Paper 5-27

Using Dynamic Data Exchange to Export Your SAS® Data to MS Excel — Against All ODS, Part I —

Koen Vyverman, Alphanor Data Mining Inc.

Abstract

Of all the different ways in which the SAS System allows data export into a Microsoft Excel spreadsheet, Dynamic Data Exchange (DDE) is the only technique providing total control over the Excel output. As is often the case however, this high level of control comes at a price ... The DDE formalism can appear quite daunting at times, even downright obscure! The purpose of this paper, liberally sprinkled with recyclable Working Code™, is to provide a detailed walkthrough of how to programmatically create and customize an Excel workbook containing SAS data. More or less in order of decreasing obviousness, we will cover the following specific topics: firing up Excel from within a SAS session; loading and saving an Excel workbook; inserting SAS data into a given worksheet; formatting a range of spreadsheet cells; adding an Excel macro sheet to a workbook; renaming a worksheet; finding out the names of existing worksheets in an Excel workbook. As a bonus, a SAS macro is presented, harnessing the power of DDE in an easy-to-use one-liner!

1 Introduction

With the version 7 release of the SAS System, the Output Delivery System (ODS) made its first appearance. As compared to earlier versions, the ODS has greatly increased the number of file-formats that SAS output can be directed to. However, an easy and flexible mechanism for generating custom MS Excel and MS Word documents is still lacking. The ubiquity of the MS Office suite of applications has lead to a situation where most business users are very familiar with Word and Excel. They know how to handle the files, they know how to edit them, etc. Generally they'll feel most comfortable when presented with SAS output formatted either as a Word document (for stationary reporting, tables & graphs) or as an Excel workbook (for interactive reporting, data exploration).

The key word here is *custom:* of course it is possible to go for the 'common file format' strategy and produce ODS RTF (Rich Text Format) output that can be read by Word. Similarly, Excel can read CSV-files (Comma Separated Values) as well as tabular HTML. The improved `proc export` makes it easier than ever to simply dump the contents of a SAS data set into an Excel file. But none of the above will allow populating existing Word and Excel files with bits of SAS data precisely *where* they are wanted, and formatted exactly *how* they need to appear. For an overview of different methods and their varying degrees of functionality, see Kuligowski (1999) and Mumma (1999).

This is where Dynamic Data Exchange (DDE) comes to the rescue. Think of DDE as the way to achieve free format Word and Excel file generation. Using DDE, most things that can be done manually in Word and Excel can be automated from within a base SAS program. DDE enables a SAS session to take control of the Word and Excel applications and, like puppets on a string, tell them exactly what to do.

Since DDE is only available on the OS/2 and Windows platforms, this pretty much sets the stage for what follows. The SAS System supports DDE as of the 6.08 release. All SAS code in the present paper was tested on a Windows NT box (SP4), running MS Office 97 and release 8e of the SAS System. As the title indicates, we will only be concerned with the matter of using DDE between the SAS System and Excel. In a forthcoming paper (which will be subtitled "Against All ODS, Part II"), the possibilities of DDE for the purpose of creating custom MS Word documents will be explored.

2 DDE Preliminaries

Before barging ahead with code samples, it is necessary to explain a few basic things about DDE. A bit of theory, so to speak, but without going into too much detail. First of all, what *is* DDE? Dynamic Data Exchange is a communication protocol available on Windows and OS/2, enabling certain applications on these platforms to talk to each other in a client/server fashion. Among the DDE-compliant applications, we find Borland Paradox, Lotus 1-2-3, MS Access, MS Excel, MS Word, MS FoxPro, MS Word, Corel Quattro Pro, and presumably a swath of others as well.

The SAS System on Windows and OS/2 is only DDE-compliant in the sense that as of release 6.08 SAS can act as a DDE-client but not as a server. Remember that in the client/server paradigm, the client application is the one that initiates a conversation with the server application, and asks the server app to perform a specific task. In our case, the internal language by means of which the client and the server converse is DDE. See the Institute's Technical Support document TS325 (SAS Institute, 1999) for more information about the general idea of using DDE between the SAS System and the aforementioned pc applications.

A prerequisite for using DDE is that both client and server applications are up and running. Some consider this a nuisance, but it is actually only a minor inconvenience considering the many fancy possibilities offered by DDE. It is possible to programmatically launch and kill Excel, but we will come to that in due course. Furthermore, if data are to be written to — or read from — one of the server's files, this file must be open and active in the server application.

Once both a SAS session and Excel are running, a client/server DDE communication link is initiated by means of a special form of the SAS `filename` statement. It comes in two flavors:

```
filename <fileref> dde
  '<server app>|<topic>!<item>';
```

and

```
filename <fileref> dde
  '<server app>|system';
```

In the first of these, the expression `<server app>|<topic>!<item>` is known as the DDE-triplet. It simply tells the SAS session which server application to talk to, which file the server has available, and what part of this file to read/write data from/to. An example will clarify things. In our specific case of writing SAS data to a range of cells in an Excel workbook/worksheet, the components of the triplet become:

```
<server app> = Excel
<topic>      = [<workbook.xls>]<worksheet>
<item>       = a cell range of the form
               RiCj:RmCn
```

Suppose *e.g.* that a file 'Some Data.xls' is open and active in the Excel application. Excel files (or 'workbooks') are container documents that can hold a number of spreadsheets ('worksheets'), graphics sheets, macro sheets, ... Suppose that 'Some Data.xls' has a worksheet named 'Stuff from SAS' to which we wish to write data in the cell-range defined by cell A1 in the upper left corner and E3 in the lower right corner. The full triplet-style `filename` statement to set up a DDE communication channel then becomes:

```
filename sasstuff dde 'excel|[some
  data.xls]stuff from sas!r1c1:r3c5';
```

Once a SAS fileref has been defined in this manner, it can be used just like any other SAS fileref on `file` and `infile` statements in subsequent data steps. The usual `put` and `input` statements will then write to and read from the specified cell-range.

The second form of the DDE `filename` statement uses the special value `system` as the DDE topic, and has no item at all. Since it can therefore hardly be called a triplet any longer, we will refer to it as a DDE doublet, or system-doublet. A real-life example in our case will look as follows:

```
filename xlsystem dde 'excel|system';
```

The functionality of a doublet-style DDE fileref is profoundly different from the triplet-style fileref. Whereas the triplet opens a link to a specific cell-range in an Excel workbook/worksheet, the system-doublet will allow the client application to send system-level commands to the server. In our present case, this means that from within a base SAS program, using a DDE doublet like the above, we will be able to tell Excel what to do, exactly as if we were manually operating menus and clicking around in the Excel workspace.

Obviously, those system-level commands need to be sent in a language that the server is capable of understanding. For Excel, this means that the Excel version 4 Macro Language (X4ML) must be used. As of Excel version 5, Microsoft has replaced X4ML by Visual Basic for Applications (VBA), which is actually more like a proper programming language than X4ML. Unfortunately it is impossible to send VBA commands through a DDE link. It *is* possible to run existing VBA macros that are stored in *e.g.* the active workbook. Roper (2000b) shows how to do this through a DDE connection, and Stetz (2000) discusses the same technique applied to Word rather than Excel.

This method of running stored VBA macros has one drawback though: when Excel opens a workbook that has embedded macros, a warning box to that effect will pop up, requiring user interaction — a simple click, but still … — and interrupting an otherwise fully hands-free process. It is of course possible to turn these pesky macro-warnings off, but that may not always be a wise decision. Moreover, if one designs code that is supposed to run seamlessly on other people's machines, it is probably better to avoid VBA altogether.

However, depending on what one is actually trying to accomplish by means of DDE, the use of stored VBA macros is at times unavoidable. Consider this: the later versions of Excel are nicely backward compatible and will eat any X4ML that is thrown at them. Well, almost any, but we will come to that later in the section about renaming a worksheet. X4ML as a language however has presumably stopped evolving when VBA took over as Microsoft's macro language of choice. As a consequence, features that were added to Excel *after* version 4, may *not* have an X4ML equivalent. When running into one of those, a small VBA macro would need to be written or recorded.

A commonly voiced complaint regarding X4ML is that the commands are arcane — true, but then, isn't SAS at times? — and that one cannot figure out which commands are available, and what their precise syntax is. The latter point should really not deter anyone from using DDE: by far the easiest thing to do is to surf to the main Microsoft web-site, locate the technical support area, and use its search facility to find a file named 'macrofun.exe'. It sounds somewhat like a suspicious e-mail attachment, but it really is the installer for the 'Macrofun.hlp' help file. This help file is a priceless tool for the X4ML developer as it explains full syntax of hundreds of functions, has some examples, and has cross-references to related commands and functions. A printed manual equally exists (Microsoft, 1992) for those who prefer to browse hardcopy. Finding one will be a matter of luck however, as it used to be part of the printed documentation for Excel 4, which is of course long since out of print.

In fact, having a copy of both the help file *and* the printed manual is ideal. The topics covered largely overlap, but there *are* things which only appear in either the manual or in the help file. For example, the manual discusses a second argument to the 'workbook.activate' function, while there is no mention thereof in the help file. On the other hand, the help file has an entry for the 'filter' function, which is lacking in the printed version.

3 Starting Up Excel

Now we can get serious. Over the past few years, several people have suggested different methods for complying with the first premise of DDE: our server application, Excel, must be up and running on the same pc as the SAS session from which we want to initiate a DDE connection. How to start up Excel programmatically from a SAS session? And equally important, how to avoid launching another instance of Excel should there be one running already?

The solutions offered vary in speed and reliability. A solution that works on one Windows-box may not work on a seemingly identical one. Generally, one can distinguish three distinct techniques, one of which relies on the use of the SAS `call modulen` function to run external Winapi routines. This would lead us too far out into dangerous waters, so we shall concentrate on the remaining two techniques, the first and oldest of which uses a combination of the SAS `sleep` function and an `x` statement to issue a host command. The code — in a simple macro-wrapper for ease of use — goes as follows:

```
%macro startxl;
  filename sas2xl dde 'excel|system';❶
  data _null_;
    file sas2xl;❷
  run;
```

```
    options noxwait noxsync;❸
    %if &syserr ne 0 %then %do;
      x '"c:\program files\microsoft office\
          office\excel.exe"';❹
      data _null_;
        x=sleep(10);❺
      run;
      %end;
  %mend startxl;
  %startxl;
```

❶ A doublet-style DDE filename is defined. This can be done, independent of the fact whether Excel is running or not.

❷ In this tiny data step, all we do is gently poke the DDE fileref. At this point, it *does* make a difference whether Excel is running or not: if Excel is unavailable, the automatic macro variable `syserr` will receive a non-zero value, and an error message is displayed in the log.

❸ The `noxwait` and `noxsync` options are there to ensure that the upcoming `x` statement executes independently from the SAS session.

❹ The `x` statement simply contains the full path of the Excel executable on the current system. Thanks to the bit of conditional macro logic, it will fire up Excel only when it is necessary to do so.

❺ While Excel starts up, we briefly put the SAS session to sleep, this to avoid that further attempts at DDE communication take place before Excel is entirely ready for it.

The disadvantages of the above technique are clear: the path of the executable needs to be hard-coded, and the number of seconds that SAS is put to sleep must be sufficient to allow Excel to start up also in worst case scenarios, *e.g.* when the system load is high and everything simply takes much longer than usual. The technique's main advantage is that it will almost certainly work in most imaginable circumstances. Recently, Roper (2000a) published a clever variation on the above, wherein he takes advantage of the Windows Registry and some of the SCL functions that have been ported to base SAS. An implementation of his solution may look like this:

```
options noxsync noxwait;
filename sas2xl dde 'excel|system';
data _null_;
  length fid rc start stop time 8;
  fid=fopen('sas2xl','s');❶
  if (fid le 0) then do;
    rc=system('start excel');❷
    start=datetime();
    stop=start+10;❸
    do while (fid le 0);
      fid=fopen('sas2xl','s');❹
      time=datetime();
      if (time ge stop) then fid=1;
      end;
    end;
  rc=fclose(fid);
run;
```

❶ The SCL-function `fopen` will return a positive integer if the fileref `sas2xl` is working properly, *i.e.* if Excel is running (this is the equivalent of poking the fileref in the first technique).

❷ If such is not the case, the host command 'start excel' is issued. If all goes well, this is where Excel starts coming up. The proper functioning of this command depends on whether the Excel application has been decently registered in the system's Registry upon installation. This step is the equivalent of the `x` statement used in the first technique, only here it is not necessary to provide the path, as the 'start' command will use the Registry.

❸ The moment that the attempt at starting Excel is made, is then captured in the date-time variable `start`, and a maximum waiting period of ten seconds is defined.

❹ During at most these ten next seconds, SAS keeps checking whether Excel has finally started up. This is a definite improvement over putting the entire SAS session to sleep, because as soon as Excel is indeed ready to listen to DDE commands, the `do while` loop exits and the SAS session can continue processing.

Note that while this second technique should be the preferred one — to say the least, it will be the more efficient one on faster machines — the fact that it relies on the Registry has been known to make it fail on older machines, notably Windows95 systems.

Also, both techniques refrain from opening a new instance of Excel if one is already available. This means that subsequent use of DDE will hijack the active Excel application. For this reason one should take care when running scheduled batch SAS jobs on users' machines. If a SAS session suddenly takes control over Excel while a user is working on some spreadsheet, nasty things might happen.

4 Loading and Saving a Workbook

As a next step we need to give Excel a workbook. We can either create a new one, or open an existing file. To create a new workbook containing a single empty worksheet (note that the default worksheet name is 'Sheet1'), we use:

```
data _null_;
  file sas2xl;
  put '[new(1)]';
run;
```

The parameter value of the X4ML function 'new' determines the type of sheet created in the active workbook. A value of '1' will produce a new normal worksheet. The value '2' makes a graphic sheet, and '3' creates an X4ML old-style macro sheet. The statement as used above omits some parameters which therefore revert to their default values. We will not go into full syntax detail for each X4ML function used in this paper. Suffice it to say that X4ML functions tend to have a more varied usage than what is shown, and that the reader is encouraged to find out about the possibilities via the 'Macrofun.hlp' help file.

If we'd rather work with an existing workbook, we use the X4ML function 'open' as follows:

```
data _null_;
  file sas2xl;
  put '[open("c:\koen\sugi 26 paper\
       existing workbook")]';
run;
```

This will open the file 'Existing Workbook.xls' in the specified location, and make it the active document. Again, the 'open' function takes many other parameters than just the filename: if one attempts to open a password-protected workbook, it is possible to specify the password; for a workbook containing external references, one can dictate whether or not to update the linked values; a read-only flag can be set; and so on …

The file 'Existing Workbook.xls' is a typical Excel Workbook. It contains four sheets: 'Random Numbers' has three columns of, well, random numbers … 'Bubble Graph' is a graphic sheet, visualizing those three random variables in a bubble-chart. The first two columns from 'Random Numbers' define x and y positions, the third column determines bubble-size. 'Early Morning Reminder' is a normal worksheet again, with some dummy text-data. And 'Records' is an empty worksheet.

Readers wishing to try out all the sample code as we go along, are advised to re-create on their system an 'Existing Workbook.xls' as outlined above. The order of the sheets has no importance, only their name and the type of content have. Obviously, some paths will need to be modified depending on where the workbook is stored in the file system.

Saving a workbook can be equally simple:

```
data _null_;
  file sas2xl;
* put '[error(false)]';
  put '[save.as("c:\koen\sugi 26 paper\
        Saved Before Modifying")]';
  put '[file.close(false)]';
run;
```

Note the statement that is commented out. The X4ML 'error' function is used to toggle Excel error-checking on ('true') and off ('false'). When trying to automate document creation in Excel, it is advisable to switch off the error-checking, because it will keep Excel from displaying any message-boxes that would stop the process until they are clicked away.

It is important to realize that turning the error-checking off will also suppress warning messages, like the one asking if changes should be saved before closing a file, or the one pointing out that a file with the same name already exists and if it really should be overwritten with the one that is being saved …

Unsurprisingly, the 'file.close' function closes the active workbook. The 'false' parameter prevents another save.

5 Inserting SAS Data

For most of the remainder of this paper we will work with the 'Existing Workbook.xls' rather than with a new blank workbook. First we create a sample of SAS data, say, an excerpt from a record store's catalogue with two character variables for band-name and album-title, and two numerical variables for the album's release date and the unit retail price (the latter are fictitious):

```
data price_list;
  length band album $20 date price 8;
  informat date date9.;
  input band $1-10 album $12-23 @25 date
        @35 price;
  cards;
Waits, Tom Bone Machine 23FEB1992 19.95
VdGG       Godbluff     15MAY1975 8.50
Pixies     Bossanova    08NOV1990 14.99
Cave, Nick Tender Prey  10AUG1988 17.49
;
run;
```

Suppose we want to write the contents of this `price_list` data set into the empty 'Records' worksheet that exists on 'Existing Workbook.xls'. Since we have four variables and four observations, we need to define the 4x4 target cell-range by means of the following triplet-style DDE filename statement as discussed in section 2:

```
filename recrange dde 'excel|[existing
  workbook.xls]records!r1c1:r4c4' notab;
```

Note that the '.xls' filename-extension is necessary here. Omitting it will result in a 'Physical file does not exist' error when trying to use the fileref `recrange` later on. The `notab` option is also a must: for some reason, the default for DDE communications between SAS and Excel, is to translate *every* blank between any two tokens in the output stream into a tab-character. This is bad because the implicit delimiter between two Excel cells is also a tab-character. So, by default, a SAS character variable containing words separated by blanks would end up smeared out over consecutive worksheet cells, which is mostly undesirable. The `notab` option will prevent this blanks-into-tabs translation.

It is of course not necessary to start writing in the top left cell of the worksheet. Taking an offset into account in the definition of the fileref, we could position the output anywhere on the 'Records' worksheet. The following data step will do the actual writing:

```
data _null_;
  set price_list;
  file recrange;
  put band '09'x album '09'x
      date '09'x price;
run;
```

And lo, the contents of `price_list` are now in the 'Records' worksheet. Well, sort of … Some explanations and improvements are due.

For starters, what are those funny `'09'x` hex-values good for, separating the variables on the output-stream? As explained above, Excel delimits its cell values by tab characters. Therefore we need to output a tab character between each pair of SAS variables, so as to make the subsequent variables end up in subsequent worksheet cells. And 09 is the hex value for the tab character in the ANSI character set. Omitting these explicit tabs would cause an entire SAS observation to be written to a single cell. For a more in-depth description of all the intricacies involved in using tabs and the `notab` option, see Bodt (1996) and Schreier (1998).

Inspecting the 'Records' worksheet, it should come as no surprise to see that the variable `date` looks a bit meaningless. Indeed, Excel does not know what to do with the internal SAS date format which counts the days since January 1st 1960. To repair this, we apply a format on the SAS date variable *before* writing the data to Excel:

```
proc datasets library=work nolist;
  modify price_list;
    format date date9.;
quit;
```

Repeating the above `data _null_` step to re-write the `price_list` data to the 'Records' worksheet, it now appears that Excel has correctly interpreted the date values because SAS sends *formatted* values through the DDE connection, even if it is not explicitly asked to do so by using formats on the `put` statement. It is important to realize that this implicit use of formats associated with data set variables takes place. In the case of a

SAS date variable, it is fine. In other cases this behavior might not be desirable, and can lead to strangely formatted values in the worksheet. Especially since, on the data-receiving end, Excel will attempt to be clever and try to guess the nature of the data that come through the DDE pipeline, and format it according to its own intuition. One way to prevent this from happening is by pre-formatting the target cell-range in the receiving worksheet, which leads us to the next topic: how to format cells via DDE.

Well, almost: another noteworthy point is that the logical record length can be important. The default value for `lrecl` seems to be 255, meaning that if we would want to `put` a stream of variables through to Excel, with a total formatted length exceeding this number, truncation will occur. If this happens, then one needs to add the `lrecl` option to the triplet-style filename statement, and set it to a sufficiently large value. A bit of data step logic combined with querying the dictionary tables actually allows to take care of this in a very clean way.

6 Formatting of Excel Worksheet Cells

As an example of how to apply formatting to worksheet cells, we will now modify the font and the background color of the cells which we just filled with `price_list` data. While we are at it, we also adjust the column widths to 'best fit', so that all the data are visible at once:

```
data _null_;
  file sas2xl;
  put '[error(false)]';❶
  put '[workbook.activate("records")]';❷
  put '[select("r1c1:r4c4")]';❸
  put '[format.font("Verdana",24,false,
      false,false,false,4,false,false)]';❹
  put '[patterns(1,0,6)]';❺
  put '[column.width(0,"c1:c4",false,
       3)]';❻
run;
```

❶ We turn the Excel error-checking off to avoid even the remotest possibility that one of those pesky 'are you sure you want to do this or that' boxes pops up and stops our program in its tracks.

❷ While in the preceding section the receiving worksheet did not need to be selected in the active workbook for the data transfer to function properly — the triplet-style fileref took care of locating the target — we now need to appeal to the DDE system-doublet fileref, and act as if we are manually manipulating the workbook. Hence, the first thing to do is to activate the worksheet 'Records'. Interactively, this is done by clicking on the worksheet's name-tab near the bottom of the Excel workspace. The equivalent X4ML function is 'workbook.activate'.

❸ The next thing to do is to select the range of cells that needs formatting. The X4ML 'select' command takes its argument in a number of related forms. The one used here specifies a cell-range. For selecting a bunch of rows, a shorter form omitting the column references can be used, *e.g.* 'r3:r16', and vice versa for selecting entire columns. Disjointed cell-ranges can be specified by using a comma as separator, as in 'r1c1:r1c2,r4c1:r4c2' which will select the first two cells of row one *and* the first two of row four. This is equivalent to manually selecting cells while keeping the control-key pressed down.

❹ After having selected the cells, changing the font and font-attributes is done by means of the 'format.font' X4ML function. Its first two arguments are the font-name and the font-size in points. Then follow a number of true/false flags allowing combinations of bold, italic, underline and strikethrough formatting. The numerical parameter is a color code. A value of four seems to translate to some kind of green, but any value in the range 0–56 may be used. The zero value corresponds to automatic coloring, while the others are mapped to the 56 colors in Excel's font dialog box, although in a rather non-transparent way so it may take some experimenting to find the desired value. The last two parameters determine the outline and shadow attributes.

❺ To set the background color of our cells, we use the 'patterns' function. Its first argument determines the pattern and takes values in the 0–18 range. Zero means no pattern, one yields a solid pattern, and the remaining values correspond to the patterns that one can choose in the Excel format cells dialog box. The next two arguments determine foreground and background colors respectively. Their values follow the same rules as outlined in the 'format.font' discussion above. Six yields yellow, so the effect of this statement is to give the selected cells a solid yellow background color. Green on yellow, not exactly aesthetically pleasing …

❻ As a final example of automated output formatting via DDE, we adjust the column widths to 'best fit' so that all the inserted values are visible at a glance. The first argument of the 'column.width' function is the desired column width in units of one character of the font corresponding to the 'normal' cell style. We set it to zero, because it is ignored when a fourth argument is present. Next comes the selection of columns to which to apply the new width, followed by the 'standard width' true/false flag. We set it to false because we do not want to apply the standard column width as defined in Excel's column width dialog box. Finally, setting the fourth parameter to the value three will apply a best-fit width to the selection.

7 Insert an Excel Macro Sheet

Until now we have worked with an existing workbook wherein a worksheet ('Records') was already available for receiving our SAS data. How about *adding* a *new* worksheet to a workbook, preferably with a sheet-name of our own choice rather than the default 'Sheet<n>' naming convention employed by Excel? Can we do that with DDE? Sure, but we will be heading for some trouble …

First, the easy part. The following will add a new worksheet to the active workbook:

```
data _null_;
  file sas2xl;
  put '[workbook.next()]';❶
  put '[workbook.insert(1)]';❷
  put '[workbook.move("sheet1",
       "existing workbook.xls",1)]';❸
run;
```

❶ When using the Excel menu command Insert → Worksheet, the number of inserted sheets is equal to the number of sheets that are currently selected in the workbook. For example, if the current workbook has three sheets of which two are selected, two new ones will be inserted in front of the selected two. To avoid this from happening, we run the X4ML 'workbook.next' function which will relinquish the current selection and select the next available sheet in the workbook, thereby assuring that only one single sheet is selected.

❷ The 'workbook.insert' function will then insert a single new sheet in front of the one currently selected. Its parameter determines the nature of the sheet, similar to the 'new' function discussed in section 4. A value of one will make it a normal worksheet.

❸ Using the 'workbook.move' function, we then reposition the new sheet — which has received the default name 'Sheet1' — to be the first, or top-most sheet in the workbook.

So far, so good. The help file lists a function 'workbook.name' that is supposed to rename a sheet (this one is also nowhere in the printed manual, by the way):

```
data _null_;
  file sas2xl;
  put '[workbook.name("Sheet1","No Go")]';
run;
```

Upon attempting to use 'workbook.name' to rename 'Sheet1' as 'No Go', the following error appears in the SAS log:

```
ERROR: DDE session not ready.
FATAL: Unrecoverable I/O error detected in
       the execution of the data step
       program.
       Aborted during the EXECUTION phase.
```

We have come across an X4ML function that obviously doesn't work as advertised, at least not when we try to feed it to Excel through a DDE connection. The misbehavior of 'workbook.name' is not a unique nuisance either. Occasionally, while trying to perform fancy tricks via DDE, one will stumble onto something that simply doesn't work. Luckily there exists a workaround for this problem, the implementation of which for the 'workbook.name' case will occupy most of the remainder of this paper. The same technique applies however to other failing X4ML code.

What is this magic trick, then? Various sources have pointed out that X4ML functions like 'workbook.name' that fail when they are being passed to Excel straight through a doublet-style DDE connection, *will* actually work properly when used within an Excel macro. Thus, our strategy runs as follows: create a temporary X4ML macro sheet (*i.e.* an old-style, non-VBA macro sheet) in the current workbook. Use DDE to write an X4ML macro into this macro sheet that will perform the desired renaming operation. Use more DDE to run this Excel macro. Then delete the macro-sheet from the workbook.

The first step of this procedure takes exactly the same form as the code discussed at the start of the present section, except then that the 'workbook.insert' function is used with 3 as the argument, which leads to an X4ML macro sheet with the default name 'Macro1' being inserted. We also move our new macro sheet to the first position within the workbook. This will prove necessary for the correct working of the code in section 9:

```
data _null_;
  file sas2xl;
  put '[workbook.next()]';
  put '[workbook.insert(3)]';
  put '[workbook.move("macro1",
       "existing workbook.xls",1)]';
run;
```

By the way, note that the internal Excel names of sheets and workbooks are case-insensitive. Even if the sheet's name-tab claims that the macro sheet is named 'Macro1', we can still reference it as 'macro1'.

8 Rename a Worksheet

If we want to write to the cells on the 'Macro1' sheet, we need to define a triplet-style DDE fileref pointing to, say, the first 100 cells in the first column of the sheet:

```
filename xlmacro dde 'excel|macro1!r1c1:
  r100c1' notab lrecl=200;
```

The number 100 is arbitrary, it only needs to be chosen large enough so that sufficient cells are available to receive the X4ML macro which we want to store there. Note however that the processing of a DDE filename statement takes longer the larger the target cell-range is. It is therefore not a good idea to be too generous!

The following data step will then create the X4ML macro in the 'Macro1' sheet, and subsequently run it:

```
data _null_;
  file xlmacro;
  put '=workbook.name("sheet1",
       "Works Fine Now")';❶
  put '=halt(true)';❷
  put '!dde_flush';❸
  file sas2xl;
  put '[run("macro1!r1c1")]';❹
  put '[error(false)]';
run;
```

❶ We start writing to the triplet-style fileref `xlmacro`. The first line of X4ML code is precisely the 'workbook.name' function that we attempted to use earlier on. It will rename the sheet 'Sheet1' into 'Works Fine Now'.

❷ The 'halt(true)' command is required to stop the macro from running at this point, even if there is really nothing else in the following cells.

❸ At this point, we have created our little sheet renaming Excel macro, and we are ready to tell Excel to run it via the doublet-style `sas2xl` fileref. However, we need to insert the special '!dde_flush' command here because of an important difference between the way output to a triplet is handled by SAS, as opposed to output sent to a doublet. When writing to a triplet-style fileref, SAS actually *buffers* the output, and only sends it to the server application when the data step terminates. On the other hand, anything written to a doublet-style fileref is passed on to the server *immediately*. The '!dde_flush' command effectively flushes the DDE buffer and sends anything it finds in it through to the triplet-style fileref. In the above data step, failing to flush the buffer at this point would result in telling Excel to run our macro before it has even received it. Only after the flushing of the buffer will our two lines of X4ML code really have been inserted in the two upper-left cells of the 'Macro1' sheet.

❹ The X4ML function to run a macro is used to execute what we just wrote into the cells of the 'Macro1' sheet. Incidentally, the same 'run' function can be used to fire up a stored VBA macro.

For the set purpose of renaming our worksheet we are now done with the macro-sheet, so we could delete it at this point. For practical reasons however — we will need the 'Macro1' sheet in the next section — let's just keep it around for now. In case we should wish to delete it, the code to do so would be the following:

```
data _null_;
  file sas2xl;
```

```
* put '[error(false)]';
  put '[workbook.delete("Macro1")]';
run;
```

The line to turn the Excel error-checking off may not be necessary here, provided that the formatting data step in section 6 has been submitted. Once turned off, error-checking remains off during the remainder of the Excel session, until explicitly turned on again. In this case, failing to have it turned off would cause a warning box to pop up asking if it is ok to delete the sheet …

9 Get Existing Sheet Names

Being able to insert new sheets and renaming them at will is a nice thing of course, but unfortunately there are some pitfalls … Looking back at the code we have used till now, it is easy to see that things have worked out fine because we *knew* certain things about our ‘Existing Workbook.xls’ file. We knew the names of the sheets it contained, and therefore we knew that the new sheet we inserted would be named ‘Sheet1’. We also knew that the sheet ‘Works Fine Now’ did not exist yet in the workbook, so we could safely perform the renaming without causing a DDE error.

When developing DDE code to act upon *unknown* Excel workbooks, not knowing what sheet-names are already in use poses a problem. As an example, suppose a workbook ‘Fooled You.xls’ has two sheets filled with data: ‘Sheet1’ and ‘Sheet3’. Not knowing this, we want to add an extra sheet to it, ‘Stuff from SAS’. If we use the ‘workbook.insert’ function to create a new sheet, Excel will make an empty ‘Sheet2’, because that’s the first available name in its internal sheet-naming logic. Without knowing that there already are a ‘Sheet1’ and a ‘Sheet3’ on the workbook, we can impossibly perform the renaming operation because we don’t know exactly which sheet to rename. Not to mention the possibility that a ‘Stuff from SAS’ sheet might already exist on the current workbook.

All this to show that a basic premise for handling the insertion and renaming of sheets correctly, is to have a means of getting the existing sheet-names out of a random workbook. The solution to this problem is not for the faint of heart. But it is a nice example of the kind of tricks that can be pulled off with the help of a temporary macro sheet and some trial-and-error Excel macro-programming.

First we empty the ‘Macro1’ sheet:

```
data _null_;
  file sas2xl;
  put '[workbook.activate("macro1")]';
  put '[select("r1c1:r100c1")]';
  put '[clear(1)]';
  put '[select("r1c1")]';
run;
```

Setting the ‘clear’ function’s argument to ‘one’ is equivalent to selecting ‘Clear All’ from the Excel menu.

Then we define a SAS macro `%loadnames` that will create a data set `work._sheet_names` with all the sheet-names available on the current workbook:

```
%macro loadnames;
  %local sh wn nsheets;❶
  %let sh=0;
  %let wn=0;
  %let nsheets=0;
  data _null_;❷
    file xlmacro;
    put '=set.value($b$1,
          get.workbook(4))';❸
    put '=halt(true)';
    put '!dde_flush';
    file sas2xl;
    put '[run("macro1!r1c1")]';
    put '[error(false)]';
  run;
  filename nsheets dde "excel|macro1!r1c2:
    r1c2" notab lrecl=200;❹
  data _null_;❺
    length nsheets 8;
    infile nsheets;
    input nsheets;
    call symput('nsheets',
      trim(left(put(nsheets,3.))));
  run;
  %let nsheets=%eval(&nsheets-1);❻
  data _null_;❼
    file sas2xl;
    put '[workbook.activate("macro1")]';
    put '[select("r1c1:r100c1,
          r1c2:r1c2")]';
    put '[clear(1)]';
    put '[select("r1c1")]';
  run;
  data _null_;❽
    length maccmd $200;
    file xlmacro;
    %do sh=1 %to &nsheets;
      maccmd="=select(!$b$&sh,!$b$&sh)";
      put maccmd;
      put '=set.name("cell",selection())';
      %do wn=1 %to &sh;
        put '=workbook.next()';
        %end;
      put '=set.value(cell,
            get.workbook(3))';
      put '=workbook.activate("Macro1",
            false)';
      %end;
    put '=halt(true)';
    put '!dde_flush';
    file sas2xl;
    put '[run("macro1!r1c1")]';
    put '[error(false)]';
  run;
  filename sheets dde "excel|macro1!
    r1c2:r&nsheets.c2" lrecl=200;
  data _sheet_names;❾
    length bookname sheetname $100;
    infile sheets delimiter=']';
    input bookname sheetname;
    bookname=substr(bookname,2);
    bookname=left(reverse(substr(left(
      reverse(bookname)),5)));
  run;
  filename nsheets clear;
  filename sheets clear;
%mend loadnames;
```

Without going into too much detail, this is how it works:

❶ Some local macro variable declarations: `sh` and `wn` are simply counters for the nested do-loops in step 8, while `nsheets` will be set to the number of worksheets present in the current workbook. Perhaps unnecessary in the present case, but generally good SAS macro design practice, all macro variables are given an initial value.

❷ The first data step will write a short Excel macro to the *first* column of the 'Macro1' sheet (using the `xlmacro` triplet-style fileref), and subsequently run it (using the `sas2xl` doublet-style fileref). The net effect of running this Excel macro will be that the number of sheets in the current workbook is inserted in the top cell of the *second* column ('b1') of the 'Macro1' sheet.

❸ This is done by means of the X4ML function 'set.value'. It will set the value of the cell referenced by its first argument, to the value specified by its second argument. In the present case, 'get.workbook(4)' returns the number of worksheets in the active workbook.

❹ After having run this Excel macro, the number of worksheets will be in the r1c2-cell of the 'Macro1' worksheet, and we define a triplet-style fileref `nsheets` pointing exactly to this cell.

❺ Then, we briefly use DDE to *read* from Excel rather than write to it. We pick up the number of sheets and store it in the SAS macro variable `nsheets`. Note that the `3.` format is wide enough, at least with the Office97 release of Excel, where the number of worksheets in a workbook is limited to 255.

❻ Also note that when Excel coughed up the number of worksheets, it included our temporary 'Macro1' sheet in the count. What we will need however is the real number of sheets contained in the workbook before we started meddling with it. So we subtract one.

❼ We then clear the 'Macro1' sheet once more.

❽ Only to write another Excel macro into it: the following data step will, with the help of two nested SAS macro do-loops, write an Excel macro to the *first* column of the 'Macro1' sheet that, when executed, will make Excel enter the names of all the worksheets into the cells of the *second* column of the 'Macro1' sheet. The idea is this: select the next free cell where to insert a sheetname ('select' function). Define a reference name 'cell' to point to this cell ('set.name' function). Advance the worksheet selection as many times as necessary to get to the next worksheet of which the name is sought ('workbook.next'). Get the sheet-name ('get.workbook(3)') and stuff it into the cell referenced by the name 'cell' ('set.value'). Then return to the 'Macro1' sheet ('workbook.activate') and start again …

❾ As before, we can now define a triplet-style fileref pointing to the information that Excel has just entered in the second column, and read it into a data set `_sheet_names`.

A few more noteworthy things, perhaps. The Excel macro in step 8 was actually cobbled together pretty much by trial and error. It is not very efficient when being applied to workbooks having hundreds of sheets. A quick calculation shows that if presented with the maximum number of worksheets in an Excel97 workbook (255), step 8 would generate an Excel macro of over 33,000 statements long. Others who are better versed in X4ML than the author may well know of a more elegant and efficient way to obtain the same result, perhaps involving an iterative loop in the Excel macro, cycling through the worksheets.

Remember that the `xlmacro` fileref was defined as pointing to the first 100 cells in the 'Macro1' sheet (section 8). In the presence of many worksheets, this number needs to be taken a bit more liberally to accommodate all the code that step 8 generates.

The 'select' function in step 8 has a SAS macro variable in its arguments. When attempting to use `put` statements combined with SAS macro variables to write to DDE filerefs, one quickly finds out that SAS macro variables are *not* being resolved before entering the DDE buffer. The easiest solution is to use a dummy data step text variable assignment (`maccmd`) to force SAS macro variable resolution, and then simply `put` the dummy variable instead.

10 Is This Really Necessary?

Incidentally, the Institute's Technical Support Document on DDE (SAS Institute, 1999) hints at some intriguing possibilities that might provide a simpler solution than the one presented in the above. At least with respect to the problem of getting to know what worksheets are present in an Excel workbook.

Consider the following code, which uses a hybrid type of DDE doublet-style fileref:

```
filename topics dde 'excel|system!topics';
data _null_;
  length topics $1000;
  infile topics pad dsd notab dlm='09'x;
  input topics $ @@;
  put topics;
run;
```

The SAS log shows this:

```
[:]:
[existing workbook.xls]Bubble Graph
[existing workbook.xls]Early Morning
                       Reminder
[existing workbook.xls]Macro1
[existing workbook.xls]Random Numbers
[existing workbook.xls]Records
[existing workbook.xls]Works Fine Now
[PDFWriter.xla]Sheet1
Syste
```

Obviously, these returned values could be captured in a data set, and knowing the workbook name, one could proceed to parse the proper strings and extract the sheet-names!

However, as this feature is largely undocumented, there is a certain risk attached to using it for production quality work. For one, the final character is invariably missing, which in the above example results in 'Syste' being returned, rather than the correct 'System'. It may not be guaranteed that none of the actual sheet-names could end up in the last position of the returned list, and have its name truncated.

Another reason why the more elaborate method presented in the previous section might be preferred, is that there we get the sheet-names in order of appearance on the workbook. This extra bit of knowledge may come in handy for positioning new sheets with some data step logic and the 'workbook.move' function. In the above, the sheets merely appear sorted by name.

11 Bonus: the %SASTOXL Macro

Two of the nice features of DDE are that it can be used from within the data step environment, and that it lends itself extremely well to automation by means of the SAS macro language. An easy-to-use SAS macro `%sastoxl` is available from the author's web-site www.vyverman.com The macro harnesses the power of DDE in a one-liner that can be included in any base SAS program.

`%sastoxl` incorporates all of the features discussed in the present paper, including transparent handling of sheet-names, the possibility of formatting the spreadsheet cells, working with an existing workbook rather than creating a new one, and much more … The macro is fully documented and includes usage samples. A paper describing its functionality and — to some degree — its inner workings, is equally available (Vyverman, 2000).

It would be far beyond the scope of the present paper to discuss `%sastoxl` in any depth, but two brief examples might whet the appetite. First we create some dummy numerical data:

```
data _sinuous(drop=counter angle);
  length x y z counter angle 8;
  do counter=1 to 1000;
    angle=counter/80;
    x=sin(angle);
    y=cos(angle*x);
    z=x*y-x+y;
    output;
    end;
run;
```

We make a first call to `%sastoxl` as follows:

```
%sastoxl(
          libin=work,
          dsin=_sinuous,
          sheet=Sinuous Numbers from SAS,
          savepath=c:\koen\sugi 26 paper,
          savename=Made by the SAS System,
          stdfmtng=1
         );
```

What happens is this: `%sastoxl` takes the data set `work._sinuous` as specified by the `libin` and `dsin` parameters, and exports it to a new workbook 'Made by the SAS System.xls' (`savename`) which it saves at the location dictated by `savepath`. The new workbook has one single worksheet named 'Sinuous Numbers from SAS' (`sheet`). The standard formatting flag (`stdfmtng`) provides the icing on the cake, and will cause the spreadsheet cells to be formatted in 10-point Courier, with column widths adjusted to 'best fit', and the freeze-panes option turned on so that the first row containing variable labels/names remains visible while scrolling.

By means of a second call to `%sastoxl`, we will insert the same data into our 'Existing Workbook.xls' file, and save the result as 'Updated Workbook.xls':

```
%sastoxl(
          libin=work,
          dsin=_sinuous,
          tmplpath=c:\koen\sugi 26 paper,
          tmplname=existing workbook,
          sheet=random numbers,
          savepath=c:\koen\sugi 26 paper,
          savename=Updated Workbook,
          stdfmtng=1
         );
```

Using the parameters `tmplpath` and `tmplname` (short for 'template') will cause `%sastoxl` to open the specified workbook rather than create a new one. Remember that 'Existing Workbook.xls' already has a sheet named 'Random Numbers'. Hence, `%sastoxl` will over-write the contents thereof with the data from the `_sinuous` data set. The workbook also contains a graphic sheet 'Bubble Graph', visualizing the data on the 'Random Numbers' sheet. As a consequence, when the 'Updated Workbook.xls' file is opened, the chart will automatically display the newly entered data!

The above actually constitutes an example of a very powerful way to mass-generate Excel files loaded with graphics and pivot tables straight from within a SAS program. It suffices *e.g.* to create a 'template' workbook with some worksheets containing dummy figures, and some graphic sheets pointing to those dummy figures. This template is kept safely on a production pc. Using `%sastoxl` one can then open this template workbook, dump real data in all the right places, and save under a custom filename on a network drive. Users consulting the resulting file then see the graphs representing the real data. It is also possible to use the macro to refresh the contents of scores of Excel workbooks during an overnight batch job.

Prospective users of `%sastoxl` may need to tweak the macro source code to varying degrees. To name but a few potential issues: `%sastoxl` uses the first technique for starting up Excel (discussed in section 3), and so the full directory path pointing to the Excel executable must be specified, and is likely to differ from the author's settings. Some of the formatting conventions used in the code, *e.g.* the pre-formatting of cells destined to receive SAS character variables, may or may not be suitable. The macro was written to work with the English language version of Excel. Other language versions of the Excel application will *e.g.* not use 'SheetN' as the default name for a new worksheet, and the macro code will need to be made conform to whatever is being used. Summarizing: your mileage may vary!

12 Conclusion

We have shown how to do a number of basic things to customize DDE output to Excel. We have encountered some pitfalls. We have discovered workarounds. So, is this it? By no means! We have just been scratching the surface: the printed X4ML function reference (Microsoft, 1992) is a 500-page book, detailing hundreds of functions that are waiting to be used by our SAS programs to pull off fancy Excel tricks in a fully controlled and automated manner. When developing DDE code, new functionality suggests itself regularly: the sky is the limit!

References

Bodt, M. "Talking to PC Applications Using Dynamic Data Exchange". *Observations*™, vol. 5, no. 3, pp. 18–27, 1996.

Kuligowski, A. T. "Advanced Methods to Introduce External Data into the SAS System". *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*, paper 53, 1999.

Microsoft Corporation (no author specified) "Microsoft Excel Function Reference". *Document Number AB26298-0592*, 1992.

Mumma, M. T. "The Redmond to Cary Express – A Comparison of Methods to Automate Data Transfer Between SAS and Microsoft Excel". *Proceedings of the Twelfth Annual North-east SAS Users Group Conference,* pp. 654–661, 1999.

Roper, C. A. "Intelligently Launching Microsoft Excel from SAS, using SCL functions ported to Base SAS". *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference*, paper 97, 2000a.

Roper, C. A. “Using SAS and DDE to execute VBA macros in Microsoft Excel”. *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference*, paper 98, 2000b.

SAS Institute, “Technical Support Document #325 – The SAS System and DDE”. *http://ftp.sas.com/techsup/download/technote/ts325.pdf*, updated 1999.

Schreier, H. “Getting Started with Dynamic Data Exchange”. *Proceedings of the Sixth Annual Southeastern SAS Users Group Conference*, pp. 207–215, 1998.

Stetz, M. “Using SAS Software and Visual Basic for Applications to Automate Tasks in Microsoft Word: an Alternative to Dynamic Data Exchange”. *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference*, paper 21, 2000.

Vyverman, K. “Using Dynamic Data Exchange to Pour SAS Data into Microsoft Excel”. *Proceedings of the 18th SAS European Users Group International Conference*, 2000.

Acknowledgments

Thanks are due to all those who, over the years, have shared their knowledge on the topic of DDE, either in private communications, by means of published material, or via that most valuable of on-line forums: SAS-L.

Trademarks

Contacting the Author

The author welcomes and encourages any questions, corrections, feedback, remarks, both on- and off-topic via e-mail at:

```
sugi27papers@vyverman.com
```

Alternatively, snail-mail may be directed to:

```
Koen Vyverman
c/o Alphanor Data Mining Inc.
Oerenstrasse 9
D-54290 Trier
Germany
```