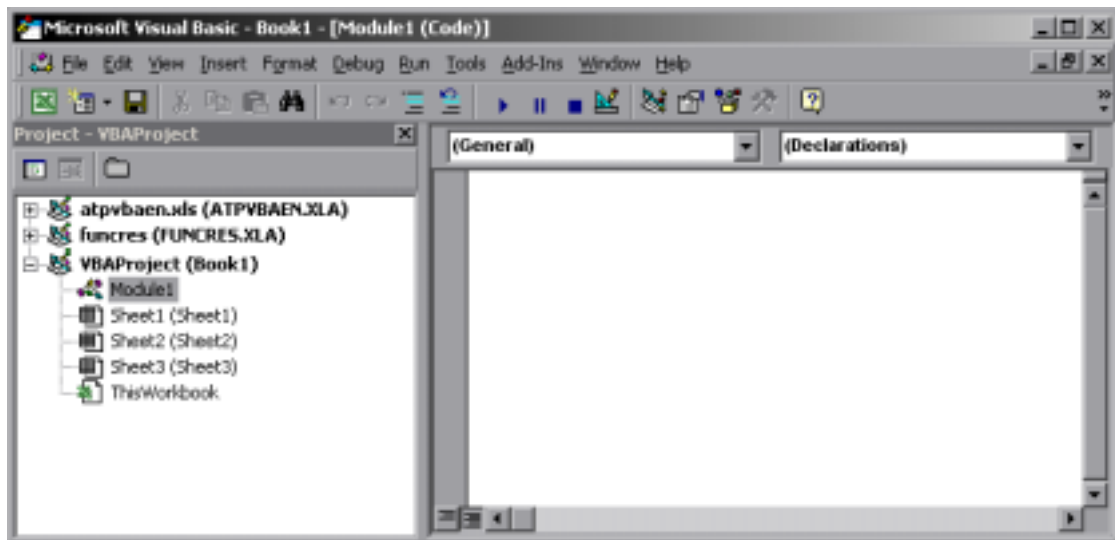
Fire up Excel and make sure a new workbook is open (Book1.xls). If the Visual Basic toolbar isn't visible then choose VIEW | TOOLBARS | VISUAL BASIC from the main menu. Click the Visual Basic Editor icon to open up the VBA development environment. Make sure the Project Explorer window is visible by using VIEW | PROJECT EXPLORER from within the VB editor. The project explorer window contains all the add-ins and normal Excel workbooks currently loaded in Excel, and these items can be expanded to reveal the different components of each add-in or workbook, but only if they aren't locked for viewing by the add-in's author. For obvious reasons you won't be able to open up Excel's standard add-ins!

The VB editor area can be a little confusing at first, but since this is the main tool of the trade, time spent understanding this environment is time very well spent.

Find VBAPProject (Book1) in the Project Explorer and expand this project. If there is a group called Microsoft Excel Objects then expand this item. You will see each worksheet contained in Book1 is represented by a worksheet object, plus an additional item called ThisWorkbook which represents the workbook object.

The only other additional item we'll need is a central repository for the bulk of our VBA code. To do this, make sure the VBAPProject (Book1) in the Project Explorer is selected, then select INSERT | MODULE. A new object should have been added to the project called Module1. The figure below should hopefully represent the VB editor after these instructions.

A word here about the VBA references. We can load and unload specific references to certain object libraries in our project via TOOLS | REFERENCES. We'll need to select the Microsoft ActiveX Data Objects 2.x library reference if using the ADO example. Just check the check box from the references list and click OK.



## USER INTERFACE

Since an add-in is essentially a hidden workbook, we need to provide some way for the user to call our code. We're going to enable this by adding an additional menu item in the TOOLS menu bar called "Open SAS Table". You could of course add a new menu item anywhere, even at the top level along with FILE, EDIT and VIEW etc so creating a new top level group called "SAS Procedures" with additional submenu items all from the one add-in is easily achievable should you wish to.

What we want to do is to create the additional menu item when the add-in is loaded, and remove it when the add-in is unloaded. This is achieved by adding our code in two specific places. The *Workbook\_Open()* and *Workbook\_BeforeClose(...)* procedures are called automatically by Excel whenever the add-in is loaded or unloaded respectively (for normal workbooks these sub procedures are called when the excel file is opened and before it is closed but when dealing with add-ins we use the terminology load and unload). Double clicking the "ThisWorkbook" object opens up the code editor. Select "Workbook" from the Objects list box and either "Open" or "BeforeClose" from the Procedures list box.

```
Private Sub Workbook_Open()
    ' Variable declaration section
    Dim toolMenus As CommandBar
    Dim toolMenuItem As CommandBarControl
    Dim newMenuItem As CommandBarControl
    Dim bFound As Boolean

    Set toolMenus = Application.CommandBars("Tools") ❶
    bFound = False

    ' check to see if our menu item already exists in the TOOLS menu
    For Each toolMenuItem In toolMenus.Controls ❷
        If toolMenuItem.Tag = "OpenSASDataset" Then
            bFound = True
            Exit For
        End If
    Next

    If Not bFound Then

        Set newMenuItem = toolMenus.Controls.Add(Type:=msoControlButton, _
            Temporary:=True) ❸
        newMenuItem.Caption = "Open SAS Ta&ble" ❹
        newMenuItem.OnAction = "ImportSASTable" ❺
        newMenuItem.Tag = "OpenSASDataset" ❻
    End If
End Sub
```

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

    Application.CommandBars("Tools").Controls("Open SAS Ta&ble").Delete ❷

End Sub
```

❶ assigns the relevant object reference to the variable `toolMenus`. This isn't essential, but makes the following code easier to read.

❷ loop through each menu item in the TOOLS menu option to see if one of the entries has a *Tag* property value equal to "OpenSASDataset". In other words, is our menu item already loaded. If yes, then set the `bFound` flag variable to true. The *Tag* property is a string that holds information to help identify the control. Obviously we need to test for the same string which we assign to the menu item in step 6. The more VBA aware reader will spot that this loop is actually cycling through the command bar controls *collection*, hence the syntax.

❸ if our menu item hasn't been found then add a blank entry to the end of the TOOLS menu. Again we assign this new menu item object to a variable.

❹ give our new menu item some text by using the *Caption* property. The '&' symbol denotes the following character to be the keyboard shortcut key.

❺ the *OnAction* property should be set to the same name as the sub procedure we want to call which then does the rest of the processing for us.

❻ set the *Tag* property. Setting this property enables us to test for the presence of our menu item with more confidence.

❼ In contrast to adding our menu item, deleting the menu item is a simple affair accomplished with one line of code. This is done in the sub procedure `Workbook_BeforeClose()` which, as stated before, is automatically run by Excel when the add-in is unloaded.

This completes the user interface part of the code. The code which does the rest of the processing will reside in Module1.

Double click Module1 in the Project Explorer window to be sure we are in the correct coding section. Type `Sub ImportSASTable()` as per our *OnAction* method for the tool bar menu. Excel adds the *End Sub* part itself once you have hit enter.

```
Sub ImportSASTable()

    ... variable declarations

    ... your code to go here ...

End Sub
```

## CREATING AND LOADING THE ADD-IN

Once all the code has been entered there only remains the simple task of turning this normal workbook into an add-in. Bring up the document properties dialog box (FILE | PROPERTIES), then in the SUMMARY tab complete the title and comments sections. The text in the title field is used to name the add-in in the add-in manager (so keep it concise e.g. "Import SAS Table"), while the text in the comments field is a fuller description of the purpose of the add-in displayed again in the add-in manager (e.g. "Open SAS tables from within Excel").

Use FILE | SAVE AS to save the work book, but select the Save As type to be Microsoft Excel Add-in (\*.XLA). This file can now be freely distributed to your colleagues, or maintained centrally. To load the add-in into Excel, use TOOLS | ADD-INS..., browse to the add-in and click OK. Your add-in should now be listed in the ADD-INS AVAILABLE list box with it's check box checked. Click OK and use the add-in via the TOOLS | OPEN SAS TABLE menu item. The add-in is automatically loaded each time Excel starts so this process need only be done once.

## CONCLUSION

While the knowledge of getting data from SAS to Excel is widely disseminated, it's limitations are not so widely known. Hopefully this paper has not only highlighted this issue, but has also put forward some simple, and not so simple, solutions. The use of VBA can greatly enhance the efficiency and productivity of data migration while at the same time eliminating data corruption issues.

## REFERENCES

Koen Vyverman (2000) "Using Dynamic Data Exchange to Pour SAS Data into Microsoft Excel", *Proceedings of the 18th SAS European Users Group International Conference*.

Koen Vyverman (2001) "Using Dynamic Data Exchange to Export Your SAS® Data to MS Excel — Against All ODS, Part I", *Proceedings of the Twenty-sixth Annual SAS Users Group International Conference*, Paper 11.

Koen Vyverman (2002) "Creating Custom Excel Workbooks from Base SAS® with Dynamic Data Exchange: A Complete Walkthrough". *Proceedings of the Twenty-seventh Annual SAS Users Group International Conference*, Paper 190.

Ted Conway (2002) "A Better Desktop Than Windows: Using Excel to Organize, View, Launch and Document SAS Programs", *Proceedings of the Twenty-seventh Annual SAS Users Group International Conference*, Paper 101.

Darren Key (2002) "Using SAS® Data To Drive Microsoft Office", *Proceedings of the Twenty-seventh Annual SAS Users Group International Conference*, Paper 123.

Vincent DelGobbo (2003) "A Beginner's Guide to Incorporating SAS Output in Microsoft Office Applications". *Proceedings of the Twenty-eighth Annual SAS Users Group International Conference*, Paper 52.

Vincent DelGobbo (2004) "Moving Data and Analytical Results between SAS and Microsoft Office", *Proceedings of the Thirtieth Annual SAS Users Group International Conference*, Paper 136.

Perry Watts (2004) "Using Single-Purpose SAS® Macros to Format EXCEL Spreadsheets with DDE", *Proceedings of the Thirtieth Annual SAS Users Group International Conference*, Paper 89.

Brian Fairfield-Carter (2004), "Instant SAS Applications With VBScript, Jscript, and dHTML ", *Proceedings of the Thirtieth Annual SAS Users Group International Conference*, Paper 9.

Steven Feder (2004), "Reporting and Interacting with Excel® spreadsheets with ODS and DDE", *Proceedings of the Thirtieth Annual SAS Users Group International Conference*, Paper 26.

Ted Conway (2005) "Make Bill Gates and Dr. Goodnight Run Your SAS Code: Using VBA, ADO and IOM to Make Excel and SAS Play Nice", *Proceedings of the Thirtieth Annual SAS Users Group International Conference*, Paper 157.

SAS Institute Inc., "Welcome to the SAS 9.1 Data Providers: ADO/OLE DB Cookbook ",  
<<http://support.sas.com/rnd/eai/oledb/index.htm>>, Cary, NC: SAS Institute Inc.

## **ACKNOWLEDGMENTS (HEADER 1)**

Thanks to Paul Frost and Ian Taylor (AstraZeneca UK Ltd) for guidance and comments during the development of this paper.

## **CONTACT INFORMATION**

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

Timothy Adlington  
AstraZeneca UK Ltd  
Alderley Park  
Macclesfield  
SK10 4TG  
UK

Email: [timothy.adlington@astrazeneca.com](mailto:timothy.adlington@astrazeneca.com)

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.