Excel Exposed: Using Dynamic Data Exchange to Extract Metadata from MS Excel Workbooks

Koen Vyverman, SAS Institute — the Netherlands

Abstract

Dynamic Data Exchange (DDE) can be used on the Windows platform to create fully customized MS Excel and MS Word files, all from within a Base SAS program. But it also comes in handy for gathering useful metadata about your Excel files! When automating the reading from, or writing to an Excel workbook, life becomes a lot easier if you have access to certain facts about the workbook you're manipulating. Such as the names of the worksheets it comprises; the type of each worksheet — data, graphics, or macros; the number of rows and columns that are in use on each data-sheet; whether a given column contains numerical, character, or mixed type values; and so forth. In this tutorial we will go through the necessary DDE-movements that allow the extraction of such Excel workbook metadata. Wrapping up, a SAS macro is shown to load these metadata into a SAS data set for easy reference in the manner of a dictionary table.

0 What's Up?

The dictionary tables — or rather, views — that come with Base SAS are a fantastic, yet oft underestimated tool for automating in- and output processes. Think of the `SASHELP.VTABLE` view, which holds a wide variety of metadata that describe attributes of all SAS data sets in all SAS libraries: data set names, number of observations, number of variables, …

In a typical corporate data environment though, not all data are readily available in the form of SAS data sets. In the best scenario, there may be a well-kept data warehouse, which is seen as the single source of Truth™ by all and sundry. But even in such an ideal situation, there will still be a bunch of Excel spreadsheets floating around that suddenly appear to contain vital morsels of wisdom.

Would it not be cool to have a tool similar to the aforementioned `SASHELP` views, that would hold sufficient metadata about all those rogue Excel workbooks such as to enable treating them as regular data sources rather than having to examine them one by one and build kludge after kludge to control this unruly lot?

Such is the purpose of this paper: we will build a SAS data set `SASHELP.XLSHEETS` that we'll populate with all manner of metadata coaxed from Excel workbooks by means of Dynamic Data Exchange. We will even go one step further, and show how to gather metadata for individual spreadsheet columns.

1 A Very Concise Introduction to DDE

Extensive detail of what DDE is and how to access it from Base SAS code, has already been covered in previous publications: Vyverman (2000, 2001, 2002), Viergever & Vyverman (2003). In order to make the present paper self-contained, a few essential things need mentioning though. Dynamic Data Exchange is a communication protocol available on the Windows platform — and reportedly also on OS/2 and MacOS. It enables DDE-compliant applications on these platforms to talk to each other in a client/server fashion.

The SAS System is only DDE-compliant in the sense that it can act as a client but not as a server. That is: SAS can contact Excel and ask it to do certain things, but not the other way around. Metaphorically speaking, DDE can be seen as a common language that both applications use to communicate. See also the Technical Support document TS325 (SAS Institute, 1999) for a more DDE background and a list of other DDE-compliant applications.

Considering the client/server picture, it is clear that a prerequisite for using DDE is that both client and server applications are up and running, otherwise they couldn't have a chat.

Once both a SAS session and Excel are running, a client/server DDE-communication is initiated by means of a special form of the SAS `filename` statement. It basically comes in two flavours (although in the present paper we will also make good use of a more uncommon form):

```
filename <fileref> dde 'excel|
  [<workbook.xls>]<sheet>!<cell-range>';
```

and

```
filename <fileref> dde 'excel|system';
```

In the first of these basic forms, the expression `excel|[<workbook.xls>]<sheet>!<cell-range>` is known as the DDE-triplet. It tells the SAS session that it needs to talk to Excel, that the conversation is about a certain worksheet in a certain workbook, and what part this worksheet to read/write data from/to. Once a fileref has been defined in this manner, it can be used on `file` and `infile` statements just like any other SAS fileref. The usual `put` and `input` statements will then write to and read from the specified workbook/worksheet/cell-range.

The second form of the DDE `filename` statement — which for the sake of naming consistency we shall refer to as the DDE doublet, or the system-doublet — does not read/write from/to a specific bunch of cells. Rather, the system-doublet allows the SAS session to send commands to Excel, telling it what to do. These commands must be formulated in the old MS Excel 4 Macro Language from before the VBA days. We will refer to this particular lingo as X4ML.

It is oft noted that the main frustration with using DDE between SAS and Excel stems from the fact that one cannot find the proper X4ML function to get some specific task done. However, there is an X4ML help-file available from the Microsoft site: download and run the file 'macrofun.exe' from the main Microsoft site. This installs the actual help-file 'macrofun.hlp', which contains the full syntax for hundreds of X4ML functions, as well as some usage examples. The reader is urged to download the help-file before proceeding, since due to space-limitations we will not be able to furnish the full syntax for all the X4ML functions used in the code.

2 A Few Preliminary Steps

Before kicking off with the code, note that due to the columnar lay-out of the present article, a lot of the longer lines have wrapped. It is suggested not to just copy/paste code from this paper into a SAS program editor, but rather to use the SAS code files available from www.vyverman.com

As we work our way through the necessary steps to set up a SASHELP.XLSHEETS metadata table, we will be using a sample Excel workbook that has a number of ordinary worksheets filled with data, some Excel pivot-tables to create summaries, and a couple of charts of various types. This sample workbook 'Sales Demo.xls' is also included in the conference package on the web-site.

Since the sample workbook contains data-sheets as well as charts, we will be digging up metadata pertaining to both types of worksheet. For starters, we create an empty XLSHEETS data set in the SASHELP library:

```
data sashelp.xlsheets;
  length
    wb_name
    wb_path      $ 300
    sh_name      $  31
    n_rows
    n_cols          8
    sh_type      $   1
    sh_order
    fu_row
    fu_col          8
    mc_type
    oc_type      $   2
    mc_n_series
    oc_n_series     8
    ;
  label
    wb_name      = 'Workbook Name'
    wb_path      = 'Workbook Path'
    sh_name      = 'Sheet Name'
    n_rows       = 'Number of Rows'
    n_cols       = 'Number of Columns'
    sh_type      = 'Sheet Type'
    sh_order     = 'Sheet Order'
    fu_row       = 'First Used Row'
    fu_col       = 'First Used Column'
    mc_type      = 'Main Chart Type'
    oc_type      = 'Overlay Chart Type'
    mc_n_series = 'Number of Series on
                   Main Chart'
    oc_n_series = 'Number of Series on
                   Overlay Chart'
    ;
  format
    sh_type $sh_type.
    mc_type
    oc_type $ch_type.
    ;
  delete;
run;
```

This to give an idea of the kind of metadata that we're after:

- 'Workbook Name' and 'Workbook Path' will uniquely identify the Excel workbook that's being interrogated by our code. We reserve 300 characters for both the name and the path to make sure we have enough space.
- The 'Sheet Name' variable will list the names of all the sheets present on the current workbook. Sheet-names have a maximal length of 31 characters in MS Office97.
- 'Number of Rows' and 'Number of Columns' will tell us the last row and column used on a sheet — provided of course that it is of the worksheet type and not a chart.
- The 'Sheet Type' will make clear whether the current sheet is a data-sheet, a chart, an Excel 4 macro sheet, …
- 'Sheet Order' yields the position of the current sheet within the workbook, counting from left to right.
- 'First Used Row' and 'First Used Column' will contain the co-ordinates of the top-left-most cell used on a worksheet.
- The 'Main Chart Type' and 'Overlay Chart Type' will tell whether a chart (resp. an overlay chart) is a bar-chart, a pie, a donut, …
- And the two 'Number of Series' variables will yield the number of Excel data series that are plotted on a given chart (resp. an overlay chart).

To make the results look nicer, we define two character formats. One for the 'Sheet Type' variable, and one for both of the 'Chart Type' variables. We store these in the SASUSER library, so if necessary, add SASUSER to the FMTSEARCH system option…

```
proc format library=sasuser;
  value $sh_type
    '1'   = 'Worksheet'
    '2'   = 'Chart'
    '3'   = 'Excel 4 Macro Sheet'
    '4'   = 'Excel 4 International Macro
             Sheet'
    '5'   = '(Reserved)'
    '6'   = 'Microsoft Excel Visual Basic
             Module'
    '7'   = 'Dialog'
    other = 'Who Knows What This Might
             Be?'
    ;
run;

proc format library=sasuser;
  value $ch_type
    '  '  = 'N.A.'
    '0 '  = 'No Overlay Chart'
    '1 '  = 'Area'
    '2 '  = 'Bar'
    '3 '  = 'Column'
    '4 '  = 'Line'
    '5 '  = 'Pie'
    '6 '  = 'XY (scatter)'
    '7 '  = '3D Area'
    '8 '  = '3D Column'
    '9 '  = '3D Line'
    '10'  = '3D Pie'
    '11'  = 'Radar'
    '12'  = '3D Bar'
    '13'  = '3D Surface'
    '14'  = 'Donut'
    other = 'Who Knows What This Might
             Be?'
    ;
run;
```

3 Starting Up Excel

Now we can get started. To keep track of certain constants we define the following global SAS macro variables:

```
%let wb_path=c:\tu15;
%let wb_name=Sales Demo;
%let temp_macro_sheet=;
%let n_sheets=0;
```

So `wb_name` and `wb_path` hold the name and the exact whereabouts of the sample workbook. Important: note that the filename is given without the usual Excel .xls extension, and the directory path does not have a trailing backslash. The other two variables will be derived further on, they are merely being announced here.

To start up an instance of MS Excel — the server application in the DDE client/server paradigm — we use Chris Roper's SCL-function method:

```
options noxsync noxwait;
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');
    start=datetime();
    stop=start+10;
    do while (fid le 0);
      fid=fopen('sas2xl','s');
      time=datetime();
      if (time ge stop) then fid=1;
      end;
    end;
  rc=fclose(fid);
run;
```

A full discussion as to the workings of this code can be found in Roper (2000). Note that in the process of firing up Excel, we have defined a DDE system-doublet fileref `SAS2XL` through which we will be sending X4ML commands to Excel.

When Excel has started, we proceed to open our 'Sales Demo.xls' workbook:

```
data _null_;
  file sas2xl;❶
  length ddecmd $ 200;
  put '[error(false)]';❷
  ddecmd='[open("'||"&wb_path"||'\'||
         "&wb_name"||'")]';❸
  put ddecmd;
run;
```

As this is our first SAS chat with Excel, it is perhaps useful to expand briefly upon a few features that will be recurring throughout the code to come.

❶ At the risk of belabouring the point: the use of the `sas2xl` fileref here means that anything on a subsequent `put` statement will go straight to Excel.

❷ The `error(false)` X4ML command switches off the Excel error message dialogs. Surely we don't wish to be halted in our tracks time and again by 'Are you sure you want to do this?' type of interjections. Note that this is a toggle, so we need to issue the command just once in the whole process.

❸ Simply using the SAS macro variables on the `put` statement won't work. We need to make sure that the macro variables are being resolved before the X4ML command is sent to Excel. This can be achieved in various ways. Probably the most didactic one is shown here: we simply build a dummy data step character variable `ddecmd` to force resolution of the macro variables. An added advantage is that during the development of code, it is handy to be able to check the content of `ddecmd` for potential X4ML syntax errors. In the present case *e.g.* `ddecmd` will contain the string `[open("c:\tu15\Sales Demo")]`.

4 Extracting the Sheet Names

In previous publications concerning the use of DDE in a SAS-Excel context, the author has proposed and repeatedly used a SAS macro `%loadnames`, the purpose of which was to create a SAS data set containing the names of all sheets present on a given Excel workbook, in their order of appearance.

Within the present context of setting up an Excel metadata table `SASHELP.XLSHEETS`, the use of `%loadnames` has become obsolete. Which is probably for the best, because the way in which `%loadnames` operates has always seemed somewhat contrived and overly circumspect.

In order to extract the list of sheet names from our 'Sales Demo.xls' workbook, we shall use a rather uncommon form of the DDE `filename` statement, as intimated in Section 1. There exists a special 'topics'-triplet, which is documented in TS325 (SAS Institute, 1999):

```
filename xltopics dde
  'excel|system!topics' lrecl=32000;
```

To get a feel for what this does, we read in the `xltopics` stream and write the contents to the SAS log:

```
data _null_;
  length topic $ 1000;
  infile xltopics pad dsd notab dlm='09'x;
  input topic $ @@;
  put topic=;
run;
```

This gives, in part:

```
topic=[:]:
topic=[ABCMM.XLA]Sheet1
topic=[DESIGNER.XLA]Sheet1
topic=[FLOW95.XLA]Sheet1
topic=[PDFMaker.xla]Sheet1
topic=[PP.XLA]Sheet1
topic=[Sales Demo.xls]Month AM Net Pivot
topic=[Sales Demo.xls]Month BA Net Pivot
topic=...
topic=[Sales Demo.xls]Net IFA Top 10
topic=[Sales Demo.xls]Net UL Top 10
topic=System
```

There is a lot of junk here, but a pattern emerges: the values of `topic` that commence with the workbook name within a set of square brackets actually yield the sheet names in alphabetical order! Armed with this knowledge, we proceed to create a data set `_sheetnames_before`, in which we essentially store the sheet names as `sh_name`: We use the suffix '`_before`' to indicate the fact that in this data set we have captured the sheet names as they exist *before* we start tampering with the workbook. The use of this will become apparent in Section 5.

```
data _sheetnames_before(drop=str strlen
                            topic);
  length
    strlen       8
    sh_name $   31
    str
    wb_name
    wb_path $  300
    topic   $ 1000
    ;
  retain
    str     " "
    strlen  0
    wb_name "&wb_name"
    wb_path "&wb_path"
    ;
  infile xltopics pad dsd notab dlm='09'x;
  input topic $ @@;
  if _n_=1 then do;
    str="[&wb_name..xls]";
    strlen=length(trim(str));
    end;
  if topic =: "[&wb_name..xls]" then do;
    sh_name=substr(topic,strlen+1);
    output;
    end;
run;
filename xltopics clear;
```

We have already defined a global SAS macro variable `n_sheets`, which we now populate with the number of sheets present on our workbook:

```
data _null_;
  set _sheetnames_before end=last;
  if last then do;
    call symput('n_sheets',trim(left(
                put(_n_,2.))));
    end;
run;
```

5 Adding a Temporary Excel 4 Macro Sheet

Before we continue, now is probably a good point to explain something about the mechanism by means of which we will unearth the sheet-related metadata from Excel. We are going to insert a new sheet into our 'Sales Demo.xls' workbook, and make it a sheet of the type 'Excel 4 Macro Sheet'.

By means of an appropriate DDE-triplet pointing to the first column of cells on this — temporary — macro sheet, we will write an Excel 4 macro there that, when executed, will populate the other columns on the macro sheet with the desired information.

Pulling the results back into SAS is then an exercise in reading data from an Excel sheet and applying some degree of post-processing.

So, as a first step we add an Excel 4 macro sheet to the open workbook:

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

❶ The `workbook.insert` function inserts as many new sheets as are currently selected. To make sure that there is only one sheet selected, we tell Excel to release the current sheet-selection, and move on to the next one in line. A mere matter of good DDE coding practice…

❷ The argument of `workbook.insert` determines the type of sheet that is added to the workbook. The complete list of possibilities can be seen in the code that sets up the sheet type SAS format `$sh_type` (Section 2).

Upon inserting the macro sheet, we have left the business of giving it a name entirely to Excel. Indeed, Excel maintains a naming scheme for different sheet types, and chances are that our new sheet is called 'Macro1'. Except then if the workbook already contained a sheet by that name, in which case Excel will name the new one 'Macro2' by default. Except if… And so forth. So how do we now figure out the name of our macro sheet?

Remember that we have stored the list of sheet names before messing with the workbook in a `_sheetnames_before` SAS data set. So all we need to do is query the special 'topics'-triplet once more in exactly the same way as before to create a new SAS data set `_sheetnames_after`, and the difference between the two data sets will be exactly one sheet name: the name of our new macro sheet. So:

```
filename xltopics dde
  'excel|system!topics' lrecl=32000;

data _sheetnames_after(drop=str strlen
                           topic);
  ...(exact same data step)
  ...(code as used for making)
  ...(_sheetnames_before)
run;

filename xltopics clear;
```

And by comparing the `_after` and `_before` data sets, we learn the name of the new macro sheet which we store in the global SAS macro variable `temp_macro_sheet`:

```
proc sql noprint;
  select
    sh_name into: temp_macro_sheet
            separated by ''
  from
    _sheetnames_after
  where
    sh_name not in (select sh_name from
                    _sheetnames_before)
  ;
quit;
```

One important remark here: it may perhaps seem odd to specify `separated by ''`, as we only expect one value to be returned by the query. Not doing so however would yield a macro variable padded with trailing blanks to the full length of the `sh_name` variable, which is potentially 31 characters long. We will be resolving `temp_macro_sheet` in DDE triplet `filename` statements and X4ML command strings, hence the need to eliminate superfluous blanks.

Now that we know its name, we can proceed to move the temporary macro sheet into pole position. This may appear to be a quite unnecessary move — no pun intended — at this point, but the usefulness of doing so will become clear in Section 7.

```
data _null_;
  file sas2xl;
  length ddecmd $ 200;
  ddecmd='[workbook.move("'||
         "&temp_macro_sheet"||'","'||
         "&wb_name..xls"||'",1)]';
  put ddecmd;
run;
```

When submitting this step, the ddecmd string resolves to:

```
[workbook.move("Macro1","Sales
                              Demo.xls",1)]
```

Which tells Excel to move the sheet 'Macro1' to position 1 on the workbook 'Sales Demo.xls'.

We then define a triplet-style DDE fileref XLMACRO, pointing to the full first column on our temporary macro sheet:

```
filename xlmacro dde
  "excel|&temp_macro_sheet!r1c1:r65536c1"
  notab lrecl=200;
```

As explained at the beginning of this section, XLMACRO is where we will write and execute some genuine Excel 4 macro code.

6 Surfacing Workbook Metadata

And so we come to the center-piece of the entire construction. The following data step will generate, and subsequently execute, an Excel 4 macro in the XLMACRO range, to make Excel enter certain types of sheet metadata elsewhere in the temporary macro sheet. This looks quite ghastly, but explanations follow:

```
data _null_;
  set _sheetnames_before;❶
  length maccmd ddecmd $ 200;❷
  file xlmacro;
  maccmd='=select(!$b$'||
         trim(left(put(_n_,8.)))||')';❸
  put maccmd;
  put '=set.name("rows",selection())';❹
  maccmd='=select(!$c$'||
         trim(left(put(_n_,8.)))||')';
  put maccmd;
  put '=set.name("cols",selection())';
  maccmd='=select(!$d$'||
         trim(left(put(_n_,8.)))||')';
  put maccmd;
  put '=set.name("type",selection())';
  maccmd='=select(!$e$'||
         trim(left(put(_n_,8.)))||')';
  put maccmd;
  put '=set.name("pos",selection())';
  maccmd='=select(!$f$'||
         trim(left(put(_n_,8.)))||')';
  put maccmd;
  put '=set.name("furow",selection())';
  maccmd='=select(!$g$'||
         trim(left(put(_n_,8.)))||')';
  put maccmd;
  put '=set.name("fucol",selection())';
  maccmd='=set.value(rows,
         get.document(10,"'||
         trim(left(sh_name))||'"))';❺
  put maccmd;
  maccmd='=set.value(cols,
         get.document(12,"'||
         trim(left(sh_name))||'"))';
  put maccmd;
  maccmd='=set.value(type,
         get.document(3,"'||
         trim(left(sh_name))||'"))';
  put maccmd;
  maccmd='=set.value(pos,
         get.document(87,"'||
         trim(left(sh_name))||'"))';
  put maccmd;
  maccmd='=set.value(furow,
         get.document(9,"'||
         trim(left(sh_name))||'"))';
  put maccmd;
  maccmd='=set.value(fucol,
         get.document(11,"'||
         trim(left(sh_name))||'"))';
  put maccmd;
  put '=halt(true)';❻
  put '!dde_flush';❼
  file sas2xl;
  ddecmd='[run("'||
         "&temp_macro_sheet"||'!r1c1")]';
  put ddecmd;❽
run;
```

❶ Let's begin by examining the implicit data step loop. The data set _sheetnames_before has one observation for each sheet present on the workbook (minus the temporary macro sheet). For each observation/sheet, we have the sheet name stored in the variable sh_name.

❷ As before, we use a dummy character variable to force resolution of SAS macro variables before sending commands to Excel, or merely to concatenate different parts of an expression. Strictly speaking, one dummy should do the trick, but as we will be writing both to the XLMACRO cell-range, and to the SAS2XL system-doublet in this data step, we'll use maccmd for stuff that gets sent to the former, and ddecmd for stuff that goes to the latter.

❸ Here we select a single cell in column B of the macro sheet. Note the automatic data step iteration counter _n_. As we loop through _sheetnames_before, different cells get selected: b1, b2, …

❹ The set.name function attributes an Excel name, a label really, to the currently selected cell, which is referenced as selection(). As the data step loops, the cells in column B are one by one labelled as 'rows', because here we will be storing the number of rows used on each workbook sheet.

The process is repeated a number of times, and cell names/labels are defined for each of the quantities that we wish to capture. Here's a list of columns, cell labels, and their intended use:

- B rows Number of Rows
- C cols Number of Columns
- D type Sheet Type
- E pos Sheet Position
- F furow First Used Row
- G fucol First Used Column

❺ Once we have defined all the cell labels for a given value of sh_name — which in its turn corresponds to a row on the macro sheet — we can use the set.value function to enter

data into the cells. The first argument of `set.value` is the label of the cell whose value needs to be set. The second argument specifies the value. In this case *e.g.* we set the cell referenced as 'rows' to a value returned by the `get.document` function. This `get.document` function returns a host of different attributes pertaining to a specific sheet name, depending on the numerical parameter. In the present case, a parameter value of 10 will yield the number of the last used row on the specified sheet.

We repeat the `set.value` procedure a number of times, referencing the differently labelled cells, and entering sheet attributes as returned by the corresponding `get.document` calls with different parameter settings:

- 10 Number of Rows
- 12 Number of Columns
- 3 Sheet Type
- 87 Sheet Position
- 9 First Used Row
- 11 First Used Column

❻ This indicates the end of the Excel 4 macro.

❼ The DDE buffer and the importance of flushing it before running an Excel 4 macro have been dealt with in previous papers. Suffice it to say that it is necessary here because for every observation on `_sheetnames_before`, we are actually writing a different macro into the `A` column of the macro sheet.

❽ It is only after the DDE buffer has been flushed, that the cells in the `XLMACRO` range really contain the macro code. We issue a `run` command to the `SAS2XL` system-doublet, which, well, runs the macro! As an example, this is the complete Excel 4 macro code for the 16th sheet name value:

```
=SELECT(!$B$16)
=SET.NAME("rows",SELECTION())
=SELECT(!$C$16)
=SET.NAME("cols",SELECTION())
=SELECT(!$D$16)
=SET.NAME("type",SELECTION())
=SELECT(!$E$16)
=SET.NAME("pos",SELECTION())
=SELECT(!$F$16)
=SET.NAME("furow",SELECTION())
=SELECT(!$G$16)
=SET.NAME("fucol",SELECTION())
=SET.VALUE(rows,GET.DOCUMENT(10,"Net UL
                                Top 10"))
=SET.VALUE(cols,GET.DOCUMENT(12,"Net UL
                                Top 10"))
=SET.VALUE(type,GET.DOCUMENT(3,"Net UL
                               Top 10"))
=SET.VALUE(pos,GET.DOCUMENT(87,"Net UL
                               Top 10"))
=SET.VALUE(furow,GET.DOCUMENT(9,"Net UL
                                Top 10"))
=SET.VALUE(fucol,GET.DOCUMENT(11,"Net UL
                                 Top 10"))
=HALT(TRUE)
```

The columns `B` through `G` of the macro sheet now contain our metadata. Before we try to read these into SAS, we need to make sure that we'll be able to match the metadata up with the correct sheet names.

The easiest way of ensuring this is to clear the `XLMACRO` range, *i.e.* the `A` column on the macro sheet, and then write the sheet names in there:

```
data _null_;
  file sas2xl;
  length ddecmd $ 200;
  ddecmd='[workbook.activate("'||
         "&temp_macro_sheet"||'")]';
  put ddecmd;
  put '[select("r1c1:r65536c1")]';
  put '[clear(1)]';
  put '[select("r1c1")]';
run;

data _null_;
  set _sheetnames_before;
  file xlmacro;
  put sh_name;
run;
```

This concludes the work on the Excel side. All that we wanted to find out is now present on the temporary macro sheet.

7 Loading the Workbook Metadata into SAS

We then define a triplet-style fileref `SHPARAMS` pointing to the range of cells on the macro sheet containing our various sheet parameters:

```
filename shparams dde "excel|
  &temp_macro_sheet!r1c1:r&n_sheets.c7"
  lrecl=200;
```

The data step that reads all this into a SAS data set takes care to define all the variables initially as character strings. Indeed, some of the cells that should really be numerical, like the number of rows used on a sheet, can contain alphanumerical values in Excel. In particular, the value '#N/A' will appear whenever *e.g.* `get.document` was asked to return the number of rows used on a sheet that is actually of the Chart type. So the most cautious way to proceed is to read everything into SAS character variables, and then post-process.

```
data _sheet_parameters;
  length
    sh_name       $ 31
    n_rows_char
    n_cols_char
    fu_row_char
    fu_col_char   $  5
    sh_type       $  1
    sh_order_char $  5
    ;
  label
    sh_name  = 'Sheet Name'
    sh_type  = 'Sheet Type'
    ;
  format
    sh_type $sh_type.
    ;
  infile shparams notab dlm='09'x dsd
    missover;
  input
    sh_name
    n_rows_char
    n_cols_char
    sh_type
    sh_order_char
    fu_row_char
    fu_col_char
    ;
run;
```

```
filename shparams clear;
```

As explained above, _sheet_parameters now needs some post-processing:

```
data _sheet_parameters(drop=i n_rows_char
     n_cols_char fu_row_char fu_col_char
     sh_order_char);❶
  set _sheet_parameters;
  length
    wb_name
    wb_path     $ 300
    n_rows
    n_cols
    fu_row
    fu_col
    sh_order        8
    mc_type
    oc_type     $   2
    mc_n_series
    oc_n_series     8
    ;
  label
    wb_name     = 'Workbook Name'
    wb_path     = 'Workbook Path'
    n_rows      = 'Number of Rows'
    n_cols      = 'Number of Columns'
    fu_row      = 'First Used Row'
    fu_col      = 'First Used Column'
    sh_order    = 'Sheet Order'
    mc_type     = 'Main Chart Type'❷
    oc_type     = 'Overlay Chart Type'
    mc_n_series = 'Number of Series on
                   Main Chart'
    oc_n_series = 'Number of Series on
                   Overlay Chart'
    ;
  format
    mc_type
    oc_type $ch_type.
    ;
  wb_name="&wb_name";
  wb_path="&wb_path";
  if sh_type='2' then do;❸
    mc_type=fu_row_char;
    mc_n_series=input(fu_col_char,8.);
    if upcase(n_rows_char)='#N/A' then do;
      oc_type='0';
      oc_n_series=.;
      end;
    else do;
      oc_type=n_rows_char;
      oc_n_series=input(n_cols_char,8.);
      end;
    n_rows_char='.';
    n_cols_char='.';
    fu_row_char='.';
    fu_col_char='.';
    end;
  array charz(i) n_rows_char n_cols_char
                 fu_row_char fu_col_char;
  do over charz;❹
    if upcase(charz)='#N/A' then
                                charz='.';
    end;
  n_rows=input(n_rows_char,8.);
  n_cols=input(n_cols_char,8.);
    fu_row=input(fu_row_char,8.);
    fu_col=input(fu_col_char,8.);
    sh_order=input(sh_order_char,8.)-1;❺
 run;
```

❶ The more obvious part of this clean-up action is the conversion of certain alphanumerical variables that contain only digits (or missing values) into numerical variables.

❷ But, hang on a second… Where in the previous section did we write Excel 4 macro code to return these chart types? Or the number of series on charts? Well, the crux of the matter is that even with a single setting of its parameter, the `get.document` function returns different kinds of information depending on whether a sheet is a worksheet or a chart.

For example, a `get.document(9,"Sheet")` will return the number of the first used row if "Sheet" is a worksheet or a macro sheet, but it will return the type of the main chart if "Sheet" is actually a chart. Similarly, parameter settings 10, 11, and 12 also return chart related attributes for charts, and worksheet related information for worksheets. Luckily, the X4ML help-file details this quixotic behaviour. It must have seemed like a good idea at the time.

❸ But no matter: charts have a sheet type of 2, and with the help-file at hand it is easy to build a bit of data step logic that will disentangle the chart- from the worksheet-metadata.

❹ A final check on some of the numerical fields. That is, before converting them to numerical variables, we replace any remaining '#N/A' values by a SAS missing value period.

❺ Remember that we moved the temporary macro sheet into the first position on the workbook? Correcting the sheet order for the presence of our macro sheet therewith becomes a triviality. We need but diminish the returned position numbers by one.

We are nearly there. What remains to be done is to append our fresh set of sheet parameters to the `sashelp.xlsheets` dictionary table. We take care to remove any extant rows pertaining to the current workbook, so that if the workbook was already registered in `sashelp.xlsheets`, its entries now become refreshed:

```
data sashelp.xlsheets;
  set sashelp.xlsheets(where=(
   (upcase(wb_name) ne upcase("&wb_name"))
   or
   (upcase(wb_path) ne upcase("&wb_path"))
   ));
run;

proc append
  base=sashelp.xlsheets
  data=_sheet_parameters
  force
  ;
run;
```

Then we mop up: closing the workbook without saving it (`file.close(false)`) gets rid of the temporary macro sheet. Also kill the Excel session:

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

Delete intermediary data sets and close down the remaining output streams:

```
proc datasets nolist lib=work;
  delete
    _sheetnames_before
    _sheetnames_after
    _sheet_parameters  / memtype=data;
quit;
filename xlmacro clear;
filename sas2xl clear;
```

8 One Step Beyond: Extracting Excel Column Metadata

In the course of the preceding sections, we have developed and successfully effectuated a scheme for obtaining sheet related metadata from an essentially unknown Excel workbook. That is, without physically opening and inspecting the workbook, we now know the names of the various sheets contained within, the sheet types, the order in which they appear, chart types, the number of rows and columns used, and so forth.

A very similar technique can be developed to go a step further, and gather information about the content of specific columns on the workbook's data sheets. Most of the preparatory steps are identical to what we did before, so let's focus on the different parts instead.

As before, we store some essential constants in the form of SAS global macro variables:

```
%let wb_path=c:\tu15;
%let wb_name=Sales Demo;
%let sh_name=Net Data;
%let n_rows=435;
%let col_num=1;
```

What we're saying here is that we will investigate the content of the 1st column on the 'Net Data' worksheet of our 'Sales Demo.xls' workbook. The number of rows, `n_rows`, is of course known on the `SASHELP.XLSHEETS` look-up table, and we could retrieve it there, but for reasons of brevity we just plug it into a global macro variable here.

The initial steps are as before:

- Start up Excel.
- Open the 'Sales Demo.xls' workbook.
- Add a temporary Excel 4 macro sheet.
- Find out the name of the new macro sheet by using the 'topics'-triplet, and comparing the resulting sheet names with those registered in `SASHELP.XLSHEETS` Store in a SAS macro variable `temp_macro_sheet`.
- Define an `XLMACRO` triplet-style fileref pointing to the first column of the temporary macro sheet.

And then things become interesting again. What we'll do in the next data step is this:

- We copy the entire data column that we want to investigate into column `B` of the macro sheet.
- And not only that, we also paste the same column repeatedly into columns `C` through `J`.
- Then we write and run an Excel 4 macro that will replace the values in columns C through J with the results of various cell-diagnostic functions.

More details as to how this works after the code:

```
data _null_;
  file sas2xl;
  length maccmd ddecmd $ 200;
  ddecmd='[workbook.activate("'||
         "&sh_name"||'")]';❶
  put ddecmd;
  ddecmd='[copy("r1c'||"&col_num"||
         ':r'||"&n_rows"||
         'c'||"&col_num"||'")]';❷
  put ddecmd;
  ddecmd='[workbook.activate("'||
         "&temp_macro_sheet"||'")]';❸
  put ddecmd;
  put '[paste("r1c2")]';
  put '[paste("r1c3")]';
  put '[paste("r1c4")]';
  put '[paste("r1c5")]';
  put '[paste("r1c6")]';
  put '[paste("r1c7")]';
  put '[paste("r1c8")]';
  put '[paste("r1c9")]';
  put '[paste("r1c10")]';❹
  file xlmacro;
  maccmd='=for.cell("CurrentCell",'||
         '!$c$1:$c$'||"&n_rows"||
         ',false)';❺
  put maccmd;
  put '=formula(get.cell(7,CurrentCell),
      CurrentCell)';❻
  put '=next()';❼
  maccmd='=for.cell("CurrentCell",'||
         '!$d$1:$d$'||"&n_rows"||
         ',false)';
  put maccmd;
  put '=formula(isnumber(CurrentCell),
      CurrentCell)';
  put '=next()';
  maccmd='=for.cell("CurrentCell",'||
         '!$e$1:$e$'||"&n_rows"||
         ',false)';
  put maccmd;
  put '=formula(istext(CurrentCell),
      CurrentCell)';
  put '=next()';
  maccmd='=for.cell("CurrentCell",'||
         '!$f$1:$f$'||"&n_rows"||
         ',false)';
  put maccmd;
  put '=formula(isblank(CurrentCell),
      CurrentCell)';
  put '=next()';
  maccmd='=for.cell("CurrentCell",'||
         '!$g$1:$g$'||"&n_rows"||
         ',false)';
  put maccmd;
  put '=formula(isnontext(CurrentCell),
      CurrentCell)';
  put '=next()';
  maccmd='=for.cell("CurrentCell",'||
         '!$h$1:$h$'||"&n_rows"||
         ',false)';
  put maccmd;
  put '=formula(isna(CurrentCell),
      CurrentCell)';
  put '=next()';
  maccmd='=for.cell("CurrentCell",'||
```

```
        '!$i$1:$i$'||"&n_rows"||
        ',false)';
  put maccmd;
  put '=formula(get.cell(18,CurrentCell),
      CurrentCell)';
  put '=next()';
  maccmd='=for.cell("CurrentCell",'||
        '!$j$1:$j$'||"&n_rows"||
        ',false)';
  put maccmd;
  put '=formula(get.cell(19,CurrentCell),
      CurrentCell)';
  put '=next()';
  put '=halt(true)';
  put '!dde_flush';
  file sas2xl;
  ddecmd='[run("'||"&temp_macro_sheet"||
        '!r1c1")]';❽
  put ddecmd;
run;
```

❶ Writing to the `SAS2XL` system-doublet, we activate the worksheet of interest, as specified by the `sh_name` SAS macro variable.

❷ From the column we wish to examine — SAS macro variable `col_num` — we copy as many cells as are potentially in use. Remember that the SAS macro variable `n_rows` contains the number of rows used on this particular worksheet, so this is an upper limit for the length of every single column on the current worksheet.

❸ We make the temporary macro sheet active again.

❹ Into the columns 2 through 10 of the macro sheet — or `B` through `J` if you like — we paste the copied cells. We will leave column `B` alone, it will retain the actual copied values so we can read these into a SAS data set later on.

❺ For each of the other columns, we write a bit of Excel 4 macro code into the `XLMACRO` range. Each of these bits comprises a do-loop to cycle through all the cells in the column, and replace the contents with the result of some function. This particular line *e.g.* initiates the do-loop (or in X4ML speak: a `for.cell` loop) over all the cells that we pasted into column `C`. As the loop tiptoes down the column, it temporarily sticks the label 'CurrentCell' onto each cell it encounters, enabling us to reference the cell in a function call. The 'false' flag indicates not to skip empty cells. If there are any, we'd like to know that.

❻ The body of each do-loop consists of a single `formula` function, which enters the result of its first argument into the cell referenced by its second argument. In this case, that means the contents of the CurrentCell are being replaced by the action of `get.cell` with numerical parameter 7 on the CurrentCell itself. And what does `get.cell(7,CurrentCell)` do? It returns the Excel format that is used to display the cell contents. So this do-loop here reveals the formatting for the entire column we are diagnosing!

❼ This marks the end of the first `for.cell` loop. The following loops work in precisely the same way and reveal some more cell-characteristics:

- `isnumber(CurrentCell)` returns `TRUE` if Excel deems the contents of the CurrentCell to be numerical.
- `istext(CurrentCell)` returns `TRUE` if Excel deems the contents of the CurrentCell to be alphanumerical.
- `isblank(CurrentCell)` returns `TRUE` if Excel deems the CurrentCell to be empty.
- `isnontext(CurrentCell)` returns `TRUE` if Excel deems the contents of CurrentCell not to be alpha-numerical. This one will also return `TRUE` if a cell is empty.
- `isna(CurrentCell)` returns `TRUE` if the CurrentCell contains a `#N/A` (value not available) error value.
- `get.cell(18,CurrentCell)` returns the name of the font used to display the CurrentCell.
- `get.cell(19,CurrentCell)` returns the font-size in points used to display the CurrentCell.

The possibilities are endless. We could apply any kind of function of interest to the cell values.

❽ After finalizing the macro with a `halt(true)` and flushing the DDE buffer, we run the thing. The actual Excel 4 macro produced by our data step is this:

```
=FOR.CELL("CurrentCell",!$C$1:$C$435,FALSE)
=FORMULA(GET.CELL(7,CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$D$1:$D$435,FALSE)
=FORMULA(ISNUMBER(CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$E$1:$E$435,FALSE)
=FORMULA(ISTEXT(CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$F$1:$F$435,FALSE)
=FORMULA(ISBLANK(CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$G$1:$G$435,FALSE)
=FORMULA(ISNONTEXT(CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$H$1:$H$435,FALSE)
=FORMULA(ISNA(CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$I$1:$I$435,FALSE)
=FORMULA(GET.CELL(18,CurrentCell),CurrentCell)
=NEXT()
=FOR.CELL("CurrentCell",!$J$1:$J$435,FALSE)
=FORMULA(GET.CELL(19,CurrentCell),CurrentCell)
=NEXT()
=HALT(TRUE)
```

When the Excel macro has finished running, the temporary macro sheet is filled with goodies, and we merely need to reap them:

```
filename colparms dde
  "excel|&temp_macro_sheet!
  r1c2:r&n_rows.c10" lrecl=2500;

data _column_diagnostics;
  length
    col_value     $ 2000
    col_format    $  100
    col_isnumber
    col_istext
    col_isblank
    col_isnontext
    col_isna      $ 5
    col_font      $ 50
    col_fontsize  $ 5
    ;
  label
    col_value     = 'Value'
    col_format    = 'Format'
    col_isnumber  = 'Is Number'
    col_istext    = 'Is Text'
```

```
    col_isblank   = 'Is Blank'
    col_isnontext = 'Is Not Text'
    col_isna      = 'Is Not Available'
    col_font      = 'Font'
    col_fontsize  = 'Fontsize'
    ;
  infile colparms notab dlm='09'x dsd
    missover;
  input
    col_value
    col_format
    col_isnumber
    col_istext
    col_isblank
    col_isnontext
    col_isna
    col_font
    col_fontsize
    ;
run;
```

Then, as before, close the workbook without saving, quit Excel, and clean up temporary data sets.

We now have a SAS data set with a variable `col_value` that contains an alphanumerical copy of the contents of the column we wish to study, plus a whole slew of diagnostic variables that can help *e.g.* in determining whether the column is text, numbers, date, time, whether it has a top row label… To do all that, some serious data step logic is in order of course. As the matter is out of scope for the present paper, it is left to the reader as an exercise.

9 Wrapping It Up

Like all DDE code, the two applications we have discussed here are easily wrapped inside a SAS macro for ease of use. As a matter of fact, the download package on www.vyverman.com contains two extra macro source codes. One to register a workbook into `SASHELP.XLSHEETS`, and another one to create column-diagnostics data sets. These macros may be called as follows:

```
%register_workbook(
                   wb_path=c:\tu15,
                   wb_name=Sales Demo
                  );

%column_diagnostix(
                   wb_path=c:\tu15,
                   wb_name=Sales Demo,
                   sh_name=Net Data,
                   n_rows=435,
                   col_num=1,
                   ds_lib=work,
                   ds_name=net_data_col_1
                  );
```

The intrepid reader may of course modify these sources, choose different `get.document` and `get.cell` parameters, depending on personal judgement of what is useful and what is not. Both functions offer a lot more possibilities than what is shown in the present paper. Experimentation is encouraged!

10 In Closing

A final idea? Looking at the kind of sheet metadata that we've managed to trap into our `SASHELP.XLSHEETS` dictionary table, it is not altogether inconceivable to postulate the feasibility of building a second dictionary table which would focus on the attributes of worksheet columns rather than on the sheets themselves.

Imagine a data set, let's call it `SASHELP.XLCOLUMNS`, that would list a number of useful attributes for each column of every data-sheet registered in `SASHELP.XLSHEETS`. Attributes like: some solid indicator of the data type. The maximum length if the data type is text. Minimum and maximum values for numerical columns. A sparsity index. A first row label flag… Sounds decidedly cool. Any takers?

References

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.

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

Viergever, W. W. & Vyverman, K. "Fancy MS Word Reports Made Easy: Harnessing the Power of Dynamic Data Exchange — Against All ODS, Part II." *Proceedings of the 28th SAS Users Group International Conference*, 2003.

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

Vyverman, K. "Using Dynamic Data Exchange to Export Your SAS Data to MS Excel — Against All ODS, Part I." *Proceedings of the 26th SAS Users Group International Conference*, 2001.

Vyverman, K. "Creating Custom Excel Workbooks from Base SAS with Dynamic Data Exchange: a Complete Walkthrough." *Proceedings of the 27th SAS Users Group International Conference*, 2002.

Acknowledgments

Thanks go to Ed Heaton and Ian Whitlock, SESUG10 Tutorials Section Chairs, for making this contribution possible. And of course to my dear friends Pico Dolt and Kilo Volt, without whom this piece of work would have remained unfinished and gathering dust in some drawer.

Trademarks

Contacting the Author

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

`sesug10paper@vyverman.com`