Paper 2220-2015

Are You a Control Freak? Control Your Programs – Don't Let Them Control You!

Mary F. O. Rosenbloom, Edwards Lifesciences LLC, Irvine, CA
Art Carpenter, California Occidental Consultants, Anchorage, AK

ABSTRACT

You know that you want to control the process flow of your program. When your program is executed multiple times, with slight variations, you will need to control the changes from iteration to iteration, the timing of the execution, and the maintenance of output and LOGS. Unfortunately in order to achieve the control that you know that you need to have, you will need to make frequent, possibly time consuming, and potentially error prone manual corrections and edits to your program.

Fortunately the control you seek is available and it does not require the use of time intensive manual techniques. List processing techniques are available that give you control and peace of mind that allow you to be a successful control freak. These techniques are not new, but there is often hesitancy on the part of some programmers to take full advantage of them.

This paper reviews these techniques and demonstrates them through a series of examples.

KEYWORDS

Control file, macro list, CALL SYMPUTX, CALL EXECUTE, macro language, macro arrays

INTRODUCTION

Both of the authors have lived in California and know that traffic there is no joke. However, you can now get into the carpool lane when driving solo if you have a special sticker. The stickers for hybrid cars have all been claimed, but there are still some available for fully electric cars. Once you set your sites on finding a fully electric car and obtaining that sticker, you see fully electric cars everywhere. Using control files to drive your code is a powerful tool, to say the least. If you get only one thing out of this paper, let it be that you will start seeing control files everywhere you look! We contend that once you start seeing them, you'll want to start using them. And once you get into that fast lane, you won't ever want to drive with the slow traffic again.

Control files are abundant, and can be used to automate your code. But why should you bother? The first few times coding this way will be cumbersome and time consuming. Then you'll start to improve your code. Then you'll realize that your code is data-driven and that it is *portable*. Not only is it self-adapting to the data or other driving force, but this results in code that; requires little-to-no maintenance by you, is much less error-prone, doesn't miss emergent subgroups, and doesn't need to be re-validated every time the data change. Not only that, but your code might be just as useful with little to no changes for an entirely different task!

We have so many examples of data-driven programming in our own work that we have had to limit the list. Here are some favorites:

- A macro that creates summary statistics for every variable in a dataset (with an optional drop) by writing macro calls based on variable types, as determined by PROC CONTENTS
- Code that writes a label statement that can be called within a DATA step where two datasets were merged – The label statement tacks on the dataset name of origin at the front of the old label
- Code that writes a PROC REPORT call for all datasets in a library
- An excel file where clients update study names and numbers – which is then read in as a dataset and transformed into a SAS® format using PROC FORMAT
- A macro that writes a macro call for a study center report for each new center in a growing study
- A control file that contains macro parameters with one macro call per row in the table

As you might have noticed, just because you are using a control file, doesn't mean that you will use a macro program. In most cases, you will use a macro variable; however we will see a couple cases where no macro tools are employed.

Two excellent papers have already been written on the mechanics of this process: "List Processing Basics: Creating and Using Lists of Macro Variables" (Fehd and Carpenter, 2007) gives us all of the tools that we need to build and process macro variable lists, to automate the writing of macro calls, and even demonstrates the old school technique of writing macro calls to a text file and then using %INCLUDE with that file to execute the calls. As if that weren't enough, CALL EXECUTE, with all of its caveats and idiosyncrasies, is also placed in the tool box. The second important paper, "Manual to Automatic: Changing Your Program's Transmission" (Carpenter, 2009) shows us how to go under the hood and convert a program that we use with manual changes into an automated macro.

So now that the tools have been laid out and we have gone under the hood, this paper aims to take you on a tour of all the different ways that you can use control files to drive your programs into the fast lane. Here is our roadmap: First we will review the elements of a data driven program ("The Six Step Program"). Then we will talk about %PUT _USER_, which will act as our rearview mirror. Then we'll review the techniques for building horizontal and vertical lists. Then we'll look at several examples, based on our own work, that employ horizontal and vertical list processing. We will also look at an example or two where the entire macro call is written. All the while, we are going to take advantage of a multitude of different types of control files to drive our programs. So, sit back and enjoy the ride!

THE SIX STEP PROGRAM

Most successful, not all, but most, data driven programs will have certain elements present in order to coordinate the information in the control source with the program that is performing the task. The "Six Step Program" was described by Carpenter (2009) to aid the reader in the conversion of programs that did not take advantage of a control file to ones that did use external information. Even when not converting a program, these same steps/elements can be applied to help you make sure that you have taken all the necessary elements into consideration. Several of the examples in this paper will refer back to these steps. This will be our road map that will help us navigate the twists and turns of list processing.

The Six Step Program

1. Identify the control file
2. Convert your program to accept a macro variable list
3. Process the macro variable list with a %DO loop
4. Within the %DO loop, access each element from the list using
    a. &&item&I for a vertical list (one macro variable per item)
    b. %QSCAN(&itemlist, &I, %STR()) for a horizontal list (one macro variable containing all of the "words" or items)
5. Build a vertical or horizontal macro variable list
6. For vertical lists only (a list of macro variables): count the number of elements in the list and store it as a macro variable.

OUR REARVIEW MIRROR: %PUT _USER_

You may be familiar with the system options MPRINT, MLOGIC, and SYMBOLGEN, which are helpful during macro program development. Another important tool is %PUT _USER_. When this code executes, the list of user-defined macro variables, both global and local, is written to the log. It is helpful to use this statement often to check that the macro variable definitions that you create are what you expect them to be. You may find it handy to assign this to a function key. One way to do this is to click on TOOLS →OPTIONS → KEYS in the menu bar of the program editor. The KEYS window will open, and you can simply type the command in one of the available spots. We have chosen SHF F11 (shift +F11). Now, every time that you want to view the macro variable definitions in the global symbol table, hit this key combination and look at the LOG. (Rosenbloom and Lafler, 2012)

```
gsubmit "%put _user_;"
```

BUILDING VERTICAL AND HORIZONTAL LISTS: A REVIEW

If you have not yet read the foundation papers for list building, allow us to get you up to speed. There are two kinds of lists: vertical and horizontal. Both use macro variables to hold control information. A vertical list consists of several macro variables, each holding one piece of information. A horizontal list is just one macro variable holding all of the information in a delimited string. Both kinds of lists can be built with either a data step or PROC SQL.

In this code, we build a horizontal list of names from the SASHELP.CLASS data using PROC SQL with the SELECT INTO methodology, separating the names with spaces. ❶ This list of values is stored in one macro variable, &NAMESQL. We can also build a vertical list, storing each name in a different macro variable, as shown in the second SELECT statement. ❷ In this case, since there are 19 observations in the dataset, we will create 19 macro variables, &NAMESQL1, &NAMESQL2, etc, each holding one name. We can also create a macro variable that holds the total number of variables that we created. This is &TOTSQL, in this case, which is created using the automatic macro variable &SQLOBS. ❸ Since &SQLOBS effectively counts the number of observations in our dataset, and since we are

```
proc sql noprint;
      /*create a horizontal list*/
      select name into :nameSQL separated by ' ' ❶
      from sashelp.class;

      /*create a vertical list*/
      select name into :nameSQL1-:nameSQL999 ❷
      from sashelp.class;
quit;
%let totSQL=&sqlobs; ❸
```

creating one macro variable per record, we can use &SQLOBS to tell us how many macro variables we have just created in the *most recent* SELECT statement.

Although generally easier in a SQL step, we can also create horizontal and vertical lists using the DATA step.

```
data _names;
      set sashelp.class end=last; ❹
      format nameDATA $1000.;
      retain nameDATA ''; ❺

      /*build concatenated list of names*/
      nameDATA=catx(' ', nameDATA, name); ❻
      /*assign to macro variable -- horizontal list*/
      if last then call symputx('nameDATA', nameDATA); ❼

      /*assign macro variable to each name -- vertical list*/
      call symputx(catt('nameDATA',_n_), name); ❽
      /*assign macro variable to total number of names*/
      if last then call symputx('totDATA', _n_); ❾
run;
```

When we read in the SASHELP.CLASS data, we now add a variable LAST, using the END= SET statement option, which will indicate the end of file. ❹ Building a horizontal list is more work in the DATA step, and it is limited by the length of the DATA step variable that will be used to hold the list. This technique is shown here for completeness, but in practice, we rarely build horizontal lists in the DATA step. In this case, we retain our variable NAMEDATA, and specify a length of 1000 characters by assigning a format. This will be long enough not to truncate our list this time, but not necessarily with other data. ❺ To build the concatenated list of names with our retained variable, we use CATX. ❻ When we get to the end of file marker (LAST=1), we use CALL SYMPUTX to assign the string to a macro variable of the same name. ❼ Building a vertical list in a data step is a bit easier, and is commonly used because we don't have the same issues with variable length when storing values in several macro variables. Again, we use CALL SYMPUTX to create macro variables, only now we must add a numeric suffix to the name, which we do with _n_. ❽ This results in the creation of &NAMEDATA1, &NAMEDATA2, etc. When we reach the end of file marker (LAST=1) then we also create &TOTDATA, which holds the value of the number of records in the dataset, which is also the number of macro variables in our vertical list. ❾

```
GLOBAL TOTSQL 19
GLOBAL NAMEDATA Alfred
Alice Barbara Carol Henry
James Jane Janet Jeffrey
John Joyce Judy Louise
Mary Philip Robert Ronald
Thomas William
GLOBAL NAMEDATA5 Henry
GLOBAL NAMEDATA9 Jeffrey
GLOBAL NAMEDATA10 John
GLOBAL NAMEDATA11 Joyce
GLOBAL NAMESQL6 James
GLOBAL NAMESQL1 Alfred
GLOBAL NAMESQL10 John
GLOBAL NAMESQL Alfred
Alice Barbara Carol Henry
James Jane Janet Jeffrey
John Joyce Judy Louise
Mary Philip Robert Ronald
Thomas William
GLOBAL TOTDATA 19
```

A simple call of "%PUT _USER_;" (or SHF+F11) will put the values of these variables in the log. Only part of the list is shown, but you can see that the variables don't exactly follow parking rules. They come out in a funny order, but they are all there!

So now that we've done our tune up, let's hit the road and visit some examples based on what we have encountered in our own work.

EXAMPLE 1: PROC FORMAT

***model**: format*

***tool**: EXCLUSIVE option in PROC MEANS*

***fuel**: format definition*

***destination**: control the class levels in the output*

```
proc format;
   value $type
       'SUV' = 'SUV'
       'Truck' = 'Truck'
       'Wagon' = 'Wagon'
       'Hybrid' = 'Hybrid';
run;
```

When you think of control files, formats are probably not at the top of your list, but they can, in fact, be used as control files. Using the SASHELP.CARS data, we can create a format for the variable TYPE. Even though the data values will already match the formatted values, we will be able to use this format as a control file in a couple of ways. The data contain information for more types of cars than are specified in our format ("Sedan" and "Sport" are not included in the format). When we run the following code, which purposely excludes the records for hybrid cars, we can see that the output includes summaries for only the values found in our format.

Because we use COMPLETETYPES, we get values for hybrids, ❶ but because we use the PRELOADFMT and EXCLUSIVE options, we get summaries for all of the groups specified in the format, while also excluding the types of vehicles not included in the format. ❸ So, we have controlled which values are summarized, both by setting options in our procedure and by taking the time to create a format.

```
title 'preloadfmt exclusive completetypes';
proc means data=sashelp.cars
  (where=(type ne 'Hybrid')) completetypes; ❶
       format type $type.; ❷
       class type/preloadfmt exclusive; ❸
       var MPG_City;
run;
```

Formats are powerful tools for controlling the data and how it is summarized. And, since formats can be defined using datasets (Carpenter, 2003) we can control them with control files, too! Several SAS procedures have options that allow us to use formats to control the output. For more information on this, see Carpenter, 2012.

Analysis Variable : MPG_City MPG (City)

Type	N Obs	N	Mean	Std Dev	Minimum	Maximum
Hybrid	0	0	.	.	.	.
SUV	60	60	16.1000000	2.8206262	10.0000000	22.0000000
Truck	24	24	16.5000000	3.2302914	13.0000000	24.0000000
Wagon	30	30	21.1000000	4.2128703	15.0000000	31.0000000

EXAMPLE 2: WRITING A LABEL STATEMENT

***model**: horizontal list*

***tool**: select into*

***fuel**: sashelp view: vcolumn*

***destination**: label statement*

In a recent project, study centers within a clinical trial reported adverse events. These events were reviewed by a clinical events committee (CEC). Data from the two sources needed to be merged together, but before doing so, we wanted to add the origin of the data point to the beginning of the label for each variable.

PROC DATASETS provides a nice facility for modifying the descriptor data on a dataset without processing each record. Since we are working with SASHELP.CARS data, we will need to read the dataset into the WORK library so that we can modify it with PROC DATASETS. For simplicity, we will create this data set in a SORT step and keep only four variables at the same time. Note that in this dataset the variable MAKE doesn't have a label defined.

```
proc sort data=sashelp.cars
     out=_cars (keep=EngineSize MPG_City Weight make);
     by make model;
run;
```

The code that is used to update the labels is straightforward. We are simply appending the prefix 'Cars Data:' to the existing label. However, doing this for a wide (many columns) dataset or for multiple datasets would be tedious. Furthermore, if new columns are added, it would be nice to have the labels updated automatically. Did I hear 'automatically? Sounds like a great opportunity for some macro list processing.

```
proc datasets lib=work nolist;
     modify _cars;
          label EngineSize='Cars Data: Engine Size (L)'
                MPG_City  ='Cars Data: MPG (City)'
                Weight    ='Cars Data: Weight (LBS)'
                 Make      ='Cars Data:';
quit;
run;
```

To get started, we need to re-run the PROC SORT code from above in order to reset the labels to the values from SASHELP.CARS. This is especially important if you just modified the labels in WORK._CARS using PROC DATASETS, because the labels have already been appended with the string "Cars Data:". Now, let's follow Art's six step process for changing our transmission from manual to automatic.

STEP 1: Identify the control file. We see that the fuel for this process will be the existing labels. We can use the SASHELP VIEW called VCOLUMN to get this information into macro variable lists. (Lafler, 2010) See Step 5 for the SQL step code.

STEP 2: Convert your program to accept a macro variable list. A list of one element would not require a %DO loop ❶. In general, we know that we will want a macro variable to contain a dataset variable name, and we will set that equal to some new label assignment, surrounded by quotes.

STEP 3: Process the macro variable list with a %DO loop. ❷ We will use a %DO loop to work through the items in the macro list.

STEP 4: Within the %DO loop, access each element from the list using %QSCAN(&itemlist, &I, %STR()). ❸ Since we are going to be working with a horizontal list, we will use %QSCAN to access each variable name and corresponding label assignment.

```
proc datasets lib=work nolist;
    modify _cars;
       label
         &var = "&label"; ❶
quit;
```

STEP 5: Build a vertical or horizontal macro variable list. For this example, we will build a horizontal list. We will create one macro variable &VLIST which will hold the list of variable names separated by spaces. ❹ Additionally, we will create the macro variable &LLIST which will hold the variable labels. ❺ We use the slash as the delimiter since the labels will likely contain spaces. Notice that we are using SASHELP.VCOLUMN, as we discussed in *STEP 1*, as the fuel for this process. ❻

```
proc datasets lib=work nolist;
    modify _cars;
       label
        %do i=1 %to &vcnt;❷
          %qscan(&vlist,&i) = "%qscan(&llist,&i,/)"❸
        %end;❷
        ;
quit;
```

STEP 6: For vertical lists only (a list of macro variables): count the number of elements in the list and store it as a macro variable. Since we are working with a horizontal list, we can skip this step.

```
proc sql noprint;
      select name, cat('Cars Data: ',strip(label))
         into :vlist separated by ' ',❹
              :llist separated by '/ ' ❺
      from sashelp.vcolumn ❻
      where libname='WORK' and memname='_CARS';
      %let vcnt=&sqlobs;
quit;
```

We can place all of this code into the macro %LABEL, as shown below. We can simply call the macro ❼, using no input parameters, and the labels will be assigned automatically. Notice the %PUT statements that were used for &VLIST and &LLIST. ❽ When we create these macro variables within the %LABEL macro, they are probably going to be stored in the local symbol table. So if we try to use our function key assigned to %PUT _USER_ to view them after the macro executes, they will not be available because they will have already been deleted. There are several ways around this. One way is to place the PROC SQL in open code outside of the macro program definition. Then &VLIST and &LLIST will then be stored in the global symbol table, which we can always access with %PUT _USER_. Another way would be to initialize them

```
%global vlist llist;
```

to the global symbol table within the macro, using a %GLOBAL statement prior to PROC SQL. As a general rule it is safer to place them in the local symbol table.

```
%macro label;
proc sql noprint;
      select name, cat('Cars Data: ',strip(label))
         into :vlist separated by ' ',
              :llist separated by '/ '
      from sashelp.vcolumn
      where libname='WORK' and memname='_CARS';
      %let vcnt=&sqlobs;
quit;
%put vlist is &vlist;❽
%put llist is &llist;
proc datasets lib=work nolist;
    modify _cars;
       label
        %do i=1 %to &vcnt;
          %qscan(&vlist,&i) = "%qscan(&llist,&i,/)"
        %end;
        ;
quit;
%mend label;
%label ❼
```

Using the global symbol table is not a best practice method, however, since global macro definitions within a programming suite can interact with each other in ways that are difficult to keep track of (Carpenter, 2006). So, it is always best to keep macro variable definitions as local as possible. But, in order to develop a solid understanding of writing data-driven code, it is vital that we understand the macro variable assignments at all times. In this example, we have chosen to embed the %PUT statements in the macro. Using the other methods mentioned here to move the macro variable definitions to the global symbol table is okay for learning purposes, but not as desirable for production work. In order to remove macro variable definitions from the global symbol table, %SYMDEL can be employed. (Rosenbloom and Lafler, 2012)

Cars Data:	Cars Data: Engine Size (L)	Cars Data: MPG (City)	Cars Data: Weight (LBS)
Acura	3.5	18	3880
Acura	3.5	18	3893
Acura	3.5	17	4451
Acura	3.2	17	3153

EXAMPLE 3: USE A DATA DICTIONARY TO SELECT VARIABLES TO READ, PRINT, AND SUMMARIZE

model*: vertical list*

tools*: SELECT INTO*

fuel*: data dictionary*

destination*: listing and summary statistics*

Data dictionaries are often supplied to a programmer as part of a statistical analysis plan, and can be an excellent control file. Generally data dictionaries contain one table for each analysis dataset that is planned for a suite, with one row for each variable. The variable name, label, data type, and description are often provided. Recently, a study statistician also provided an additional column containing a numeric code for each variable. This code could be used to determine if each variable should be placed into a listing, or if it should be summarized.

Variable	Type	Format	Label	Analysis Code
Make	Char			0
Model	Char			0
Type	Char			1
Origin	Char			0
DriveTrain	Char			0
MSRP	Num	DOLLAR8.		3
Invoice	Num	DOLLAR8.		1
EngineSize	Num		Engine Size (L)	1
Cylinders	Num			0
Horsepower	Num			1
MPG_City	Num		MPG (City)	3
MPG_Highway	Num		MPG (Highway)	3
Weight	Num		Weight (LBS)	0
Wheelbase	Num		Wheelbase (IN)	0
Length	Num		Length (IN	1

To simulate the data that might be read in from a data dictionary, we can run the data step shown here. In this case, a variable name and a code number are given. Now we are set up to walk through the six step process.

STEP 1: Identify the control file. In this case, the data dictionary will power our analysis. Here we have shown a portion of the control file after it has been read into a SAS data set. For this example we only needed two of the columns of information, VAR and CODE.

VIEWTABLE: Work.Control_cars

	var	code
1	Cylinders	0
2	DriveTrain	0
3	EngineSize	1
4	Horsepower	1
5	Invoice	1
6	Length	1

STEP 2: Convert your program to accept a macro variable list.
We would like to call PROC MEANS for every variable with CODE=3. PROC MEANS will use a macro variable in the VAR statement, ❶ and in the TITLE ❷ statement.

```
title "Summary for &&summarize_list&i";❷
proc means data=sashelp.cars;
       var &&summarize_list&i;❶
run;
```

STEP 3: Process the macro variable list with a %DO loop. We would like to run the PROC MEANS code once for each variable in our summarization list that matches our summarization criteria in &&CODE&I. Here we will only summarize those variables that have a CODE of 3. ❸ In the final macro we will make this value a macro parameter to increase flexibility. Finally this part of the code should be embedded in a %DO loop that increments across the list. ❹

```
%do i=1 %to &summarize_tot;❹
   %if &&code&i = 3 %then %do; ❸
       title "Summary for &&summarize_list&i";
        title2 "Variable with Code Level &&code&i";
       proc means data=sashelp.cars;
              var &&summarize_list&i;❺
       run;
   %end;
%end;❹
```

STEP 4: Within the %DO loop, access each element from the list using &&item&I for a vertical list (one macro variable per item). We will access each of the macro variables within the VAR and TITLE statements by referencing the macro variable array &&SUMMARIZE_LIST&I, ❺ indexed from 1 to &SUMMARIZE_TOT. ❹

STEP 5: Build a vertical or horizontal macro variable list. We create the vertical list of variables to summarize: &SUMMARIZE_LIST1, &SUMMARIZE_LIST2, etc. ❻, and their associated codes: &CODE1, &CODE2, etc. ❼

STEP 6: For vertical lists only (a list of macro variables): count the number of elements in the list and store it as a macro variable. We will create &SUMMARIZE_TOT which will represent the number of elements in our vertical list. ❽

```
proc sql noprint;
   select var, code
      into :summarize_list1-:summarize_list99 ❻
           :code1 – :code99 ❼
   from control_cars;
   %let summarize_tot=&sqlobs; ❽
quit;
```

Now let's put it all together. Since we are using a %DO loop, we need to place it into a macro. But we chose to leave the PROC SQL code outside of the macro. This is helpful for training purposes, but in production, best practices would usually be to place the PROC SQL code inside of the macro and to explicitly initialize all of the macro variables to the local symbol table. We can call the macro, using no input parameters, and dynamically produce the requested analysis and listing. The macro is called with the value of &&CODE&I that is to be selected. ❾

```
%macro summary(codeval=0);
   %do i=1 %to &summarize_tot;
      %if &&code&i = &codeval %then %do;
         title "Summary for &&summarize_list&i";
         title2 "Variable with Code Level &&code&i";
         proc means data=sashelp.cars;
            var &&summarize_list&i;
         run;
      %end;
   %end;
%mend;
%summary(codeval=3)❾
```

Data dictionaries can serve as powerful control files, as well as a communication tool between the study statistician and programmer. Moreover, the data dictionary can be updated by the statistician easily, without necessitating reworking of SAS programs. Excel files are often good vehicles for this, since they are easily read in with SAS/ACCESS or PROC IMPORT, and they can be maintained by someone other than the SAS programmer. With a little upfront planning by the study team, this control file can be standardized enough so that it can be used to order variables on the program data vector, assign labels, direct analysis, or more.

EXAMPLE 4: WRITE PATIENT SUMMARIES

***model**: horizontal list*

***tool**: select into*

***fuel**: the data itself*

***destination**: write one report per patient*

The form for patient summary reports is one page (or table) per event, with the first column of the table containing the attribute name, such as the site and subject id, event date, etc, and the second column containing the attribute value. A simple example of this type of report is shown here. This report could have been generated with a simple PROC REPORT step where the user specifies the name of the attribute column using a macro variable. In this example, which utilizes the SASHELP.CLASS data set, we want to generalize to create one report for each student without writing each one individually.

Summary for Alfred

Atribute	Value
Age	14
Height	69
Weight	112.5
Sex	M

The SASHELP.CLASS dataset contains one record per student, with information about height, weight, age, and more. As was shown above, we would like to create a table using PROC REPORT for each student in SASHELP.CLASS.

In this example we will get the names of the students from the original data, but will transpose the data to create one column for each student prior to generating the report. For the transposed data set (WORK.TCLASS), the data for each student will be in a variable with that student's name.

NAME	Alfred	Alice	Barbara
Age	14	13	13
Height	69	56.5	65.3
Weight	112.5	84	98
Sex	M	F	F

```
proc transpose data=sashelp.class
       out=tclass;
       id name;
       var age height weight sex;
run;
```

Once the data has been transposed it will be easy to use it in the REPORT.

STEP 1: Identify the control file. We will use the data itself to build the list of names.

STEP 2: Convert your program to accept a macro variable list. We will use PROC REPORT to summarize the record for each student. If it was a single value, the REPORT step would look something like the one on the right, where &NAME is used as a place holder for every place that we will access the macro variable list. However as we add the %DO loop in Step 3, the macro variable &NAME will become a list reference, which for a horizontal list like used here will be through a %QSCAN macro function call.

```
title1 j=c "Summary for &name";
proc report data=tclass nowd;
  column _name_ &name;
  define _name_/'Atribute';
  define &name /'Value' center;
run:
```

STEP 3: Process the macro variable list with a %DO loop. The %DO %WHILE loop will be used to step through the list.❶ Notice the incrementing variable, &I, which helps us process through the loop.❷

STEP 4: Within the %DO loop, access each element from the list using %QSCAN(&itemlist, &I, %STR()). We will access the macro variable list in the TITLE, COLUMN, and DEFINE statements. ❸

```
%do %while(%qscan(&ex4,&i+1,%str( )) ne %str());❶
      %let i=%eval(&i+1);❷
      title1 j=c "Summary for %qscan(&ex4,&i,%str( ))";❸
      proc report data=tclass nowd;
        column _name_ %qscan(&ex4,&i,%str( ));❸
        define _name_/'Atribute';
        define %qscan(&ex4,&i,%str( )) /'Value' center;❸
      run;
%end;❶
```

STEP 5: Build a vertical or horizontal macro variable list. We will build a horizontal list, and will store it in the macro variable &EX4.

```
proc sql noprint;
   select name
      into :ex4 separated by ' '
         from sashelp.class;
quit;
```

STEP 6: For vertical lists only (a list of macro variables): count the number of elements in the list and store it as a macro variable. Since we are working with a horizontal list, there is nothing to do for this step. In our example the %DO %WHILE automatically detects when the last item in the list has been processed. If we did know the number of items, which we could have retrieved from &SQLOBS in the SQL step, we could have used an iterative %DO instead of a %DO %WHILE.

So now we can put it all together into a macro. This time, we show the best practices method of storing &EX4 and &I in the local symbol table,❹ which we are able to do because we have moved the PROC SQL code into the macro. This means that after the macro executes, &EX4 and &I will be deleted until they are re-defined upon the next invocation of the macro. In our case, we will call this macro only once, so we do not lose processing time. Remember, if you ran the PROC SQL code, just above, you will have &EX4 defined in your global symbol table, which you will see if you use %PUT _USER_ to list all of the user-defined macro variables. To avoid confusion, you may want to use %SYMDEL to delete &EX4 before running the %PATIENT macro, below. ❾

```
%macro patient;
   /*build a list of names*/
   %local ex4 i;❹
   proc sql noprint;
       select name
              into :ex4 separated by ' '
       from sashelp.class;
   quit;
   %put &ex4;

   %let i=0;❺
   %do %while(%qscan(&ex4,&i+1,%str( )) ne %str());❻
       %let i=%eval(&i+1);❼
       title1 j=c "Summary for %qscan(&ex4,&i,%str( ))";
       proc report data=tclass nowd;
         column _name_ %qscan(&ex4,&i,%str( ));
         define _name_/'Atribute';
         define %qscan(&ex4,&i,%str( )) /'Value' center;
       run;
   %end;
%mend patient;
```

When we process through a horizontal macro list, we have to do a little more work. We start by initializing an accumulator variable, &I, to zero. ❺ Then, we use %DO %WHILE with %QSCAN to work through the list of names, one at a time.❻ After the selection of each name, we increment the word counter, &I. ❼

To create RTF output, we create an "ODS sandwich"❽, indicating the place where we would like to save the file, and then place the call to the macro %PATIENT within the "sandwich". ❾

This example could have been addressed with by-group processing. It is important to be aware of several ways to address a problem, because some are more appropriate in certain situations. For more information on customizing titles with by-group processing, see Carpenter, 1998.

```
options nodate nonumber papersize=letter
orientation=portrait nobyline;

ods rtf file="C:\Patient Summaries.rtf"
    style=Journal bodytitle; ❽
%patient ❾
ods rtf close; ❽
```

EXAMPLE 5: BREAKING UP AND APPENDING DATASETS FROM A LIBRARY

***model**: vertical list*

***tool**: DATA step/call symputx*

***fuel**: INFILE statement / a text file*

***destination**: obtain a cumulative (appended) dataset*

When working with data sets we will sometimes need to either breakup a given data set according to some classification criterion, or conversely we might want to append a series of unknown datasets. In either case we may well not know the names of the data sets or how many there are. In both cases we need the data to drive the process.

In order to set up the datasets for this example, we can use an X statement to create a file folder called 'example 5 data'. Then, we reference this folder with a LIBNAME statement, and break up our SASHELP.CARS data into three datasets that are saved there, one dataset for each value of REGION in the data. The NOXWAIT and NOXSYNC system options control the operation of the system sub-session that is spawned by the X statement.

```
options noxwait noxsync;
x md "C:\example 5 data." ;
```

Of course, we can use data-driven techniques to create our datasets. We write the macro %MULTI, initializing an incrementing variable, &I, to the local symbol table. ❶ We will store distinct values for the variable ORIGIN ❷ in the vertical list of macro variables &ORG1, &ORG2, etc. ❸ We will also create &ORGCNT, which will represent the number of macro variables that we created (since it will give us a count of the number of distinct values in our data). ❹ Then, we use a %DO loop to process through our vertical list, both within what will become the DATA statement, ❺ and to build a series of OUTPUT statements. ❻ The result of this is that we obtain a file folder, "Example 5 Data" containing one dataset for each distinct value of ORIGIN in the SASHELP.CARS data.

```
***create a libref for that file folder;
libname cars "C:\example 5 data";

%macro multi;
%local i; ❶
proc sql noprint;
select distinct origin ❷
   into :org1 - :org99 ❸
      from sashelp.cars;
%let orgcnt = &sqlobs; ❹

data
   %do i = 1 %to &orgcnt; ❺
     cars.&&org&i ❻
   %end;;
   set sashelp.cars;
   %do i = 1 %to &orgcnt;
      %if &i>1 %then else;
      if origin ="&&org&i" then
             output cars.&&org&i;
   %end;
   run;
%mend multi;
```

Now that the example is set up, let's see how we might use these datasets as control files. At this point, we will take advantage of the power of X statements. We can use an X statement, along with the appropriate switches, to create a file called "datalist.txt" which contains a vertical list of the file name and location for every SAS dataset in the "example 5 data" folder.

In addition to automating the breaking up of data sets, we may also need to take a series of SAS data sets and append or stack them into a single data set. Again we are going to assume that we do not know the number of data sets or even their names. There are a number of ways of creating a list of all the data sets within a folder. These include DICTIONARY.TABLES, SASHELP.VTABLES, PROC CONTENTS DATA=CARS._ALL_, and others. In the following example the X statement is used to build the list and write it to a temporary file where it will be read as data (a control list). This is a Windows® solution but similar solutions exist for other operating systems.

First we build a list of all the data sets in the specified directory. In this case we will pull from the same location that we just wrote to.

```
x dir "c:\example 5 data\*.sas7bdat"  /b /o:n >"c:\example 5 data\datalist.txt";
```

This text file now contains a list of all the SAS data sets within the specified folder. This list becomes the control file that we will use to drive the process of appending the list of data sets. First we need to use this list to create our list of macro variable – we need to get the information into SAS. We do this by using a DATA step and an INFILE statement, we can read the text file in as data, and after a little manipulation using the SCAN function to isolate the dataset names, we use CALL SYMPUTX to assign each dataset name to a macro variable; &DATASET1, &DATASET2, etc.

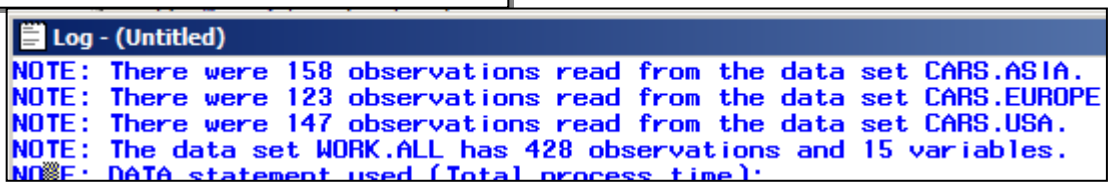
```
asia.sas7bdat
europe.sas7bdat
usa.sas7bdat
```

```
data pgms;
   infile "c:\example 5 data\datalist.txt"
          lrecl = 200 pad end=last;
   length progline $200;
   input progline $char200.;

   /*get the dataset name*/
   name=scan(progline,1,'.');

   /*assign macro variable to each name -- vertical list*/
   call symputx(catt('DATASET',_n_), name);
   /*assign macro variable to total number of names*/
   if last then call symputx('totDATASETS', _n_);
run;
```

Finally, we use a %DO loop to process the macro variable array and stack the data back together. Notice that the %DO loop is inside what will become the SET statement. Alternatively we could have used PROC DATASETS with the APPEND statement to concatenate the data sets.

```
%macro humpty_dumpty;
data all;
   set
      %do i=1 %to &totDATASETS;
         cars.&&dataset&i
      %end;
       ;
run;
%mend humpty_dumpty;
%humpty_dumpty
```

```
%macro humpty_dumpty2;
proc datasets lib=work nolist;
   %do i=1 %to &totDATASETS;
   append base=all data=cars.&&dataset&i;
   %end;
   quit;
%mend humpty_dumpty2;
%humpty_dumpty2
```



Another information source for additional control is the operation system itself. In SAS we have macro variables. The operating system -- in this example Windows -- has a similar concept known as environmental variables. These environmental variables are named, and can be a great source of information that our macros can exploit.

The %GRABPATH macro shown here (Carpenter, 2008) utilizes two of these environmental variables;

```
%macro grabpath ;
   %qsubstr(%sysget(SAS_EXECFILEPATH),
   1,
   %length(%sysget(SAS_EXECFILEPATH))-%length(%sysget(SAS_EXECFILEname))
   )
%mend grabpath;
```

SAS_EXECFILEPATH and SAS_EXECFILENAME. When you are executing code from within the SAS Display Manager you can use the %SYSGET macro function to retrieve their values:

- SAS_EXECFILEPATH holds the path and name of the executing program
- SAS_EXECFILENAME holds the name of the executing program

The %GRABPATH macro is written to be a macro function that returns the path portion of the location of the executing program.

In the expansion of the previous program in Example 5 we want to store the data sets in the same directory that contains the calling program. We can discuss whether this is a good idea, but for now let's do it. Let us assume that we do not necessarily know the location of the executing program so we need to write it to self-determine the location. In the statements used above, wherever we need the path (it was c:\ in the earlier examples) we can now replace the path information with a call to the %GRABPATH macro. The LIBNAME and X statements are shown here.

```
x "%str(md %"%grabpath\example 5 data.%" )";

libname cars "%grabpath\example 5 data";

x "%str(dir %"%grabpath\*.sas7bdat%" /b /d /s >%"%grabpath\datalist.txt%")";
```

And the INFILE statement in the step that reads the text file becomes:

```
data pgms;
   infile "%grabpath\datalist.txt" lrecl = 200
                pad end=last;
   length progline $200;
```

EXAMPLE 6: GET A LIST OF SAS PROGRAMS AND THEIR PURPOSE

***model**: X statement*

***tool**: DATA step processing*

***fuel**: SAS program headers*

***destination**: read your program headers and report on them*

Many programmers are required to populate the top or "header" section of each program with standard information that documents the program name, location, version purpose, and programmer, among other things. Many working groups create a standardized header to ensure that all necessary information is present, and documented consistently. When programs are validated by another programmer, there is often a controlled document that is created and signed by both the programmer and validator. Creating these signoff sheets can be error prone, hard to keep track of, and tedious. We were able to simplify this process by treating the .sas files as data, and reading them into a DATA step. Because each program contained a standardized header, we knew where to look for key information. This data was used along with dynamic data entry (DDE) techniques to populate fields in a signoff sheet template, generating the forms programmatically, and saving days if not weeks of work over the course of several years.

To simulate this situation, the programs from each of the examples in this workshop have been saved with common headers in the same folder. The header section of each program contains information on the "model", "tool", "fuel" and "destination" associated with the example. Additionally, the final line of every program contains the string "/*END OF PROGRAM*/".

```
/*HOW EXAMPLE 6: GET A LIST OF SAS PROGRAMS AND THEIR PURPOSE*/
/*model: infile statement*/
/*tool: ???*/
/*fuel: SAS program headers*/
/*destination: read your program headers and report on them*/
```

Our program will use the SAS programs themselves as the control files for the process. Using an INFILE statement, we can read in all of the programs. In fact, this statement will read each line of each of the programs, as if it were a dataset, stacking each program together. Once we have the data, it takes a bit of work to trim off the inline commenting and to keep the information from each key line, but since the header is standardized, we know what to expect and can program around it.

```
data pgms;
   infile "%grabpath\*.sas" lrecl = 200 pad; ❶

   length progline $200;
   input progline $char200.;

   format example model tool fuel destination $200.;
   retain example model tool fuel destination;

   progline = compress(upcase(progline),'/*'); ❷

   if substr(progline,1,6)='HOW EX' then example=progline; ❸
   else if substr(progline,1,6) ='MODEL:' then model=progline;
   else if substr(progline,1,5)='TOOL:' then tool=progline;
   else if substr(progline,1,5)='FUEL:' then fuel=progline;
   else if substr(progline,1,12)='DESTINATION:' then
                                        destination=progline;

   drop progline;
   if progline='END OF PROGRAM' then output pgms;❹
run;
```

❶ %GRABPATH is used to determine the path to the SAS programs. This effectively concatenates all the SAS programs in this directory.

❷ Each line of code is read and comment symbols are removed.

❸ The value is examined for keywords in the standard header.

❹ When we reach the string “END OF PROGRAM” (comment symbols have been removed) we output the record, which will contain the retained values from the SAS program header.

The dataset, PGMS, contains variables that summarize what we have done, and where we are about to go. If we change any of the program headers (and save the program) we can repeat the process and get updated information. There are several practical uses for this, beyond generating signoff sheets. This could be used to keep tabs on validation progress, or to create a “table of contents” for macros in a macro library. The possibilities are only limited by imagination.

example	model	tool	fuel	destination
HOW EXAMPLE 1: PROC FORMAT	MODEL: FORMAT	TOOL: EXCLUSIVE OPTION IN PROC MEANS	FUEL: FORMAT DEFINITION	DESTINATION: CONTROL THE CLASS LEVELS IN THE OUTPUT
HOW EXAMPLE 2: WRITING A LABEL STATEMENT	MODEL: HORIZONTAL LIST	TOOL: SELECT INTO	FUEL: SASHELP VIEW: VCOLUMN	DESTINATION: LABEL STATEMENT
HOW EXAMPLE 3: USE A DATA DICTIONARY TO SELECT VARIABLES TO READ, PRINT, AND SUMMARIZE	MODEL: HORIZONTAL LIST AND VERTICAL LIST	TOOL: SELECT INTO	FUEL: DATA DICTIONARY	DESTINATION: LISTING AND SUMMARY STATISTICS
HOW EXAMPLE 4: WRITE PATIENT SUMMARIES	MODEL: HORIZONTAL LIST	TOOL: SELECT INTO	FUEL: THE DATA ITSELF	DESTINATION: WRITE ONE REPORT PER PATIENT
HOW EXAMPLE 5: STACK DATASETS FROM A LIBRARY	MODEL: VERTICAL LIST	TOOL: DATA STEPCALL SYMPUTX	FUEL: INFILE STATEMENT	DESTINATION: OBTAIN A CUMULATIVE DATASET
HOW EXAMPLE 6: GET A LIST OF SAS PROGRAMS AND THEIR PURPOSE	MODEL: INFILE STATEMENT	TOOL: ???	FUEL: SAS PROGRAM HEADERS	DESTINATION: READ YOUR PROGRAM HEADERS AND REPORT ON THEM
HOW EXAMPLE 7: PRINT ALL OF THE DATASETS IN A LIBRARY, IN LOGICAL ORDER	MODEL: WRITE THE MACRO CALL	TOOL: SELECT INTO, CATX	FUEL: DICTIONARY TABLE, FORMAT	DESTINATION: EXCEL OUTPUT
HOW EXAMPLE 8: CALL THE SUMMARY MACRO FOR ALL VARIABLES IN A DATASET	MODEL: WRITE MACRO CALL	TOOL: SELECT INTO, CATX	FUEL: PROC CONTENTS	DESTINATION: CALL SUMMARY MACRO, INPUT VARIABLE NAME, TYPE AND LABEL
HOW EXAMPLE 9 PRINT ALL OF THE DATASETS IN A LIBRARY IN LOGICAL ORDER	MODEL: WRITE MACRO CALL AND EXECUTE AFTER A DATA STEP	TOOL: CALL EXECUTE	FUEL: SASHELP TABLES	DESTINATION: EXCEL OUTPUT
HOW EXAMPLE 0: BUILDING LISTS	MODEL: PRACTICE	TOOL: SELECT INTO, CALL SYPMPUTX	FUEL: SASHELP.CLASS DATA	DESTINATION: EXPLORE SEVERAL WAYS TO BUILD A LIST

EXAMPLE 7: PRINT ALL OF THE DATASETS IN A LIBRARY, IN LOGICAL ORDER

***model**: write the macro call*

***tool**: SELECT INTO, CATT*

***fuel**: SASHELP view, format*

***destination**: excel output*

It is often desirable to create listings for all of the datasets in a library. We do this all the time for the permanent analysis datasets that are created in a program suite, so that the study team can understand the data that make up the summary statistics. When hard codes are performed we can compare the data before and after using PROC COMPARE, and we can save the output datasets from this procedure into a hard code library where they can be printed for review. The ExcelXp TAGSET provides a fast and easy way to create professional-looking output in Excel. For a very accessible introduction to this, see DelGabbo, 2012.

From what we have already discussed, you probably have some ideas about how you might address the task of printing all of the data from a library dynamically. So now, let's add another layer of customization to it. We are often asked to print that data in a logical order. In Excel, this means that the leftmost tab might contain the information on the analysis cohort, and then the next tab might contain baseline data, followed by treatment data, and then towards the right we would present the exit data. Of course, we could write a macro to do this, and we could write explicit calls to that macro. The first call to PROC REPORT would create the leftmost tab, and then each subsequent call would add a tab to the right. However, if new datasets come into the library, we will have to remember to write macro calls for them, which we don't want to do.

PROC FORMAT once again serves as our control vehicle. We create the format RAWDS which will take the names of the datasets in our library, and assign them to formatted values which we can order. We will gather our list of dataset names from SASHELP.VMEMBER, which is a view that contains one record for each dataset in each library within the SAS session. Notice that the dataset names, stored in the variable MEMBER, are in all caps. We need to maintain this case in our format definition.

```
proc format;
    value $ rawds
       'CARS'='a'
       'SHOES'='b'
       'DEMOGRAPHICS'='c'
        other=z;
run;
```

We will be working with a pretty simple PROC REPORT call for this example. We place the call into the macro program %REPORT, and specify one input parameter, &DSN, which is where the dataset name will be specified. We will reference this name in the SHEET_NAME option, the TITLE statement, and in the DATA= statement.

```
%macro report(dsn=);
  ods tagsets.ExcelXP options(sheet_name="&dsn");
  title  j=l "Hard Codes for &dsn Data";
  proc report data=SASHELP.&dsn headline missing nowd;
  run;
%mend report;
```

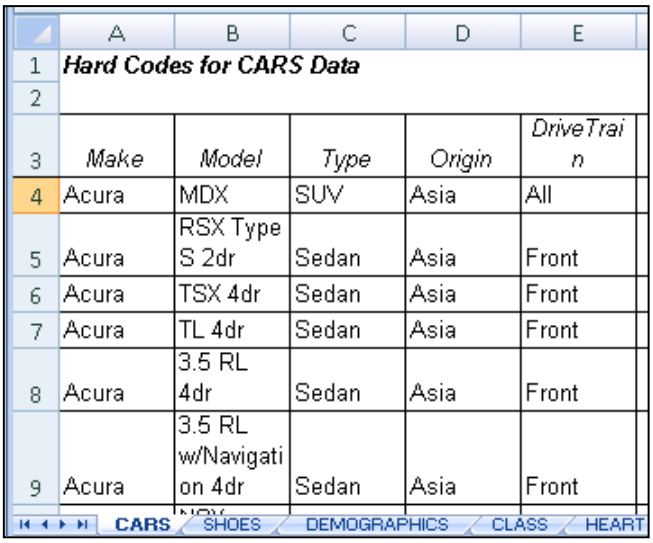
Next, we will use PROC SQL to accomplish two tasks. First, we will create the table _TEMP which will contain one variable, MEMNAME. We will limit our records to the datasets CARS, DEMOGRAPHICS, CLASS, SHOES, and HEART from the SASHELP library. Not all of these datasets are explicitly assigned to the format, above. However, we have added OTHER to our format definition, so HEART and CLASS should be formatted as 'z'. The most important part of this step is the ORDER BY clause. ❶ We use this to order the values of MEMNAME, however the order is based on the formatted values that we have created. The resulting table, _TEMP, will have the unformatted names of the data sets, however they will have been ordered by their formatted values.

```
PROC SQL noprint;
  create table _temp as
  select memname
  from sashelp.vmember
  where upcase(libname)='SASHELP' and
       memname in('CARS', 'DEMOGRAPHICS',
              'CLASS', 'SHOES', 'HEART')
  order by put(memname, $rawds.); ❶

  SELECT catt('%report(dsn=', memname, ')')❷
       into :reptmac separated by ' ' ❸
          FROM _temp;
QUIT;
```

Next, we use the second SELECT statement to read the names of the data sets. The CATT function ❷ is used to write the successive %REPORT macro calls that include the value of MEMNAME for the parameter value of DSN. Since the names are ordered values of MEMNAME, the resulting macro calls will also be in the appropriate order. We use the INTO clause to create the global macro variable &REPTMAC, ❸ which will hold these space-delimited macro calls, in the formatted order. After submitting this code we can run "%PUT _USER_;" to view the value of &REPTMAC. If you forget to add the NOPRINT option to PROC SQL, the values of MEMNAME in the _TEMP dataset will show in the results viewer, which you may find helpful.

```
GLOBAL REPTMAC %report(dsn=CARS) %report(dsn=SHOES) %report(dsn=DEMOGRAPHICS)
%report(dsn=CLASS) %report(dsn=HEART)
```

Even though we did not explicitly define format values for CLASS and HEART, we still generated macro calls for the use of those data sets. Those calls are placed at the end of the string, since they fell into the "other" category in the format and were assigned a formatted value of 'z'.



	A	B	C	D	E
1	***Hard Codes for CARS Data***				
2					
3	*Make*	*Model*	*Type*	*Origin*	*DriveTrain*
4	Acura	MDX	SUV	Asia	All
5	Acura	RSX Type S 2dr	Sedan	Asia	Front
6	Acura	TSX 4dr	Sedan	Asia	Front
7	Acura	TL 4dr	Sedan	Asia	Front
8	Acura	3.5 RL 4dr	Sedan	Asia	Front
9	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front

CARS | SHOES | DEMOGRAPHICS | CLASS | HEART

```
ods tagsets.ExcelXP path="C:\"
    file="SASHELP Data Listings.xml"
    style=Journal;

ods tagsets.ExcelXP
      options(embedded_titles='yes'
        embed_titles_once='yes'
        suppress_bylines='yes'
        sheet_interval='None'
        sheet_label='none'
        orientation='landscape'
        autofit_height='yes');

  &reptmac

ods tagsets.ExcelXP close;
```

Now that we have everything ready, we can just set up the ODS sandwich, add some options for ExcelXp, and reference the macro variable &REPTMAC. Since we are not using a %DO loop, we do not need to place this inside of a macro.

EXAMPLE 8: CALL THE SUM1 MACRO FOR ALL VARIABLES IN A DATASET

***model**: write macro call*

***tool**: select into, CATX*

***fuel**: proc contents*

***destination**: call summary macro, input variable name, type and label*

In the paper "Best Practices: Subset Without Getting Upset", (Rosenbloom and Lafler, 2014), a macro, %SUM1, is discussed. This macro incorporates many of the techniques demonstrated in that paper, tying them all together into one routine. The macro must be called once for every variable of interest, and parameters, which must all be specified, include: the input dataset name, a subsetting WHERE clause, a corresponding string to explain the WHERE clause (used in titles), the variable label, and format. Each time the analysis is run, the results are appended to a dataset and can be displayed with BY group processing techniques. An example call is shown here.

```
%sum1(dsn=heart2,
where=not missing(status),
titlewhere=All Records,
var=age_group,
label=Generation,
format=$generation)
```

All of this typing only produces analysis for one variable. There are a lot of ways to mess that up! Furthermore, the paper aims to demonstrate techniques that help when one is asked to subset analysis on the fly. If we want to repeat the analysis for a subgroup, writing one long macro call per variable again – even with cut and paste – is error-prone and a poor use of time.

Instead, we can use the macro %SUMMARY, which writes a macro call for each categorical variable (excluding SEX) in the input dataset, using information from PROC CONTENTS to fill in the values for &LABEL, &FORMAT, and &VAR. Furthermore, the values are sorted by the variable VARNUM, the order of the variables on the program data vector (PDV). This results in the macro calls also following the order of the variables on the PDV, and the resulting output will be ordered from top to bottom following the left to right order of the variables on the PDV.

```
%macro summary(dsn=, where=, titlewhere=,);
proc contents data=&dsn out=_look ❶
  (where=(type=2 and name ne 'Sex')) noprint;
run;

proc sort data=_look out=_look2;
   by VARNUM; ❷
run;

data _null_;
   set _look2 end=last;

   /*Create call to macro*/ ❸
   format callit $2000.;
       callit=catt(
              '%sum1(dsn=',
              "&dsn",
              ", where=&where",
              ', var=',
              name,
              ', label=',
              label,
              ', format=',
              format,
              ')'
   );
   put callit; ❹

   call symputx(catt('var',_N_), callit, 'L');❺

   if last then
       call symputx('totvars',left(_n_), 'L');❻
run;

%do i=1 %to &totvars;❼
      &&var&i
%end;
%mend summary:
```

The macro starts with a call to PROC CONTENTS, outputting the dataset _LOOK, which will contain the variable name, label, format, and other metadata, in the form of one record per variable from the dataset specified with &DSN. ❶ This dataset is sorted by VARNUM, which contains the order of the variables on the PDV. ❷ Calls to the macro %SUM1 are built using the CATT function within the DATA step. ❸ These macro calls are displayed in the log with the PUT statement, ❹ and are also assigned to the macro variables &VAR1, &VAR2, etc using CALL SYMPUTX. They are placed in the local symbol table when we specify the value 'L' in the third argument of CALL SYMPUTX. ❺ The overall count of macro variables in the vertical list is assigned to &TOTVARS, which is also in the local symbol table. ❻ We process the vertical list of macro variables, thus calling the %SUM1 macro, using the %DO loop. ❼ When we call the %SUMMARY macro, the call is much simpler. We specify the input dataset name, WHERE clause, and corresponding descriptive text string. Calls for three different data groups are shown here.

```
%summary(dsn=heart2, where=not missing(status), titlewhere=All Records)
%summary(dsn=heart2, where=status='Dead', titlewhere=Expired Subjects)
%summary(dsn=heart2, where=age_group='Traditionalist',
      titlewhere=Traditionalist Subjects)
```

The log (truncated for display purposes) shows the calls that are generated to %SUM1. We have limited the number of variables on the HEART2 dataset for discussion purposes.

```
%sum1(dsn=heart2, where=not missing(status), var=smoking_status, label=Smoking
Status, format=$SMOKING_STATUS)
%sum1(dsn=heart2, where=not missing(status), var=status, label=, format=$STATUS)
%sum1(dsn=heart2, where=not missing(status), var=age_group, label=,
format=$GENERATION)
%sum1(dsn=heart2, where=status='Dead', var=smoking_status, label=Smoking Status,
format=$SMOKING_STATUS)
%sum1(dsn=heart2, where=status='Dead', var=status, label=, format=$STATUS)
%sum1(dsn=heart2, where=status='Dead', var=age_group, label=, format=$GENERATION)
%sum1(dsn=heart2, where=age_group='Traditionalist', var=smoking_status,
label=Smoking Status, format=$SMOKING_STATUS)
%sum1(dsn=heart2, where=age_group='Traditionalist', var=status, label=,
format=$STATUS)
%sum1(dsn=heart2, where=age_group='Traditionalist', var=age_group, label=,
format=$GENERATION)
```

So we can see that quite a bit of time and typing has been saved.

EXAMPLE 9: PRINT ALL OF THE DATASETS IN A LIBRARY, IN LOGICAL ORDER - REVISITED

***model**: write macro call and execute after a data step*

***tool**: CALL EXECUTE*

***fuel**: SASHELP tables*

***destination**: excel output*

In example seven we created a macro, %REPORT, and then wrote and ordered calls to %REPORT using PROC SQL. This allowed us to print all of the datasets from a library in a logical order. There is another way to accomplish that task within the DATA step, and it uses CALL EXECUTE. CALL EXECUTE allows us to generate macro calls within the DATA step, and these macro calls are executed immediately after the DATA step is finished. We would still like to order the macro calls in a meaningful way, as indicated in our FORMAT definition in Example 7. In this case, we can use the dataset _TEMP from Example 7, which contains all of the dataset names, ordered by the formatted values.

Instead of creating a macro variable that contains a series of macro calls, in this example we will issue the series of macro calls directly using the DATA step's CALL EXECUTE routine.

In this example CALL EXECUTE is used with CATT to build the macro call. As a result we construct a series of macro calls within the DATA step (one for each incoming observation. These macro calls are held in a stack for execution after the DATA step terminates.

```
options mprint;
ods tagsets.ExcelXP path="C:\"
file="SASHELP Data Listings.xml" style=Journal;

data _null_;
   set _temp;
   call execute(catt('%report(dsn=', memname, ')'));
run;

ods tagsets.ExcelXP close;
```

The entire DATA step is embedded within the ODS sandwich so that the output that we create will go to Excel. Each macro call will execute after the DATA step terminates.

Part of the log is shown below. You can see that the macro was called for the CARS data, and the sheet name and DATA= fields were populated with the value “CARS”. Only the portion of the LOG associated with the CARS data set is shown here.

```
NOTE: CALL EXECUTE generated line.
1    + ods tagsets.ExcelXP options(      embedded_titles='yes'
embedded_footnotes='yes'      embed_titles_once='yes'
suppress_bylines='yes'
sheet_interval='None'      sheet_name="CARS"      sheet_label='none'
frozen_headers='3'
2    + frozen_rowheaders='11'      orientation='landscape'
row_repeat='3'      autofit_height='yes');   title  j=l "Hard Codes for CARS
Data";   proc report
data=SASHELP.CARS headline missing nowd;   run;

NOTE: Multiple concurrent threads will be used to summarize data.
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: PROCEDURE REPORT used (Total process time):
      real time           1.37 seconds
      cpu time            1.28 seconds
```

SUMMARY

This paper has reviewed several concepts for building macro variable lists. Most of these methods have been introduced previously in other papers, however through the examples shown in this paper we are hoping to make these methods more easily understood. The methods discussed for building a macro list include the use of:

--SQL INTO:
--CALL SYMPUTX

Information driven programs can derive the needed knowledge from a variety of sources. These sources provide us control and that control can appear in various forms. In this paper we have seen examples using several kinds of control files:

--data dictionary
--proc contents output
--proc datasets with ODS or SASHELP tables/dictionary views
--the data itself
--OS environmental variables

When control files are used to power the code, then the code self-updates. When new study centers are introduced into the data, the program detects them in the data and creates macro calls. By using control files and data to write your program, the programming environment becomes more flexible and more nimble. When the data dictionary is modified, the analysis is also modified. In this way, the program constantly fits the data that it processes, rather than the data fitting the program. In a regulated environment, SAS programs must be validated for every change made. By writing data-driven code, we can eliminate this need.

Our programs are our vehicles that we drive to generate analyses and reports. Imagine a car equally suited for taking the family to the beach, racing on the speedway, and 4 wheeling in the backcountry. Your programs can be this type of car if you take control, if you become a control freak.

ABOUT THE AUTHORS

Mary Rosenbloom is a statistical programmer at Edwards Lifesciences in Irvine, California. She has been using SAS for over 15 years, and is especially interested in using macros to generate data-driven code, DDE, and program validation methods.

Art Carpenter's publications list includes five books, and numerous papers and posters presented at SUGI, SAS Global Forum, and other SAS user group conferences. Art has been using SAS® since 1977 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Advanced Professional programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Mary F. O. Rosenbloom
Edwards Lifesciences, LLC
One Edwards Way
Irvine, CA 92614

949 250-2281
mary.rosenbloom.sas@gmail.com

View Mary Rosenbloom's paper presentations page at:
http://www.sascommunity.org/wiki/Presentations:Otterm1_Papers_and_Presentations

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

View Art Carpenter's paper presentations page at:
http://www.sascommunity.org/wiki/Presentations:ArtCarpenter_Papers_and_Presentations

ACKNOWLEDEMNENTS

Mary would like to thank Art for collaborating on this fun and challenging work. She would also like to thank him for being an enthusiastic mentor and friend.

REFERENCES

Carpenter, Arthur L., 1997, "Better Titles: Using the #BYVAR and #BYVAL Title Options", presented at the 5th Western Users of SAS Software, WUSS, meetings (October, 1997) and published in the conference proceedings. Also presented at the 23rd SAS User's Group International, SUGI, meetings (March, 1998) and published in the Proceedings of the Twenty-Third Annual SUGI Conference, 1998.

Carpenter, Arthur L., 2003, "Building and Using User Defined Formats", Proceedings of the 11th Annual Western Users of SAS Software, Inc. Users Group Conference, Cary, NC: SAS Institute Inc. Also in the proceedings of the Twenty-ninth SAS User Group International Conference (SUGI), 2004, Cary, NC: SAS Institute Inc.

Carpenter, Arthur L., 2004a, *Carpenter's Complete Guide to the SAS® Macro Language, 2nd Edition*, Cary, NC: SAS Institute Inc.

Carpenter, Arthur L., 2004b, "Storing and Using a List of Values in a Macro Variable", Proceedings of the 12th Annual Western Users of SAS Software, Inc. Users Group Conference (WUSS), Cary, NC: SAS Institute Inc. Also in the proceedings of the Thirtieth SAS User Group International Conference (SUGI), 2005, Cary, NC: SAS Institute Inc., paper 028-30, and in the proceedings of the Pharmaceutical SAS User Group Conference (PharmaSUG), 2005, Cary, NC: SAS Institute Inc.

Carpenter, Arthur L., 2005, "Make 'em %LOCAL: Avoiding Macro Variable Collisions", proceedings of the Pharmaceutical SAS User Group Conference (PharmaSUG), 2005, Cary, NC: SAS Institute Inc., paper TT04.

Carpenter, Arthur L., 2008, "The Path, The Whole Path, Nothing But the Path, So Help Me Windows", presented at the SAS Global Forum 2008 Conference in San Antonio, Texas (paper 023-2008).

Carpenter, Arthur L., 2009, "Manual to Automatic: Changing Your Program's Transmission". Presented at the 13th Annual Western Users of SAS Software Conference, WUSS, Cary, NC: SAS Institute Inc., paper APP-Carpenter. Also presented at the Vancouver SAS Users Group, 2010, and the PharmaSUG conference, 2010, paper AD25.

Carpenter, Art, 2012, Carpenters Guide to Innovative SAS® Techniques, SAS Press, SAS Institute Inc., Cary NC.

Fehd, Ronald and Art Carpenter, 2007, "List Processing Basics: Creating and Using Lists of Macro Variables" by Ronald Fehd and Art Carpenter which was presented at the 2007 SAS Global Forum (Paper 113-2007). The discussion of the paper looks at different approaches used in the automation of programs by using various kinds of macro variable lists. This paper appears in proceedings of a number of conferences, including: SASGF(2007), WUSS (2008), MWSUG (2009), SESUG (2009).

Michel, Denis, 2005, "CALL EXECUTE: A Powerful Data Management Tool", presented at SUGI 30 (Paper 027-30), this paper includes a number of references to other CALL EXECUTE papers.

Lafler, Kirk Paul, 2010, "Exploring DICTIONARY Tables and SASHELP Views", presented at SAS Global Forum 2010, Cary, NC: SAS Institute Inc.

DelGobbo, Vincent, 2012, "An Introduction to Creating Multi-Sheet Microsoft Excel Workbooks the Easy Way with SAS", presented at SAS Global Forum 2012, Cary, NC: SAS Institute Inc.

Rosenbloom, Mary F. O and Lafler, Kirk Paul, 2012 "Assigning a User-Defined Macro to a Function Key" ", presented at SAS Global Forum 2012, Cary, NC: SAS Institute Inc.

Rosenbloom, Mary F. O and Lafler, Kirk Paul, 2014 "Best Practices: Subset Without Getting Upset" ", presented at SAS Global Forum 2012, Cary, NC: SAS Institute Inc.

TRADEMARK INFORMATION