

Using Datasets to Define Macro Loops and Local Macro Variables

Steven Ruegsegger
IBM Microelectronics, Burlington, VT

ABSTRACT

A common and optimal way to execute repeated tasks in SAS[®] is to write a template of the task in a macro function and then call that macro function the desired number of times. This can be difficult to implement when running the code in batch mode. Often, the number of times to execute the macro or the parameter values to use are not known until data is pulled. This paper will look at defining a macro loop from a Control Dataset, where each row (observation) is one loop iteration and each column (variable) is one local macro variable. Two implementation methods will be investigated. Both use Macro Language %do loops to repeatedly call the analysis template. The first method defines all local macro variables for all iterations *a priori*. Within each loop iteration, the appropriate local macro variables will be referenced with the format “&&var&i”. The second method is a bit more elegant, using nested %do loops to define the local macro variables only for the current iteration. A technique for defining a Control Dataset will then be explored, which involves inner joins, outer joins, and data blocks.

INTRODUCTION

User-defined macro functions in the SAS[®] Macro Language are often used to repeatedly execute a code template to produce common analysis. Macro variables are used to alter the code template to the desired result for each iteration. One efficient way to implement repeated calls is through a macro loop. The goal of this paper is to demonstrate the power and convenience of using a SAS dataset to define the loop and the macro variables. We'll call this the Control Dataset.

This paper will first define a Control Dataset. Then two methods of implementing a macro loop using a Control Dataset will be discussed. The first is a typical approach, and the second is more novel.¹ A technique for creating the Control Dataset will be presented. Finally, a real-world example is given.

USER-DEFINED MACRO FUNCTIONS

In general, SAS is a succinct language. That is, one can get a large amount of complex analysis with just a few lines of code. Also, one statement (e.g. a by statement) or one word (e.g. nodupkey) can drastically change the analysis results. Nonetheless, SAS programs can grow large and sections of code get repeated. The Macro Language is very helpful for re-using blocks of common code by putting them into a macro function and calling that function each time the analysis is needed. Within the macro function, macro variables change what's unique about each iteration. For example, if there are several distinct, independent data populations in a factory, the macro function would generate a common analysis report, but each iteration would use a different data population for that analysis. A macro variable &population might be used in the macro function like this:

```
proc summary data=measurements(where=(product_PN in ( &population )));
```

where &population is a list of partnumbers.

If there is a small, fixed-number of times the analysis macro function is to be executed, then the repeated macro calls and macro variable values can be hard-coded right in the source code. For example, a QC (Quality Control) engineer at a factory may want to create a series of SPC (Statistical Process Control) charts using proc shewhart. Generally, the analysis proc is preceded with standard “data-prep” code which might pull the data, calculate univariate statistics, remove flyers from the data, etc. Rather than cut-and-paste the SAS blocks over and over for each data population in the factory, a user-defined macro function can be created. Continuing with the illustration, here is the macro function shell:

¹ I've never seen this method published as implemented in a macro loop. However, I recently discovered a similar version of this method used in a data block in order to export a dataset to a text file [Aster, p91].

```
%macro spc_charts(dsn=, parm=, toolset= );
  /* 1. analyze data in &dsn for 1 parameter (&parm) and 1 toolset
     - find mean, median, and ptiles
     2. prep data - remove flyers, etc.
     3. create reports, distributions, CDFs, histos
     4. create SPC charts, EWMA, ANOVA, Cpk, etc.
     5. send alarms
  */
%mend spc_charts;
```

Our macro function to be repeatedly called is named %spc_charts. It has three arguments -- &dsn is the dataset to be analyzed, &parm is the name of the column of measurements, and &toolset is the name of the manufacturing process step to analyze for statistical control. These 3 arguments define one data population in the factory for SPC analysis.

How many charts need to be made? If the factory had 3 different populations with 10 measured parameters and 2 toolsets each, then the %spc_charts macro would need to called 60 times.

```
%spc_charts(dsn=tech180nm, parm=parm1, toolset=etch)
...
%spc_charts(dsn=tech130nm, parm=parm2, toolset=photo)
...
%spc_charts(dsn=tech090nm, parm=parm8, toolset=etch)
...
```

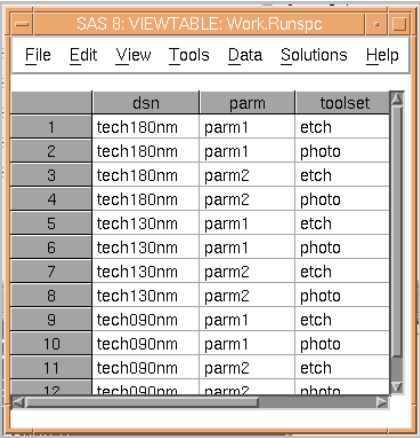
However, in most situations, it's probably not a good idea to keep all these macro calls individually hardcoded in the source code. First, the macro arguments may change frequently in a dynamic factory and it would be difficult (and risky) to keep up with the changes in the source code. Second, it may be too impractical to maintain individual calls in source code. A total of 60 calls is much too small for most factories. Permutations of 1000's or more are likely. Third, the desired calls may not be known *a priori*, but rather a function of real-time data pull and summary analysis. Or the analysis may be requested in real-time from a web page. For all these reasons, a macro loop defined at run-time is a much better approach. A %do %while macro loop in a *control* macro function can call the *analysis* macro function (e.g. %spc_charts) as often as needed and with the proper arguments.

```
/* control macro loop;
%do %while I = 1 %to &N;
  %spc_chart( <arguments> ); /* analysis template to run;
%end;
```

In order to do this, &N needs to be known and the arguments need to be defined for each iteration. The best way to do this is with a Control Dataset.

USING A CONTROL DATASET

A Control Dataset contains all the information needed to setup the macro loop in order to execute the analysis macro function the correct number of times with the correct macro variables. In the Control Dataset, each row of the dataset defines one iteration of the loop and each column represents a local macro variable's value. For example, **Figure 1** is a snapshot of a SAS dataset being used as a Control Dataset. For the first iteration, the variable &dsn will be equal to "tech180nm", for the fifth iteration it will be equal to "tech130nm", and for the ninth iteration it will equal "tech090nm".



	dsn	parm	toolset
1	tech180nm	parm1	etch
2	tech180nm	parm1	photo
3	tech180nm	parm2	etch
4	tech180nm	parm2	photo
5	tech130nm	parm1	etch
6	tech130nm	parm1	photo
7	tech130nm	parm2	etch
8	tech130nm	parm2	photo
9	tech090nm	parm1	etch
10	tech090nm	parm1	photo
11	tech090nm	parm2	etch
12	tech090nm	parm2	photo

Figure 1 - An Example Control Dataset

Where does the input for a Control Dataset come from? There are two main sources for defining a Control Dataset:

1. A summary of the real-time datapull
2. User input of what's desired

Real-time Summary. Very often, it is desired to have standard analysis done only on “what’s new,” *e.g.* what’s currently running in the factory. A `proc sort` with a `nodupkey` on a data pull of recently processed goods can be used to create the beginning of the Control Dataset. Each unique key (*e.g.* `product_PN`) corresponds to an iteration of the macro loop.

```
proc sort data=data_pull nodupkey out=run_products;
  by product_PN;
run;
```

User Input. A second method of creating a control file is from user input. A web-page front-end may have a list of possible populations, and the user may select a subset of them. The output from this web-page form can then become the input to a SAS job and used as the beginnings of a Control Dataset.

Once the Control Dataset is defined, there needs to be an implementation method for the macro looping. There are two tasks to perform –

1. loop through each observation, and
2. defining the macro variables for each loop count.

Two implementation methods will be discussed. The first method separates these two tasks – all the macro variables are defined first, and then the main macro loop is defined. The second method integrates these two tasks – the macro variables are defined in an inner, nested loop while looping through the rows.

THE “&&VAR&I” METHOD

The first implementation method is called the “&&var&i” method (pronounced “amper-amper-var-amper-i”). It gets its funny name from how the local macro variables are referenced in the analysis macro function (see Burlew, pg 65 for further explanation). This method pre-defines *all* the macro variables for *all* iterations at once. The most common method to do this is by using `call symput()` in a `data _null_;` block where the Control Dataset is the input.

```
1  data _null_;
2      set run_charts;
3      call symput ('dsn' || _n_, dsn);
4      call symput ('parm' || _n_, parm);
5      call symput ('toolset' || _n_, toolset);
6      call symput ('nobs', _n_); * keep overwriting value until end = total obs.;
7      run;
```

Line 2 shows the Control Dataset (`run_charts`) as the input dataset. There is no output dataset (line 1) as the purpose of this code block is simply to setup the macro variables used to define the subsequent macro loop.

Line 6 is one method for getting the total number of observations in the Control Dataset, which represents the total number of macro loops.² Lines 3-5 define the local macro variables to be used in the analysis macro function. For example, if there are 3 obs in the `run_charts` Control Dataset, then 10 variables have been

² This method is a little inefficient in that it rewrites the `&nobs` variable over and over with the current observation number (`_n_`) before finally stopping at the last. However, as this is a small Control Dataset and it has to be looped through anyway for all the other variables, the overhead is negligible. A more efficient method would use an `(end=)` option in the dataset.

defined: `dsn1`, `dsn2`, `dsn3`, `parm1`, `parm2`, `parm3`, `toolset1`, `toolset2`, `toolset3`, and `nobs`. The “`&&var&i`” format promotes the ability to think of these variables as single dimension arrays (`dsn[1..3]`, `parm[1..3]`, `toolset[1..3]`), but in reality they are actually 9 individual variables with a naming convention. Notice that all 10 macro variables have been defined by this one `data` block before any macro loops have started.

With the macro variables all defined, and the number of desired iterations known (`&nobs`), a “run” macro function is defined, which executes these loops.

```
1  %macro run_loops;
2      /* &nobs and all the “&&var&i” variables defined above */
3      %do i = 1 %to &nobs;
4          %spc_charts(dsn=&&dsn&i, parm=&&parm&i, toolset=&&toolset&i)
5      %end;
6  %mend run_loops;
7  %run_loops
```

Line 3 shows the `%do` loop from 1 to `&nobs`. Line 4 calls the analysis macro function `%spc_charts()`. At each iteration, the macro arguments change accordingly. As the loop counter variable `&i` changes, the macro arguments point to a new macro variable. It can be thought of as passing the “*i*th value” of all the macro variables.

It can be tricky to see how this works at first. The key is to know three rules of the SAS Macro Language:

1. the double ampersand (`&&`) simply resolves to a single ampersand (`&`)
2. the macro engine resolves variables left to right
3. the macro engine continues passing through the code until no more `%`'s or `&`'s remain

Consider “`&&dsn&i`”. On the first pass through the macro code, the macro engine resolves `&&` to `&` and `&i` resolves to the current loop number, let's say 1. With these two changes, `&&dsn&i` becomes `&dsn1`. Since an ampersand still exists, the macro engine does another pass. This time `&dsn1` is simply a macro variable which has already been defined from the `call symput`'s discussed above. Specifically, `&dsn1` resolves to the first observation of the `dsn` column from the Control Dataset. This repeats for the other arguments -- `&parm1`, `&toolset1`. Then, upon the 2nd iteration, `i = 2`, and the macro engine resolves `&dsn2`, `&parm2`, and `&toolset2`.

This method is perfectly valid and used often, but it has a few drawbacks. One is that the form `&&var&i` is a little cumbersome. Unless all the macro variables are used as `%macro` arguments, the `&&var&i` form will have to be used throughout the analysis macro function. However, one does get used to it. Also, these macro variables are not local, but global since they are defined with `call symput`.³ Finally, this method is slightly suboptimal in its memory use as it forces the definition of *all* variables for *all* loops *a priori*. But, most of the time, this is not a memory-limiting issue.

THE “FETCH” METHOD

A different implementation method for defining macro loops through a Control Dataset is the “fetch” method. It nests the tasks – an outer loop over the rows and a nested inner loop over the columns. The outer loop uses the Macro Language to open the Control Dataset and read it line by line using several `%sysfunc` commands. The inner loop gets the column name and value using one of the “`getvar`” commands. The column value is then assigned to a local macro variable of the same name. After the completion of the inner loop, the analysis macro function is executed.

Below is the template for this method.

³ Code could be written to get `&nobs` first and then create an additional initial loop to define `%local` for each variable to be defined later, but in practice this is not done. The variables are simply left global.

```

1  %local dsid rc now total cols;
2  %let dsid = %sysfunc(open(runme));
3  %let now = 0;
4  %let rows = %sysfunc(attrn(&dsid, nobs));
5  %let cols=%sysfunc(attrn(&dsid, nvars));
6
7  %do %while (%sysfunc(fetch(&dsid)) = 0);    /* outer loop across rows;
8
9      %let now=%eval(&now + 1);
10     %*if &now ne 1 %then %goto out;    /* debugging;
11
12     %do i = 1 %to &cols;    /* inner nested loop;
13         %local v t;
14         %let v=%sysfunc(varname(&dsid,&i));
15         %local &v;
16         %let t = %sysfunc(vartype(&dsid, &i)); /* N or C;
17         %let &v = %sysfunc(getvar&t(&dsid, &i));
18         %*if 0 %then %put ** &v = &&&v; /* show var and value in log;
19         %end;
20
21     %put ** loop # &now of &rows **;
22     %*if <condition> %then %goto out;
23
24     /* analysis macro function -- to be run for each loop iteration */
25
26     %out:
27     %end;    /* while fetch loop;
28 %let rc = %sysfunc(close(&dsid));

```

Line 2 uses the `%sysfunc` macro function to open the Control Dataset, called `runme` in this example.

```
%let dsid = %sysfunc(open(runme));
```

This dataset is now is “in use” by SAS, and cannot be modified until it is closed in line 28. The `open` function returns an integer file number, assigned to `&dsid`, which is used throughout the code to reference this opened file.

Line 3 initializes the loop counter `&now` to zero and line 9 uses `%eval` to increment the counter by one for each loop. This is used primarily in debugging to print to the log or to limit the number of rows executed.

The outer loop is defined in line 7, which will execute once for each observation (row) in the `&dsid` dataset.

```
%do %while (%sysfunc(fetch(&dsid)) = 0);
```

The `fetch` function reads the current observation and populates the DDV (Dataset Data Vector) with the values from each variable (column). It will return 0 if successful, another integer if an error, and -1 if at end of file. This line loops as long as a 0 is returned.

However, before the outer loop, line 5 defines the number of columns in the Control Dataset using the `nvars` option of the `attrn` function.

```
%let cols=%sysfunc(attrn(&dsid, nvars));
```

This will define the range for the inner loop from lines 12 to 19. It takes three `%sysfunc` calls to automatically define a local macro variable. The first, in line 14, calls the `varname` function which returns the variable name of the i^{th} column. This is stored in local variable `&v`.

```
%let v=%sysfunc(varname(&dsid,&i));
```

Line 15 is short, but a little tricky.

```
%local &v;
```

It creates a local macro variable from the *value* of a local macro variable. That is, a new local macro variable is

defined to have the *same name* as the i^{th} column retrieved from line 14. This line only defines the name of the local macro variable – its value will be retrieved and assigned next.

Recall that SAS has 2 types of variables – numeric and character. SAS treats these variable types differently and often has different functions dependent upon the type. So, for each column in the Control dataset, the variable type must be retrieved. This is done in line 16 with the `vartype` function which returns either an “N” or “C.”

```
%let t = %sysfunc(vartype(&dsid, &i)); %* N or C;
```

The column name (`&v`) and column type (`&t`) are used together in line 17 to retrieve the *value* of the i^{th} column in the DDV. Rather than use an `%if` statement, the “N” or “C” `vartype` from `&t` is used right in the `%sysfunc` command. The `getvarc` or `getvarn` function is *created*, depending on the appropriate `vartype` of the i^{th} column.

```
%let &v = %sysfunc(getvar&t(&dsid, &i));
```

This `%let` statement is a bit unusual in that the variable name being assigned (left of the equal sign) is also a variable (`&v`). The macro engine will resolve `&v` first and then assign to it the value returned from the `getvar&t`.

Before continuing, we should consider the ramifications of lines 12-19. Some may think this dangerous. This is indeed a concern – macro variables are *automatically* being created simply for being a column in a dataset. This very succinct code does not show the programmer the macro variables being created. So, its easy to imagine that a column added to a dataset may inadvertently make its way into the Control Dataset, where it is automatically defined as a macro variable.

What are the concerns? First, the variable may inadvertently collide with a previous assignment of the same macro variable and corrupt the program. However, the `%local` definition of `&v` in line 15 mitigates that risk by limiting the variable scope. If a global macro variable does exist, the explicit `%local` definition will ensure it will *not* be corrupted. But what if we want to use the global variable in the macro loop or the analysis macro function? Or what if there is a local variable defined in a nested macro with the same name? Both of these situations are problematic.

They are indeed risks, but no more than normal programming. There is nothing in this technique which *enhances* those typical risks. Nonetheless, one could argue that this programming technique may be a little more difficult since the variable names are *not* explicitly seen in the source code and cannot be searched or ‘grep-ed.’ However, additional programming techniques can be used to reduce this difficulty. For example, one technique used is to have the variables of the control dataset explicitly defined in the `select` of a `PROC SQL` block. This ensures ‘extra’ columns do not inadvertently get into the Control Dataset since they were not specified in the `select` statement. It also allows the variable names to be searched for. A second technique is to use the top-level macro solely to run the macro, so no other `%local` variables exist.

At this point in the macro, through line 19, we have concluded all the setup for the analysis macro call. Each macro variable was defined, and defined with a local scope so as to not collide with any global variables. If we want to print out the variable names and values to the log for debugging, line 18 can be uncommented. This has a cool triple-`&` structure:

```
%put ** &v = &&&v;
```

After the first macro engine pass, the first `&v` becomes the variable name (e.g. “days”), the `&&` after the equals sign resolves to a single `&`, and the subsequent, second `&v` (at the end of the line) also resolves to “days” like the first one. It now looks like:

```
%put ** days = &days; %* after 1st macro pass;
```

At the next macro engine pass, the `&days` then becomes the value (e.g. “30”). So, the log prints:

```
** days = 30
```

Line 22 is a commented debugging statement used for skipping out of the macro loop early and not executing the analysis macro function. For example, this line is often left in the source as a comment:

```
%*if &now ne 1 %then %goto out;
```

When the asterisk (*) is removed during debugging, then this line skips calling the analysis macro for every observation but the first. However, this technique is not only used for debugging purposes. There may be times when a loop is to be skipped based on the column values. For example, this line:

```
%if &alarm_only and &spc_alarm eq 0 %then %goto out;
```

could be used as a switch to only plot a chart where an SPC alarm has been set.

PREPPING THE CONTROL DATASET

One of the most useful features of this “fetch” method is the ability to edit the row and column dimensions of the Control Dataset using all the power of SAS. As previously described, editing *rows* adds or removes loop iterations, and editing *columns* adds, deletes, or modifies the local macro variables. A three step process has been developed to define and modify the Control Dataset. It starts with a `proc sql` inner join to build the initial set of valid observations. Then a series of `proc sql` left outer joins are used to add desired columns from other datasets into the Control Dataset. Finally, a `data` block allows all the power of the SAS language to be used for various row and column modifications.

Inner Join. First, an inner join is used create the initial ‘bare-bones’ Control Dataset. In an inner join, only observations are kept which match all criteria from all merging datasets. Therefore, this first join is used to ensure valid observations. For example, this SQL makes the Control Dataset `runspc`.

```
1 proc sql noprint;
2   create table runspc as
3   select a.parm, a.oper,
4         b.parmname, b.parmdesc, b.meastool, b.proctool,
5         c.operdesc, c.toolset, c.opertype
6   from charts a
7   inner join validparms b on b.parm = a.parm
8   inner join validopers c on c.oper = a.oper
9   where c.opertype not in ('EXP', 'EWR', 'PCN')
10  ;
11  quit;
```

The `charts` dataset (line 6) is the dataset containing the loop instructions – in this case, which SPC charts to make. This initial dataset is often a very simple list, for example, containing only pairs of parameters (parms) and operations (opers). This could come from a user’s input through a web page, or from the summary output of another SAS job, or from a standard report config file.

In order to ensure valid charts, the list of charts is joined with datasets containing valid parms (line 7) and valid opers (line 8). Line 9 shows a `where` clause which excludes making charts for any opers of a certain type. Lines 4 and 5 add a few columns of extra information to our Control Dataset.

This inner join could have been done with a `data` step and a `merge` command. But when joining several datasets together in an inner join, a `proc sql` is very powerful, natural and easy to read.

Left Outer Join. After the inner join, extra information can be added with left outer joins. In a left outer join, all the rows from the “left” dataset are included in the result, and only the information from the “right” dataset is included if it meets the join criteria. Therefore, these left outer joins have the affect of only adding columns to the rows which already exist from the inner joins above. These left outer joins can be thought of as simply adding desired local macro variables from auxiliary datasets.

```
1 proc sql noprint;
2   create table runspc as
3   select a.*,
```



```

4          b.target, b.lsl, b.usl, b.spc, b.lcl, b.ucl, b.spectype,
5          c.y1 as ymax, c.y2 as ymin, c.y3 as yby, c.ylog,
6          d.customflyer
7  from runspc a
8  left outer join targets      b on a.parm = b.parm
9  left outer join yaxis       c on a.parm = c.parm
10 left outer join (select parm, value as customflyer
11                  from parmfilter
12                  where action eq "filt" ) as d on a.parm = d.parm
13 ;
14 quit;

```

The query above joins 3 other tables with the `runspc` Control Dataset. Line 3 shows that all the variables from `runspc` are being kept. Lines 8 and 9 are joining datasets which have special information about the parms. The `targets` dataset has some spec and control targets for some parms, and the `yaxis` dataset has information for an `axis order=` statement. Not all parms have this extra information. The left outer join will still keep the row in the Control Dataset even if there is no entry for the parm in the auxiliary datasets.

How are these variables used? They are used throughout the analysis macros for all sorts of configurable options. For example, in the analysis code, one might see a line like:

```
%if &y_order_flag %then order=(&y1 to &y2 by &3);
```

The True/False flag variable `&y_order_flag` is defined below in the third step. If True, then the y-axis limits of the chart will be defined by the user. If False, then the SAS defaults are used.

Lines 10-12 show a nested query. The `parmfilter` dataset has multiple observations per parm. The nested query only keeps rows with a certain action. Then the column `customflyer` is joined to the `runspc` dataset for the appropriate parms.

Data Step. Finally, after `proc sql` is used for inner and left outer joins, the final massaging of the Control Dataset is done with a data step. This makes available all the power of the SAS language for any row or column editing.

```

1  data runspc;
2    set runspc;
3
4    * formatting;
5    target = put(target, best5.);
6    usl = put(usl, best5.);
7    lsl = put(lsl, best5.);
8
9    * conditions;
10   if y1 < y2 and y3 ne . then y_order_flag = 1 ; * order= (y1 to y2 by y3);
11   else y_order_flag = 0;
12   if lsl > usl then delete;
13
14   run;

```

Lines 4-7 show an example of formatting variables for better printing when used a local macro variable. Lines 9-12 show that SAS conditional statements can be used to modify columns or even filter iterations. Lines 10-11 are properly formatting a T/F flag variable for easy use in an `%if` statement. Line 12 is used to delete observations which don't follow the assumed conventions.

A REAL-WORLD EXAMPLE

As this paper was being written, a real-world example of using this method was needed. The volume for this type of data is very large – on the order of 3E6 rows per lot. Therefore, it is expedient to not re-process previous data

already analyzed, and also to only run one lot at a time (otherwise there are buffer and work space issues).

The code below is a more succinct version of the “fetch” method (block #4) and is preceded by the Control Dataset setup (blocks #1-3). After the detailed description above, it is hoped that the reader can understand the code below based on the comments before the blocks. A “lot” is a single manufacturing batch unit (usually 25 wafers per lot and usually 100’s of chips per wafer).

```
* 1. get lots recently measured (recent_lots);
proc sort data=big_database(keep=lotlabel testprogramec test_date)
    out=recent_lots nodupkey;
    where test_date > &sysdate - 2;
    by lot_label;
run;

* 2. get lots already analyzed - no need to do again (old_lots);
proc sort data=analyzed(keep=lot_label) out=old_lots nodupkey;
    by lot_label;
run;

* 3. find the 'new' lots - the difference between the two (the Control Dataset);
data new_lots;
    merge recent_lots(in=a) old_lots(in=b);
    by lot_label;
    if a and not b;
run;

* 4. for each 'new' lot only, run a loop to get data, process, store & analyze;
%macro runlot;
    %local dsid rc now rows cols;
    %let dsid = %sysfunc(open(new_lots));
    %let rows=%sysfunc(attrn(&dsid,nobs));
    %let cols=%sysfunc(attrn(&dsid,nvars));
    %do %while (%sysfunc(fetch(&dsid)) = 0);
        %do i = 1 %to &cols;
            %local v t;
            %let v=%sysfunc(varname(&dsid,&i));
            %local &v;
            %let t = %sysfunc(vartype(&dsid, &i)); %* N or C;
            %let &v = %sysfunc(getvar&t(&dsid, &i));
            %end;

            %* analysis program setup and call;
            %let population = twp.lotlabel in ( &lot_label );
            %include "/afs/btv/u/ruegsegs/.../lbist_freq.sas";

        %end; %* while fetch loop;
    %let rc = %sysfunc(close(&dsid));
%mend runlot;
%runlot
```

CONCLUSION

Consolidating frequently used code into macro functions is a great way to optimize and organize your code. Using a macro loop is a great way to repeatedly execute the macro function. A Control Dataset can be used to define the macro loop easily and neatly. In a Control Dataset, each observation represents one loop iteration and the columns represent the local macro variables. Two different implementation methods were described. The first was called the “&&var&i” method which defines all the variables for all the loops *a priori*. The second method is called the “fetch” method. It nests the column-definition loop inside the row-definition loop. A three-step programming technique can be used which makes defining the Control Dataset simple and easy to read.

REFERENCES

Aster, Rick. *Professional SAS Programming Shortcuts*. Breakfast Communications Corp, Paoli, PA. 2002

Burlew, Michele M. *SAS Macro Programming Made Easy*. SAS Publishing. Cary, NC. 2003.

Carpenter, Art. *Carpenter's Complete Guide to SAS Macro Language*. SAS Publishing. Cary, NC. 2004.

SAS Language Reference: Dictionary, Version 8. SAS Institute, Inc. Cary, NC. 1999.

CONTACT INFORMATION

Steve Ruegsegger
IBM Microelectronics
1000 River Road
Essex Junction, VT 05465
Email: ruegsegs@us.ibm.com

ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.