PharmaSUG 2013 - Paper CC05

Can you export all the datasets into Excel for me? – A small dynamic macro that does it as fast as they think it should.

Steven Black, W. L. Gore & Associates Inc., Flagstaff, AZ

ABSTRACT

The purpose of this paper is to demonstrate the power of utilizing PROC SQL, the macro language, and the EXCELXP tagset to complete a seemingly simple task of exporting a number of datasets into a single Excel workbook. The secondary purpose of the paper is to briefly explain each section of code moving from basic syntax to a dynamically driven efficient code which let's SAS® do all the hard work for you. The final result is code and an understanding which is transferable to any project with a limited number of key strokes.

INTRODUCTION

If you've ever been asked to 'simply' create an Excel spreadsheet with all of the data in it, and do not have some sort of preset program to do so, you know that this seemingly simple task can be quite a chore. Especially if it is wanted in a single workbook including labels and formatting applied. The request is doable, but it can take a lot more work and time than others may realize.

The intent of this paper is to provide a compact dynamic macro which provides a nicely formatted Excel workbook that contains all the data requested. The second intent is to provide basic information regarding each of the procedures used and how the SAS macro language is utilized to dynamically create the code.

In this paper I will:

- Discuss the benefits of using PROC SQL to dynamically obtain the dataset names.
- Provide MACRO language basics and illustrate how these principles are utilized within the code.
- Provide a quick and easy example of PROC TEMPLATE.
- Demonstrate the capability the EXCELXP tagset.
- Illustrate a simple example of PROC REPORT.
- Put it all together, using each of the procedures above into a simple but effective dynamic macro.

Throughout the paper I will discuss only those options that are relevant to the example provided as there are many other papers and information available to obtain details of each section.

PROC SQL

Using PROC SQL, I am able to utilize a number of essential items that will enable the program to be dynamic and efficient. PROC SQL allows users to access the dictionary tables which are automatically generated at runtime and the tables' contents become available once the SAS session is started. There are a number of dictionary 'tables' that are available, in this example I use the 'tables' options, which provides the meta-data of the various datasets within the SAS libraries, including the commonly used WORK library. Within the dictionary.tables option I can pull the libname, table names (memname), number of observation (nobs), and a number of other metadata items (see Lafler reference below).

Basic PROC SQL Example Code:

```
proc sql;
select memname
from dictionary.tables
quit;
```

Adding a few options such as the noprint suppresses the output from being printed to the output window, and using a where clause limits the data to a particular library (I have selected the default 'WORK' library but this will be changed later to be more dynamic). I also use the upcase function to remove any potential case issues with the library name. Notice that there is no semicolon after the text "from dictionary.tables" due to the use of a subsequent where statement.

```
   proc sql noprint;
   select memname
   from dictionary.tables
   where upcase(libname)="WORK";
   quit;
```

Next I'll limit some of the datasets being pushed out, in this case these are constant and do not change so I will hard code them into my dynamic program, if needed I could macrotize them but for this example will leave them as hardcoded. I also want to collect the total number of datasets in the library besides those be excluded and store this number as a macro variable. This is done using a %let statement creating a new macro variable which in this case is called num_sasdata and setting it equal to the &sqlobs macro variable automatically created within the SQL procedure. The &sqlobs macro variable is a sum of the number of rows processed, which in this case, is the number of datasets or tables within the specified library, again excluding those datasets not wanted.

```
   proc sql noprint;
   select memname
   from dictionary.tables
   where upcase(libname)="WORK"
   and upcase(memname) not in ('FREEZE_DATE','LOG','TEMP');
   %let num_sasdata=&sqlobs;
   quit;
```

The final option I want to use is the ability to create macro variables from a column value. The macro variable is assigned using the value of the column specified in the select list. A colon is used in conjunction with the macro variable name being defined. In this example I am macrotizing the values of the memname and storing each of the values in a different macro variable suffixed with the number 1 through 99, or up to the value of iterations of rows processed (&sqlobs). This is done by creating macro variable and suffixing it with a number then using a '-' and ':' then the same macro variable with a suffix up to the highest possible number of rows.

```
   proc sql noprint;
   select memname into: sasdata1 - : sasdata99
   from dictionary.tables
   where upcase(libname)="WORK"
   and upcase(memname) not in ('FREEZE_DATE','LOG','TEMP');
   %let num_sasdata=&sqlobs;
   quit;
```

Lastly, instead of guessing the number of datasets that may be in the library, I'll use the same principles as shown earlier to get the true number of datasets within the selected library (macro variable mename_cnt) and use this in the into statement to have truly dynamic code. PROC SQL has a number of summary functions that are available to use within the PROC SQL code, in this example I use the COUNT summary function (for more information regarding summary functions, see Lafler reference below) I use a %eval macro function to remove any spaces that may append themselves between the sasdata and resolved &memname_cnt variable and insure the data is numeric. I can do all this within the same PROC SQL step. I will use the 'members' dictionary table which also contains the dataset names in each library.

```
   proc sql noprint;
   select count(memname) into: memname_cnt
   from dictionary.members
   where upcase(libname="WORK";

   select memname into: sasdata1 - : sasdata%eval(&memname_cnt)
   …
```

So using this code we can dynamically pull all the dataset names within a certain library, excluding those not wanted and hold these names within a macro variable per dataset name. We also have the total number of datasets processed.

MACRO BASICS

Now that we have the names and count of all datasets stored within macro variables we are able to utilize this and move to the next step of using these macro variables and creating more dynamic fields. First it will be essential to illustrate some of the principles of the macro language and how macro variables resolve.

Some basic principles of using and creating macros:

- Macro names must start with a letter or underscore
- Macro variables must be referenced by using an ampersand (&)
- A period can be used after a macro variable name to concatenate text onto the resolved value of the macro variable, as a suffix.
- %do loops must be used within a macro shell.
- Double quotes must be used when macro variables need to be resolved within quotes.
- Using the %put statement allow us to see in the output window how a macro variable resolves.
- When two or more ampersands appear next to each other, successive passes are required to make the final resolution.

Principles of resolving macro variables:

Example of appending two macro variables together, using three %let statments

```
%let macro_var=sample;
%let n=2;
%let macro_var2=example;
```

Original Macro Variable	**Mid Resolution**	**First Resolution**	**Mid Resolution**	**Second Resolution**
&macro_var	&macro_var	sample		
&macro_var&n	&macro_var&n	sample2		
&&macro_var&n	&&macro_var&n	&macro_var2	&macro_var2	example

Table 1. Example of resolution of macro variables

Now if we wanted to do this same procedure for each of the datasets we will need to use a %do loop and replace the dataset name with the macro variable for each of the dataset created. We also want to limit the calls to the number of rows read and so we use &num_sasdata as our limiting statement. We create a new macro variable called x which will equal 1 to the value of &num_sasdata.

Utilizing the %do loop also requires the use of a macro shell. The most simple form of this is to create a macro name, lets call this one *EXCEL*. We begin with a macro statement *%MACRO* then that name of the macro, so we have *MACRO %EXCEL*, we end the macro statement with *%MEND* statement. To invoke or call this macro we use the *%* in front of the macro name we would like to invoke, in our first example we'll have *%EXCEL;*. All code within the macro is then run. However unless specified as %global macro variable, most macro variables are only created temporarily when within the macro statements. This requires that most PROC SQL statements which create the macro data be included within the macro shell.

```
%macro excel;

%do x = 1 %to &num_sasdata;
%end;

%mend excel;
```

Next will bring in the PROC SQL code created above resulting in the code below

```
%macro excel;

proc sql noprint;
select count(memname) into: memname_cnt
from dictionary.members
where upcase(libname)="WORK";

select memname into: sasdata1 - : sasdata%eval(&memname_cnt)
from dictionary.tables
where upcase(libname)="WORK"
and upcase(memname) not in ('FREEZE_DATE','LOG','TEMP');
%let num_sasdata=&sqlobs;
quit;
```

```
%do x = 1 %to &num_sasdata;
%end;

%mend excel;
```

In subsequent steps I'll discuss the elements of proc template, EXCELXP tagsets, and proc report which will add more functionality to the program, but first I'll discuss how to increase the dynamics by adding more functionality of the macro.

In my example above we can add macro parameters to tell SAS what libname to pull from and provide the name and location of the output xls file. To accomplish this we will first add the parameters to the top of the macro call within parenthesis, for this example I'll call the libname parameter 'lib' and the name and location of the output xls file as 'file_loc', each macro parameter is separated by a comma, and in this example I'll set both to missing, however if a default value is wanted then this can be specified here.

```
%excel (lib=,file_loc=);
```

I'll then use this same code and place it at the bottom of the call and fill in the information with the desired parameters.

```
%excel (lib=WORK,file_loc=c:\Excel_Data.xls);
```

Finally I'll replace the text initially used with these new dynamic macro variables ('WORK' will be replaced with "&lib"). In our example we have not yet created the xls output but this will be done shortly.

```
%macro excel (lib=,file_loc=);

    proc sql noprint;
    select count(memname) into: memname_cnt
    from dictionary.members
    where upcase(libname)="&lib";

    select memname into: sasdata1 - : sasdata%eval(&memname_cnt)
    from dictionary.tables
    where upcase(libname)="&lib"
    and upcase(memname) not in ('FREEZE_DATE','LOG','TEMP');
    %let num_sasdata=&sqlobs;
    quit;

    %do x = 1 %to &num_sasdata;
    %end;

%mend;
%excel (lib=WORK,file_loc=c:\Excel_Data.xls);
```

This will allow us to easily change the location of the data to be exported and the file location of the export.

PROC TEMPLATE

In this paper I want to present a quick, simple, and easy to understand example of proc template. Many times I have found examples of proc template to be discouragingly complicated with a lack of description of which code overrules another code applied later. In this example of proc template there are 5 sections which are laid out as follows: The procedure call - including the new style name and parent reference, the body style, the table style, the header style, and the data style.

```
proc template;
define style mystyle;
     parent = styles.default
     ;
    style body from body /
        topmargin=.5in
        bottommargin=.5in
        leftmargin=.5in
        rightmargin=.5in
     ;
     style table from table /
         foreground = black
         font_face = 'arial'
         font_size = 10pt
         just = center
         font_weight = bold
         borderwidth = 1
         vjust = center
         bordercolor = black
     ;
     style header from header /
         foreground = black
         font_face = 'arial'
         font_size = 10pt
         just = center
         font_weight = bold
         borderwidth = 1
         vjust = top
         bordercolor = black
     ;
     style data from data /
         background = white
         foreground = black
         font_face = 'arial'
         vjust = center
         just = center
         font_size = 10pt
         borderwidth = 1
         bordercolor = black
     ;
end;
quit;
```

To create a new style template is moderately simple - first create a name for the style, mine happens to be the very common 'mystyle'. Then determine a parent style which to modify from, I chose 'default', there are over 50 other parent styles that you can choose from, the code '`proc template;list styles;run;`' can be used to see all available styles, however some look better than others.

The body section I use to change the margins in the output document to fit my specific needs. The next section the table style which primarily controls the borders of the table, I set mine to the attributes as seen above, if no change is made then the default value will remain. To see a list of all default values and additional information regarding proc template see the reference paper by Hayworth. The header section defines the headers of the table including the justification of the header labels and the size and color of the header border. The data section modifies all of the cells within the excel table. The vjust option will vertically adjust the data in the cells to whatever you specify (top, center, or bottom). You close the proc template procedure with an 'end;quit;' block of code.

EXCELXP TAGSET

The relatively new EXCELXP tagset has greatly increased the ability to create an excel document that requires little or no post production manipulation. The tagset has been updated a number of times and increased in function and scope after each iteration. The tagset can be found at http://support.sas.com/rnd/base/ods/odsmarkup/, this page contains the current version and previous versions of the EXCELXP tagset. SAS also provides a very handy quick reference guide which contains all of the options available and a brief description of the option and default values http://support.sas.com/rnd/base/ods/odsmarkup/excelxp_help.html. To use the tagset you must download it and copy/paste into the editor window then run the code. This will update your tagset library with the proper code needed. This paper will only discuss a few of the options used as excellent additional references have been provided. The tagset itself contains a fuller description of all of the options which will be displayed in the log when the following code is used:

```
  ods tagsets.excelxp file="test.xml" options(doc="help")
```

To begin using the tagset you use the ods statement then reference the tagset and create an output file with a either a .xml or .xls suffix.

```
ods tagsets.excelxp file="c:\temp\example.xls"
```

Then state the options desired in parenthesis, each option is followed by an equals sign and then the value in single or double quotes. After the close parenthesis the style is referenced. In this example the style 'mystyle' created in the proc template code above is used.

```
options(sheet_name='Example Data' autofit_height='yes' wraptext='no'
frozen_headers='1' orientation='landscape' absolute_column_width='20')
style=mystyle;
```

There are numerous options within the tagset which allow the user to get quite detailed in how the resulting Excel output will look. To limit the scope of this paper I will only address a few of those that I have found particularly helpful.

- Frozen_headers: can be 'yes' or a number, which will be the row count frozen.
- Frozen_rowheaders: can be 'yes' or a number, which will be the column count frozen.
- Autofilter: values 'all', or a range such as '3-5' columns in which the filter will be applied.
- absolute_column_width: a number or list of numbers but this will set the width of all columns of data to the number listed a value of 13 roughly equals 1 inch.
- Sheet_name: names the sheet within the workbook.
- Orientation: 'portrait' or 'landscape'
- Fittopage: 'yes' or 'no'. Accompanied by pages_fitwidth: number and pages_fitheight: number.
- Print_header: creates the title for the document there are a lot of options to this (see tagset references).
- Print_footer: creates the footer for the document, with similar options for titles however, there is a limit of 250 characters for the whole footer.
- Wraptext='yes' coupled with autofit_height: 'yes' resizes the row height based off the data.

To close and run the tagset the following code is used:

```
ods tagsets.excelxp close;
ods listing;
```

As a standard coding I close the ods listing prior to running the tagset and then open it again once run.

PROC REPORT

Within the tagset open and close statements sits the proc report code. To begin using the report procedure you call the procedure and reference the datasource. For this example I'll use the sashelp.class dataset, then you can use the various options defined below:

```
proc report data=sashelp.class nowd missing;
```

The missing option considers missing values as valid values for group, order, or across variables. If you omit the MISSING option, then PROC REPORT does not include observations with a missing value for any group, order, or across variables in the report. When you use NOWD or NOWINDOWS option, PROC REPORT runs without the REPORT window and sends its output to the open output destinations.

In this example I am not limiting the data in anyway or removing any variables. I am also going to use the labels already established within the each of the datasets, so no define statements or even a column statement is used. I am simply pulling each of the datasets using my macro variable loop datasets (&&sasdata&x) from the specified library (&lib) and reporting them out to the output destination.

```
proc report data=&lib..&&sasdata&x nowd missing;
run;
quit;
```

If there are some variables that are not wanted within certain datasets then you could again use the macro variable to specifically create keep, drop, or where statements semi-dynamically.

```
%if x=1 %then %do;
(drop=site age race);
```

```
%end;
%else %if x=2 …
```

FINAL CODE

Despite the length of text it took to explain the principles of each step, the actual code to export all datasets within a specified library into a single workbook is quite compact.

```
proc template;
define style mystyle;
     parent = styles.default
     ;
    style body from body /
        topmargin=.5in
        bottommargin=.5in
        leftmargin=.5in
        rightmargin=.5in
     ;
     style table from table /
         foreground = black
         font_face = 'arial'
         font_size = 10pt
         just = center
         font_weight = bold
         borderwidth = 1
         vjust = center
         bordercolor = black
     ;
     style header from header /
         foreground = black
         font_face = 'arial'
         font_size = 10pt
         just = center
         font_weight = bold
         borderwidth = 1
         vjust = top
         bordercolor = black
     ;
     style data from data /
         background = white
         foreground = black
         font_face = 'arial'
         vjust = center
         just = center
         font_size = 10pt
         borderwidth = 1
         bordercolor = black
     ;
end;
quit;

%macro excel (lib=,file_loc=);

    proc sql noprint;
    select count(memname) into: memname_cnt
    from dictionary.members
    where upcase(libname)="&lib";

    select memname into: sasdata1 - : sasdata%eval(&memname_cnt)
    from dictionary.tables
    where upcase(libname)="&lib"
    and upcase(memname) not in ('FREEZE_DATE','LOG','TEMP');
    %let num_sasdata=&sqlobs;
    quit;

ods tagsets.excelxp file="&file_loc"

options(autofit_height='yes' wraptext='no' frozen_headers='1' row_repeat='1'
orientation='landscape' column_repeat='1' center_horizontal='yes'
absolute_column_width='20') style=mystyle;

    %do x = 1 %to &num_sasdata;

            ods tagsets.excelxp options(sheet_name="&&sasdata&x");

            proc report data=&lib..&&sasdata&x nowd missing;
            run;
            quit;
    %end;

ods tagsets.excelxp close;
ods listing;

%mend;
%excel (lib=WORK,file_loc=c:\Excel_Data.xls);
```

CONCLUSION

The purpose of this paper was to provide a compact dynamic macro which creates a nicely formatted Excel workbook that contains all the data requested. The second intent was to provide basic information regarding each of the procedures used and how the SAS macro language is utilized to dynamically create the code. Using this example and gaining an understanding of how each of these different procedures work in harmony with each other allows SAS users to be able to utilize this code and knowledge for a number of different applications.

REFERENCES

Lafler, Kirk Paul. 2004. *PROC SQL*: Beyond the Basics Using SAS®. Cary ,NC: SAS Institute Inc.

Haworth Lauren - PROC TEMPLATE: The Basics
http://www2.sas.com/proceedings/sugi31/112-31.pdf

DelGobbo Vincent - Creating AND Importing Multi-Sheet Excel Workbooks the easy way with SAS
http://www2.sas.com/proceedings/sugi31/115-31.pdf

Options for Report Procedure
http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a002473620.htm

ACKNOWLEDGMENTS

Many thanks to Greg Newgard, Leela Muppana, Art Carpenter, and Vince DelGobbo

CONTACT INFORMATION

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

Name: Steven Black
Enterprise: W. L. Gore & Associates
Address: 3250 W. Kiltie Lane
City, State ZIP: Flagstaff AZ 86001
E-mail: sblack@wlgore.com