PharmaSUG 2012 – Paper TF05

Interesting technical mini-bytes of Base SAS® – From Data step to Macros

Airaha Chelvakkanthan Manickam, Cognizant Technology Solutions, Teaneck, NJ

ABSTRACT:

Over the last several decades, SAS® has improved and added thousands of features to Base SAS®. It is almost impossible for someone to know all of the tricks and tips of Base SAS®. This paper highlights some useful tips and advanced tricks of Base SAS® that I have encountered during my experience working with SAS®. These technical mini-bytes are divided into two sections—DATA step mini-bytes and macro mini-bytes. In this paper, I present interesting examples involving common and advanced DATA step executions such as CALL SET, dynamically accessing SAS® data sets, and PERL pattern matching. Similarly, simple to advanced macro tips including CALL EXECUTE and QUOTE functions in macros are discussed.

INTRODUCTION:

Base SAS® consists of hundreds of functions and statements for efficient data processing. It is practically impossible for a user to know every aspect of each function and statement. This paper discusses few tips and tricks of certain rarely known aspects of base SAS® that users may find interesting during their experience with SAS®. Many sections of this paper were discussed among the SAS-L user community as discussions originated by the author. Every section of this paper is assigned with one or many of the following logos that describes the nature of the tip discussed in it.

- Error Handling – provides solution to commonly occurring errors
- Problem Solving – provides an easy way of solving a problem
- Knowledge Gain – Information not commonly known to basic SAS® users
- Performance Improvement – tip to improve the performance of a situation

All examples in this paper are considered from pharmaceutical industry, though the same solution can be applied to any industry which faces relevant situation.

INTERESTING MINI-BYTES OF SAS® DATA STEP PROCESSING:

LETS START WITH *_ALL_*:

all is an interesting automatic variable in base SAS® which helps you capture all variables in a data step – both user defined and automatically defined. However, how many of us tried to use it in an array declaration and failed?

```
Array abc{*} _character_;   - valid
Array def{*} _numeric_;     - valid
Array def{*} _all_;         - invalid.
```

The reason is that all array variables must be of same data type, either all character or all numeric and cannot be both.

WELL KNOWN PROBLEM – DATA STEP STOPPED DUE TO LOOPING:

Many of us would have come across the following note “accidentally”.

```
NOTE: DATA STEP stopped due to looping.
```

The reason for this note is that if a DATA step is constructed such that no data reading statements (e.g. SET, INPUT) are executed (even though they exist within the data step), the step is terminated just after one iteration and this note is written to the SAS® log.

For example, in the following DATA step, the INPUT statement is conditionally executed. Due to the IF condition, there is no data read during first iteration and the step terminates with the note as shown in Log Output (1).

```
DATA DS_One;
If _n_ > 1 then do;
   Infile 'sample.txt';
   Input a $2.; Put a=;
End;
Put "Will I print it? " _n_=;
Run;
```

```
Will I print it? _N_=1
NOTE: DATA STEP stopped due to looping.
NOTE: 0 records were read from the infile sample.txt.
```

Log Output 1

VARIABLE LISTS AND FORMAT LISTS – ALWAYS INTERESTING:

Variable list is an abbreviated method of referring to a list of variable names. For example, `_NUMERIC_` and `_CHARACTER_` are variable lists that refer to all numeric and character variables in a data step. Format list is an easy way of assigning different formats to different variables.

Following is an interesting example of how variable list and format list can be used in creating comma separated value (CSV) files in SAS® with few lines of code.

```
DATA _NULL_;
     Set DS_One;
     Put (_NUMERIC_) (z4. ',') @;
     Put ',"' @;
     Put (_CHARACTER_) ($8. '","')@;
     Put '"';
 Run;
```

```
0001,0011,0009,"aa      ","sample1 "
0022,0222,0099,"bbb     ","sample2 "
0003,0003,0999,"cccc    ","sample3 "
```

Log Output 2

BY GROUPING READS ONE EXTRA OBSERVATION. ISN'T IT CALLED STEALING?

Whenever BY grouping is used in a data step, it performs a "look-ahead" – means it reads one extra observation. It can be proved using the following example.

```
DATA DS_Sample1;
Input Sum_Var
Product;
Cards;
100 3
100 2
100 1
;
DATA DS_Sample2;

  Set DS_Sample1;
  by Sum_Var;
  cnt+1; If CNT > 1 then stop;
Run;
```

```
NOTE: There were 3 observations read from the data set WORK.DS_SAMPLE1.
NOTE: The data set WORK.DS_SAMPLE2 has 1 observations and 3 variables.
```

Log Output 3

Note that even though the data step has to stop after 2 iterations, it has passed 3 iterations. The reason being SAS® has to **"**look-ahead**"** to set the values for the automatic variables FIRST. and LAST. associated with the BY variables. We can confirm this behavior of BY grouping by running the same data step without BY group – as given below.

```
DATA DS_Sample2;
  Set DS_Sample1;
  cnt+1;
  If CNT > 1 then
stop;
Run;
```

```
NOTE: There were 2 observations read from the data set
WORK.DS_SAMPLE1.
NOTE: The data set WORK.DS_SAMPLE2 has 1 observations and 3 variables.
```

Log Output 4

As expected, only two observations are read as there is no "look-ahead" is performed without any BY grouping.

MULTI-WAY RANDOM ACCESS WITH MULTIPLE SET WITH KEY STATEMENTS:

The SET statement in a data step allows the user to read the observations sequentially from a data set. The key= option of SET statement is special and it helps to randomly select an observation from a data set based on the value assigned to the key variable. This option is commonly used to treat a data set as a look-up table.

Beyond the normal use of SET statement with key= option, there can be situations where multiple SET statements with key= option may be needed. Consider a scenario in the pharmaceutical industry where the master data set contains doctors' sales by zip code and the look-up data set contains the specialties of all doctors and the goal is to assign all specialties of a doctor from the look-up data set to the master data set.

```
DATA DS_Sales;  /* Sample Sales data (Master data set) */
  DR_ID = 1; ZIP_CODE=’07034’; SALES=100; output;
  DR_ID = 1; ZIP_CODE=’07045’; SALES=200; output;
  DR_ID = 2; ZIP_CODE=’19623’; SALES=300; output;
  DR_ID = 2; ZIP_CODE=’19764’; SALES=400; output;
Run;

DATA DS_Spec(index=(DR_ID)); /* Sample Specialties data (look-up data set) */
  DR_ID = 1; SPEC = ‘AA’; output;
  DR_ID = 1; SPEC = ‘BB’; output;
  DR_ID = 1; SPEC = ‘CC’; output;
  DR_ID = 2; SPEC = ‘XX’; output;
  DR_ID = 2; SPEC = ‘YY’; output;
Run;
```

In the following data step, master sales data set is read sequentially using a SET statement and look-up data set is accessed randomly inside a loop until all specialties of a doctor are obtained.

```
DATA DS_Sales_Spec;
Put "Starting iteration: " _n_=;
Set DS_Sales;
Do while (_IORC_ = 0);
  Set DS_Spec key = DR_ID;
  Select (_IORC_);
     When (0)       do; put "SPEC found.  " DR_ID= SPEC=; output; end;
     When (%sysrc(_DSENOM)) do; put "SPEC no more." DR_ID= SPEC=; end;
  End;
End;
_error_=0; _iorc_=0;
Run;
```

```
Starting iteration: _N_=1
SPEC found.  DR_ID=1 SPEC=AA
SPEC found.  DR_ID=1 SPEC=BB
SPEC found.  DR_ID=1 SPEC=CC
SPEC no more.DR_ID=1 SPEC=CC
Starting iteration: _N_=2
SPEC no more.DR_ID=1 SPEC=CC
Starting iteration: _N_=3
SPEC found.  DR_ID=2 SPEC=XX
SPEC found.  DR_ID=2 SPEC=YY
SPEC no more.DR_ID=2 SPEC=YY
Starting iteration: _N_=4
SPEC no more.DR_ID=2 SPEC=YY
Starting iteration: _N_=5
```

```
Obs    DR_ID    ZIP_CODE    SALES    SPEC
1        1      07034       100      AA
2        1      07034       100      BB
3        1      07034       100      CC
<== Sales for ZIP_CODE=07045 missing
4        2      19623       300      XX
5        2      19623       300      YY
<== Sales for ZIP_CODE=19764 missing
```

Log Output 5. PROC PRINT of DS_Sales_Spec

Log Output 6. Log trace of put statements

As shown in the Log Output (6) above, we are missing the 2nd occurrence of every doctor from the sales data set to resultant data set. Why? In order to obtain multiple specialties of a doctor, look-up data set DS_Spec is called within a loop; however once the loop is over for a doctor (DR_ID), in order to access the same doctor (DR_ID) again, the pointer needs to be re-set. If it is not reset, SAS® assumes end-of-search for the immediate subsequent records with the same key value (DR_ID).

To avoid this situation and to reset the point, multiple SET statements with key option can be used. Because all the key pointers are independent of each other, each pointer can traverse the look-up table from top to bottom independently. It is explained in the code segment below.

```
DATA _null_;
Set DS_Sales nobs=nobs;
Call Symput('Sales_Obs',nobs);
Run;

%Macro Expand(Sales_Obs);
%Do iter=1 %to &Sales_Obs;
  Put "Starting iteration: &iter";
  Set DS_Sales point=pos;
  Do while (_IORC_ = 0);
    Set DS_Spec key = DR_ID;
    Select (_IORC_);
       When (0)        do; put "SPEC found.  " DR_ID= SPEC=; output; end;
       When (%sysrc(_DSENOM)) do; put "SPEC no more." DR_ID= SPEC=; end;
    End;
  End;
_iorc_=0; pos=pos+1;
%end;
%Mend Expand;

DATA DS_Sales_Spec;
Retain pos 1;
%Expand(&Sales_Obs);
_error_=0;
Stop;
Run;
```

```
Starting iteration: 1
SPEC found.  DR_ID=1 SPEC=AA
SPEC found.  DR_ID=1 SPEC=BB
SPEC found.  DR_ID=1 SPEC=CC
SPEC no more.DR_ID=1 SPEC=CC
Starting iteration: 2
SPEC found.  DR_ID=1 SPEC=AA
SPEC found.  DR_ID=1 SPEC=BB
SPEC found.  DR_ID=1 SPEC=CC
SPEC no more.DR_ID=1 SPEC=CC
Starting iteration: 3
SPEC found.  DR_ID=2 SPEC=XX
SPEC found.  DR_ID=2 SPEC=YY
SPEC no more.DR_ID=2 SPEC=YY
Starting iteration: 4
SPEC found.  DR_ID=2 SPEC=XX
SPEC found.  DR_ID=2 SPEC=YY
SPEC no more.DR_ID=2 SPEC=YY
```

Log Output 8. Log trace of put statements from previous code

```
Obs    DR_ID    ZIP_CODE    SALES    SPEC
1      1        07034       100      AA
2      1        07034       100      BB
3      1        07034       100      CC
4      1        07045       200      AA
5      1        07045       200      BB
6      1        07045       200      CC
7      2        19623       300      XX
8      2        19623       300      YY
9      2        19764       400      XX
10     2        19764       400      YY
```

Log Output 7. PROC PRINT of DS_Sales_Spec

Multiple SET=key statements worked. No more missing observations.

Points to note:

- Multiple SET statements with key option may not be an efficient approach at all situations. If the master data set or the look-up data set is huge, this approach may take a long time to execute.
- As the %EXPAND macro may produce a large number of statements, it is advisable not to use MACROGEN and SYMBOLGEN options while using this approach.
- Another way of obtaining this result is to have PROC SQL with Cartesian join between Sales and Spec tables.

DATA STEP VIEWS: YOUR COMPANION TO SAVE TEMP SPACE:

We have used views in Oracle and other relational databases. How many of us know that base SAS® provides similar view functionality in data step processing? Views are basically stored compiled program statements. They don't contain physical data instead they contain the compiled code.

In SAS®' terms, a SAS® view is a type of SAS® data set that retrieves data values from other files. A SAS® view contains only descriptor information such as the data types and lengths of the variables (columns), plus information that is required for retrieving data values from other SAS® data sets or from files that are stored in other software vendors' file formats. SAS® views are of member type VIEW. In most cases, you can use a SAS® view as if it were a SAS® data file.

The major use of the views is to save memory and avoid memory overrun problems. This can be achieved by keeping large intermediate data sets in views. Most commonly, the intermediate data sets, if not needed for any data investigation purposes, formed by merging multiple huge data sets can be avoided by using views. This saves lot of money and space overhead issues in an organization.

Data step views are created either with a data step or PROC SQL. It works as an input data set to any PROC or DATA step that uses it. Whenever the data or proc step needs an observation, the view is executed and an observation is returned.

Consider the scenario where an enterprise contain 10GB of sales data in a SAS® data set called FACT and 4GB of reference data in a SAS® data set called REFERENCE. In a relational data warehouse, it is not allowed to combine FACT data with REFERENCE data. In order to provide the users a combine view of FACT with REFERENCE data, a SAS® view is created in the master SAS® library. As mentioned below, this SAS® view contains only compiled code and not the actual data.

It provides multiple benefits. 1) It does not occupy any space 2) It provides a "virtual" image of the FACT + REFERENCE combined data to the users; so the users need not to go through the hassle of combining them 3) Any business rules to be applied while combing FACT + REFERENCE data is taken care within the view and hidden from the users. Lets see the template of such a view below.

```
libname REF1 '....\SAS® Data set Library';
libname REF2 '....\SAS® Master Library';

DATA REF2.COMBINED_FACT_REFERENCE / VIEW=REF2.COMBINED_FACT_REFERENCE;
    merge REF1.FACT REF1.COMBINED;
    by PRIMARY_KEY;
run;
```

The view `COMBINED_FACT_REFERENCE` is published to all users. If user wants any data, he/she can use the view as if it is a SAS® data set.

```
libname REF2 '....\SAS® Master Library';
libname REF3 '....\User Library';

DATA REF3.PRODUCT1_FACT (keep=PRODUCT_FAMILY SALE_OF_LAST_YEAR);
set REF2.COMBINED_FACT_REFERENCE (where=(PRODUCT_FAMILY='XXX' and INTRO_DATE =
'31JAN2012'd and ACTIVE_FLAG = 'Y'));
/* User performs conditional checks on REFERENCE fields and pull FACT fields in
output */
run;
```

Lets consider another example where VIEW can be useful. Assume that the source data to a SAS® data mart is huge in size and also frequently changes (say every 4 hours). Several users want, not whole, but a segment of this source data for their analytical calculations. In this situation, storing the frequently changing, large in size, source data to a permanent SAS® data set unnecessary consumes space. Instead, a SAS® view can be created on top of this source data. In the pharma industry, such a frequently changed data can be prescriber dimension data from a Master Data Management (MDM) system. Lets see how VIEW can help with to utilize this data.

```
Libname REF1 '....\SAS® Master Library';

DATA REF1.CUST_DIM / VIEW=REF1.CUST_DIM;
    infile '..../Source File' dsd;
    input Unique_Key : $5.
             Prescriber_Id : $3.
```

```
                Prescriber_Name : $50.
                Address_1 : $100.
                Address_2 : $100.
                City : $30.
                State : $2.
                Zip : 5.
                Specialty : $3.;
       if Unique_Key NOT = '';
       if State = 'NH' then Specialty = '999';
       If address_1 = '' and address_2 NOT = '' then do; address_1 = address_2;
   address_2 = ''; end;
       /* any other business rules */
   run;
```

- The view CUST_DIM will not store the physical data; instead it gets data from the frequently modified source file whenever this view is called by anyone. So the need of storing the frequently modified source data to a physical SAS® data set is avoided and it saves space.
- The complicated business rules that need to be applied on the source data are inside the view and completely hidden from the users.

% INCLUDE – IS NOT A MACRO STATEMENT – NEED FOR DOUBLE SEMICOLON:

%INCLUDE is not a macro statement. It is a data step statement which is executed at compile time. The common use of it is to bring external files or commonly stored routines and business rules into the processing of a data step. However the users should be careful when %INCLUDE is used alongside a macro. It may cause trouble due to the way semi-colons are consumed by macros. Lets perceive this with an example:

```
%Macro Mac1(var1=);
  %if &var1=1 %then
  %include 'sample.txt';
  Put "And I am following: ";
%Mend;
DATA _null_;
   %Mac1(var1=1)
Run;
```

Assume that the content of sample.txt is:

```
put "I am inside";
```

```
52   data _null_;
53   %mac1(var1=1)
ERROR: Invalid file, AND I AM FOLLOWING:.
ERROR: Incorrect %INCLUDE statement will not be executed. There is a syntax
error.
54   run;
```

Log Output 9

The semicolon after the %INCLUDE statement is consumed by the %IF statement of macro, which caused the error in Log Output (9) while constructing the data step.

In order to correct it, all that we need is 2 semicolons. Simple!

```
%Macro Mac1(var1=);
  %if &var1=1 %then
  %include 'sample.txt';; <== Two semi colons
 Put "And I am following: ";
%Mend;
DATA _null_;
   %Mac1(var1=1)
Run;
```

```
I am inside
And I am following:
```

Log Output 10

CALL SET AND DYNAMICALLY ACCESSING SAS® DATA SETS:

CALL SET routine helps the user to have complete control over a SAS® data set. The user gets the ability to pin point to what he needs from a SAS® data set and make use of only that data instead of reading the entire content and loosing the control over a SAS® data step. The other important feature of CALL SET routine is its ability to access SAS® data sets from within a macro. Along with CALL SET routine, the dynamic nature of macro facility helps the user to build an efficient and powerful program. Lets see it with an example.

Consider a scenario in a pharmaceutical industry where a master parameter data set lists all products and the respective reports to be printed. In order to process these reports, CALL SET with macro can help to build an efficient program.

```
DATA DS_Master_Parameter;
Length Type $10 Value $300;
/* TYPE - Metadata variable that describes type of action to be taken
   Value - value assigned to Metadata variable TYPE */
Type = 'PRODUCT'; Value = 'PRD1'; output;
Type = 'BUS_RULE'; Value = 'Sales > 5000'; output;
Type = 'REPORT'; Value = 'INCENTIVE_REPORT'; output;
Type = 'PRODUCT'; Valye = 'PRD2'; output;
Type = 'BUS_RULE'; Value = 'MON1_UNITS > 1000 and MON24_UNITS > 1000'; output;
Type = 'REPORT'; Value = 'VALIDATION_REPORT'; output;
Run;
```

The macro MASTER_PROCESS works as a command center to process the business reports for each product based on the business rules that need to be applied.

```
%macro MASTER_PROCESS;
%let open_dsid=%sysfunc(open(DS_Master_Parameter));
%let total_obs=%sysfunc(attrn(&open_dsid,nobs));
%do obs_no=1 %to &total_obs;
   %syscall set(open_dsid);
   %let fetch_rc=%sysfunc(fetch(&open_dsid));
   %put Type= &Type Value= &Value;
   %if &Type = PRODUCT %then %let Product = &Value;
   %if &Type = BUS_RULE %then %do;
           /* Build WHERE conditions to include business rules for reports */
   %end;
   %if &Type = REPORT %then %do;
           /* Build statements for each type of report being processed */
   %end;
%end;
%let close_rc=%sysfunc(close(&open_dsid));
%mend MASTER_PROCESS;

%MASTER_PROCESS;
```

CALL SET and Variable Declaration:

Lets take a look at an interesting difference in using CALL SET inside a data step and inside a macro. The difference is that the usage of CALL SET within a data step mandates defining all variables that are part of the data set referred by CALL SET routine whereas the same restriction doesn't apply to macros. Following example explains it.

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83.0

Table 1 Sample content of data set SASHELP.CLASS

```
DATA _null_;
  Retain name ' ' sex ' ';
  Rc=open('SASHELP.CLASS');
  Call set(rc);
  Rc1=fetchobs(rc,1);
  Rc2=sysmsg();
  Rc3=curobs(rc);
  Height=getvarn(rc,4);
  Rc4=close(rc);
  Put 'Returned Variables : ' name= sex= Height=;
  Put 'Return Codes         : ' Rc2= Rc3=;
Run;
```

```
Returned Variables : name=  sex=  Height=69
Return Codes       : Rc2=ERROR: ERROR: DATA
step variable Age not defined.. Rc3=1
```

Log Output 11. Output of data step

As evident from Log Output (11), the data step expects the user to declare (or) use all variables that are sourced from SASHELP.CLASS data set. In other words, all variables sourced from SASHELP.CLASS must be part of the PDV (Program Data Vector) of the data step.

From SAS®9.2 onwards, this error can be avoided by using NOSET option of FETCHOBS function.

Lets perform the same CALL SET, but within a macro.

```
%let name=;
%let sex=;
%let rc=%sysfunc(open(SASHELP.CLASS));
%syscall set(rc);
%let rc1=%sysfunc(fetchobs(&rc,1));
%let rc2=%sysfunc(sysmsg());
%let rc3=%sysfunc(curobs(&rc));
%let Height=%sysfunc(getvarn(&rc,4));
%let rc4=%sysfunc(close(&rc));
%put Returned Variables : name=&name sex=&sex Height=&height;
%put Return Codes       : RC2=&Rc2 Rc3=&Rc3;
```

```
Returned Variables : name=Alfred   sex=M Height=69
Return Codes       : RC2= Rc3=1
```

Log Output 12

As evident from Log Output (12), there is no variable-not-defined error returned while using CALL from within a macro. Also note that the macro processor has assigned values to macro variables “name” and “sex” even though they are not explicitly assigned.

From SAS®9.2 onwards, automatically assigning values to macro variables can be avoided by using NOSET option of FETCHOBS function.

PERL PATTERN MATCHING – HELPS TO WRITE CRISP PROGRAMS:

Pattern matching is the technique of searching a string or file containing text for specific set of characters based on a specific search pattern. The search pattern is called regular expression. In SAS®, the Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns.

There are four PRX functions provided in base SAS®.

- PRXPARSE – Compiles a PRX expression that can be used for pattern matching.
- PARXCHANGE – Performs a pattern-matching replacement
- PRXMATCH – Searches for a pattern match and returns the position of match.
- PRXPOSN – Returns a character string that contains the value for a capture buffer

The common use of PRX functions is to make sure well known patterns of strings (such as phone numbers, SSN numbers, address, etc) confirm to the pre-defined list of patterns. However pattern matching can also be effectively used to write crisp and shorter programs in places where lengthy programs are otherwise required. We will see such an example here.

Consider a situation in the pharmaceutical industry where a correction is performed on the sales & marketing data of a particular product across all zip codes of US. The supplier provides a corrected feed which needs to be compared with the previous feed from the supplier. The corrected feed is exactly identical to the previous feed in layout & format except the fact that the sales are corrected. In this situation, PRX functions can be used to write an intelligent program which can be very crisp and generic enough to compare any number of months of sales data.

In the following example, Var_1 to Var_5 refer to Month-1 to Month-5 of sales data. In reality, it could be any number of months and this example can compare all of them.

```
/* Building current and previous sample feeds */

Libname CURRENT 'Folder where corrected feed is stored';
DATA CURRENT.DS_1;
Informat Zip_Code $5.;
input Zip_Code Var_1 Var_2 Var_3 Var_4 Var_5;
cards;
07034 01 02 03 04 05
07864 12 13 14 15 16
08988 34 56 78 99 99
;

Libname PREVIOUS 'Folder where previous feed is stored';
DATA PREVIOUS.DS_1;
Informat Zip_Code $5.;
input Zip_Code Var_1 Var_2 Var_3 Var_4 Var_5;
cards;
07034 01 02 03 04 05
07864 11 12 13 14 15
08988 34 56 78 99 00
;

/* Obtaining variable list and count of variables */

Proc Contents Data=PREVIOUS.DS_1 out=TEMP_PREV_DS_1 (Keep=Name Type VarNum)
NOPRINT; Run;

Proc SQL;
Select distinct Name into :Var_List separated by ' '
From TEMP_PREV_DS_1
Where Type=1
order by VarNum;

Select count(Name) into :Var_Count
From TEMP_PREV_DS_1
Where Type=1;
Quit;

/* Following PRX statements help to rename and list multiple variables in a short
and crisp way */

%LET PRX_ID=%sysfunc(PRXPARSE(s/(\w+)/$1=$1_P/));
%LET RENAME_VAR_P=%sysfunc(PRXCHANGE(&PRX_ID,-1,&Var_List));

%LET PRX_ID=%sysfunc(PRXPARSE(s/(\w+)/$1_P/));
%LET LIST_VAR_P=%sysfunc(PRXCHANGE(&PRX_ID,-1,&Var_List));

%LET PRX_ID=%sysfunc(PRXPARSE(s/(\w+)/$1=$1_C/));
%LET RENAME_VAR_C=%sysfunc(PRXCHANGE(&PRX_ID,-1,&Var_List));

%LET PRX_ID=%sysfunc(PRXPARSE(s/(\w+)/$1_C/));
%LET LIST_VAR_C=%sysfunc(PRXCHANGE(&PRX_ID,-1,&Var_List));

%LET PRX_ID=%sysfunc(PRXPARSE(s/(\w+)/$1_D/));
%LET DIFF_VAR=%sysfunc(PRXCHANGE(&PRX_ID,-1,&Var_List));

/* Performing current to previous comparison */

Libname RESULT 'Folder where resultant difference to be store';
DATA RESULT.DS_1 (Keep = Zip_Code &DIFF_VAR);
      Merge PREVIOUS.DS_1 (in=a rename=(&RENAME_VAR_P)) CURRENT.DS_1 (in=b
rename=(&RENAME_VAR_C));
      By Zip_Code;
        Array PREVIOUS_VAR{&Var_Count} &LIST_VAR_P;
      Array CURRENT_VAR{&Var_Count}  &LIST_VAR_C;
      Array DIFF_VAR{&Var_Count}     &DIFF_VAR;
```

```
        If a & b then;
        Do i = 1 to &Var_Count;
           DIFF_VAR(i) = CURRENT_VAR(i) - PREVIOUS_VAR(i);
        End;
  Run;
```

```
Zip_Code     Var_1_D      Var_2_D      Var_3_D      Var_4_D      Var_5_D
07034           0            0            0            0            0
07864           1            1            1            1            1
08988           0            0            0            0           99
```

Log Output 13. PROC PRINT of resultant difference data set

As shown above, with the help of PRX functions, a generic comparison program is written in few lines of code to compare any months of data.

INTERESTING MINI-BYTES OF SAS® MACRO PROCESSING:

HOW TO UNDERSTAND A MACRO EXECUTION?

Lets start the macro section with an example of how macro works in SAS®. Macro expansion and data step execution are interleaving processes controlled by the word scanner. This is demonstrated in the following example.

```
  %Macro Mac1;
  %Put "inside macro: 1";
     DATA DS_Sample1;
     Put "inside data step :";
     Call sympput('macvar1','XXXXX');
  %Put "inside macro: macro variable's value: " &macvar1;
     Run;
  %Put "inside macro: 2 <--- check here :";
  %Put "macro variable's value: " &macvar1;
  %Mend Mac1;

  %Put "before calling the macro:";
  %Mac1;
```

```
"before calling the macro:"
"inside macro: 1"
WARNING: Apparent symbolic reference MACVAR1 not resolved.
"inside macro: macro variable's value: " &macvar1
inside data step :
"inside macro: 2 <--- check here :"
"macro variable's value: " XXXX
```

Log Output 14

As shown in Log Output (14), the first occurrence of MACVAR1 is not resolved and the 2nd occurrence is resolved. By understanding the fact behind this behavior, we can easily understand how macros work. Lets understand this by seeing the step by step execution of the above program.

```
%Put "before calling the macro:"; <== S1: Execute it
%Mac1;                            <== Call macro

%Macro Mac1;                      <== Start of macro
%Put "inside macro: 1";           <== S2: Execute it
   DATA DS_Sample1;               <== S4: Construct data step
   Put "inside data step :";      <== S5: Construct data step
   Call sympput('macvar1','XXXXX');<== S6: Construct data step
%Put "inside macro: macro variable's value: " &macvar1; <== S3: Execute it
   Run;                <== S7: Data step construction is over. Execute data step.
%Put "inside macro: 2 <--- check here :"; <== S8: Execute it
```

```
%Put "macro variable's value: " &macvar1; <== S9: Execute it
%Mend Mac1;                             <== End of macro
```

As shown above, Step S3 is executed before S6 where MACVAR1 is defined. So S3 does not know MACVAR1 and since the warning message in Log Output (14). As soon as word scanner sees RUN statement (S7), the data step (S3 – S6) is executed immediately and MACVAR1 is created. Because of this, when S9 runs, it returns the value of MACVAR1.

CALL EXECUTE – LETS EXECUTE IT INSIDE A MACRO:

CALL EXECUTE is a prominent attribute in base SAS®. It resolves the argument, and issues the resolved value for execution at the next step boundary. It takes an argument as a string. The string could be either,

- A macro invocation – the macro executes immediately and DATA step execution pauses while the macro executes. If macro returns any data step statements, they are put in the input stack and executed after the current data step is over.
- A data step statement – that statement is put in the input stack and executed after the current data step is over

Lets take the same example that we have seen to understand the macro trivial processing and just change the macro invocation to be within CALL EXECUTE and see what happens.

```
%Macro Mac1;
%Put "inside macro: 1";
   DATA DS_Sample1;
   Put "inside data step which is inside macro:";
   Call symput('macvar1','XXXX');
%Put "inside macro: macro variable's value: " &macvar1;
   Run;
%Put "inside macro: 2 <--- check here :";
%Put "macro variable's value: " &macvar1;
%Mend Mac1;

Data _null_;
Put "inside master datastep";
Call Execute('%Mac1');
Run;
```

```
inside master data step
"inside macro: 1"
WARNING: Apparent symbolic reference MACVAR1 not resolved.
"inside macro: macro variable's value: " &macvar1
"inside macro: 2 <--- check here :"
WARNING: Apparent symbolic reference MACVAR1 not resolved.
"macro variable's value: " &macvar1

NOTE: CALL EXECUTE generated line.
1   + data sample; put "inside data step which is inside macro:"; call
symput('macvar1','kantu'); run;

inside data step which is inside macro:
NOTE: The data set WORK.SAMPLE has 1 observations and 0 variables.
```

Log Output 15

Evidently the macro variable MACVAR1 is not resolved at all for both the macro %PUT statements. It is slightly different from how macro trivial processing works. This is because when the macro %Mac1 is called by CALL EXECUTE, all macro statements (%put statements) inside it are executed first, then the constructed data step DS_Sample1 is executed.

CALL EXECUTE With Double/Single Quotes:

SAS® users should be careful when using double quotes inside CALL EXECUTE because the parts inside double quotes are expanded/executed at compile time. Lets understand it with an example.

```
Options Macrogen Symbolgen;
%Macro Mac1;
  DATA _null_;
  Put "inside macro:";
  Run;
%Mend Mac1;

Data _null_;
  Call Execute ("%Mac1");
Run;
```

```
NOTE: Line generated by the invoked macro "MAC1".
125  data sample; put "inside macro:"; run;
                      ------
                      388
                      76
ERROR 388-185: Expecting an arithmetic operator.
```

Log Output 16

The reason for the error at Log Output (16) is that the macro %Mac1 is expanded at compile time and as a result of which, we will end up having: **`CALL EXECUTE ("DATA SAMPLE; PUT "inside macro:";RUN;");.`**

The above statement is syntactically incorrect due to unbalanced quotation marks. This error can be resolved in two ways.

1. By having single quotes on the put statement inside data step – Now, macro is still executed at compile time, but quotes of the resolved statements within CALL EXECUTE are balanced.

2. By having single quotes on the CALL EXECUTE macro invocation – Now, macro is executed at run time and data step is substituted to CALL EXECUTE during run time.

CALL EXECUTE – A Practical Example:

The principal characteristics of CALL EXECUTE routine are its ability to dynamically build and insert a DATA/PROC step at run time and to conditionally invoke a macro at run time. These features help to build a complex decision making program more efficiently with fewer lines of code. One such example is given here.

Lets take into account a situation in a pharmaceutical company wherein a DIS (Drop-In-Sales) investigation report has to be built for products with lower sales volume.

```
/* Macro to build DIS Report */

%Macro BUILD_DIS_REPORT;
  DATA DIS_Report_&Product;
   /* Apply business rules in building DIS Report */
  Run;

  PROC PRINT Data=DIS_Report_&Product;
  Run;
%Mend BUILD_DIS_REPORT;

/* Driving data step to conditionally invoke the DIS Report macro */

DATA _null_;
 Set DS_Sales; /* DS_Sales contain Sales data for every product */
 If Sales <= 100 then do;
    Call Sympunt ('Product',PRD);
```

```
        Call Execute ('%BUILD_DIS_REPORT');
    end;
  Run;
```

The result of this program will conditionally invoke the macro BUILD_DIS_REPORT at run time and print the Drop-In-Sales investigation report for low sales products.

QUOTE FUNCTIONS IN SAS® MACROS:

There are several papers submitted at international and regional SAS conferences about QUOTE functions in SAS® macros and also about the technical details behind them. But there are very few papers that explain the real-time use of the QUOTE functions. One important area where QUOTE functions can be leveraged in any industry is to process the “free-hand” data such as surveys, customer feedback, product feedback or any public/private forums where users are allowed to enter any information that they can. To obtain program samples of QUOTE functions to process the free-hand data, please contact the author; considering the length of the paper, they are not provided.

ACKNOWLEDGEMENTS

The author likes to thank PharmaSUG-2012 organizers for giving the opportunity to present this paper. He also likes to thank Cognizant Technology Solutions, his managers (Dhananjay Kelkar & Sunil Varanasy) and SAS® leads (Ramya Purushothaman & Anurag Katare) for their motivation & contribution towards writing this paper and his Cognizant-Pune counterparts Anay Ambure and Adney Fernandes for providing certain practical examples that are captured in this paper.

CONTACT INFORMATION

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

Name	: Airaha Chelvakkanthan Manickam
Enterprise	: Cognizant Technology Solutions US Corporation
Address	: 500 Frank W.Burr Blvd
City, State, Zip	: Teaneck, NJ, 07666
Phone	: 610 316 5780
Mail Id	: Airahachelvakkanthan.Manickam@cognizant.com