

An Introduction to SAS[®] Applications of the Windows Scripting Host

Stephen Hunt, PRA International, Victoria, BC
 Tracy Sherman, PRA International, Victoria, BC
 Brian Fairfield-Carter, PRA International, Victoria, BC

ABSTRACT

Analysis Programmers in the Pharmaceutical industry are often confronted with a variety of requirements that are peripheral to the more common tasks of data manipulation and summarization. These include supporting multiple output file types (.doc, .rtf, .ps, .pdf), making operating system calls (for example, to retrieve file attribute information), and carrying out file-handling operations. These tasks are often satisfied through more obscure components of Base SAS (DDE, System functions, 'pipe' filename access, x/DOS commands), and through code-writing macros (custom output 'drivers'), that allow SAS interaction with other applications.

This paper introduces the Windows Scripting Host (WSH) as a simple, powerful, and comprehensive tool for meeting all of the requirements listed above. The WSH allows VBScript, Jscript and Perl programs to run on the host operating system as stand-alone applications, and thus gives SAS programmers not only an environment in which to program in COM (the 'Component Object Model'), but also to develop dynamic applications. This paper focuses on the fundamentals and practical application of VBScript for running Operating System and File System commands and for instantiating and controlling Windows applications, and on methods for integrating WSH applications with a SAS session. The final section introduces the ActiveX Data Object for writing data between SAS, Excel, and Access.

INTRODUCTION

A REVIEW OF COMMON FILE-TYPE CONVERSION METHODS

File-type conversion methods commonly seen in the SAS world can be roughly divided into 4 categories:

- (1) Those that use data _null_ 'drivers' to write .ps, .rtf or .html code,
- (2) Those that use SAS ODS output destinations,
- (3) Those that use the DDE (Dynamic Data Exchange) interface to interact with Windows applications, and
- (4) Those that run Visual Basic for Applications (VBA) macros in Windows applications.

Each of these methods comes with its unique set of attractive features and disadvantages (table 1). In the long run, it is likely that ODS will be the most versatile standard for producing non-SAS output from SAS, simply because it functions seamlessly with the rest of the SAS system, and uses syntax and language elements which are familiar to SAS users. However, even with continued advancements in ODS, it is likely that Windows scripting technologies will continue to provide a valuable supplement.

OPERATING AND FILE SYSTEM CALLS

Operating and file system calls in SAS tend to be made through 'Call System', 'x/DOS', 'pipe' filename access, and system function ('Sysfunc') commands. While these tend to allow SAS programs to be fairly seamless, and use familiar syntax, the overall functionality is incomplete: system functions do not return all file attribute information, and much of the functionality relies on archaic DOS commands rather than Windows Component Object Model (COM) objects and methods.

Table 1. Common methods for file type conversion and for Operating and File system calls

Method	Advantages	Disadvantages
Production of alternate file types		
Data _null_ -based output drivers	Low-level control over the placement and format of output	May be a major investment in time to learn post-script, rtf (etc.) languages; static (driver must be updated to keep pace with changes to standard language definitions)
Output Delivery System	Uses syntax and language familiar to SAS users	Can be difficult to set up output templates with PROC TEMPLATE; some post-processing of ODS html and rtf output may still be required.
Dynamic Data Exchange	Can take advantage of output drivers built into target Windows applications	Requires a hard-coded reference to the application executable (not consistent between OS releases); non-intuitive and difficult to debug, and cannot be tested externally to SAS; slow, because of the need to use the 'sleep' function to control process timing; target applications cannot be run in the background (inadvertent key-strokes may corrupt output)

VBA macros in Word	Can take advantage of Word's output drivers; no VBA programming ability required to record and run VBA macros	Running VBA macros directly from SAS requires the use of DOS commands or DDE; Word provides an awkward and error-prone development environment, and macros can't be run as stand-alone applications; recorded macros contain large volumes of redundant commands, making them difficult to read and edit.
File and Operating System commands		
SAS System Functions	Syntax and language are familiar to SAS users; seamless	SAS system functions are incomplete and do not provide the full range of file system operations
SAS x DOS commands	Seamless	Requires knowledge of DOS; may not perform the same in different OS releases; archaic – doesn't use the Windows Component Object Model
Windows Scripting Host (both to produce alternate file types, and for file and operating system commands)	Can be tested externally to SAS; takes advantage of output drivers and database engines built into target applications; provides a very simple interface between SAS and ActiveX automation/COM; scripts can be re-used in dynamic HTML applications with very little or no modification; allows the rapid development of stand-alone applications.	Debugging facilities are limited; differences between VBScript and VBA can be a source of frustration

THE WINDOWS SCRIPTING HOST

The Windows Scripting Host is a component of Windows Operating Systems, introduced with Windows 95, which allows 'Visual Basic Scripting Edition' (VBScript), Jscript, and Perl to be run 'natively on the host operating system' (as stand-alone applications). This paper limits its focus to VBScript; however, the techniques for writing and executing Jscript and Perl would follow those described here for VBScript. Note that in order to run Perl on the WSH, it is first necessary to install a Perl language interpreter, such as ActiveState ActivePerl or MKS Pscript.

VISUAL BASIC SCRIPTING EDITION (VBSCRIPT)

VBScript is closely related to VBA, which serves as the standard scripting component to all MS applications. Practically speaking, the major difference between the two (other differences are briefly discussed in table 2) is that variables in VBScript can't be 'strictly typed'. Whereas variables in VBA can be declared as specific types ('floating point integers', 'boolean', etc.), in VBScript all variables are 'variants', such that values are resolved in a context-dependant way. In other words, the digit '2' may be treated as text or as a number depending on the context.

Table 2. Comparison of Visual Basic Scripting Edition/WSH and Visual Basic for Applications

Aspect	VBScript	VBA
Variable types	Variants (context-dependent interpretation)	Strictly typed (i.e. floating point integers, boolean, etc.)
Debugging	Limited; typically involves use of Message-Box (MsgBox) and PopUp commands to display values during execution.	Extensive, particularly within an integrated development environment, where programs can be 'stepped' through.
Development Environment	Any old text editor will do; functions as stand-alone since compiler facilities are built into the operating system.	Typically (in the SAS world) Windows applications such as Word and Excel, and less frequently 'Integrated Development' tools; if relying on compiler facilities in Word or Excel, can't function as stand-alone.
Intended purpose	Light version of VBA for web page automation, rapid development of custom utilities	Universal scripting component for Windows applications; large applications-development projects.

INTERACTION BETWEEN SAS AND THE WINDOWS SCRIPTING HOST

USING DATA_NULL_ TO WRITE 'EPHEMERAL' SCRIPTS

SAS can be involved in a WSH automation application in a variety of ways. The first and most obvious way is simply where SAS is used to write VBScript or Jscript source code in a data_null_. In this context, the 'dynamic' element exists in the SAS macro writing the script, where macro variables can be used to substitute specific data values each time the macro is called, but the resultant script is static. A common example would be where a path and/or file name was hard-coded into a script via a macro variable. For example:

```
%macro vbs_(InString);
filename "c:\wutemp\hi.vbs"; data _null_; file filename;
put 'MsgBox("&InString|'|")'; run;
%mend vbs_;
%vbs_(Hi);
```

The term 'ephemeral' applies to this type of script because once it has been called it can be over-written or deleted – there is no reason for the file to persist, since it can be re-created from the SAS macro, tailored to the requirements of the particular instance.

PASSING ARGUMENTS TO 'PERMANENT' SCRIPTS

The second form of interaction is in executing a script from SAS. Scripts generated through a data _null_ typically do not require arguments; however, stand-alone scripts are usually designed such that variables in the script are set dynamically through arguments passed in the script call.

For the script shown above, the call would simply be:

```
x 'c:\wutemp\hi.vbs';
```

If the script had been written as a stand-alone (receiving arguments), for example the following saved as "c:\wutemp\hi.vbs":

```
Dim InString
InString=WScript.Arguments(0)
MsgBox(InString)
```

, then the script call from SAS would look like:

```
x 'c:\wutemp\hi.vbs "Hi"';
```

In other words, arguments are passed to a stand-alone script simply by listing them after the path/file of the script.

USING SCRIPTS TO WRITE AND EDIT SAS PROGRAMS

The third form is in the use of VBScript text-streaming operations to write and edit SAS programs, which can then be executed in a SAS session launched as an ActiveX automation object. (This is essentially the reverse of the data _null_ generation of .vbs code). For example, this script generates and runs a simple PROC PRINT:

```
Const SasFolder = "C:\SAS\SASUSER"
Dim FSO, SasPth, TextStream
Set FSO = CreateObject("Scripting.FileSystemObject")
Set SasPth = FSO.GetFolder(SasFolder)
Set TextStream = SasPth.CreateTextFile("test.sas")
TextStream.Write("data test;a=1;output;proc print;run;")
TextStream.Close
Dim objSAS
Set objSAS = Wscript.CreateObject("SAS.Application.8")
objSAS.Visible = True
objSAS.Submit("%INCLUDE 'C:\SAS\SASUSER\test.sas';")
```

SWAPPING DATA BETWEEN SAS AND THE SCRIPTING ENVIRONMENT

SAS and the WSH need a way of swapping data back and forth. This would be necessary if, for instance, you wanted a script to respond to SAS output, to read and edit ODS HTML or rtf output, to transfer data between SAS and other applications (Access, Excel), or to provide a list of files and file attributes to SAS. The simplest and most obvious way of doing this is through simple text files: data _null_ and infile/input statements in SAS, and 'text streaming' operations using the File System Object in the scripting environment. Obvious drawbacks to this method are that you can't specify text and numeric data types in the text file, and your reading/parsing program in either environment must be very specific to the structure of the incoming file. A much more powerful method is to make use of the 'ActiveX Data Object' (ADODB Recordset) which enables you to transfer data directly from the native format of one application to the native format of another (well, not quite – Excel does not support ADODB Recordset, but you can still read data from an Excel file and write it to a SAS Recordset using SQL). Use of the ActiveX data object is illustrated in the last section of the paper.

DYNAMIC HTML

Although dynamic HTML does not strictly speaking make use of the Windows Scripting Host (although it is possible in dHTML scripts to pass commands such as 'PopUp' and CurrentDirectory to a WSH 'Wscript.Shell' object), it is worth mentioning here simply because VBScript and Jscript programs written to function as stand-alone applications on the Windows Scripting Host can usually be dropped verbatim into dHTML applications, and vice versa. For an introduction to dHTML/SAS applications, refer to Fairfield-Carter *et al.* (2004).

SCRIPT COMPONENTS – A BRIEF INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Although the terminology used to describe VBScript language components differs somewhat from that for SAS, and although VBScript makes use of 'object-oriented' architecture, after spending a bit of time writing VBScript code you will discover some fairly strong analogies to the more familiar SAS components.

OBJECTS

The fundamental component of an object-oriented language is of course the 'object'. The object can most easily be defined by means of examples: an instance of an application, a file, a chunk of text, and a directory are all objects. There are also some 'intangible' objects, such as the 'File System Object', which carries out file manipulation (and many other) operations.

PROPERTIES

Specific attributes of an object are referred to as 'properties'. Examples include 'visible' (the process is visible on the desktop), and 'CurrentDirectory' (the directory the script is launched from). Properties are set and/or determined through 'Object.Property' notation, as in

```
Dim objIE
Set objIE=CreateObject("InternetExplorer.Application")
objID.visible=true
```

METHODS

Specific actions that can be taken by objects are referred to as 'methods'. Examples include 'open' (a file), 'navigate' (to a web page), 'copy' (a file), and 'submit' (to a SAS session). Methods are called through 'Object.Method' notation, as in

```
Dim objIE
Set objIE=CreateObject("InternetExplorer.Application")
objIE.Visible=true
objIE.Navigate "c:\wutemp\test.htm"
```

OTHER LANGUAGE COMPONENTS

Once you have a grasp of 'Object.Property' and 'Object.Method' notation, many of the remaining core language elements (variables, functions, conditionals, and looping structures) bear a very strong resemblance to SAS. Rather than attempting to provide a complete language definition here, these features will be illustrated in the examples provided. For greater detail on scripting languages, it is well worth spending some time on the Microsoft Developers Network (MSDN) web site.

SAMPLE APPLICATIONS

WORD-PROCESSOR-READY DOCUMENTS AND FILE TYPE CONVERSION

Outside of data manipulation and summarization, creating word-processor-ready documents and file type conversion are probably the most common programming challenges for SAS programmers. The example below uses a data _null_ to generate an 'ephemeral' script. Each macro call generates a script that is hard-coded to a specific piece of output, so the script would ideally be deleted immediately after being called. For simplicity, this macro assumes that SAS output (for example, created via PROC PRINTTO) has been given a '.doc' file extension, though until it has been saved as a Word document it is actually just a text file.

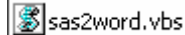
A point of particular interest is in the use of the 'ReloadAs' method. SAS form characters will often cause Word to assume that the file it is loading is 'encoded', and Word will not automatically default to Windows default encoding, but may select something extremely obscure. As a result, fonts and other document features will be incorrect in the output document. For some reason, you can't specify a default encoding when the file is being opened, but you can in the 'ReloadAs' method. In other words, the script must first open the file (incorrectly) and then immediately re-load it with the correct encoding. To further confuse things, the 'ReloadAs' method can't be called for files that have been opened using the 'Add' method (used to open a blank Word document, or open a file as a Word document), but can only be called for files that have been opened using the 'Open' method.

```
%macro vbs_word(outfile);
data _null_;
  attrib str format=$180.;
  file 'c:\sas\sasuser\sas2word.vbs'; /*(text file for vbscript code);
  put 'dim objwd';
  put 'set objwd = wscript.createObject("word.application");/*Create Word instance;
  str='objwd.documents.open("||"&outfile"||")'; put str; /*Open target file;
  put 'ObjWD.ActiveDocument.ReloadAs msoEncodingWestern'; /*Reload;
  put 'objwd.selection.wholestory'; /*Select the entire document;
  put 'with objwd.selection.font'; /*Set font attributes;
  put '.name = "courier new"'; put '.size = 9'; put '.bold = 0'; put 'end with';
  put 'with objwd.activedocument.pagesetup'; /*Set page orientation and margins;
  put '.orientation = 1'; put '.topmargin = 85'; put '.bottommargin = 73';
  put '.rightmargin = 57'; put '.leftmargin = 58'; put 'end with';
  str='objwd.activedocument.saveas("||"&outfile"||")'; /*Save as a word document;
  put str;
  put 'wscript.sleep 2000'; /*stall process for 2 seconds before quitting word;
  put 'objwd.documents.close'; put 'objwd.application.quit';
run;
options noxwait xsync; /*Set synchronous execution of 'x' process;
x "c:\sas\sasuser\sas2word.vbs"; /*(run the script);
%mend vbs_word;
```

A macro call to convert a SAS listing (with '.doc' file extension) to a Word document would look like:

```
%vbs_word(c:\wutemp\test1.doc);
```

In the 'c:\sas\sasuser' directory, assuming that you hadn't submitted instructions to delete the script after calling it, you would see the generated script file, where the '.vbs' extension associates the file with the Windows Scripting Host and displays it with the following icon:



For comparison, if instead of generating the script in a data _null_, the script had been written as a 'permanent' stand-alone, values would be assigned in the following way:

```
Dim objWD, InFile
InFile=WScript.Arguments(0)
Set objWD = WScript.CreateObject("Word.Application")
objWD.Documents.Open(InFile)
(etc.)
```

The script call from SAS would then look like this:

```
x 'c:\sas\sasuser\sas2word.vbs "C:\WUTemp\test1.doc";
```

A macro like that shown above can be easily re-tooled to import SAS/GRAPH output into Word documents. The key instructions for importing SAS/GRAPH cgm output would be:

```
put 'objwd.selection.inlineshapes.addpicture("c:\wutemp\temp.cgm").select';%*Insert;
put 'objwd.selection.inlineshapes(1).height = 360'; %*Set height;
put 'objwd.selection.inlineshapes(1).width = 580'; %*Set width;
```

, and of course there is considerable scope for additional functionality, such as setting up rules to determine values to pass to the 'MoveDown' method for positioning of the cursor before inserting the figure.

SAVING UNDER OTHER FILE TYPES

By supplying a parameter after the 'save as' file reference, you can use Word's output drivers to save under alternative file types:

- objwd.activedocument.saveas("test.doc"), 1 → saves as MS Word (file type parameter is optional)
- objwd.activedocument.saveas("test.rtf"), 6 → saves as RTF (position 6 in the 'save as' file type list)
- objwd.activedocument.saveas("test.htm"), 8 → saves as HTML
- (etc.; any file type in Word's Save As File Type drop-down list)

Writing post-script and pdf files is a bit more involved, since these are produced by printing to other output drivers; this process is described in the next section.

THE 'CURRENTDIRECTORY' PROPERTY AND MULTIPLE-FILE PROCESSING; HARD-COPY, POST-SCRIPT AND PDF PRINTING

The example above showed the use of a dynamically-generated, 'ephemeral' script to carry out processing on a single file, and the use of a 'permanent' stand-alone script, also to process one file at a time. A bit of additional book-keeping would allow you to pass output file references to this SAS macro via a macro variable (for example, using a system function to return the full path and file name of a 'filename' reference) rather than as hard-coded literal text. An alternative would be to get a reference to the target directory containing all SAS listing files (and/or SAS/GRAPH output files), and use a loop to process each file in that directory having the correct file extension.

The key ingredients to this alternative are the 'File System Object', which is the universal scripting object for file-handling and directory operations, the CurrentDirectory property of the WScript.Shell object (which returns the path name for the directory the script file is sitting in), and a bit of string manipulation in order to work with path/file names. The string manipulation functions used here are:

- StrReverse, which reverses the input string
- Split, which splits an input string at a specified character, and writes the 2 parts to an array
- Lbound, which retrieves the lower bound of an array (the number "1", in this example)
- Trim, which removes leading and trailing blanks
- Replace, which performs text replacement
- "&", the concatenation operator

Although the sample application given here deals with file printing (hard-copy, post-script, and pdf), it could easily be applied to the file type conversion discussed in the previous section, as well as to the task of concatenating multiple output files into a single document. Syntax for the 'PrintOut' method is a bit cryptic; for an explanation, refer to the section below entitled 'Finding Methods and Method Syntax'.

```
Dim objwd, wshshe, fs
Set WshShe = WScript.CreateObject("WScript.Shell")
Set fs= CreateObject("Scripting.FileSystemObject")
Set f = fs.GetFolder(WshShe.CurrentDirectory)
'(The CurrentDirectory property of the WScript.Shell object is passed to the
' GetFolder method of the File System Object, so 'f' is now a reference to the
```

```

' folder the script is sitting in)
Set fc= f.files          'The collection of files in the current directory
Set objwd = WScript.CreateObject("Word.Application")
objwd.visible=false     '(This can be run as an invisible process)
For Each fl in fc       'Loop through each file in the collection
    extends=(split(strreverse(fl.name), ".", -1, 1))
    '(Split the file name into file and file extension strings)
    prefix=strreverse(extends(lbound(extends))) 'Get the file extension
    torep=trim(".") & prefix 'Concatenate '.' to file extension
    suffix=trim(replace(fl.name, torep, " ")) 'Just the file name, no extension

    if ucase(prefix)="RTF" then 'Only if file is an rtf file
        size=int(((fl.size)/102.400)*3) 'Ad hoc calibration to estimate a sleep time
        if size<1000 then ' to allow the file to finish printing before
            size=1000 ' closing and going on to the next file
        end if
        pname=f & "\" & suffix & ".pdf" 'Create file name to copy the temporary file
        fname=f & "\" & fl.name 'Get the full path/file name of the rtf file to open
        tname="c:\temp.pdf" 'Temporary/default pdf output file name
        gname="c:\temp.pdf.pdf" 'To cope with an Acrobat PDFWriter idiosyncrasy -
            'always seems to create an additional copy with doubled file extension
        objwd.documents.open(fname) 'Open the rtf file
        objwd.activeprinter="Acrobat PDFWriter" 'Set active printer to pdf device
        objwd.ActiveDocument.PrintOut 0,0,0,tname,,,,,1 'Print to active printer
        wscript.sleep size 'Stall script execution while file prints
        objwd.ActiveDocument.Close() 'Close the rtf file
        wscript.sleep size
        fs.movefile gname, pname 'Move/rename temporary pdf file to final file
        set badfile=fs.getfile(tname) 'Set a reference to the duplicate temporary pdf
        badfile.delete 'Delete the duplicate
    end if
next
objwd.application.quit() 'Terminate the instance of Word

```

In order to create a post-script file, all you would need to do would be to change objwd.activeprinter to point to a post-script driver (you could also dispense with the code for dealing with the duplicate output file). For example, if you had installed the .ps device downloaded from the PharmaSUG web site, and called it Acrobat Distiller (Copy 1), then you would set the active printer to "Acrobat Distiller (copy 1)":

```
objwd.activeprinter="Acrobat Distiller (copy 1)" 'Set active printer to .ps device
```

You could then print to this device (you'd probably want to set the file extension of the output file to '.ps' to avoid confusion, though this is optional), and then, if you wanted, run the post-script file in an application such as Adobe PDFWriter, to produce a pdf file:

```
Dim objwd
set objwd=wscript.createobject("Word.Application")
objwd.Documents.open "c:\wutemp\test.doc"
objwd.activeprinter="Acrobat Distiller (Copy 1)"
objwd.ActiveDocument.PrintOut 0,0,0,"c:\wutemp\test.ps"
```

```
Dim wshell
Set wshell = WScript.CreateObject("WScript.Shell")
wshell.run Chr(34) & "C:\Program Files\Adobe\Acrobat 4.0\Distillr\AcroDist.exe " &
Chr(34) & "c:\wutemp\test.ps"
```

For more information on command-line syntax passed to the 'Run' method, refer to the section below entitled "Finding Application Object references and Command-line Syntax on the System Registry".

Finally, you could print the file to a hard-copy printer, simply by setting that as the active printer:

```
objwd.activeprinter="\\My Network\My Printer"
```

Or, alternately, setting that as the default printer:

```
Dim net
Set net = CreateObject("WScript.Network")
net.SetDefaultPrinter "\\My Network\My Printer"
```

, and then printing:

```
objwd.ActiveDocument.PrintOut '(No parameters are necessary)
```

What makes this attractive is that it gives you the opportunity to set a start time for your print job (for example, if you

wanted to print at 2:30am). You would specify the start time, the script would calculate the duration between that time and current system time, and the 'Sleep' function would suspend the process (without using up system resources) for that duration:

```
StartHour=InputBox("Enter Hour of Print Start")
StartMin=InputBox("Enter Minute of Print Start")
If StartHour < 12 Then StartHour=StartHour+12 'Assume a.m. hour means next day
If StartHour >= Hour(Time) Then 'If duration is a positive value
    Diff=(StartHour*60*60 + StartMin*60) - (Hour(Time)*60*60 + Minute(Time)*60)
    Diff_=Diff*1000 'Sleep time in milliseconds, as expected by the Sleep function
    Wscript.Sleep Diff_ 'Sleep for the desired duration
Else
    MsgBox("Start time is before current system time - cannot calculate sleep time")
End If
```

As a final point of interest, the CurrentDirectory/file-collection architecture could also be applied to SAS programs:

```
Dim ObjSAS
Set ObjSAS = WScript.CreateObject("SAS.Application.8")
...[get current directory, set up loop, check for .SAS file extension]...
ObjSAS.Submit("%include '" & f & "\" & f1.name & "';")
```

RE-ORDERING A LIST OF FILES

One deficiency of the above code is that it does not allow for printing of documents in a specific order (files are printed alphabetically, rather than consecutively by table number); this would also be a problem if you were using this type of multiple-file processing to concatenate files into a single document (note that in this case page numbers should be 'locked' first, if they are rendered by rtf code rather than being part of the body text; syntax would be: objwd.selection.fields.locked=true). One way around this is to embed the desired file order in the file name (for example, mydoc1.doc, doc2.doc, xdoc3.doc, adoc4.doc, etc.), parse the file name and print order into arrays, re-order the arrays by numeric print order, and then pass file names from the ordered array rather than from the file collection. File names can be written to array elements in a simple for/next loop:

```
Dim array(5)
Set fc= f.files '(The collection of files in the directory 'f')
i=1
For Each fl in fc
    Array(i)=fl.name
    i=i+1
Next
```

, and the ordering numbers captured by means of fairly straight-forward text manipulation. Here is one of many possible alternatives for ordering a VB array (this one 'shunts' values leftward in the array until all values to the left are smaller; it assumes the array has been set up initially with the left-most element empty):

```
Function SortArray(InArray)
    i = 3 'Start at position 3; assumes position 1 is empty to begin
    Do Until i = UBound(InArray) 'Work to the right up to the last array element
        If InArray(i) < InArray(i-1) Then 'If current array element smaller than preceding
            k = i
            Do Until (InArray(k-1) < InArray(k)) or k = 2 'Work leftwards, 'shunting' the
                Right_ = InArray(k) 'value by swapping places with the left adjacent one, until
                Left_ = InArray(k-1) 'either the next left value is smaller, or the left end
                InArray(k) = Left_ 'of the array is reached
                InArray(k-1) = Right_
                k = k - 1
            Loop
        End If
        i = i + 1
    Loop
End Function
```

MANIPULATION OF RTF CODE

Rtf code is extremely useful, because it can be parsed and edited as simple text, but rendered as visually complex documents. Occasionally you may run into special characters that need to appear in SAS output that can't be retrieved through the BYTE function. In these cases, it may be convenient to create SAS output as .rtf, with a literal text 'place-holder' (such as /*Infinity*/), and then edit the .rtf as a text stream in order to replace the literal with the rtf code to produce the desired symbol (in this case, ∞):

```
Dim FSO, File, textin, textout, string, stringout
Set FSO = CreateObject("Scripting.FileSystemObject")
Set File = FSO.GetFile("c:\wutemp\test.rtf")
```

```

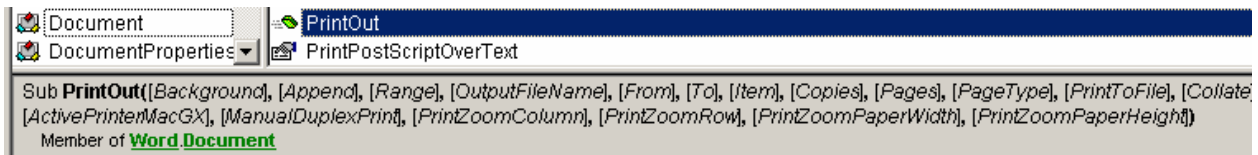
Set textin = File.OpenAsTextStream 'Open the source file as a text stream
Set textout = FSO.CreateTextFile("c:\wutemp\test_modified.rtf", True)
'(Open the output file as a text file for writing)
Do While Not textin.AtEndOfStream 'Loop through each line of the source file
string = textin.ReadLine & NewLine 'Read the line
stringout = Replace(string, "/*Infinity*/", "\u8734\`38")
'(Replace literal text with rtf code for desired character)
textout.Write(stringout) 'Write the modified line to the output file
textout.WriteLine("") 'Carriage-return to the next line of the output file
Loop
textin.close 'Close the input text stream
textout.close 'Close the output text stream

```

FINDING METHODS AND METHOD SYNTAX

For methods such as 'PrintOut', the number and order of arguments may not be readily apparent. Often the quickest way to determine expectations of a particular method is by recording a VBA macro in Word (or Excel), and then 'translating' to VBScript (remembering of course that values are stored as 'variants', and that recorded macros will inevitably contain many lines of irrelevant code).

Another useful tool is the 'Object Browser' in Word's Visual Basic Editor, which provides a template for the syntax of each method. The screen-shot below shows details of the 'PrintOut' method, under 'Document'. The output file name is specified in argument 4, and 'print to file' is specified as argument 11. Many arguments can be ignored altogether, and will simply resolve to default values. Arguments such as 'PrintToFile' are Boolean (true/false), or 1/0 as VBScript variants, so referring to the object browser is helpful for deciphering cryptic VBScript method calls.

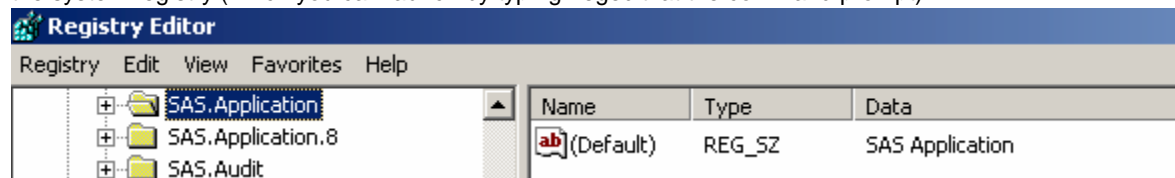


FINDING APPLICATION OBJECT REFERENCES AND COMMAND-LINE SYNTAX ON THE SYSTEM REGISTRY

In some cases, unless you have sample code to borrow from, some research may be required in order to determine the correct reference for an application object. For example, the statement

```
Set objSAS=Wscript.CreateObject("SAS.Application")
```

may launch an instance of SAS version 8 on some machines but not others. The reason for this is shown by viewing the system registry (which you can launch by typing 'regedit' at the command prompt):



If "SAS.Application" appears in the registry, and 'current version' points to "SAS.Application.8", then the CreateObject call above will work. If "SAS.Application" does not appear, then the CreateObject call must reference "SAS.Application.8".

The system registry is also useful for providing command-line syntax for certain operations. For example, if you have created an instance of SAS with a CreateObject call, you might assume that you could submit some sort of 'objSAS.Open' command in order to open a SAS program, as you would for an instance of Word. SAS does not appear to support such a method, leaving you with the following options, using the 'submit' method:

```
objSAS.Submit("dm pgm 'inc c:\wutemp\test.sas';")
```

or

```
dmstr="inc c:\wutemp\test.sas;"
objSAS.Command(dmstr)
```

, both of which perform unpredictably (as in, they may not always open the target file).

An alternative is to execute a command-line using the 'Run' method on an instance of "WScript.Shell" (the command syntax is determined from the system registry, as shown in the screen-shot below):

```

Dim wshell
Set wshell = CreateObject("WScript.Shell")
wshell.Run Chr(34) & "C:\PROGRA~1\SAS\INS~1\SAS\V8\CORE\SASEXE\SASOACT.EXE" & Chr(34)
& " action=Open datatype=SASFile filename='c:\wutemp\test.sas'

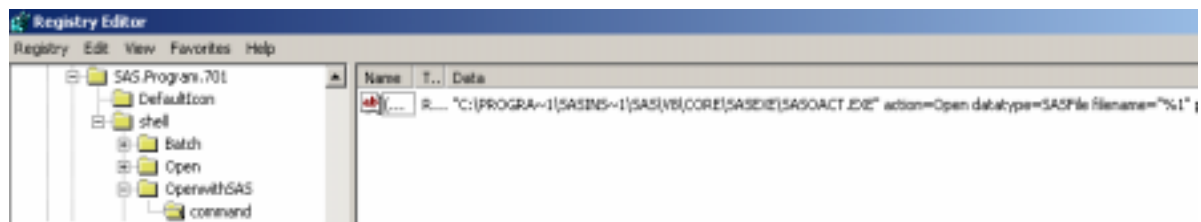
```

```
progid=SAS.Application.8"
```

The 'Chr(34)' function calls return a double-quote, which is useful when you need to double up on the double-quotes (as is required in this example, since the path name to the executable file contains blank spaces, and so must be enclosed in double quotes, while the string passed to the 'Run' method must also be in double-quotes). Of course, the instance of SAS that you now have was not spawned as an ActiveX object, so in order to get an object reference to it you need to call the 'GetObject' method:

```
dim objSAS
on error resume next
While IsObject(objSAS)=False
Set objSAS = GetObject(, "SAS.Application.8")
If IsObject(objSAS)=True Then
objSAS.Submit("%let testvar = TESTVALUE;")
End If
Wend
```

As you can see, this is not without its own set of problems, since you need to wait until the application instance exists before you can create a reference to it and start submitting program statements. Interestingly, the fact that this works appears to contradict SAS Institute claims that you can't get an object reference to an instance of SAS that was not spawned as an ActiveX process.



FILE HANDLING

Previous examples have illustrated the use of the File System Object for retrieving file and path names, for looping through collections of files, for reading and writing text files, and for copying, moving, re-naming and deleting files. Additional capabilities include retrieving file attribute information (size, date created, etc.) as properties of the 'file' object, creating directories and subdirectories, and testing for the existence of target files.

ADODB RECORDSET – USES OF THE ACTIVEX DATA OBJECT

INTRODUCTION

Although the use of simple text files is an inviting starting point for swapping data between SAS and the scripting environment, the 'ActiveX Data Object' (ADO) is by far the preferred method, first for the obvious reason that you don't have to manage stray text files, but more importantly because it provides a very powerful data interchange between SAS and other applications such as Excel and Access. With very little work, you can create your own SAS <--> Excel and SAS <--> Access import/export tools, tailored with any additional data manipulation steps you might care to add. A key point is that you can do this without necessarily creating an instance of either application.

Working with ADO is similar to working with other 'application objects' described so far – you create an instance of 'ADODB.RecordSet' by using the 'CreateObject' method, and call methods in this object. However, the requirements for a data interchange also include a database connection ('ADODB.Connection'), also created using 'CreateObject' and qualified by a specific 'connection string'. The connection string indicates the data 'provider' (the 'engine' that reads and writes the particular data files) and the data source.

CREATING SAS DATA

To illustrate, say you wanted to create a SAS dataset. First you would create instances of the necessary objects:

```
Dim RecSet, Connection
Set RecSet = WScript.CreateObject("ADODB.Recordset")
Set Connection = WScript.CreateObject("ADODB.Connection")
```

Then you would use the ADO.Connection object to connect to SAS (specifically, to SAS's database engine) by passing a connection string as an argument to the 'Open' method, and set the 'ActiveConnection' property of the ADO.RecordSet object to this connection:

```
Connection.Open "Provider=SAS.IOMProvider.1;Data Source=_LOCAL_"
RecSet.ActiveConnection = Connection
```

Now all you need to do in order to create the SAS dataset is pass an SQL sentence (as literal text) as an argument to the 'Execute' method on the active connection (since a SAS session is not launched, an automatic library such as 'sasuser' or 'sashelp' must be used):

```
Connection.Execute "create table sasuser.recset (test_text char(20), test_num num)"
```

This creates a SAS dataset with 2 variables, the first character with length 20, and the second numeric, and with no

records. In order to populate this dataset, we first need to open it for writing on the active connection:

```
RecSet.Open "sasuser.recset", Connection, 2, 4, 512
```

Next we need to create records ('AddNew'), assign values to variables, and add the records to the data set ('UpdateBatch'):

```
Dim i
i=10
Do Until i = 0
  RecSet.AddNew
  RecSet.Fields(0).Value="number" & i
  RecSet.Fields(1).Value=i
  RecSet.UpdateBatch
  i = i - 1
Loop
```

Finally, say we wanted this data set to be re-ordered in increasing values of the numeric variable – all we'd need to do would be to call the 'Execute' method again with the appropriate SQL sentence:

```
Connection.Execute "create table sasuser.recset1 as select * from sasuser.recset
order by test_num"
```

WRITING EXCEL DATA TO SAS

Now that we have the basics of creating and manipulating SAS data, we can move into some practical examples. Probably the first and most obvious one is that of writing Excel data to a SAS dataset. From the code above, we can see that we need to know

1. The location of the source data, and how to read it
2. The variables and attributes of the source data, to plug into the first 'Create Table' statement
3. The number of records on the source data, to determine the size of the loop used to populate the SAS data set.

Because Excel lacks the database engine component of a true database application, data can only be read by first creating an instance of the application (which, fortunately, can be invisible), then opening the target spreadsheet, and reading values via row/column references. For the sake of simplicity, the file reference and cell range have been hard-coded into this example, though you will no doubt recognize the potential for passing these in as arguments. (The potential for determining the cell range 'dynamically', by parsing rows and columns to determine the starting point of missing values on each dimension, should also be apparent.)

First, create the ADO objects and open the connection:

```
Dim RecSet, Connection
Set RecSet = WScript.CreateObject("ADODB.Recordset")
Set Connection = WScript.CreateObject("ADODB.Connection")
Connection.Open "Provider=SAS.IOMProvider.1;Data Source=_LOCAL_"
RecSet.ActiveConnection = Connection
```

Next, create an instance of Excel, and select the target data (column A row 1 to column F row 10) in the target workbook page:

```
Dim appXL
Set appXL = WScript.CreateObject("Excel.Application")
Set GetBook = appXL.Workbooks.Open("C:\WUTemp\xls2sas\test.xls")
Dim GetSheet, GetRange
On Error Resume Next
Set GetSheet = GetBook.Sheets("Sheet1")
Set GetRange = GetSheet.Range("A1:F10")
```

Next, starting at row 2, parse each column to determine an appropriate data type (it is assumed that row 1 contains the variable names, with no blank spaces such that they can be used as SAS variable names):

```
Function ParseColumn(Col)
  Dim Lngth, Lngth_test, Typ
  Lngth=8      'Start with a default of numeric, length 8
  Typ="num"
  For Rw = 2 to GetRange.Rows.Count - 1 'For every value in the column
    Test = GetRange.Cells(Rw,Col).Value 'Capture the value
    Lngth_test = Len(Test) 'Capture the length of the value
    If Lngth_test > Lngth Then Lngth = Lngth_test 'Capture the greatest length
    If Typ="" or Typ="num" Then
      If IsNumeric(Test)=False Then
        Typ="char" 'If ANY non-numeric found, set type to character
      End If
    End If
  Next
  If Typ="num" Then Lngth=8
```

```

    ParseColumn=Typ & ":" & Lngth 'Return a 'type:length' string
End Function
'Call the function for each column
Dim Typ, Lngth '(note that arrays must be 'ReDim'ed to the desired size)
ReDim Typ(GetRange.Columns.Count-1)'Array to hold appropriate data type for each var
ReDim Lngth(GetRange.Columns.Count-1) 'Array to hold appropriate length for each var
For i = 1 to GetRange.Columns.Count
    RString=ParseColumn(i) 'Get the 'type:length' return value
    Typ(i)=Left(RString, Instr(ParseColumn(i),":")-1) 'Pull off the type...
    Lngth(i) = Right(RString, Len(RString)-Instr(ParseColumn(i),":")) '...and length
Next

```

Using variable names captured from row 1 and attributes determined by parsing each column, generate and execute an SQL sentence to create a shell SAS data set:

```

sql = "create table sasuser.recset ("
bFirst=True
For j = 1 to GetRange.Columns.Count
    If bFirst=False and j<GetRange.Columns.Count Then sql = sql & ", "
    bFirst = False
    sql = sql & " " & GetRange.Cells(1,j).Value & " " & Typ(j) & "(" & Lngth(j) & ")"
Next
sql = sql & ")"
Connection.Execute sql

```

Finally, open and populate the SAS data set:

```

RecSet.Open "sasuser.recset", Connection, 2, 4, 512
For i = 2 To GetRange.Rows.Count 'For each row
RecSet.AddNew 'Add a record
    For j = 1 To GetRange.Columns.Count 'For each column
        RecSet.Fields(j - 1) = GetRange.Cells(i,j).Value 'Assign the value
    Next
RecSet.UpdateBatch 'Update the SAS data set with the newly populated record
Next
appXL.application.quit 'Terminate the instance of Excel

```

WRITING ACCESS DATA TO SAS

The process of writing Access data to a SAS dataset is quite similar, but streamlined in that you don't need to launch an instance of Access, and variable names and attributes are provided as properties of RecordSet.Field. The major steps are:

1. Create RecordSet and Connection objects for SAS (as above)
2. Create RecordSet and Connection objects for Access, the only difference being the connection string, and the absence of a data library reference required with SAS:

```

Connection.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\WUTemp\test.mdb"
AccessRecordSet.Open "test", Connection, 2, 4, 512

```

3. Capture variable names and attributes from the Access table, and use these to create the shell SAS data set (note that numeric codes for different Access data types can be determined by using a MsgBox(field.Type) function call for each field in RecordSet.Fields):

```

bFirst = True
sql = "CREATE TABLE " & SASName & " ("
For Each fld In AccessRecordSet.Fields
    If Not bFirst Then sql = sql & ", "
    sql = sql & fld.name
    If fld.Type=129 Or fld.Type=200 Or fld.Type=130 Or fld.Type=202 Then
        sql = sql & " CHAR(" & fld.DefinedSize & ")"
    ElseIf fld.Type=5 Or fld.Type=11 or fld.Type=3 Then
        sql = sql & " NUM"
    ElseIf fld.Type=7 Then
        sql = sql & " DATE"
    Else
        sql = sql & " Char(255)" '(If no attribute is provided, force as text)
    End If
    bFirst = False
Next
sql = sql & ")"

```

4. Read values from the Access table, and write to the SAS data set (note that SAS and Access use different default arbitrary cut-off dates when storing date values, so a conversion is required):

```

Dim col

```

```

While Not AccessRecordSet.EOF
  SASRecordSet.AddNew
  For col = 0 To AccessRecordSet.Fields.Count - 1
    If AccessRecordSet.Fields(col).Type=7 Then 'If source value is a date...
      SASRecordSet.Fields(col).Value=AccessRecordSet.Fields(col).Value - 21916
      '(Convert to a SAS date equivalent (different cut-off dates!!))
    Else 'Otherwise, just use the value as is
      SASRecordSet.Fields(col).Value=AccessRecordSet.Fields(col).Value
    End If
  Next
  SASRecordSet.UpdateBatch
  AccessRecordSet.MoveNext
Wend

```

FURTHER APPLICATIONS OF ADO

Although the above examples show the process of writing *to* SAS datasets, in a way that mimics PROC IMPORT, the direction can easily be reversed to mimic PROC EXPORT. One thing to note is that SQL syntax in Access has its own peculiarities, so some adjustment is necessary if you are used to working with PROC SQL in SAS. ADO can also be used for moving data between SAS and other native formats, notably Oracle. Data manipulation in ADO offers quite a number of interesting possibilities: text files can be written to record sets, and SQL then used for processing (for example, to edit rtf code, and to do things like parse SAS log files for errors and warnings). A particularly interesting instance of rtf manipulation is in modifying ODS/rtf output to switch floating headers and footers to body text. In addition, combining multiple Access tables or Excel spreadsheets can be done in a single export step, rather than importing each table to SAS and then merging.

CONCLUSION

The Windows Scripting Host provides a powerful and comprehensive tool for all of the 'secondary' problems encountered by a SAS programmer, including file handling (moving, copying, renaming, deleting), creating word-processor-ready output, and supporting alternate file types (post-script, pdf, html, rtf, etc.). ActiveX Data Object programming extends this list to include transferring data between SAS and other native formats such as Excel and Access.

REFERENCES

Fairfield-Carter, Brian, Sherman, Tracy, and Hunt, Stephen (2004), "Instant SAS® Applications with VBScript, Jscript, and dHTML", Proceedings of the 2004 Pharmaceutical Industry SAS Users Group Conference.

ACKNOWLEDGMENTS

The authors wish to thank first and foremost Jeff Carter of Equinox Software Design (www.equinox.ca) for introducing the possibilities offered by the Windows Scripting Host and for providing many helpful insights and programming tips. We would also like to thank Cara, Kelly, Gavin, Kurtus, PJ, Nicholas, our friends in Biostatistical Services, and Tim Williams for their consistent encouragement and support.

CONTACT INFORMATION

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

Stephen Hunt,
 Tracy Sherman,
 Brian Fairfield-Carter
 PRA International
 600-730 View Street
 Victoria, BC (CAN) V8W 3Y7
 Work Phone: 250-480-0818
 Fax: 250-480-0819
 Email: HuntStephen@PRAIntl.com
 ShermanTracy@PRAIntl.com
 FairfieldCarterBrian@PRAIntl.com
 Web: www.prainternational.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.