Paper 092-30

%sas2xl: A Flexible SAS® Macro That Uses Tagsets to Produce Complex, Multi-Tab Excel Spreadsheets with Custom Formatting

David Brown, Amylin Pharmaceuticals, Inc., San Diego, CA

ABSTRACT

Converting SAS data to Excel format spreadsheets is not necessarily hard. But creating highly user-friendly spreadsheets with multiple tabs and complex formatting options is much more difficult. This paper explores how a custom SAS macro, %sas2xl, can be used to generate better spreadsheets more easily than other methods available. It harnesses the power of ODS tagsets to generate custom XML code that Excel reads in as sheet formats. %sas2xl is designed to be user-friendly and flexible, and it can produce spreadsheets with options such as multiple tabs, frozen column headers (first row doesn't scroll off the screen), print headers and footers, and traffic highlighting. This paper concentrates on the technical aspects of this solution, which works on all platforms and Versions 8 or 9 of SAS. The intended audience is intermediate to advanced users.

INTRODUCTION

It's late on a Friday, and management has just requested last months sales data ahead of schedule, by Monday morning, in fact. Of course, they are not programmers and don't use SAS, where the data reside. You need to get it into a format that's easy to understand, and you need it in an application everyone already knows how to use. A Microsoft Excel® workbook with multiple tabbed spreadsheets separating the data into easy to digest units is a good start. But the data are complicated, and you want to make it easy for everyone to understand. You'll need to make sure you freeze the column names at the top so the user always knows what he/she is looking at, and you'll need to make sure everything is formatted clearly. Maybe you should also highlight problem data, so they are not missed. That would certainly make things easy for everyone.

This could be a long weekend since you have to be done by Monday morning. But it won't be. The powerful macro %sas2xl, will help you easily produce multi-tabbed Excel spreadsheet workbooks complete with user-friendly features like traffic-light type highlights, frozen column headers and even nicely labeled printouts, if the user finds it easier to review the data on paper.

CONCEPT

By leveraging SAS's powerful ODS capabilities to produce customized HTML output that Excel natively understands, the challenging task of creating elegant Excel spreadsheets based on SAS data is made much easier. Packaged into a powerful macro, the task of creating these spreadsheets becomes as easy as a PROC REPORT call.

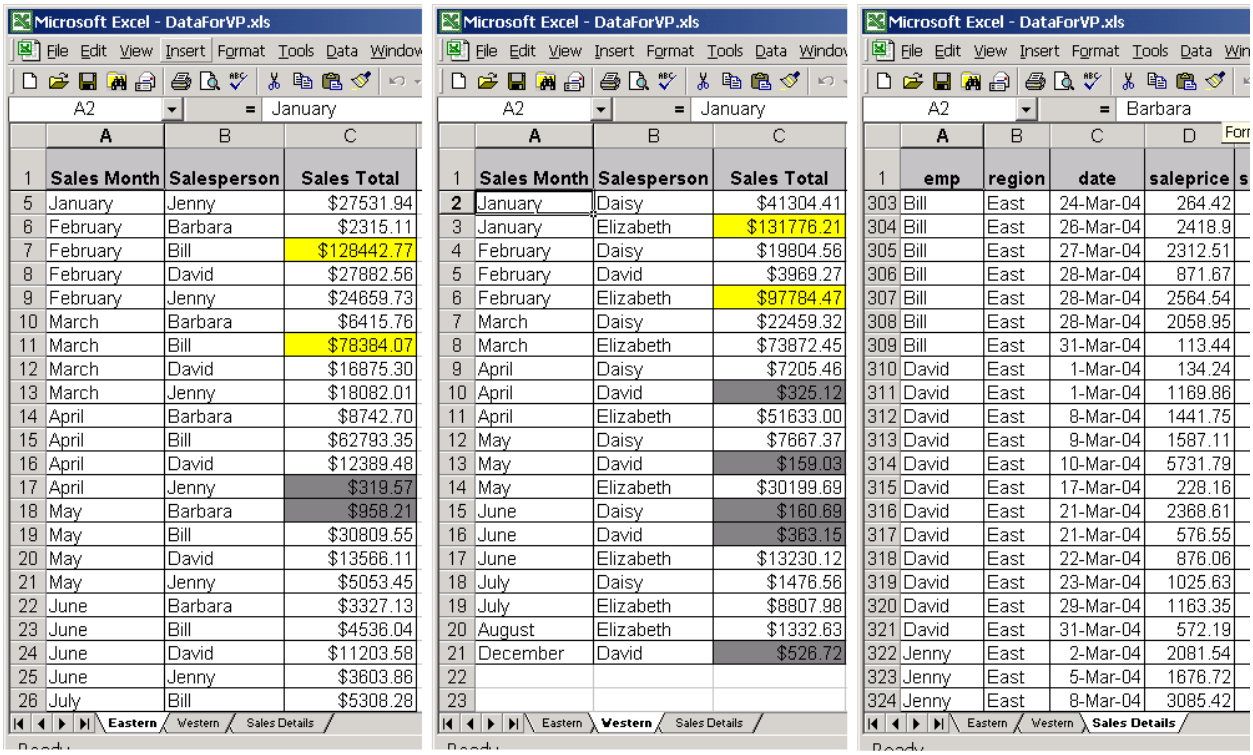
Here is a sample call to %sas2xl:

```
%sas2xl(path=C:\,file=DataForVP,
        def= @sheet=Eastern @dset=MonthlySales(where=(region="East"))
             @xlhead=Monthly Sales Data for Eastern Region
             @fitwide=1
             @cols=!month   / "Sales Month"
                   !emp     / "Salesperson"
                   !sales   / "Sales Total" w=15 f=8.2 bg=salesf. xlf=$00.00

             @sheet = Western @dset=MonthlySales (where=(region="West"))
             @xlhead=Monthly Sales Data for Western Region
             @fitwide=1
             @cols=!month   / "Sales Month"
                   !emp     / "Salesperson"
                   !sales   / "Sales Total" w=15 f=8.2 bg=salesf. xlf=$00.00

             @sheet=Sales Details @dset=sales
       )
```

This produced the multi-tabbed workbook DataForVP.xls (see Figure 1). The first tab is for the Eastern Region Monthly Sales, the second tab is for Western Region Monthly sales, and the third tab is the detailed sales data. It highlights in color low or high monthly sales, and the top row is frozen, so as you scroll down, the column labels never disappear. The second row displayed in tab one below is actually row 5; row 1 did not scroll off the screen. The first 2 sheets are produced by explicitly describing the columns to include. The last sheet includes all variables in the sales data set in the resulting spreadsheet. When printed, the first 2 sheets will be reduced in size by Excel (if required) to always fit one page wide.



Microsoft Excel - DataForVP.xls

File Edit View Insert Format Tools Data Window

A2 = January

	A	B	C
1	Sales Month	Salesperson	Sales Total
5	January	Jenny	$27531.94
6	February	Barbara	$2315.11
7	February	Bill	$128442.77
8	February	David	$27882.56
9	February	Jenny	$24659.73
10	March	Barbara	$6415.76
11	March	Bill	$78384.07
12	March	David	$16875.30
13	March	Jenny	$18082.01
14	April	Barbara	$8742.70
15	April	Bill	$62793.35
16	April	David	$12389.48
17	April	Jenny	$319.57
18	May	Barbara	$958.21
19	May	Bill	$30809.55
20	May	David	$13566.11
21	May	Jenny	$5053.45
22	June	Barbara	$3327.13
23	June	Bill	$4536.04
24	June	David	$11203.58
25	June	Jenny	$3603.86
26	July	Bill	$5308.28

Eastern Western Sales Details

Microsoft Excel - DataForVP.xls

File Edit View Insert Format Tools Data Window

A2 = January

	A	B	C
1	Sales Month	Salesperson	Sales Total
2	January	Daisy	$41304.41
3	January	Elizabeth	$131776.21
4	February	Daisy	$19804.56
5	February	David	$3969.27
6	February	Elizabeth	$97784.47
7	March	Daisy	$22459.32
8	March	Elizabeth	$73872.45
9	April	Daisy	$7205.46
10	April	David	$325.12
11	April	Elizabeth	$51633.00
12	May	Daisy	$7667.37
13	May	David	$159.03
14	May	Elizabeth	$30199.69
15	June	Daisy	$160.69
16	June	David	$363.15
17	June	Elizabeth	$13230.12
18	July	Daisy	$1476.56
19	July	Elizabeth	$8807.98
20	August	Elizabeth	$1332.63
21	December	David	$526.72
22			
23			

Eastern Western Sales Details

Microsoft Excel - DataForVP.xls

File Edit View Insert Format Tools Data Win

A2 = Barbara

	A	B	C	D
1	emp	region	date	saleprice
303	Bill	East	24-Mar-04	264.42
304	Bill	East	26-Mar-04	2418.9
305	Bill	East	27-Mar-04	2312.51
306	Bill	East	28-Mar-04	871.67
307	Bill	East	28-Mar-04	2564.54
308	Bill	East	28-Mar-04	2058.95
309	Bill	East	31-Mar-04	113.44
310	David	East	1-Mar-04	134.24
311	David	East	1-Mar-04	1169.86
312	David	East	8-Mar-04	1441.75
313	David	East	9-Mar-04	1587.11
314	David	East	10-Mar-04	5731.79
315	David	East	17-Mar-04	228.16
316	David	East	21-Mar-04	2368.61
317	David	East	21-Mar-04	576.55
318	David	East	22-Mar-04	876.06
319	David	East	23-Mar-04	1025.63
320	David	East	29-Mar-04	1163.35
321	David	East	31-Mar-04	572.19
322	Jenny	East	2-Mar-04	2081.54
323	Jenny	East	5-Mar-04	1676.72
324	Jenny	East	8-Mar-04	3085.42

Eastern Western Sales Details

Figure 1: DataForVP.xls

HOW IT WORKS

%sas2xl exploits Excel's ability to read and write spreadsheets in HTML format. Almost all Excel functions and formatting are preserved when you save a file in HTML format. By using tagsets to customize SAS's HTML output, SAS can write out files that Excel understands perfectly, with detailed formatting and Excel options built into the file.

One of the benefits of using ODS tagsets is that it should work on platforms that do not support Microsoft Excel, such as Linux. Files that can be read by Excel are produced directly from SAS. Although this method hasn't been tested on those platforms, %sas2xl doesn't require Microsoft products to produce the spreadsheets (though it does use Excel to convert the result file to a native Excel format file). The techniques described in this paper have been tested under SAS 8.2 and Excel 2000.

In an excellent paper, Parker (2003) described the method of using tagsets to produce Excel spreadsheets, including details on custom formatting. %sas2xl builds on these techniques by creating a custom tagset for each spreadsheet based on the parameters specified in the macro call. In fact, a different tagset is created by the macro for each worksheet in a multi-tab workbook.

The purpose of this paper is not to go into extensive detail on these techniques again, but rather to show how they can be used to build a robust macro-driven utility that allows quick and easy creation of Excel files from SAS data. However, a review of the concepts involved should be helpful. When an Excel file is saved in HTML format, Excel writes out various formatting and options using an XML schema that most web browsers can read. Excel can also read in these HTML formatted files. Giving these files a .xls extension will cause the files to be opened automatically by Excel (on the Windows platform). Because custom tagsets allow SAS to customize HTML output,%sas2xl can write out a file that uses Excel's XML schema to produce an HTML file that Excel will read in as a complete spreadsheet with complex formatting and options built right in.

Determining the specific XML required to turn on a given formatting option in Excel is fairly easy. Parker describes many options in some detail, but it may be easiest to create an Excel file with the formatting you need and save it as an HTML file. If you open the file in a text editor, you can review the XML data for those formatting options. Below is an excerpt from the first sheet in DataForVP.xls that shows some of the XML data used by Excel.

```
<html xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns:x="urn:schemas-microsoft-com:office:excel">
<head>
<style>
<!--
@page
  {mso-header-data:"&L&CSales Data for Eastern Region&RPage &P of &N";
  mso-footer-data:"&LC:\\&F (Current Sheet: Eastern), Generated from SAS on 02JAN05&C&R";
     margin:1.0in .75in 1.0in .75in;
     mso-header-margin:.5in;
     mso-footer-margin:.5in;
     mso-page-orientation:landscape;}
.sas2xl2 {mso-width-source:userset;width:106px;mso-number-format:"$00.00"}
.excelth{font-weight:700;background:silver}
-->
</style>
<meta http-equiv="Content-type" content="text/html;  charset=windows-1252">
<title>SAS Output</title>
<!--[if gte mso 9]><xml>
<x:ExcelWorkbook>
  <x:ExcelWorksheets>
   <x:ExcelWorksheet>
    <x:WorksheetOptions>
     <x:DisplayPageBreak/>
       <x:FitToPage/>
         <x:Print>
          <x:FitWidth>1</x:FitWidth>
          <x:FitHeight>1000</x:FitHeight>
          <x:ValidPrinterInfo/>
          <x:VerticalResolution>0</x:VerticalResolution>
         </x:Print>
     <x:Selected/>
          <x:FreezePanes/>
          <x:FrozenNoSplit/>
          <x:SplitHorizontal>1</x:SplitHorizontal>
          <x:TopRowBottomPane>1</x:TopRowBottomPane>
          <x:ActivePane>2</x:ActivePane>
               <x:Panes>
                <x:Pane>
                 <x:Number>3</x:Number>
                </x:Pane>
                <x:Pane>
                 <x:Number>2</x:Number>
                </x:Pane>
               </x:Panes>
     <x:DisplayZeros/>
     <x:DisplayHeadings/>
     <x:DisplayOutline/>
    </x:WorksheetOptions>
   </x:ExcelWorksheet>
  </x:ExcelWorksheets>
</xml><![endif]-->
</head>
```

Among the options defined in this XML data are page margins and orientation, the frozen header pane information, and print options such as fit to one page wide. It should also be noted that %sas2xl supports a limited number of XML elements, so this is shorter than an HTML file produced directly from Excel. The XML schema in HTML files produced by Excel can be a bit complicated, so it is also a good idea to refer to Microsoft's Microsoft Office HTML and XML Reference, which details the use and syntax of each XML element.

STEPS IN THE PROCESS

To produce an Excel file, %sas2xl goes through the following steps:

1. Parse the def= parameter, which defines the spreadsheet workbook parameters
2. For each worksheet, generate a custom tagset based on the requested options
3. For each worksheet, use the custom tagset and a PROC REPORT to generate the HTML files
4. For multi-tab workbooks, generate the files that tie each individual sheet into a single Excel workbook
5. Optionally, convert HTML output to a native Excel format file and delete the HTML output.

PARSE THE DEF= PARAMETER

The goal of this step is to convert the spreadsheet workbook definition from the macro call's def= parameter and save it in a series of macro variables that can be used later to construct a custom tagset and a custom PROC REPORT. The def= parameter allows you to describe exactly how the spreadsheet workbook will be produced. The '@' symbol is used as a delimiter for parameters that define the structure of each sheet. Some of the available parameters are:

```
@sheet=<name for worksheet tab>
@dset=<data set name for this tab>
@headrows=<number of header rows to freeze in Excel>
@vertheadrows=<number of columns to freeze at left in Excel>
@xlhead=<text to show up on Excel page header (shows up on printed output only)>
@footer=<SAS footnote statements for this sheet (shows up at the end of the data)>
@by=<by vars to sort the data for this sheet>
@cols=<column definitions for this sheet>
```

A minimal set of parameters for defining a spreadsheet tab in a workbook are @sheet= and @dset=. @sheet serves two purposes: it tells %sas2xl that a new tab is being defined and it presents the name of the sheet to display on the tab in the resulting Excel workbook. @dset is the data set to be converted to Excel for this tab. If these are the only parameters defined, all variables from the data set will be included in the Excel file.

You may also use the @cols= parameter, which allows you to define the columns in the resulting spreadsheet. The @cols= parameter uses an exclamation point (!) as the delimiter for each column definition. To make it as user-friendly as possible, I chose to model this parameter on the define statements from PROC REPORT. After each variable name, optional parameters, including column header, formatting, column width, and alignment, can be specified after a slash. You can also define a traffic lighting format. Some of the supported column options are:

```
"Title Text" (must be in quotes)
order=<type> (this can be "noprint" or a proc report order type [internal, data, etc.])
width = <width> (for the column widths)
format=<format> (SAS format)
bg=<format> (ODS traffic lighting format)
xltxt (this forces Excel to treat the values as text)
center (center the output)
left (left align)
right (right align)
group (group this column with the next column)
xlformat=<format string> (This is to define Excel cell formats)
```

The actual work of parsing the parameters is done by complicated macro code. Due to space limitations, only important excerpts can be presented here, but the concepts should become clear. The idea is to cycle through the @def= text and identify the parameters by splitting on the delimiters. First, we need one counter to keep track of which parameter is being processed, and another counter to track which sheet is being processed.

```
%let counter = 1;
%let sheet=0;
```

The @ sign indicates a new parameter, and %sas2xl puts the parameter and its value -- everything after one @ sign and before the next @ sign -- into a temporary macro variable with:

```
%let word = %scan(&def,&counter,@));
```

Next, %sas2xl separates the parameter name from the parameter values. The parameter name (for example 'sheet') is stored in a temporary macro variable named part1, and its value (for example 'Eastern') is stored in a temporary macro variable named part2. The position of the equal sign ('=') is used to determine where to split the text in the substring function.

```
%let part1 = %substr(&word,1,%eval(%index(&word,=) - 1));
%let part2 = %substr(&word,%eval(%index(&word,=) + 1));
```

Because new sheets are defined with the @sheet= parameter, %sas2xl will trap for this and update the sheet counter macro variable:

```
%if %upcase(&part1) = SHEET %then
  %let sheet = %eval(&sheet + 1);
```

The macro is now ready to hold the value of the parameter in a macro variable for storage until it is needed later when creating the tagset or PROC REPORT.

```
%let &part1&sheet = &part2;
```

In this example, the macro variable will become sheet1 because &part1 will resolve to 'sheet' and &sheet will resolve to '1'. The value is pulled from &part2, which resolves to 'Eastern'.

Similar logic is used to parse out column definitions when an @col= is encountered. However, the delimiter is an exclamation point (!) for each column/variable. The variable name is separated from the various attributes, such as column header and column formatting, by a slash (/). Attributes such as width have values after equal signs (possibly separated by a space), while others are simply single-word attributes like 'left'. But the processing for parsing those elements is similar: break the @def=text down into parts and store those parts in macro variables so they can be used later in building the tagset and the PROC REPORT.

Trapping for column headers is more complex, and involves looking for a quotation mark and then adding each character to the title macro variable until another quotation mark is encountered. Unmatched quotation marks must be quoted. Assuming a single character had been split off and stored in the macro variable current_letter, this would be the conditional statement to look for a quotation mark:

```
%if &current_letter = %nrstr(%") %then...
```

Because SAS allows macros to be redefined by resolving the current value of the macro variable on the right side of the equal sign, letters can be added to the title with a %let statement:

```
%let column_header = &column_header&curletter;
```

Put this inside a %do loop that goes character by character, each time looking for the ending quotation mark, and the column headers can be stored for later use.

Put all of these parsing techniques inside a `%do %until (&end=1);` loop and the macro will continue to process until the entire @def= parameter is completely parsed and stored in macro variables. This allow for a great amount of flexibility when using the macro. The definition can be as long or as short as needed by simply included or omitting parameters and options.

It is important to take time when developing the macro to determine which Excel features are important and need to be user-defined within the macro call, and which ones are not. For example, %sas2xl does not allow you to change the page margins, but adding the ability to specify those during the macro call would not be difficult. In addition to the above code, %sas2xl performs a considerable amount of error checking to trap for mistakes in the macro call. Also, careful thought went into macro quoting to give the user flexibility in the macro call.

CREATE THE CUSTOM TAGSET

Once the @def= parameter is parsed, the spreadsheet workbook definition is now stored in a series of macro variables which are used by %sas2xl to produce a unique tagset for each worksheet within a workbook. A new tagset is conditionally produced by the macro every time it is called. The tagset excelhtml is created by inheriting events from the htmlcss tagset and redefining only the events that need to be modified with Excel-specific HTML or XML codes.

```
define tagset tagsets.excelhtml;
  parent=tagsets.htmlcss;
```

For %sas2xl, I created an event called excelstyle that writes out HTML-style class statements which are included in the HTML header. These statements define cell attributes such as background color, Excel formats, and column

widths. Here is an example of a class definition that defines the column width to be 106 pixels wide and uses a numeric cell format appropriate for U.S. dollars.

```
.sas2xl3 {mso-width-source:userset;width:106px;mso-number-format:"$00.00"}
```

In %sas2xl, a class statement is defined for each column, so there is a %do loop that the cycles through each variable that was defined in @cols= parameter of the def= macro parameter. So, %sas2xl builds the needed tagset code based on the macro call:

```
define event excelstyle;

  %do i = 1 %to &varcount;

    /* this conversion approximates widths in Excel, and must be done if a width was given */
    %if &&var&i._width ne %str( ) %then
    %let var&i._width=%sysevalf(&&var&i._width * 7.1,INTEGER);

    /* if both column width and an Excel format are specified, write out this style */
    %if &&var&i._width ne %str( ) and var&i._xlformat ne %str( ) %then
      %do;
        put ".sas2xl&i {mso-width-source:userset;width:&&var&i._width.px;mso-number-format:
            ""&&var&i._xlformat""}";
      %end;
    %else %if...<other format combinations>
  %end;

  /* this style is for column headers */
  put ".excelth {font-weight:700; background:silver}" NL;
end;
```

The data event also requires customization based on the spreadsheet definition because the style attributes for the cell are specified in the “<td>” HTML tag. A complete <td> tag might look like this:

```
<td class="sas2xl3" align="right" style=" background-color: #FFFFFF;">
```

Excel will then pick up the width and any formatting specified in the sas2xl3 style statement contained in the HTML header. In the following code, remember that the PROC REPORT will define “htmlclass” if it was specified in a style= statement, pass “section” to indicate if it is writing column headers or data, and define “just” based on the justification of the column. The macro %do loop dynamically builds the needed PUTQ statements to conditionally check for and write out the appropriate “class=” statement inside the <td> HTML tag. So this event writes out the required HTML <td> tags, then the data, and finishes by writing the closing </td> tag.

```
define event data;
  start:
    put "<td";

    /* create if cmp statements for the various styles created above in excelstyle event*/
    %do i = 1 %to &varcount;
      putq " class=""sas2xl&i"""/if cmp("sas2xl&i",htmlclass);
    %end;

    putq " class=""excelth"""/if cmp("head", section);
    putq " align=""center""" /if cmp("c",just);
    putq " align=""right""" /if cmp("r",just);
    putq " align=""left""" /if cmp("l",just);

    trigger style_inline; /* this taps into ODS traffic highlighting */

    put ">";

    put VALUE;

  finish:
    put "</td>";
end;
```

The doc_head event is the most complicated because it has the most Excel-specific syntax to write out and because it must process single-tab worksheets differently from workbooks with multiple tabs. This is because single-tab files only require one HTML file, but multi-tab workbooks require a separate file for each tab.

```
define event doc_head;
  start:
    put "<head>" NL;
    put VALUE NL;
    put "<style>" NL ;
    put "<!--" NL;
    put "@page" NL;

    /* print header text (Quote ampersands that are part of Excel's header syntax */
    put "  {mso-header-data:""%nrstr(&L)%nrstr(&C)&xlhead %nrstr(&R)Page%nrstr( &P) of
        %nrstr( &N)"";" NL;
    put "   margin:1.0in .75in 1.0in .75in;" NL;
    put "   mso-header-margin:.5in;" NL;
    put "   mso-footer-margin:.5in;" NL;
    put "   mso-page-orientation:landscape;}" NL;

    /* write out the excel styles, using the event defined above */
    trigger excelstyle;

    put "-->" NL;
    put "</style>" NL;

  finish:
    put "<!--[if gte mso 9]><xml>" NL;
    put "<x:ExcelWorkbook>" NL;
    put "  <x:ExcelWorksheets>" NL;
    put "   <x:ExcelWorksheet>" NL;

    /* if name is present, this is for a single sheet workbook, and it won't be combined
     later, so add name*/
    %if %superq(name) ne %then
      %do;
        put "    <x:Name>&name</x:Name>" NL;
      %end;

    put "    <x:WorksheetOptions>" NL;
    put "     <x:DisplayPageBreak/>" NL;
    put "     <x:DisplayZeros/>" NL;
    put "     <x:DisplayHeadings/>" NL;
    put "     <x:DisplayOutline/>" NL;
    put "    </x:WorksheetOptions>" NL;
    put "   </x:ExcelWorksheet>" NL;
    put "  </x:ExcelWorksheets>" NL;

    /* if name is present, this is for a single sheet workbook, and it won't be combined
       later, so display gridlines here. */
    %if %superq(name) ne %then
      %do;
        put " <x:DoNotHideVerticalScrollBar/>" NL;
        put " <x:DoNotHideWorkbookTabs/>" NL;
        put " <x:DisplayFormulas/>" NL;
        put " <x:DisplayGridlines/>" NL;
      %end;
    put "</x:ExcelName>" NL;
  %end;
    put "</xml><![endif]-->" NL;
    put "</head>" NL;
end;
```

The complete tagset code is considerably more complicated, but it is built from these principles. The goal is for the tagset to write out the HTML code for this specific spreadsheet—based on the def= workbook definition—that Excel will read in as a formatted spreadsheet.

GENERATE THE PROC REPORT

Once the tagsets are ready, they can be used along with a simple PROC REPORT to create the spreadsheet. Remember that this is still part of a complex macro and the power this allows is leveraged to build the PROC REPORT variable by variable, if needed. First, use an ODS MARKUP statement and specify the custom tagset:

```
ods markup File="&path.&file..xls" tagset=tagsets.excelhtml;
```

Within the PROC REPORT, the COLUMN statement will be built by cycling through the macro variables created when parsing the @def= parameter. The following code fragment assumes that the variable names are stored in a series of macro variables called var1, var2, var3, etc., and that the number of variables is stored in varcount.

```
/* column statement */
column

  %do i = 1 %to &varcount;
    &&var&i
  %end; ; /* the extra semicolon ends the column statement */
```

Building the define statement is more complicated because the column attributes were stored in separate macro variables, but the process is conceptually similar. This code fragment would create define statements for each variable. Note that %sas2xl determines if it is necessary to write out a style= statement that will trigger HTML class= statements based on whether the macro call contained either an Excel format or a column width, both of which are defined in the HTML file with class statements.

```
%do i = 1 %to &varcount;
  define &&var&i / display

  /* add title */
  %unquote(&&var&i._title)

  /* add any SAS format  if present */
  %if &&var&i._format ne %str( ) %then
    %do;
      format=&&var&i._format
    %end;

  /* create style statement if necessary */
  %if &&var&i._width ne %str( ) or &&var&i._xlformat ne %str( ) %then
    %do;
      style = {htmlclass="sas2xl&i"}
    %end;

  ; /* semicolon to end define statement */

%end;
```

These complex loops are only necessary if columns were defined in the macro call. If no columns were defined, such as in the third tab of DataForVP.xls, all columns should be set to display type:

```
define _all_ /display;
```

After %sas2xl generates the PROC REPORT based on the @def= call and executes it by using the custom tagset it generated, an Excel-readable HTML file is produced.

PROCESS MULTIPLE TABS

To create multi-tab files, you simply create separate files for each worksheet and then produce two files that tie the individual worksheet HTML files into a single workbook. However, Excel has strict rules about where files are located. If the files are not exactly as expected, Excel will ask for confirmation before opening, which is not acceptable for a robust utility. Getting around this is as simple as following Excel's expectations.

Excel places the individual files for each tab in a subdirectory whose name is the name of the Excel file without the .xls extension and "_files" appended. For example, DataForVP.xls needs to have its individual output files in a subdirectory called "DataForVP_files". Then each tab is in a series of files named and numbered sheet001.html, sheet002.html, etc.

In addition to placing the files in the required location, it is necessary to create two additional files. The first is the main file, the one that would be double clicked to open the workbook. It is a fairly simple HTML file that contains a limited amount of XML data that tells Excel where each worksheet's HTML file is located. For each worksheet tab, you must write XML code for ExcelWorksheet, which includes the sheet name in <x:Name>" and the file location in <x:WorksheetSource>. This file is generally short. For the DataForVP.xls file, this is all that is needed:

```
<html xmlns:o="urn:schemas-microsoft-com:office: office"
xmlns:x="urn:schemas-microsoft-com:office:excel"
<head>
<meta name="Excel Workbook Frameset">
<!--[if gte mso 9]><xml>
<x:ExcelWorkbook>
 <x:ExcelWorksheets>
  <x:ExcelWorksheet>
   <x:Name>Eastern</x:Name>
   <x:WorksheetSource HRef="C:\DataForVP_files\sheet001.htm"/>
  </x:ExcelWorksheet>
  <x:ExcelWorksheet>
   <x:Name>Western</x:Name>
   <x:WorksheetSource HRef="C:\DataForVP_files\sheet002.htm"/>
  </x:ExcelWorksheet>
 </x:ExcelWorksheets>
 <x:ActiveSheet>0</x:ActiveSheet>
</x:ExcelWorkbook>
  <x:ExcelName>
  <x:Name>Print_Titles</x:Name>
  <x:SheetIndex>1</x:SheetIndex>
  <x:Formula>='Eastern'!$1:$1</x:Formula>
</x:ExcelName>
  <x:ExcelName>
  <x:Name>Print_Titles</x:Name>
  <x:SheetIndex>2</x:SheetIndex>
  <x:Formula>='Western'!$1:$1</x:Formula>
</x:ExcelName>
</xml><![endif]-->
</head>
</html>
```

The second additional file is in the "*_files" subdirectory. It is an XML file called filelist.xml that contains a list of the individual files. For DataForVP.xls, this is the filelist.xml file:

```
<xml xmlns:o="urn:schemas-microsoft-com:office:office">
 <o:MainFile HRef="../DataForVP.xls"/>
 <o:File HRef="sheet001.htm"/>
 <o:File HRef="sheet002.htm"/>
 <o:File HRef="filelist.xml"/>
</xml>
```

In addition to writing out these files, %sas2xl does a considerable amount of error checking when creating these files. Although it's unlikely that a subdirectory with the name "*_files" would exist, %sas2xl checks for this to make sure nothing is going to be overwritten in error.

With each individual sheet processed and the required files created, the workbook is ready to be read into Excel. When the main file is double clicked on, Excel will read in the HTML files and display a single workbook.

CONVERT TO A NATIVE EXCEL FILE

Although %sas2xl does not rely on Excel to produce the Excel-readable HTML files, it can convert files to native Excel format if it is run on the windows platform and Microsoft Excel is installed. The benefit of converting to Excel format is the creation of a single file that loads more quickly. This is done with dde commands, so SAS can talk directly with Excel (see Vyverman, 2002, for details on these techniques).

```
options xsync noxwait xmin;

filename sas2xl dde 'excel|system';

/* this opens xl if not already open */
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;

data _null_;
  file sas2xl;
    put "[OPEN(""&path&file"")]";
    put '[error(false)]';
    put "[SAVE.AS(""&path&file"", 1, , FALSE, , FALSE)]";
    put '[File.Close(TRUE)]';run;
run;
```

After converting to a native Excel format file, %sas2xl can clean up the various HTML files it created. This is done with a series of X commands such as

```
X "del ""filelist.xml"" ";
```

CONCLUSION

%sas2xl has proven that you do not have to accept limitations when it comes to generating great user-friendly Excel spreadsheets quickly and easily. While it takes a large amount of up front work to develop a macro-based approach to writing SAS data to Excel files, the benefits and time saved in the long run make it well worth the effort. %sas2xl has proven to be a very robust and flexible utility. I use it inside of other macros, including one that produces separate spreadsheets for each patient in a clinical trial data base (these spreadsheets contain only the specific subject's data). These files are delivered to medical writers who have found them to be extremely useful for writing patient narratives and general data review. %sas2xl has made this process quick and easy.

Remember the data you need by Monday morning? Don't worry. It's done, so have a good weekend.

REFERENCES

DelGobbo, V. (2003), A Beginner's Guide to Incorporating SAS® Output in Microsoft® Office Applications, Proceedings of the Twenty-eighth Annual SAS Users Group International Conference, 28.

Parker, Chevell (2003), Generating Custom Excel Spreadsheets using ODS, Proceedings of the Twenty-eighth Annual SAS Users Group International Conference, 28.

Vyverman, Koen (2002), Creating Custom Excel Workbooks from Base SAS® with Dynamic Data Exchange: A Complete Walkthrough, Proceedings of The Twenty-seventh Annual SAS Users Group International Conference

Microsoft Office HTML and XML Reference
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnoffxml/html/ofxml2k.asp

ACKNOWLEDGEMENTS

I would like to thank Jay Zhou for his review and encouragement of this paper.

CONTACT INFORMATION

David Brown
Amylin Pharmaceuticals, Inc.
9360 Towne Centre Drive, Ste. 110
San Diego, California 92121
dbrown@amylin.com