Paper 21-25

Using SAS® Software and Visual Basic for Applications to Automate Tasks in Microsoft Word: An Alternative to Dynamic Data Exchange

Mark Stetz, Amgen, Inc., Thousand Oaks, CA

ABSTRACT

Using Dynamic Data Exchange (DDE) to drive Microsoft Office applications such as Word and Excel has long been a technique utilized by SAS programmers. While alternatives exist (e.g., generating RTF code, OLE Automation), the use of DDE in conjunction with the native scripting languages of these applications has been appealing since implementation is simple and the macro recording capabilities of Office applications make script generation nearly automatic. Harris (SUGI 24 Proceedings, 1999) described a particularly elegant design for using the SAS System and DDE to populate Microsoft Word documents.

Now that the more sophisticated Visual Basic for Applications (VBA) is the common development environment among all Office applications, however, DDE only works with Office application legacy macro languages. Employing Harris' design, this paper describes a technique to simulate the ease of use of the DDE methodology while taking full advantage of VBA to automate Microsoft Word.

INTRODUCTION

DDE is a feature of Microsoft Windows that allows two programs to share data or send commands directly to each other. SAS programmers have long taken advantage of DDE with Microsoft Office applications to perform such tasks as generating Word documents or Excel spreadsheets from within their SAS programs.

Among the advantages of using DDE is its straightforward implementation. SAS programs simply use PUT statements within a DATA step to send scripting commands to an Office application. In addition, the macro recording capabilities of Office applications make generating script effortless.

Prior to Office 97, Office applications such as Word and Excel each utilized their own individual scripting languages to automate tasks within their respective programs. Beginning with Office 97, however, the programming environment was standardized such that all Office applications use the VBA development environment.

Unlike the previous scripting languages which simply send keystrokes or command strings to an application, VBA works by directly controlling Office applications, hooking into objects within the applications. This increased level of sophistication makes VBA incompatible with DDE.

This paper presents a technique for using VBA to automate tasks in Microsoft Word. The technique is similar enough to the DDE methodology to rival its ease of use, yet provides additional capabilities not possible with DDE. Implementing the design outlined by Harris further simplifies and optimizes this approach.

DDE AND THE HARRIS DESIGN

While DDE implementation is simple, the mixture of SAS and WordBasic syntax can result in confusing, difficult to maintain code. The following shows a simple DATA step for populating a Word document using DDE and WordBasic (assuming Word is currently running):

```
filename cmds dde 'winword|system' ;

data _null_  ;
   file cmds ;

   put '[Insert "Hello World"]' ;
   put '[FileSaveAs '
         '.Name = "c:\My Documents\Hello", '
         '.Format = 0, '
         '.AddToMru=0]' ;
  put '[FileClose]' ;
run ;
```

Harris suggested a design to simplify the coding of such DATA steps by wrapping WordBasic commands in intuitively named SAS macros. For example, the WordBasic Insert command would be encapsulated as follows:

```
%macro puttext(text) ;
   put "[Insert ""&text""]";
%mend ;
```

Assuming macros exist that wrap other WordBasic commands similarly, the above DATA step could be re-coded as follows:

```
filename cmds dde 'winword|system' ;

data _null_  ;
   file cmds ;

   %puttext(Hello World)
   %saveas(c:\My Documents\Hello)
   %fileclos
run ;
```

Harris provides examples of several such macros which can be grouped into libraries and customized according to a user's specific requirements. Among the advantages of incorporating this design are code re-use and complexity hiding (users of the libraries aren't required to know WordBasic). In addition, both the libraries and programs utilizing the libraries are significantly easier to develop and maintain.

CONVERSION TO VBA

The conversion of WordBasic commands to VBA in many cases is a relatively minor issue (assuming some knowledge of VBA). Word Help provides a mapping of WordBasic commands to VBA and macro recording provides another resource. For example, the VBA

equivalent for the macro PUTTEXT from above is:

```
%macro puttext(text) ;
   put 'Selection.TypeText '
         "Text:=""&text""" ;
%mend puttext ;
```

The important distinction is that the above VBA statement, unlike its WordBasic counterpart, is not executable on its own. It is only valid in the context of a VBA procedure. This distinction precludes the use of DDE to send VBA statements to Word.

THE RUN_VBA DELIVERY MECHANISM

Because VBA statements cannot be delivered to Word using DDE, an alternative delivery mechanism is required. The new delivery mechanism employs a SAS macro and a Word macro specifically designed to work together to read in and execute a given Word macro stored in an external (text) file.

THE RUN_VBA SAS MACRO

The RUN_VBA SAS macro (shown in **Appendix 1**) serves as the SAS component of the delivery mechanism for any program using VBA. The macro provides an interface to Word through the use of a Word command line startup switch and Windows environment variables.

DETAILS

The Word command line startup switch ***/mmacro-name*** instructs Word to execute the specified ***macro-name*** when Word is started. In the case of the delivery mechanism the ***macro-name*** is the Run_VBA Word macro described later.

Because Word macros cannot have input arguments, the delivery mechanism makes use of Windows environment variables to pass information to the Run_VBA Word macro. The RUN_VBA SAS macro allows the user to specify the values of these variables through its parameters.

The RUN_VBA SAS macro simply generates and executes a Windows batch file. The batch file consists of commands to create and set the values of several Windows environment variables (based on the values of the macro parameters), invoke Word (and thereby invoke the Run_VBA Word macro), and finally, delete itself.

RUN_VBA SAS MACRO PARAMETERS

The RUN_VBA SAS macro has the following parameters:

Parameter	Default	Description
WORDPATH=	None	The MS-DOS directory path of the Word executable
VBA_PATH=	None	The Windows directory path of the file containing the VBA macro
VBA_PGM=	None	The name of the file (sans extension) containing the VBA macro (Note: A .bas extension is assumed.)
VISIBLE=	Y	N= Run Word without a Graphical User Interface (GUI) (this can speed up execution) [Note: There will be NO visible indication (e.g., a button in the Toolbar) that Word is active or when the macro has completed. To have the GUI restored when the macro has completed, set NOTIFY=Y and EXITWORD=N.]
MINIMIZE=	N	Y=Minimize Word while the macro executes (this can speed up execution)
NOTIFY=	N	Y=Have Word display a notification dialog box when the macro has completed running
EXITWORD=	N	Y=Exit (close) Word upon completion of the macro execution (otherwise Word remains open)

THE RUN_VBA WORD MACRO

The Run_VBA Word macro (shown in **Appendix 2**) is the Word component of the delivery mechanism. The macro is permanently stored in the Normal template and designed to execute at Word startup as described previously. The Run_VBA Word macro uses the environment variables created by the RUN_VBA SAS macro to read in, execute, and delete a Word VBA macro generated in a SAS DATA step.

Other than the initial installation and set-up (described in the next section), the user has no *direct* interaction with the Run_VBA Word macro (although indirect interaction occurs through the RUN_VBA SAS macro).

MICROSOFT WORD SET-UP

The following steps describe how to create the Run_VBA macro in Microsoft Word and perform some additional set-up:

1. Under the **Tools** menu select **Macro** and then **Visual Basic Editor** (VBE)
2. Make sure the Project Explorer and Properties windows are visible (these are usually located along the left side of the VBE). If not, under the **View** menu select **Project Explorer** and then **Properties Window**.
3. In the Project Explorer window (usually located on the upper left side of the VBE), make sure Normal is currently selected. The Project Explorer window title bar should display "**Project – Normal**".
4. Under the **Insert** menu select **Module** (Module1 should have been added to the tree in the Project window)
5. Enter the code shown in **Appendix 2**
6. In the Properties window (usually located on the

lower left side of the VBE, below the Project Explorer window), click in the text field labeled "**(Name)**" and change "**Module1**" to "**Run_VBA**" (do not include the quotes). Press the Enter key so that the new value is reflected in the Project Explorer window.

7. Under the **Tools** menu select **References…**
8. In the list box, check the box next to "**Microsoft Visual Basic for Applications Extensibility**" (you will need to a scroll down to find that selection)
9. Click the **OK** button to close the References dialog box
10. Under the **File** menu, select **Save Normal** to save the changes
11. Exit the VBE

The Run_VBA Word macro creates a temporary module in the Normal template that is automatically deleted under normal circumstances. If, however, an error occurs, the temporary module may not be deleted and will cause further errors until it has been deleted. Therefore, it is strongly recommended that Word be set-up to prompt for saving changes to the Normal template. Having Word prompt to save changes (and selecting **No**) will prevent the temporary module from being unintentionally saved in the Normal template. (Note: If you elect to follow this recommendation Word will always prompt to save changes to the Normal template whether your program runs correctly or not since some change occurs. In either case you should always select **No** when prompted.)

To set-up Word to prompt for saving changes to the Normal template do the following:

1. Under the **Tools** menu select **Options…**
2. Select the **Save** tab
3. Select the checkbox labeled "**Prompt to save Normal template**"

EXAMPLE

The following shows a simple VBA macro library based on the example DATA step shown previously:

```
%macro vba_lib ;

   %macro puttext(text) ;
      put 'Selection.TypeText '
          "Text:=""&text""" ;
   %mend puttext ;

   %macro saveas(fname,
                 format=wdFormatDocument) ;
      put 'ActiveDocument.SaveAs '
           "FileName:=""&fname"", "
           "FileFormat:=&format" ;
   %mend saveas ;

   %macro fileclos ;
      put 'ActiveDocument.Close '
      'SaveChanges:=wdDoNotSaveChanges' ;
   %mend fileclos ;

%mend vba_lib ;
```

The DATA step presented earlier is modified as follows:

```
%vba_lib

%let wordpath=
    c:\progra~1\micros~1\office ;
%let path=c:\My Documents ;
%let program=Hello ;

data _null_ ;
   file "&path\&program..bas" ;

   put "Sub &program()" ;
   %puttext(Hello World)
   %saveas(&path\&program)
   %fileclos
   put 'End Sub' ;
run ;

%run_vba(wordpath=&wordpath,
         vba_path=&path,
          vba_pgm=&program,
          visible=Y,
         minimize=Y,
           notify=Y,
         exitword=Y) ;
```

As the example demonstrates, the main differences from the DDE methodology are:

- The FILE statement now specifies an external text file (Note: A .bas extension is required.)
- VBA Sub and End Sub statements are required to define a Word macro (Note: The Word macro name must be the same as the filename.)
- The RUN_VBA SAS macro must be invoked to execute the Word macro created in the DATA step

(Note: Macro variables are used to make the code more maintainable.)

HOW IT WORKS

The following outlines the key events when the example is executed:

- The DATA step generates a Word VBA macro written to a text file
- The RUN_VBA SAS macro generates and executes a Windows batch file that:
  - sets Windows environment variables based on the macro parameter values specified
  - invokes Word using the **/mRun_VBA** startup switch
- The Run_VBA Word macro:
  - reads the values of the environment variables set in the batch file
  - minimizes Word (`minimize=Y`)
  - creates a new module in the Normal template
  - reads in the text file containing the VBA macro generated in the SAS DATA step
  - executes the VBA macro
  - deletes the new module from the Normal template
  - deletes the text file containing the VBA macro
  - notifies the user via a dialog box that the macro has completed (`notify=Y`)
  - closes Word after the user acknowledges the dialog box (`exitword=Y`)

ADVANTAGES OF THE VBA TECHNIQUE

The VBA technique described in this paper offers several important advantages over DDE and WordBasic. Foremost, VBA is the scripting/programming language for Office applications. While backward compatibility with WordBasic exists, it comes at the expense of efficiency since Word must convert each WordBasic command to run under VBA. Using DDE also incurs additional overhead since each statement must be interpreted individually as opposed to the VBA technique which executes a single compiled macro.

The VBA technique, while providing the complexity hiding inherent in the Harris design, also allows the use of the entire programming language including control statements such as For…Next, Do, and If…Then…ElseIf. Therefore, experienced VBA programmers can take advantage of this capability to build even more efficient Word macros. Consider the following DDE example:

```
filename cmds dde 'winword|system' ;

data _null_  ;
   file cmds ;

   do i=1 to 1000 ;
      %puttext(Hello World)
   end ;
   %saveas(c:\My Documents\Hello)
   %fileclos
run ;
```

While a similar VBA version is possible, the following takes advantage of a VBA looping construct to produce a significantly more efficient Word macro—and SAS program (Note: This example assumes the existence of the macro variables as defined in the previous VBA example.):

```
data _null_ ;
   file "&path\&program..bas" ;

   put "Sub &program()" ;
   put 'Dim i as Integer' ;
   put 'For i = 1 To 1000' ;
   %puttext(Hello World)
   put 'Next i' ;
   %saveas(&path\&program)
   %fileclos
   put 'End Sub' ;
run ;
```

Another advantage of the VBA technique is that each application (the SAS System and Word) execute their functions independently. The SAS System generates the VBA macro and invokes the RUN_VBA macro (which in turn generates and executes the Windows batch file). Word is started as an independent process (by the batch file) and executes the macro generated by the SAS System. While Word executes the macro the SAS System is free to begin the sequence again in which case a new instance of Word is invoked (regardless of whether or not the previous instance of Word is still running). This allows for great flexibility in batch processing and takes advantage of multi-tasking not possible using DDE.

Finally, the VBA technique is more easily extendible to other Microsoft Office applications as well as to many other non-Office applications that support VBA. Having each of the applications "speak the same language" considerably simplifies the task of the developer in building and maintaining the libraries described by Harris.

CONCLUSION

The use of DDE, particularly in conjunction with the design described by Harris, provides a simple, powerful, and maintainable method of taking advantage of the scripting capabilities of Microsoft Office applications. The emergence of VBA as the programming environment for Office, however, provides new opportunities in extending and enhancing these capabilities. The technique described in this paper attempts to maintain the simplicity of using DDE while opening the new doors provided by VBA.

TRADEMARKS

REFERENCES

Harris, Michael (1999), "Using the SAS System and Dynamic Data Exchange to Populate Microsoft Word Documents with Text, Tables, and Graphs", *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*, 24, 156-160.

CONTACT INFORMATION

Comments and questions are welcomed. Contact the author at:

Mark Stetz
Amgen, Inc.
One Amgen Center Drive, MS 17-2-B
Thousand Oaks, CA 91320-1789

mstetz@amgen.com

The source code used in this paper can be obtained at:

http://home.earthlink.net/~equizole/

APPENDIX 1: RUN_VBA.SAS

```
%macro run_vba(wordpath=,
               vba_path=,
                vba_pgm=,
                visible=Y,
               minimize=N,
                 notify=N,
               exitword=N) ;

   options noxwait ;

  /*-------------------------------------------------------------------------*
   | Generate a batch file to set environment variables and invoke Word using |
   | the /m option to run the Run_VBA Word macro. The Run_VBA Word macro uses |
   | the environment variables' values to read in and execute the VBA macro   |
   | source code file(s) generated by the calling SAS program.                |
   *-------------------------------------------------------------------------*/
   data _null_ ;
      file "&vba_path\&vba_pgm..bat" ;

      put "set VBA_PATH=&vba_path" ;
      put "set VBA_PGM=&vba_pgm" ;
      put "set VISIBLE=&visible" ;
      put "set MINIMIZE=&minimize" ;
      put "set NOTIFY=&notify" ;
      put "set EXITWORD=&exitword" ;

      put "start &wordpath\winword.exe /mRun_VBA" ;

      put "del ""&vba_path\&vba_pgm..bat"" " ;
   run ;

  /*-------------------------------------------------------*
   | Run the batch file created in the previous DATA step |
   *-------------------------------------------------------*/
   data _null_ ;
      call system("""&vba_path\&vba_pgm..bat""") ;
   run ;

%mend run_vba ;
```

APPENDIX 2: RUN_VBA.BAS

```
Sub Run_VBA()

  '*** Declare local variables ***
   Dim vbcs As VBComponents
   Dim vbc As VBComponent
   Dim vba_num As Long
   Dim vba_filename As String

   Dim vba_path As String
   Dim vba_pgm As String
   Dim minimize As String * 1
   Dim visible As String * 1
   Dim notify As String * 1
   Dim exitword As String * 1

  '*** Read environment variables (created in RUN_VBA.SAS) ***
   vba_path = Environ("VBA_PATH")
   vba_pgm = Environ("VBA_PGM")
   minimize = UCase(Environ("MINIMIZE"))
   visible = UCase(Environ("VISIBLE"))
   notify = UCase(Environ("NOTIFY"))
   exitword = UCase(Environ("EXITWORD"))

  '*** Set the Word application window state ***
   If visible <> "Y" Then
      Application.visible = False
      If notify <> "Y" Then exitword = "Y"
   ElseIf minimize = "Y" Then
      Application.WindowState = wdWindowStateMinimize
   End If

  '*** Create a new Word document ***
   Documents.Add
   Documents(1).Activate

  '*** Get a reference to the Normal template component collection ***
   Set vbcs = VBE.VBProjects("Normal").VBComponents

  '*** Construct the filename of the SAS generated VBA macro ***
   vba_filename = vba_path & "\" & vba_pgm & ".bas"

  '*** Add a new code module to the Normal template ***
  Set vbc = vbcs.Add(vbext_ct_StdModule)

  '*** Read in the SAS generated VBA macro ***
   vbc.CodeModule.AddFromFile FileName:=vba_filename

  '*** Run the SAS generated VBA macro ***
   Application.Run MacroName:=vba_pgm

  '*** Delete the code module from the Normal template ***
   vbcs.Remove VBComponent:=vbc

  '*** Delete the file containing the SAS generated VBA macro ***
   Kill vba_filename

  '*** Notify the user that program has completed ***
   If notify = "Y" Then
      Application.visible = True
      MsgBox vba_pgm & " completed."
   End If

  '*** Exit Word ***
   If exitword = "Y" Then
      Application.Quit SaveChanges:=wdDoNotSaveChanges
   End If

End Sub
```