

Creating Custom Excel Workbooks from Base SAS® with Dynamic Data Exchange: A Complete Walkthrough

Koen Vyverman, Alphanor Data Mining Inc.

Abstract

If you find yourself besieged by hordes of business users and analysts, all clamoring for SAS output in nicely formatted Excel workbooks, then you'll need to go beyond what proc export offers ... Dynamic Data Exchange (DDE) will do the job for you. DDE is a communication protocol that lets pc-based applications talk to each other. In particular, DDE allows a SAS session to take control of the Excel application, and perform manipulation of cells, spreadsheets and entire workbooks just as if you were doing so manually. The purpose of this workshop is to provide a complete walkthrough for building good-looking Excel workbooks from scratch, all from within Base SAS code. As you run the provided sample code and observe what it accomplishes, you'll learn how to create, rename and move spreadsheets, fill them with SAS data, apply Excel formatting to selected cells, change fonts and colors, center titles, enter Excel formulas, specify header and footer, and much more. More than enough in fact to keep those hordes at bay!

0 Introduction

The present paper is intended as a companion to one previously published—Vyverman (2001)—wherein the basics of using Dynamic Data Exchange (DDE) for outputting SAS data to Excel are discussed at some length. Since there is little point in duplicating the same information, the idea is to maximize the code content here, and to point to the previous paper where applicable. Acquaintance with Vyverman (2001) is not specifically required for the digestion of what follows.

Between the two papers, a certain amount of overlap is inevitable, in terms of functionality offered and techniques used. However, whereas Vyverman (2001) primarily focuses on the use of DDE in conjunction with existing ‘template’ Excel workbooks, we shall do no such thing here and create all output from scratch instead.

The topics covered in this bag of DDE tricks have been selected both from private exchanges with other SAS-DDE-Excel users, and from questions asked / answers provided via the SAS-L online forum between April and December of 2001. As such, most of the following sections may rightly be said to deal with frequently asked questions...

The remainder of this introductory section is concerned with setting the scene and lining up the necessary props. First of all, since DDE requires the Excel application to be running, we launch Excel in the manner suggested by Roper (2000):

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

Note that after ❶ telling the OS to kick Excel awake, ❷ the data step keeps trying for at most ten seconds to set up a working DDE system-doublet fileref sas2xl through which we will be sending all sorts of Excel version 4 Macro Language (X4ML) functions to remotely control the Excel application from within our SAS programs.

We proceed to generate some SAS data sets for later use. The first of these, mixed_data, contains one alphanumeric variable filled with a string of five random characters in the {A,...,Z} range, and three random numerical variables, one of which can in fact be interpreted as a SAS date variable:

```
data mixed_data(drop=i j);
  length
    product $ 5
    x y date 8;
  do i=1 to 100;
    product='';
    do j=1 to 5;
      product=trim(left(product)||
        byte(int(65+26*ranuni(0))));
    end;
    x=5*ranuni(0);
    y=10*rannor(0);
    date=today()+int(y);
  output;
end;
run;
```

Then brackets_data, consisting of a single ten-character string seemingly populated with positive integers, some of which are enclosed within parentheses:

```
data brackets_data;
  length string $ 10;
  input string;
  cards;
54290
(1180)
68386
(235)
(9480)
;
run;
```

A last data set, `leading_zeroes_data`, has one numerical variable formatted to show leading zeroes.

```
data leading_zeroes_data;
  length zip 8;
  format zip z5.;
  input zip;
  cards;
54290
1180
68386
235
9480
;
run;
```

Finalizing the set-up, we define a global SAS macro variable `tab`, which, unsurprisingly, will resolve to a tab-character:

```
%let tab='09'x;
```

1 Basic Manipulations

As a preliminary remark, it should be pointed out that most of the X4ML functions used in this paper have many more arguments than are shown here. To find out about those other useful function modifiers and their range of values, it is highly recommendable to download the ‘macrofun.exe’ X4ML help file installer from the Microsoft web-site, or to locate a copy of the out-of-print manual (Microsoft, 1992). For more details on these documents, see section two of Vyverman (2001).

Now then, when the Excel application starts up, it presents a default workbook containing a number of empty worksheets as specified in the local application preferences. Since on most machines this number of empty worksheets is set to some ridiculously high value, we commence by ditching the default workbook and roll our own instead:

```
data _null_;
  file sas2xl;
  put '[ file.close(false)] '; ❶
  put '[ new(1)] '; ❷
  put '[ error(false)] '; ❸
  put '[ save.as("c:\temp\1 Basic
  Manipulations")] '; ❹
run;
```

❶ Talking to Excel via the DDE-doublet `sas2xl`, we use the X4ML function ‘file.close’ to dismiss the default workbook. The boolean parameter indicates that the file should not be saved prior to closing it.

❷ Instead, we use the ‘new’ function which pops up a new workbook with a single sheet of the type specified by the numerical argument. In this case, a value of one results in a worksheet which is by default named ‘Sheet1’.

❸ An important thing to do early on in the DDE-session, is to toggle off the Excel error-checking feature. When using DDE to automate the creation of Excel files, the last thing we want is to have Excel dialog boxes popping up all over the place, asking *e.g.* whether we are absolutely sure to over-write an existing file with the one we are saving? Sending an ‘error(false)’ will get rid of all these automation show-stoppers, but obviously assumes that we know what we are doing...

❹ Using the ‘save.as’ function we save our workbook somewhere, thereby fixating the filename so that we can use it in

code hereafter. Not specifying a file-extension in the double-quoted pathname / filename string will result in the default Excel file-type ‘.xls’. Mixed case can be used in the naming, but remember that the Windows file-system is only case-sensitive in appearance. Meaning that it is not: if the Explorer shows a file named ‘Bogus.xls’ and Excel writes a ‘BOGUS.XLS’ to the same directory, the file ‘Bogus.xls’ will be effectively replaced, even if it will still show as ‘Bogus.xls’ in the Explorer.

We then define a triplet-style DDE fileref `rechange`, pointing to a block of cells 4 columns wide and 100 rows tall on the ‘Sheet1’ worksheet.

```
filename rechange dde
  'excel|[1 basic manipulations.xls]
  sheet1!r4c1:r103c4' notab;
```

Note the use of the ‘notab’ option, more about which may be read in Vyverman (2001). The syntax here requires specification of the full file-name, including the ‘.xls’ extension. No need to bother with uppercase / lowercase subtleties however.

Now we can write the contents of the SAS data set `mixed_data` to the block of worksheet cells referenced by `rechange`. After it has served its purpose, we delete the fileref.

```
data _null_;
  set mixed_data;
  file rechange;
  put
    product &tab
    x &tab
    y &tab
    date;
run;
filename rechange clear;
```

The SAS macro variable `tab` separates the data set variables on the put statement, ensuring that the values end up in different columns on the target worksheet. In similar vein, we write some column headers to the third row of ‘Sheet1’:

```
filename rechange dde
  'excel|[1 basic manipulations.xls]
  sheet1!r3c1:r3c4' notab;

data _null_;
  file rechange;
  put
    'Product' &tab
    'X' &tab
    'Y' &tab
    'Date';
run;

filename rechange clear;
```

Inspecting the worksheet ‘Sheet1’ at this point, we notice that at first sight all seems to be as expected, including the fact that the SAS date variable looks rather unusable in an Excel spreadsheet.

This minor adversity is easy to overcome though, since DDE sends formatted data to Excel whenever SAS formats are defined. We use this feature to get the SAS date right: first we add a SAS date-format to the `mixed_data` data set:

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

And then we re-run the above code to export the data set. The worksheet-cells on ‘Sheet1’ have now been over-written, and lo, the ‘Date’ column displays readable dates! It is important to realize though that two distinct things have happened here. First, SAS applied the ‘date9.’ SAS format to the `date` variable. Meaning that a value like 15354 *e.g.* gets piped through to Excel as a formatted string ‘14JAN2002’.

Secondly, Excel sees this string coming in and says “Hey, looks like a date. I’ll be clever and format the target cell with one of my own date-formats!” As a result, what we see in the worksheet cell is an internal Excel value of ‘14-01-2002’, and a formatted Excel value reading ‘14-Jan-2002’. Not exactly what we were aiming for, but in this case acceptable. In section four, we will see what to do in those cases where Excel getting clever leads to undesirable results.

Before we move on, there is another potential pitfall here. Looking at the ‘Product’ values on ‘Sheet1’, notice how an extra space got tacked onto the end of each five-character string. This is not a DDE-fluke nor Excel meddling, but a simple consequence of using the SAS `put` statement for list-style output. We can get rid of these spaces by using pointer control on the `put` statement as follows:

```
put
  product +(-1) &tab
  x &tab
  y &tab
  date;
```

These extra spaces inserted into the output stream by the `put` statement may seem harmless, but it is good practice to avoid them altogether. Users might *e.g.* wish to further process the ‘Sheet1’ data with, say, an Excel ‘vlookup’ function to translate the product codes into meaningful names. Any unexpected character would be a show-stopper there...

Now on to some file-saving examples. First, we simply save the current workbook.

```
data _null_;
  file sas2xl;
  put '[ save] ';
run;
```

The ‘save.as’ X4ML function which we used earlier takes more parameters than just the path / filename string. Adding a second parameter, we can stipulate the file-type. As an example, the following creates a CSV file ‘1 Basic Manipulations.csv’ rather than the default Excel workbook format:

```
data _null_;
  file sas2xl;
  put '[ save.as("c:\temp\1 Basic
    Manipulations",6)] ';
  put '[ file.close(false)] ';
run;
```

To open an existing Excel workbook we use the ‘open’ function in its basic form. Not specifying the file extension means that it is the ‘.xls’ one that gets loaded:

```
data _null_;
  file sas2xl;
  put '[ open("c:\temp\1 Basic
    Manipulations")] ';
run;
```

To end this section, let us juggle some worksheets around:

```
data _null_;
  file sas2xl;
  put '[ workbook.insert(1)] ';❶
  put '[ workbook.move("sheet2", "1 basic
    manipulations.xls",3)] ';❷
  put '[ workbook.activate("sheet1")] ';❸
  put '[ workbook.insert(1)] ';
  put '[ workbook.delete("sheet2")] ';❹
run;
```

❶ We add a new worksheet to the open workbook. It receives the default name ‘Sheet2’ and appears to the left of ‘Sheet1’.

❷ We then move the new ‘Sheet2’ to the right of ‘Sheet1’. That is, counting from left to right, we want to get it in position #3. Hence the value of the third parameter.

❸ Making ‘Sheet1’ active, we insert yet another new worksheet, which yields a ‘Sheet3’ in the first position.

❹ To practise deleting worksheets, we kill ‘Sheet2’.

2 Worksheet and Workbook Formatting

Time to save our document with a new name:

```
data _null_;
  file sas2xl;
  put '[ save.as("c:\temp\2 Worksheet -
    Workbook Formatting")] ';
run;
```

In order to format the data which were previously entered in ‘Sheet1’, we make sure said worksheet is the active one, and select the entire block of cells before applying specific font formatting. We choose a different font for the header cells:

```
data _null_;
  file sas2xl;
  put '[ workbook.activate("sheet1")] ';
  put '[ select("r4c1:r104c4")] ';
  put '[ format.font("Courier New",18,
    false,false,false,false,3,false,
    false)] ';
  put '[ select("r3c1:r3c4")] ';
  put '[ format.font("Verdana",24,false,
    false,false,0,false,
    false)] ';
run;
```

Since font formatting is such a common task, the entire list of parameters is exceptionally reproduced in the code above. In order of appearance, there’s the font name within double quotes, followed by a menagerie of numerical and boolean values: font size in points, bold, italic, underline, strike-through, color (a number between 0 and 16), outline, shadow.

The background color of a selection of cells can be set by means of the ‘patterns’ function. The following gives our table a solid 25% grey background:

```
data _null_;
  file sas2xl;
  put '[ select("r3c1:r104c4")] ';
  put '[ patterns(1,0,15)] ';
run;
```

It is considered good form to produce tables such that the recipient can actually see the contents of the cells without having to fiddle around with column width settings. So we adjust the column widths to a best fit:

```
data _null_;
  file sas2xl;
  put '[ column.width(0,"c1:c4",false,3)] ';
run;
```

The important parameter here is the fourth one, which yields auto-fit when set to a value of three. Another useful application of the ‘column.width’ function is for hiding columns. To this effect, use a value of one for the fourth parameter.

Formatting can be applied to disjoint cell-ranges. We will use bold type for the first and fourth columns of the data in our table.

```
data _null_;
  file sas2xl;
  put '[ select("r4c1:r104c1,
              r4c4:r104c4")] ';❶
  put '[ format.font(,,true)] ';❷
run;
```

❶ To select disjoint ranges, simply list them separated by commas within the selection string of the ‘select’ function.

❷ Observe the syntax used for modifying only the third parameter of a ‘format.font’ function. Formatting operations are generally cumulative. Parameters not explicitly listed will keep the values they already had.

For further embellishment, we add some borders:

```
data _null_;
  file sas2xl;
  put '[ select("r3c1:r3c4")] ';
  put '[ border(,,,6)] ';❶
  put '[ select("r4c1:r104c3")] ';
  put '[ border(,,3)] ';❷
run;
```

❶ The fifth parameter of the ‘border’ function sets the bottom cell borders. A value of six produces a double line.

❷ The third parameter sets right cell borders. A value of three yields dashed lines.

Since we have some space left in the first two rows of ‘Sheet1’, we put a title and a subtitle there, apply a bit of formatting, and make sure they are nicely centered across the table by calling the ‘alignment’ function with a first parameter set to seven:

```
filename rechange dde
  'excel|[ 2 worksheet - workbook
  formatting.xls] sheet1!r1c1:r2c1' notab;

data _null_;
  file rechange;
  put 'Some Data from SAS';
  put 'I Kid You Not!';
run;

data _null_;
  file sas2xl;
  put '[ select("r1c1")] ';
  put '[ format.font("Verdana",24)] ';
  put '[ select("r2c1")] ';
  put '[ format.font("Verdana",24)] ';
  put '[ select("r1c1:r1c4")] ';
  put '[ alignment(7)] ';
  put '[ select("r2c1:r2c4")] ';
  put '[ alignment(7)] ';
run;
```

While scrolling through a table, it is always a hassle when those column header labels vanish off the top of the screen. Avoid “Now, what was in this column again?” user frustration and freeze those panes for easy scrolling. The third parameter of the ‘freeze.panes’ function is the number of frozen rows:

```
data _null_;
  file sas2xl;
  put '[ freeze.panes(true,0,3)] ';
run;
```

An extremely useful feature of Excel is the autofilter. Too bad it’s somewhat hidden, and too bad a lot of business users are oblivious of it. We make things easy for them by putting an autofilter on row three for easy sub-setting of the data:

```
data _null_;
  file sas2xl;
  put '[ select("r3")] ';
  put '[ filter] ';
run;
```

Time for a little teaser: how to create an Excel chart with virtually no effort. More about customizing Excel charts in some future paper, perhaps. The following inserts a graphics sheet (‘workbook.insert’ with a value of two) into the present workbook, charting two series labelled ‘X’ and ‘Y’ as based on the numerical columns of our table:

```
data _null_;
  file sas2xl;
  put '[ select("r3c2:r104c3")] ';
  put '[ workbook.insert(2)] ';
run;
```

With the graphics sheet—default name ‘Chart1’—selected, we use the ‘page.setup’ function to display the title and subtitle from ‘Sheet1’ in the header. Oh, on two lines, centered, and with custom font formatting please. And while we are at it, we might as well put our signature in the footer:

```
data _null_;
  file sas2xl;
  put '[ page.setup("&B&I&18&""Times""Some
  Data from SAS' '0d'x '&14I Kid You
  Not!") ] ';❶
  put '[ page.setup(,"&B&I&14' 'A9'x
  ' Mighty Mouse Productions")] ';❷
run;
```

❶ The first parameter of ‘page.setup’ is a string containing the header text. It can contain special text modifiers to change font formatting: &B applies bold formatting to anything following it; &I turns italic formatting on; &18 uses 18 point font size; “Times” specifies the font. A minor complication is that since the header string needs to be enclosed in a pair of double quotes, we need to use two pairs of double quotes around the font name as per Excel string handling syntax. We get the title and subtitle one above the other by embedding a carriage return character—hex value 0D—in between them.

❷ Similarly, we format the footer string in bold italic 14 point, while retaining the default footer font. Special characters like the copyright symbol—hex value A9—can be thrown in too.

We have not yet used the ‘Sheet3’ worksheet. A good place to test some Excel formulas... In the first column of ‘Sheet3’, we enter an Excel formula showing the average of the X and Y columns from ‘Sheet1’:

```

data _null_;
  file sas2xl;
  put '[ workbook.activate("sheet3")] ';
  put '[ formula.fill("XY Average",
    "r1c1")] '; ❶
  put '[ formula.fill("=average(sheet1!
    r[ +2] c2:r[ +2] c3)",
    "r2c1:r101c1")] '; ❷
run;

```

❶ The ‘formula.fill’ function may be used to simply enter text or a value into a worksheet cell. No need to define a DDE triplet and write to it with a separate data _null_ step: we specify the desired cell-content as the first parameter of ‘formula.fill’, and the target cell as the second parameter.

❷ However, the true power of ‘formula.fill’ lies in the fact that it mimics the interactive behaviour of entering a function containing relative cell references into a worksheet cell, and using the autofill feature to copy the functional expression across an entire range of cells while maintaining the correct relative cell references. Here we use the average function pointing to values on another sheet with an offset of two rows downward, and in one fell swoop enter a hundred copies of it into the specified cell-range.

Excel comes with a truck-load of predefined formats. Suppose we wish to apply the one known as ‘Fractions - As sixteenths (8/16)’ to the result of our average function:

```

data _null_;
  file sas2xl;
  put '[ select("r2c1:r101c1")] ';
  put '[ format.number("# ??/16")] ';
  put '[ select("r1c1")] ';
run;

```

The ‘format.number’ function requires an Excel format definition string as its first parameter. For those who, like the author, have little interest in learning to write their own Excel format definitions, the easy way of finding the required syntax is this: select a cell, go to the ‘Format’ menu, choose ‘Cells’. Pick the desired format. Do not click the ‘OK’ button, but click on ‘Custom’ in the list on the left. The field labelled ‘Type’ will now reveal the internal code of the named format that was selected.

Roger and out...

```

data _null_;
  file sas2xl;
  put '[ save] ';
  put '[ file.close(false)] ';
run;

```

3 FAQ: Stop Messing Up My Numbers!

As hinted at in section one, there are cases in which we want to avoid Excel getting clever on the data-receiving end, and applying all sorts of unasked for formatting. How to accomplish this is in fact a very frequently asked question.

First we make a new workbook and save it:

```

data _null_;
  file sas2xl;
  put '[ new(1)] ';
  put '[ save.as("c:\temp\3 Stop Messing
    With My Numbers")] ';
run;

```

The SAS data set leading_zeroes_data contains a numerical variable zip with the z5. format to show leading zeroes. Imagine some fake zip-codes. We send it to the first column of ‘Sheet1’:

```

filename rechange dde
  'excel|[ 3 Stop Messing With My
    Numbers.xls] sheet1!r1c1:r5c1' notab;

```

```

data _null_;
  set leading_zeroes_data;
  file rechange;
  put zip;
run;

filename rechange clear;

```

The SAS data set brackets_data contains a character variable string, the values of which happen to be integer numbers, some of them enclosed in parentheses. We send this data set to the second column of ‘Sheet1’:

```

filename rechange dde
  'excel|[ 3 Stop Messing With My
    Numbers.xls] sheet1!r1c2:r5c2' notab;

data _null_;
  set brackets_data;
  file rechange;
  put string;
run;

filename rechange clear;

```

Inspecting the worksheet, we notice that Excel has taken the liberty of zapping the leading zeroes from the data in column one. Moreover, it has also interpreted the bracketed strings of digits that we sent to the second column as negative numbers. While, at times, this may be a good thing, suppose we don’t want it to happen?

To avoid Excel cleverness and really get both leading zeroes and bracketed figures in the spreadsheet cells, we apply the generic Excel text format to the target cells prior to sending data from SAS. It is the odd one shown as an ‘@’ symbol in the ‘Format Cells’ Excel dialog box, in the ‘Custom’ category. We pre-format some cells in the 3rd and 4th column of ‘Sheet1’ in this manner:

```

data _null_;
  file sas2xl;
  put '[ select("r1c3:r5c4")] ';
  put '[ format.number("@")] ';
  put '[ select("r1c1")] ';
run;

```

We then send both data sets once more, but to the now pre-formatted cells in the 3rd and 4th columns. That is, with the same code as above, but using DDE triplets pointing to cell-ranges r1c3:r5c3 and r1c4:r5c4 respectively.

And all is now well... As a closing remark, it might be argued that in those cases where similar Excel formats exist, it is better to avoid using SAS formats when piping data through to Excel. Except for SAS date-time variables, pre-formatting the target cells with the Excel ‘@’ format, and applying some other more appropriate Excel format after insertion of the data, will leave experienced Excel users with a more usable workbook than one filled with formatted strings which they’ll need to parse out before doing some further calculations!

4 Renaming Worksheets

Slowly, we progress towards some more advanced topics. Whereas in the previous sections, getting Excel to do something in particular usually boiled down to finding the right X4ML function, some things simply don't work that way. In this and the following two sections, we will need to create temporary Excel macros to get the job done.

Let us open the Excel workbook as made in section two, and save it with a new name:

```
data _null_;
  file sas2xl;
  put '[ open("c:\temp\2 worksheet -
        workbook formatting")]';
  put '[ save.as("c:\temp\4 Renaming
        Worksheets")]';
run;
```

It would be nice if we could rename the worksheet in which we have been creating a nicely formatted table to something less bland and uninformative than 'Sheet1'. Browsing the X4ML function reference, we note the existence of a function promisingly named 'workbook.name'. We attempt to use it to rename 'Sheet1' to 'Data from SAS':

```
data _null_;
  file sas2xl;
  put '[ workbook.name("sheet1","Data from
        SAS")]';
run;
```

Observe failure: the SAS log displays some cryptic generic DDE error message, and on the Excel side, the sheet name 'Sheet1' is highlighted in the lower left corner name tab, but it has not been replaced by the desired value. Before proceeding, we make sure to de-select the highlighting of the sheet name by clicking on some worksheet cell...

To make this work, we need to insert an old-style Excel 4 macro sheet into the workbook by calling the 'workbook.insert' function with a value of three. The resulting new macro sheet will appear with the default name 'Macro1':

```
data _null_;
  file sas2xl;
  put '[ workbook.next() ]';
  put '[ workbook.insert(3) ]';
run;
```

Note that we have also coded a 'workbook.next', the effect of which is to activate the next sheet in the workbook. An unnecessary precautionary measure in our present case, but generally good coding practice as it avoids multiple sheets being accidentally inserted should more than just a single worksheet be selected...

Our strategy is now as follows: we define a triplet-style DDE fileref, pointing to a sufficiently large range of cells on the macro-sheet 'Macro1'. To this range we write an Excel 4 macro that will effectuate the renaming operation, and subsequently run it:

```
filename xlmacro dde
  'excel|macro1!r1c1:r100c1' notab;
data _null_;
  file xlmacro;
  put '=workbook.name("sheet1","Data from
        SAS")';
```

```
  put '=halt(true)';❶
  put '!dde_flush';❷
  file sas2xl;
  put '[ run("macro1!r1c1")]';❸
run;

filename xlmacro clear;
```

❶ The 'halt(true)' statement is required to end the Excel macro, even if nothing else follows underneath.

❷ A major distinction between writing output to a DDE triplet fileref like xlmacro, and a DDE system-doublet like sas2xl, is this: output to a triplet gets buffered during data step execution, and is only pumped through the DDE connection when the data _null_ step terminates.

Output to a system-doublet however, is immediately passed on to Excel. Since the writing of our Excel sheet-renaming macro code occurs in the same data step as the command to run said macro, we need to explicitly flush the DDE buffer with the special '!dde_flush' command. Otherwise we'd be attempting to run the Excel macro before it actually exists in the workbook. Admittedly, the issue is largely academic, since we could have coded two separate data _null_ steps, one writing the macro via xlmacro, and one calling it via sas2xl.

❸ The 'run' function executes the columnar Excel macro, the first cell of which is given as its parameter string.

Note that the syntax for Excel 4 macro coding is the same as what we use when writing to the sas2xl DDE doublet: simply strip off the square brackets, and stick an equality character in front of the function.

After verifying that 'Sheet1' has indeed been satisfactorily renamed, we tidy up the macro sheet with a 'clear all' function:

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

5 List Existing Worksheet Names

In Vyverman (2001) a somewhat convoluted piece of SAS macro code was presented—the %loadnames macro—in the context of automating the handling of workbook sheets by means of SAS code. It addresses the question whether there is a way of letting the SAS System know the exact names and order of the sheets in a given workbook by producing a small SAS data set containing precisely this information.

The use of %loadnames requires an empty old-style macro sheet 'Macro1' to be available on the targeted workbook. Furthermore, its proper functioning relies on the 'Macro1' sheet being in the first position. Having just cleared the contents of the 'Macro1' sheet in our open workbook, we merely need to move it into pole position:

```
data _null_;
  file sas2xl;
  put '[ workbook.move("macro1","4 Renaming
        Worksheets.xls",1)]';
run;
```

Aiming for completeness of code within the present article, the full %loadnames definition is included below:

```

%macro loadnames;
%local sh wn nsheets;
%let sh=0;
%let wn=0;
%let nsheets=0;
filename xlmacro dde
'excel|macro1!r1c1:r100c1' notab;
data _null_;
file xlmacro;
put '=set.value($b$1,
get.workbook(4))';
put '=halt(true)';
put '!dde_flush';
file sas2xl;
put '[ run("macro1!r1c1")]';
put '[ error(false)]';
run;
filename nsheets dde
'excel|macro1!r1c2:r1c2' notab;
data _null_;
length nsheets 8;
infile nsheets;
input nsheets;
call symput('nsheets',trim(left(put(
nsheets,2)))));
run;
%let nsheets=%eval(&nsheets-1);
data _null_;
file sas2xl;
put '[ workbook.activate("macro1")]';
put '[ select("r1c1:r100c1,
r1c2:r1c2")]';
put '[ clear(1)]';
put '[ select("r1c1")]';
run;
data _null_;
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")';
%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";
data _sheet_names;
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;

```

```

filename xlmacro clear;
%mend loadnames;

```

We will not dwell further upon the internal workings of this bit of code. Some of the X4ML functions that are used in it will be briefly highlighted in the next section, where they are deployed in a considerably lighter setting, and additional notes may be consulted in Vyverman (2001). Suffice it to describe the general idea: similar to the solution for the sheet-renaming problem as discussed in the previous section, SAS code is used to write an Excel 4 macro into the first column of the 'Macro1' sheet.

When this macro is called, it will cycle through the available sheets on the workbook, interrogate each sheet as to its name, and enter the discovered sheet names one by one into the cells of the second column of the 'Macro1' sheet. It is then a small matter to set up a DDE triplet pointing to these, and read the names into a SAS data set by means of infile and input statements. We try this on the open workbook:

```
%loadnames;
```

The intrepid reader who decides to examine the internals of %loadnames will quickly discover that it builds an Excel macro in an extremely inefficient manner. The reason for this is to be sought largely in the author's ignorance of Excel 4 macro programming. Donuts will be distributed to those who come up with a more elegant solution that yields an output data set identical to the one made by %loadnames.

Once again we clear the contents of the 'Macro1' sheet:

```

data _null_;
file sas2xl;
put '[ workbook.activate("macro1")]';
put '[ select("r1c1:r100c2")]';
put '[ clear(1)]';
run;

```

6 How Many Rows / Columns?

When attempting to deal with Excel workbooks in an automated fashion, another interesting piece of information to have is the number of rows and columns that are used on a given worksheet.

In this section we implement a strategy similar to the one used for the extraction of the available sheet names, in order to create two global SAS macro variables containing exactly these numbers.

We define a triplet-style DDE fileref xlmacro, pointing to a sufficiently large range of cells in the first column of the freshly cleared macro sheet 'Macro1':

```

filename xlmacro dde
'excel|macro1!r1c1:r20c1' notab;

```

Having done so, we write an Excel 4 macro there that will query the worksheet 'Data from SAS' and enter the number of rows and columns used on that specific sheet into the first two cells of the second column of 'Macro1'. After flushing the DDE triplet buffer, we run the Excel macro:

```

data _null_;
file xlmacro;
put '=select(!$b$1)';
put '=set.name("rows",selection())';
put '=select(!$b$2)';
put '=set.name("cols",selection())';
put '=workbook.activate("data from

```

```

        sas");
    put '=set.value(rows,get.document(10))';
    put '=set.value(cols,get.document(12))';
    put '=workbook.activate("macro1")';
    put '=halt(true)';
    put '!dde_flush';
    file sas2xl;
    put '[run("macro1!r1c1")]';
    put '[error(false)]';
run;

filename xlmacro clear;

```

A few words on functions used: ‘selection()’ returns the current selection, which in this case corresponds to either the first or the second cell of column B on the ‘Macro1’ sheet.

The ‘set.name’ function defines a named range, a text label corresponding to a range of cells. Here, we use it to stick the label ‘rows’ on cell \$B\$1, and ‘cols’ on cell \$B\$2.

The related function ‘set.value’ is subsequently used to enter the number of the last used row—‘get.document(10)’—and the number of the last used column—‘get.document(12)’—into the named ranges on the ‘Macro1’ sheet. Note the absence of double quotes around the names of the ranges.

As the wanted numbers are now sitting within their respective cells, reading them into SAS is merely a matter of defining triplet-style DDE filerefs and some infile / input statements:

```

filename xlrows dde 'excel|macro1!r1c2';
filename xlcols dde 'excel|macro1!r2c2';

data _null_;
  length
    xlrows
    xlcols 8;
  infile xlrows;
  input xlrows;
  call symput('xlrows',
    trim(left(put(xlrows,5)))));
  infile xlcols;
  input xlcols;
  call symput('xlcols',
    trim(left(put(xlcols,5)))));
run;

filename xlrows clear;
filename xlcols clear;

%put Rows used on Excel sheet: &xlrows;
%put Columns used on Excel sheet: &xlcols;

```

A word of caution: the number of the last used row/column as returned by the ‘get.document’ function is *not* an indicator of the number of worksheet cells that actually contain data. Suppose we have a worksheet with a single number in cell r1c1. If we, say, set the background color of cell r1500c20 to yellow, without typing any data in it, and then run the above routine, the outcome will be 1500 rows and 20 columns used. Aw...

7 Sending Keystrokes

As a final topic, let us turn to something of a more light-hearted nature. Sending keystrokes *e.g.* is great fun. We activate ‘Sheet3’ on our open workbook and do something utterly useless, like randomly coloring some randomly sized cell-ranges in a flickering flurry of flashing rectangles. But seriously, the cell-ranges are selected by using the ‘send.keys’ function in a

manner equivalent to holding the shift-key pressed down while hitting the arrow-keys:

```

data _null_;
  file sas2xl;
  put '[workbook.activate("sheet3")]';
  put '[app.activate("microsoft excel - 4
    renaming worksheets.xls")]';
  do cells=1 to 30;
    put '[send.keys("&{home}")]';
    row=floor(30*ranuni(0));
    col=floor(13*ranuni(0));
    color=floor(56*ranuni(0));
    do step=1 to row;
      put '[send.keys("&{down}")]';
    end;
    do step=1 to col;
      put '[send.keys("&{right}")]';
    end;
    put '[patterns(1,0,"color",true)]';
  end;
  put '[send.keys("&{home}")]';
run;

```

Something to realize when using ‘send.keys’ is that the key-strokes are actually being sent to the application that is currently running in the foreground. Which is our SAS session! Hence the very important ‘app.activate’, which will bring the Excel application into focus instead.

Failing to do so will in the best case send a bunch of characters flying like shrapnel all over the program editor, destroying what code is still displayed in it. In the worst case it may kill the SAS session—try sending an ALT-F-X keystroke for fun...

Furthermore, the parameter string for ‘app.activate’ must be *exactly* the same as the text on the Excel application’s button in the Windows Task Bar. Including spaces and file extension.

The parameter string for ‘send.keys’ may of course contain any text-string. In the above example, we use only some of the non-alphanumerical keys: “&{home}” is equivalent to pressing CTRL-HOME and selects the r1c1 worksheet cell. “&{down}” achieves the same thing as hitting the downward arrow-key once while holding the shift-key depressed. As such, it extends the selected cell range downwards by one row. A full list of special keystrokes is given in the X4ML help file.

The slightly farcical nature of our sample code should nevertheless not be allowed to obscure the fact that ‘send.keys’ may well be the only way out in situations where combined key-strokes are required. An important functionality to have at hand, since *e.g.*, to name but one odd construction, entering an Excel Formula Array into a worksheet cell requires pressing CTRL-SHIFT-RETURN. But then, a full discussion of the possibilities hinted at by *that* particular construct lies, alas, definitely beyond the scope of this paper.

8 In Closing

It is really recommendable to try out all the code presented here in a set-up where the SAS application occupies one half of the display—only a program editor is needed, nothing worthwhile gets written to the log—and Excel the other half. That way the code can be submitted step by step, and one observes simultaneously what happens in Excel.

The code, by the way, will be available from the author’s website as a set of 8 ‘.sas’ source files, numbered in correspondence

with sections zero through seven in this paper. Simply visit www.vyverman.com and follow the link to SAS-related material.

All code was developed, tested, and approved of on a Windows NT4 (SP6) system, running release 8.2 of the SAS System and MS Office97.

References

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

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, 2000.

Vyverman, K. "Using Dynamic Data Exchange to Export Your SAS Data to MS Excel — Against All ODS, Part I". *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*, paper 11, 2001.

Acknowledgments

My sincere thanks to everyone I have had the pleasure of exchanging DDE related ideas with in recent years. Especially so, in fact, to those who either via the SAS-L forum or in private communications have asked those questions which caused me to venture deeper into the vast *terra incognita* of DDE. Without your interest and curiosity, this paper would not be.

Trademarks

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 registered trademarks or trademarks of their respective companies.

Contacting the Author

The author welcomes and encourages any questions, corrections, improvements, 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