Paper AD10

SAS® MACROS: TIPS, TECHNIQUES, AND EXAMPLES

Andrew M. Traldi, Cingular Wireless, Atlanta, GA

ABSTRACT

Most SAS® users are aware that SAS has a macro facility but might be unsure of how they can use it or are fearful that macros are too difficult. Although macros can be complex, they can be very helpful in writing general-purpose SAS programs; in some instances, they are absolutely critical to an application.

WHAT IS THE SAS MACRO FACILITY?

The purpose of the SAS macro language is to generate text which is used in SAS programs; this text can be any valid SAS code: statements, variables, text strings, PROC steps, etc. In its simplest form, a macro variable can be used for text substitution in SAS code. Consider the following example:

```
%let state=GA;
%let month=Jul2003;
...
proc print data=permlib.sales(where=(state_code="&state" and
                              month_year=input("&month",monyy7.));
title "Sales report for &state / &month";
run;
```

These statements could be useful if you provide reporting by month and region and you want to be able to generate reports for different states and months easily. This example assumes that there is a data set PERMLIB.SALES that contains sales data and has variables `state_code` and `month_year` that we can use to select the desired observations. Note that we haven't even used a macro here, just macro variables for simple text substitution.

One important difference between macro code and SAS code is that the macro code is compiled *prior to* regular SAS code, and the code generated by the SAS macro compiler is then processed by the SAS compiler. Here is an example that illustrates this difference:

```
data dumb;
if 1 eq 2 then do;
   * this will never be executed!;
   xxxyyyzzz;
end;
else do;
   put 'Hello'; …
end;
run;

203  data dumb;
204  if 1 eq 2 then do;
205       * this will never be executed!;
NOTE: SCL source line.
206       xxxyyyzzz;
          ---------
          180
ERROR 180-322: Statement is not valid or it is used out of proper order.

207  end;

%macro dumb;
data dumb;
%if 1 eq 2 %then %do;
```

```
   %* This will never be compiled!;
   xxxxxyyyyy;
%end;
%else %do;
    put 'Hello';
%end;
run;
%mend dumb;
%dumb

MPRINT(DUMB):   data dumb;
MPRINT(DUMB):   put 'Hello';
MPRINT(DUMB):   run;

Hello
```

In the first part of this example, even though the statement in the `if 1 eq 2 then do` group will never be executed, it is still compiled and causes a syntax error. In the second case, the statement within the `%if 1 eq 2 %then %do` group is successfully compiled by the macro compiler, but because it is never executed, the invalid line of SAS code is never passed to the SAS compiler. In the first case, the SAS code is conditionally executed by the SAS compiler and therefore must have valid syntax; in the second case, the code is conditionally compiled by the SAS compiler, and only needs to be valid when the condition is met. A good question at this point is "So what – why have a program with invalid code anyway?" Here is an example showing when this technique can be very useful:

```
%exist(out.masterdsn);
%macro _go;
data results;
...
%if &exist eq yes %then %do;
    set out.masterdsn key=id/unique;
%end;
...
run;
%mend _go;
```

The statement `set out.masterdsn key=id/unique` will cause a syntax error if the data set `out.masterdsn` does not exist. However, the statement will only be sent to the SAS compiler when the data set does exist. In actuality, we might also want to verify that the data set not only exists but also has an index for the variable `id`; this can be done as well. The macro `exist` is a utility macro that I find quite useful; it is illustrated in a later example in this paper (see page 11).

ENVIRONMENT

A short discussion of the macro variable environment is probably in order. The environment can be explicitly specified with either the `%global` or the `%local` statement. The value of a global macro variable is available throughout the program – open code as well as within macros. A local macro variable's value is available only in the macro where it is defined (therefore, a `%local` statement is not valid in open code).

Consider the following example:

```
%global var1;
%let var1=hello;
%let var2=world;
%put ** in open code var1=&var1 var2=&var2 **;

%macro test;
%put ** in test: var1=&var1 var2=&var2 var3=&var3 **;
%mend test;
%test
```

```
%macro test2;
%local var2;
%let var1=hi;
%let var2=universe;
%let var3=hello, world;
%put ** in test2: var1=&var1 var2=&var2 var3=&var3 **;
%test;
%mend test2;

%test2;
%test

%put ** in open code var1=&var1 var2=&var2 var3=&var3 **;
```

Here is a piece of the resulting SASLOG:

```
** in open code var1=hello var2=world **
...
9    %test
WARNING: Apparent symbolic reference VAR3 not resolved.
** in test: var1=hello var2=world var3=&var3 **
...
19   %test2;
** in test2: var1=hi var2=universe var3=hello, world **
** in test: var1=hi var2=universe var3=hello, world **
20   %test
WARNING: Apparent symbolic reference VAR3 not resolved.
** in test: var1=hi var2=world var3=&var3 **
21
22   %put ** in open code var1=&var1 var2=&var2 var3=&var3 **;
WARNING: Apparent symbolic reference VAR3 not resolved.
** in open code var1=hi var2=world var3=&var3 **
```

The default environment for a macro variable is what I would call *downward* global. That is, the value of the macro variable can be referenced (and changed) in the environment where it first appears as well as in any macros which are invoked from that environment. In the first statement, the `%global` isn't really necessary, because the assignments are made in open code. However, note the behavior of the macro variable `var3`, which is not given an explicit environment with either a `%global` or `%local` statement when it is defined in macro `test2`. When macro `test` is invoked from macro `test2`, the value of `var3` is available, but not when it is invoked from open code. Note also that `var2` is declared as a local variable in macro `test2`, so that the value that it is assigned only remains while macro `test2` is executing – when `test` is invoked again in open code, the value given to `var2` inside of macro `test2` is no longer available. This may seem a little cumbersome at first, but it allows for a great deal of flexibility.

TIPS

Define all variables as either global or local

Set aside specific variables for `%do` loop indices and *always* define them as local to avoid inadvertently changing their values in other macros.

ASSIGNING VALUES TO MACRO VARIABLES

We have seen how macro variables are assigned values with the `%let` statement. However, there are instances where we want to reference SAS data sets to get the values for the macro variables. The `CALL SYMPUT` statement can be used to assign values to macro variables during `DATA` step execution, while the `SYMGET` function is used to retrieve values during `DATA` step execution (macro variables can also be resolved directly during `DATA` step compilation). This example illustrates the use of `SYMPUT` and `SYMGET`. The macro variables `mth` and `yr` are resolved in two different ways: first by resolving directly and then by using the `SYMGET` function. Macro variables that are set using `CALL SYMPUT` are not available to be resolved directly until *after* the DATA step is finished; this distinction is apparent in the resulting SASLOG:

```
1    %global month year;
2    %let mth=7;
3    %let yr=2003;
4
5    data _null_;
6    length test mthname $ 16;
7    * These statements are equivalent;
8    month=mdy(&mth,1,&yr);
9    put month= date9.;
10   month=mdy(SYMGET('mth'),1,SYMGET('yr'));
11   put month= date9.;
12   mthname=put(month,monname9.);
13   put mthname=;
14   * Now load into a macro variable;
15   CALL SYMPUT('mthname',mthname);
16   * Use SYMGET to get value;
17   test=SYMGET('mthname');
18   put test=;
19   test="&mthname";
WARNING: Apparent symbolic reference MTHNAME not resolved.
20   put test=;
21   run;

NOTE: Character values have been converted to numeric values at the places given by:
      (Line):(Column).
      10:11   10:29
month=01JUL2003
month=01JUL2003
mthname=July
test=July
test=&mthname

22   %put ** &mthname **;
**      July        **
```

Note that the macro variable mthname cannot be resolved directly while the `DATA` step is still executing, but is available in the `%put` statement immediately afterwards. Also, even though the macro variables `mth` and `yr` contained numeric values, SAS performs a character to numeric conversion when the `SYMGET` function is used. This is because `SYMGET` always resolves the macro variable to a text string, even when – as in this case – it contains a valid numeric value[1].

The macro variable `mthname` contains both leading and trailing spaces, even though when we used the PUT statement to show the value of the data step variable `mthname` there were no leading spaces. Often, it is desirable to remove extra spaces by using the LEFT, TRIM, or COMPRESS functions in the CALL SYMPUT statement. The following will load the value into the macro variable without leading or trailing spaces:

```
...
CALL SYMPUT('mthname',trim(left(mthname)));
...
%put ** &mthname **;
** July **
```

Question: What is the environment for `mthname`, since it is not defined with a `%local` or `%global` statement?

[1] In SAS/SCL applications, the functions `SYMGETN` and `SYMGETC` are available to retrieve macro variables as into numeric and character variables, respectively.

Answer: Since it is defined by a `CALL SYMPUT` in open code, it is a global macro variable. However, if this `DATA` step were inside a macro, then the value would not be available outside of the macro.

Another method of setting macro variables is `PROC SQL`. For example, suppose we want a separate list of character and numeric variables in data set `out.testdata`:

```
proc sql noprint;
select name into: num_vars separated by ' '
from dictionary.columns
where compress(upcase(libname|| '.' ||  memname))='OUT.TESTDATA' and type = 'num';
select name into: char_vars separated by ' '
from dictionary.columns
where compress(upcase(libname|| '.' ||  memname))='OUT.TESTDATA' and type = 'char';
quit;

%put * &num_vars *;
* units revenue month_yr *
%put * &char_vars *;
* ssn name region *
```

The `select into:` statement loads values into a macro variable; `dictionary.columns` is a virtual table which has information on SAS data sets. `PROC CONTENTS` and `SCL` statements can also be used to get information on SAS data sets.

PARAMETERS

The macro language allows passing of parameters in much the same way as other programming languages. A SAS macro can have two types of parameters: positional and keyword. Positional parameters are defined only by their order in the macro invocation and must always be included in the macro invocation, while keyword parameters are defined by the name of the parameter and do not have to be included. A macro can contain both positional and keyword parameters, but the positional parameters must come first. Here is an example of a macro with keyword parameters:

```
%macro smart_print(dsn=_LAST_,title=,by=,id=,var=,dsnopt=,options=);
%* print the specified dataset, using the specified variables in the BY, ID, and VAR
   statements and included options;

%if &by ne %then %let by=BY &by;
%if &id ne %then %let id=ID &id;
%if &var ne %then %let var=VAR &var;
%if %quote(dsnopt) ne %quote() %then
   %let dsnopt=%str ( (&dsnopt) );

TITLE "&title ";
PROC PRINT &options DATA=&dsn &dsnopt;
&by;
&id;
&var;
RUN;
%mend smart_print;
```

Here is the macro invocation and resulting SASLOG:

```
%smart_print(dsn=sales,var=customer amount, dsnopt=%str(where=(state='GA')),
          by=state, title=GA sales, options = noobs);

MPRINT(SMART_PRINT):   TITLE "GA sales ";
MPRINT(SMART_PRINT):   PROC PRINT noobs DATA=sales (where=(state = 'GA')) ;
MPRINT(SMART_PRINT):   BY state;
MPRINT(SMART_PRINT):   ;
MPRINT(SMART_PRINT):   VAR customer amount;
```

```
MPRINT(SMART_PRINT):   RUN;

NOTE: There were 100 observations read from the data set WORK.SALES.
      WHERE state='GA';
```

Note how the order of the parameters in the invocation is not the same as in the macro declaration and that we did not have to specify the `id` parameter. If we had used positional parameters, we would have had to not only specify the parameters in the same order but also use placeholders for the unneeded parameters. Here is the definition and invocation of the same macro with positional parameters:

```
%macro smart_print(dsn,title,by,id,var,dsnopt,options);
...
%mend smart_print;
%smart_print(sales,GA sales,state,,customer amount ,%str(where=(state='GA')),noobs);
```

We need to include an extra placeholder for the `id` parameter and specify the parameters in the same order as in the definition. For more complex macros, keyword parameters are preferable.

Note that this print macro did not perform any error-checking (ensuring that the data set exists, that the variables are found, that the options given are valid, etc.). Often, a decision has to be made about how much programming time is worth investing in a macro – depending on how often it will be used, whether it will be made available to other users, etc.

In some instances, you may want to define a macro with a varying number of parameters. There are two different ways to do this. In a simple case, you could define one parameter and then extract the individual values out of it within the macro. For example, here is a macro that will print multiple data sets and the resulting SASLOG:

```
%macro _printmany(dsns);
%* multiple datasets to be printed are in parameter dsns - separated by spaces ;
%local i dsname;
%let i=1;
%let dsname=%scan(&dsns,&i,%str( ));
%do %while (&dsname ne);
   proc print data=&dsname;
   run;
   %let i=%eval(&i+1);
   %let dsname=%scan(&dsns,&i,%str( ));
%end;
%mend _printmany;
%_printmany(sasuser.admit sasuser.company sasuser.credit);

MPRINT(_PRINTMANY):   proc print data=sasuser.admit;
MPRINT(_PRINTMANY):   run;

MPRINT(_PRINTMANY):   proc print data=sasuser.company;
MPRINT(_PRINTMANY):   run;

MPRINT(_PRINTMANY):   proc print data=sasuser.credit;
MPRINT(_PRINTMANY):   run;
```

This allows for printing a different number of data sets by including all of them in the macro invocation; the `%scan` function – which is analogous to the `SCAN` function in base SAS – is used to parse the passed parameter and process each data set.

Another way to allow for a variable number of parameters is to use the PARMBUFF option on the macro declaration:

```
%macro _printmany / PARMBUFF;
%* multiple SAS datasets to be printed are passed and will be in macro variable syspbuff;
%local i sysbuff dsname;
%* get rid of parenthesis in syspbuff;
%let sysbuff=%substr
```

```
   (&syspbuff,2,%length(&syspbuff)-2);
%let i=1;
%let dsname=
   %scan(%quote(&sysbuff),&i,%str( ,));
%do %while (&dsname ne );
   proc print data=&dsname;
   run;
   %let i=%eval(&i+1);
   %let dsname=%scan(%quote(&sysbuff),&i,%str( ,));
%end;
%mend _printmany;
%_printmany(sasuser.admit,sasuser.company,sasuser.credit);
```

Note that this allows the flexibility of including commas in the passed parameter; with positional or keyword parameter lists, the commas would be interpreted as delimiters between parameters.

SOME SPECIAL RULES FOR RESOLVING MACRO VARIABLES

There are many rules for macro variable resolution; here are a few of the more common ones:

You can use a period (.) to indicate to the macro compiler that you have reached the end of a macro variable name:

```
%let x=hello,;
%let x1=hello, world;
%let y=1;

%put ** &x1 **;
%put ** &xy **;
%put ** &x.y **;
%put ** &x.&y **;
%put ** &&x&y **;

** hello, world **
WARNING: Apparent symbolic reference XY not resolved.
** &xy **
** hello,y **
** hello,1 **
** hello, world **
```

The first `%put` statement produces `hello, world` – as expected. The second statement produces a warning that the macro variable `xy` does not exist. The third statement produces the string `hello,y` (note that there is no period), while the fourth produces `hello,1` (because it resolves the macro variable x and then the macro variable `y`). The final statement produces `hello, world` – it first resolves `&&x&y` to `&x1` and then resolves `&x1` to `hello, world`.

There are some instances where you *don't* want the macro compiler to attempt to resolve what follows the & or % sign. In that case, you can use the `%NRSTR` function to indicate that the text inside the parenthesis should be interpreted as text and not macro variables or macro invocations:

```
%let x=1;
%let y=2;
%let z=&x + &y;
%let zz=%nrstr(&x + &y);
%put ** &z **;
** 1 + 2 **
%put ** &zz **;
** &x + &y **
```

In the first instance, the macro variables `x` and `y` are resolved, but in the second instance the text strings &x and &y are produced. This brings up another interesting question – what is the value of the macro variable `j` after the following `%let` statement?

```
%let j=1+1;
```

Consider the following macro:

```
%macro _test(start);
%local j;
%let j=&start + 1;
%put * &j *;
%if &j eq 2 %then %put * &j=2 *;
%else %put * &j ^= 2 *;
%if &j eq 2.0 %then %put * &j=2.0 *;
%else %put * &j ^= 2.0 *;
%mend _test;
%_test(1);

* 1 + 1 *
* 1 + 1 = 2 *
* 1 + 1 ^= 2.0 *
```

Note that the text string 1+1, rather than the text string 2, has been loaded into macro variable `j`. The first `%if` statement executes while the second one does not. What happened? Obviously 2 and 2.0 have the same mathematical value. The macro compiler performed an implicit mathematical comparison in the first case, but not in the second because it doesn't recognize non-integer values. Rather than counting on this implicit behavior, it is preferable to use the `%eval` function when you want to perform arithmetic operations within the macro compiler. The statement `%let j=%eval(&start+1);` will set the macro variable `j` equal to the text string 2, rather than the text string 1+1. Note that this only works for the basic arithmetic functions and only integer results are produced (for example, the statement `%let j=%eval(6/4)` will store the value 1 – not 1.5 – in the macro variable `j`). This illustrates a difference between macro and DATA step compilation:

```
%let j=1 + 1;
data _null_;
if &j eq 2.0 then put "* &j = 2.0 *";
else put "* &j ^= 2.0 *";
run;
* 1 + 1 = 2.0 *
```

Finally, caution should be used when macro variables can contain characters that have special meaning to the macro or SAS compiler. For example, if a macro variable contains a mismatched quote and it is resolved, it will cause problems:

```
data null;
set orders;
if customer eq 123 then do;
   call symput('firstname',first_name);
   call symput('lastname',last_name);
   stop;
end;
run;

%let fullname = &lastname, &firstname;
proc print data=orders(where=(customer eq 123));
title "All orders for &fullname";
run;
```

Now, if the variable `last_name` contains a single quote (e.g. O'Brien), this will create havoc for the SAS compiler. It will keep looking for the closing quote to match the end the quoted string in the macro variable `fullname`. In this instance, you would want to use one of the quoting functions available in the macro language. Here is a brief description of the available functions:

The `%QUOTE` and `%NRQUOTE` functions are used to mask special characters and operators – NRQUOTE also masks the & and %. Unmatched quotes or parentheses must be marked with a leading %.

The `%BQUOTE` and `%NRBQUOTE` functions are similar, but unmatched quotes or parentheses do not need to be marked.

%`SUPERQ` masks everything – it is also the only one of the quoting functions that accepts the macro variable name *without* the leading ampersand.

In the above case, any of the following statements would work:

```
%let fullname= %bquote(&lastname), %bquote(&firstname)  ;
%let fullname= %nrbquote(&lastname), %nrbquote(&firstname)  ;
%let fullname= %superq(lastname), %superq(firstname)  ;
```

MACRO STYLE AND COMMENTS

There are style issues when writing macro code, just as there are for regular SAS code.

Use good, clean style. This is especially important because macro code is usually less readable than base SAS code. Some examples of good macro style include: indenting `%do` groups, using white space, and – most importantly – using comments liberally.
Use keyword parameters and define macro variables as needed.

Everyone has programming conventions that he or she likes to use. Here are a few that I use to help keep my macro code as readable as possible:

Use lower case for macro code – except text strings that must be upper case.
Avoid use of the `%goto` statement – it makes the program very hard to follow
Define and initialize all global macro variables at the beginning of the program.
Use the `%local` statement to define macro variables that will only be needed inside the current macro.
When writing comments, understand the difference between the `%*` and the * comment statements:

`%*` is a **macro compiler comment** – the macro compiler will ignore the statement
`*` is a **SAS comment** – the macro compiler will not ignore the statement, so it must be an appropriate place in the macro code for a SAS comment.

Here is an example – where would the use of a `*` comment instead of a `%*` comment cause an error?

```
%macro _missing(var=,type=);
%* this macro will set a variable var to missing, vartype=C indicates a character
   variable, else numeric - must be called from a DATA step;

%* if character, use blank ;
%if %upcase(type) eq C %then %do;
   &var = ' ';
%end;
%* if numeric, use .;
%else %do;
    &var = .;
%end;
%mend _missing;

data dumb;
if x=0 then %_missing(var=Y,type=N);
run;
```

If the first or third comments were written using a `*` comment, this `DATA` step would produce a syntax error. If the first comment had a `*` comment, the SAS compiler would see this statement: `if x=0 then * this macro will … ; Y=.` This will of course cause an error, because an inline comment must be enclosed within `/*` and `*/`. Why would the third comment cause a problem? The macro compiler is looking for an `%else` statement immediately after the end of the first `%do` loop – it ignores the `%*` comment, but would treat a `*` comment as a statement, and therefore will produce an error when it comes to the `%else` statement. Note that the /* */ comment works also in the macro language. It's also important to remember that the de-bugging process for macro code is more difficult than for regular SAS code. Because of that, it is even more critical to document programs that include macros. The `mprint`, `mtrace`, and `symbolgen` options make it easier to examine what the macro compiler is generating.

OTHER RULES FOR MACROS

It is important to place macro invocations in your SAS code in such a way that the generated code does not cause problems for the SAS compiler, even if the macro itself generates the SAS code properly:

```
data neworders;
set orders;
...
if order_amount gt 100 then do;
   %smart_print(dsn=neworders);
   stop;
end;
run;
```

This will fail because the `smart_print` macro defined earlier generates a `PROC PRINT` step, so it cannot be placed in the middle of a `DATA` step. This was a somewhat trivial example and the problem would be quickly identified and corrected; however, here is one that might be more difficult to spot:

```
%macro swap(var1,var2);
%* swap the values of two variables;
_temp_=&var1;
&var1=&var2;
&var2=_temp_;
%mend swap;

data test;
infile cards;
input x1 x2;
if x1 > x2 then %swap(x1,x2);
cards;
1 2
3 4
5 1
;
run;
```

It appears that the swap macro will exchange the values of `x1` and `x2` in the third observation, leaving their values unchanged in the first two observations. However, here are the results of the `DATA` step:

```
proc print data=test;
Title 'using swap when x1>x2';
var x1 x2;
run;

using swap when x1>x2
Obs   x1   x2
 1     2    .
 2     4    .
 3     1    5
```

and the code as it appears in the SASLOG:

```
1130  data test;
1131  infile cards;
1132  input x1 x2;
1133  if x1 > x2 then %swap(x1,x2);
MPRINT(SWAP):   _temp_=x1;
MPRINT(SWAP):   x1=x2;
```

```
MPRINT(SWAP):   x2=_temp_;
1134  cards;

NOTE: The data set WORK.TEST has 3 observations and 3 variables.
```

What went wrong? Even with the MPRINT option turned on, it may not be immediately apparent. The swap macro produces three assignment statements, but is called within a single if-then statement, so the second and third statements are always executed, regardless of the results of the if x1 > x2 comparison. It is the same as the following code:

```
if x1>x2 then _temp_=x1;
x1=x2;
x2=_temp_;
```

Therefore, when if x1<x2 is false, the value of x2 is assigned to x1 and x2 becomes missing (since _temp_ is missing). In this case, the macro either needs to generate do; and end; statements or be invoked within a do group in the DATA step:

```
%macro swap(var1,var2);
do;
   _temp_=&var1;
   &var1=&var2;
   &var2=_temp_;
end;
%mend swap;
```

OR

```
data test;
...
if x1 > x2 then do;
   %swap(x1,x2)
end;
run;
```

AN EXAMPLE WITH UTILITY MACROS

This is an example of how macros can be used to save time and run programs more efficiently. For example, suppose that there are large flat files that contain transactional records that come out of a billing system each month for different markets, and that these become available at different times. Rather than waiting for all the files to be available, we would like to provide reporting on each market as soon as possible.

The following utility macros typically would be included in a macro library or an autoexec file.

```
%macro _mprint;
%global _mp _notes;
%* return current mprint and notes setting in mp and _notes ;
%let _mp=%sysfunc(getoption(mprint));
%let _notes=%sysfunc(getoption(notes));
%mend _mprint;

%macro _options(option=);
%* Return the value of the specified option.  Calling statement should be: %let
   x=%_options(option=);

%local value;
%let value=%sysfunc(getoption(&option));
&value
%mend _options;

%macro exist(_dsn_);
```

```
%* determine if a dataset exists -- if so, return the number of obs., number of
variables, date last modified, and whether there is an index in global macro variables;

%global exist nobs nvars dsndate isindex;
%local rcid dsid;
%* try to open dataset;
%let dsid = %sysfunc(open(&_dsn_));
%if &dsid ne 0 %then %do;
   %let exist=yes;
   %let nobs=%sysfunc(attrn(&dsid,NOBS));
   %let nvars=%sysfunc(attrn(&dsid,NVARS));
   %let dsndate=%sysfunc(attrn(&dsid,MODTE));
   %let isindex=%sysfunc(attrn(&dsid,ISINDEX));
   %let rcid=%sysfunc(close(&dsid));
%end;
%else %do;
   %let exist=no;
   %let nobs =%str(.);
   %let nvars=%str(.);
   %let dsndate=%str(.);
   %let isindex=%str(.);
%end;
%mend exist;

%macro fexist(fname);  %* determines if an external file exists;
%global fexist _fdate _fdatetime;
%local rc ranx delcmd;
%let _fdate=.;
%let _fdatetime=.;

%mprint;
options nomprint nonotes;

%* create random number for directory file creation;
data _null_;
call symput('ranx',put(ranuni(0)*1000,z4.0));
run;

%let fexist=no;
%if "&fname" ne "" %then %str(
    data _null_;
    if 0 then infile "&fname";
    call symput('fexist','yes');
    stop;
    run;
);

%* get creation date of file;
%if &fexist eq yes %then %do;
    data _null_;
    %if &sysscp eq WIN %then %do;
        rc=system("dir &fname > _dir&ranx..txt");
    %end;
    %else %do;
       rc=system("ls -l &fname > _dir&ranx..txt");
    %end;
    call symput('rc',compress(put(rc,8.0)));
    run;

    %if &rc eq 0 %then %do;
       data _null_;
       infile "_dir&ranx..txt"
       %if &sysscp eq WIN %then %do;
```

```
            firstobs=6;
            input fdate mmddyy10.  @13 time time8.;
         %end;
         %else %do;
            ;
            length dummy1-dummy5 $ 18 timeyearc $ 5 mthc $ 3;
            input dummy1-dummy5 mthc date timeyearc;
            do i=1 to 12;
               if upcase(mthc) eq upcase(put(mdy(i,1,2003),monname3.))
                  then mth=i;
            end;
            if index(timeyearc,':') then do;
               * current year -- use time in stamp;
               yr=year(today());
               time=input(timeyearc,time5.2);
            end;
            else do;
            * previous year, so bump ahead to 12:00 am next day;
               time=input('00:00',time5.2);
               yr=timeyearc;
            end;
            fdate=mdy(mth,date,yr);
         %end;
         ftime=fdate * 60 * 60 * 24 + time;
         call symput('_fdate',put(fdate,5.));
         call symput('_fdatetime',compress(put(ftime,best18.)));
         %if &sysscp eq WIN %then %str(STOP;);
         run;

         %if &sysscp eq WIN %then %let delcmd=del;
         %else %let delcmd=rm;

         data _null_;
         rc = system("&delcmd _dir&ranx..txt");
         run;

      %end;
   %end;

   options &mp &_notes;
   %mend fexist;
```

The first macro simply captures the current setting of the mprint and notes SAS options. Often, a macro will generate a lot of code and/or notes and we don't want to see it all in the SASLOG so we might want to use the NOMPRINT and/or NONOTES option; however, it's nice to be able to set those values back to what they were before the macro was invoked. The second is a more general-purpose macro that will return the value of any specified SAS option. The `exist` macro determines whether a specified data set exists or not and, if it does, gathers some information about it. The `fexist` macro is a similar macro for an external (non-SAS) file.

```
* This program will read transactional files for each market and create datasets for
reporting.  It will update a dataset indicating which markets are available. Load market
codes into macro variables and set value of mkts.  Typically, this would be done either
in an autoexec file or with a format, etc. - here we are just using %let statements;

libname out 'c:\';
%let month=jan2004;    * desired month;
%let mkt1=NYC;
%let mkt2=ATL;
%let mkt3=MIA;
%let mkt4=LAX;
%let mkt5=LV;
%let mkt6=DLS;
```

```
%let mkt7=CHI;
%let mkt8=BOS;
%let mkt9=PHI;
%let mkt10=DC;
%let mkts=10;

%macro _read;
%local i filenme read;

%do i=1 %to 10;
  %local complete&i dsndate&i;
  %let filenme=c:\&month._&&mkt&i...txt;
  %* see if transactional file exists;
  %fexist(&filenme);
  %* Check to see if we already have a dataset for this market;
  %exist(out.&&mkt&i.._&month);
  %put *** &&mkt&i  &dsndate ***;
  %if &exist eq no %then %do;
     %* No dataset - proceed if transactional file is there;
     %if &fexist eq yes %then %do;
        data out.&&mkt&i.._&month;
        infile "&filenme" end=last;
        input i;
        if last then do;
           file print;
           put "reading file for &&mkt&i / &month " _n_ " records read";
        end;
        run;
     %end;
     %else %do;
        data _null_;
        file print;
        put "&&mkt&i file not available";
        run;
     %end;
  %end;
  %else %do;
     %* dataset is there, compare to date of transactional file ;
     data _null_;
     if &dsndate le &_fdatetime then
        call symput('read', 'Y');
     else call symput('read', 'N');
     run;

     %if &read eq N %then %do;
        data _null_;
        dsndate=datepart(&dsndate);
        fdate=datepart(&_fdatetime);
        file print;
        put "data set for &&mkt&i / &month already exists " /
            "Created: " dsndate date9. " File Date:" fdate date9.;
        run;
    %end;
    %else %do;
        %* read new transactional file;
        data out.&&mkt&i.._&month;
        format dsndate transdate datetime.;
        retain transdate &_fdatetime dsndate &dsndate;
        infile "&filenme" end=last;
        input i;
        if last then do;
          file print;
          put "dataset for &&mkt&i / &month being updated. " /
```

```
              "File date: " transdate " dataset date: " dsndate / _n_ " records read";
      end;
      run;
    %end;
 %end;
%end;

%* produce a dataset indicating which markets are available;
%do i=1 %to &mkts;
   %exist(out.&&mkt&i.._&month);
   %let completed&i = %upcase(&exist);
   %if &exist eq yes %then
      %let dsndate&i=&dsndate;
   %else %let dsndate&i=.;
%end;
data out.&month._markets;
length market complete $ 3;
format date_time datetime.;
%do i=1 %to &mkts;
   market="&&mkt&i";
   complete="&&completed&i";
   date_time=&&dsndate&i;
   output;
%end;
run;

proc print data=out.&month._markets;
title "Markets available for &month";
run;
%mend _read;
options mprint;
%_read;
```

Here are some quick notes about this program. Note the use of the `&&` to resolve the macro variables `mkt1`, `mkt2`, etc. This is somewhat analogous to the way arrays are used in a `DATA` step. Also, as we discussed earlier, we need an extra period in the data set name `out.&month._markets` to indicate to stop resolving the macro variable name at that point. Here is a view of the directory – showing the current data sets and files that are available. We can see that the Atlanta and Philadelphia data sets need to be updated while the Chicago and Dallas data sets need to be created; the data sets for Los Angeles, Boston, Las Vegas, and DC are fine and we haven't received the New York or Miami files yet.

```
07/16/2004  09:39 AM            87,040 atl_jan2004.sas7bdat
07/16/2004  09:39 AM            87,040 bos_jan2004.sas7bdat
07/19/2004  06:49 PM            87,040 chi_jan2004.sas7bdat
07/19/2004  06:29 PM            87,040 dc_jan2004.sas7bdat
07/19/2004  06:49 PM            87,040 dls_jan2004.sas7bdat
07/19/2004  04:28 PM            58,894 jan2004_ATL.txt
07/15/2004  04:29 PM            58,894 jan2004_BOS.txt
07/19/2004  04:29 PM            58,894 jan2004_CHI.txt
07/19/2004  04:29 PM            58,894 jan2004_DC.txt
07/19/2004  04:28 PM            58,894 jan2004_DLS.txt
07/01/2004  04:28 PM            58,894 jan2004_LAX.txt
07/15/2004  04:29 PM            58,894 jan2004_PHI.txt
07/06/2004  09:39 AM            87,040 lax_jan2004.sas7bdat
07/19/2004  06:48 PM            87,040 lv_jan2004.sas7bdat
07/06/2004  09:39 AM            87,040 phi_jan2004.sas7bdat
```

Let's look at portions of the `SASLOG` and `OUTPUT` :

```
MPRINT(_READ):   data _null_;
MPRINT(_READ):   if 1405589941.424 le 1405873680 then call symput('read', 'Y');
MPRINT(_READ):   else call symput('read', 'N');
MPRINT(_READ):   run;
```

```
MPRINT(_READ):   data out.ATL_jan2004;
MPRINT(_READ):   format dsndate transdate datetime.;
MPRINT(_READ):   retain transdate 1405873680 dsndate 1405589941.424;
MPRINT(_READ):   infile "c:\jan2004_ATL.txt" end=last;
MPRINT(_READ):   input i;
MPRINT(_READ):   if last then do;
MPRINT(_READ):   file print;
MPRINT(_READ):   put "dataset for ATL / " "jan2004 being updated." / "File " "date: "
transdate
" dataset " "date: " dsndate / _n_ " records read";
MPRINT(_READ):   end;
MPRINT(_READ):   run;
```

The Atlanta data set is being updated because the transaction file is more current; this next portion of the SASLOG shows that the Los Angeles data set is not updated:

```
MPRINT(_READ):   data _null_;
MPRINT(_READ):   if 1404725959.453 le 1404318480 then call symput('read', 'Y');
MPRINT(_READ):   else call symput('read', 'N');
MPRINT(_READ):   run;

MPRINT(_READ):   data _null_;
MPRINT(_READ):   dsndate=datepart(1404725959.453);
MPRINT(_READ):   fdate=datepart(1404318480);
MPRINT(_READ):   file print;
MPRINT(_READ):   put "data set for LAX / " "jan2004 already exists " / "Created: "
dsndate
date9. " File Date:" fdate date9.;
MPRINT(_READ):   run;
```

Here's a portion of the OUTPUT, which shows how the macro processed each market:

```
NYC file not available
...
dataset for ATL / jan2004 being updated.
File date: 19JUL04:16:28:00  dataset date: 16JUL04:09:39:01
10000  records read
...
data set for LAX / jan2004 already exists
Created: 06JUL2004 File Date:01JUL2004
...
data set for BOS / jan2004 already exists
Created: 16JUL2004 File Date:15JUL2004
...
dataset for PHI / jan2004 being updated.
File date: 15JUL04:16:29:00  dataset date: 06JUL04:09:39:26
10000  records read
...
```

Obs	market	complete	date_time
1	NYC	NO	.
2	ATL	YES	20JUL04:09:47:41
3	MIA	NO	.
4	LAX	YES	06JUL04:09:39:19
5	LV	YES	19JUL04:18:48:05
6	DLS	YES	19JUL04:18:49:11
7	CHI	YES	19JUL04:18:49:30
8	BOS	YES	16JUL04:09:39:06
9	PHI	YES	20JUL04:09:47:41
10	DC	YES	19JUL04:18:29:33

A MACRO TO LOAD MACRO VARIABLES

In the previous macro, we specified the markets via a series of `%let` statements. Often, we will want to create macro variables in a more automated fashion. Here is an example of a macro I wrote that allows me to load values from a data set into macro variables in a variety of ways.

```
%macro _loadmac(dsn=,var=,macrovar=, byvar=,sep=,series=,numvals=,where=,macfmt=);
%local i type fmt ranx _dsn;

%* This macro will create either a single macro variable or a series of macro variables
with values from a specified dataset.

Parameter list:

dsn REQUIRED - name of dataset
var REQUIRED - name of variable with values in dataset
macrovar REQUIRED- name of macro variable
byvar OPTIONAL - name of variable to sort dataset before loading
sep OPTIONAL - separator character
series OPTIONAL - indicates whether a series of macro variables is desired
numvals OPTIONAL - macro variable to hold number of values in series
where OPTIONAL - where clause for dataset
macfmt OPTIONAL - formatting for value
;

%if %scan(&dsn,2,.) eq %then %let _dsn=WORK.&dsn;
%else %let _dsn=&dsn;

%let series=%upcase(&series);
%if "&sep" eq "" %then %let sep=%str( );
%let fmt=&macfmt;

%* generate global statement for output macro variabes;
%if &series ne Y %then %do;
    %global &macrovar;
%end;
%if &numvals ne %then %do;
    %global &numvals;
%end;

%* generate random number for dataset use within macro;
data _null_;
call symput('ranx','data' || put(ranuni(0)*10000,z4.0));
run;

%* get format/type of original variable;
proc sql noprint;
   select compress(format), case when type='num' then '1' else '2' end
   into: fmt, :type
from dictionary.columns
where (compress(upcase(libname || '.' ||memname))="&_dsn") and
      upcase(name)=upcase("&var");
quit;

%* use specified format if there is one;
%if &macfmt eq NONE %then %let fmt=;
%else %if &macfmt ne %then %let fmt=&macfmt;

%* make copy of original dataset, using where to restrict obs ;
data &ranx;
set &dsn;
%if &where ne %then %str(WHERE &where;);
run;
```

```
   %if &byvar ne %then %do;
      * sort dataset as desired;
      proc sort data=&ranx;
      by &byvar;
      run;
   %end;

   %exist(&ranx);  %* get number of obs. ;
   %if &numvals ne %then %let &numvals=&nobs;
   %if &series eq Y %then %do;
      %do i=1 %to &nobs;
         %global &macrovar.&i;
      %end;

      data _null_;
      set &ranx end=last;
      %if &fmt ne %then %do;
         call symput("&macrovar" || compress(put(_n_,12.0)),put(&var,&fmt));
      %end;
      %else %do;
         %if &type eq 1 %then %do;
            call symput("&macrovar" || compress(put(_n_,12.0)),compress(put(&var,best18.)));
         %end;
         %else %do;
            call symput("&macrovar" || compress(put(_n_,12.0)) , &var);
         %end;
      %end;
      run;
   %end;
   %else %do;
      proc sql noprint;
      %if &fmt ne %then %do;
          select put(&var,&fmt)
      %end;
      %else %do;
          select &var
      %end;
      into: &macrovar separated by "&sep"
      from &ranx;
      quit;
   %end;

   * delete temporary dataset ;
   proc datasets lib=work nolist;
        delete &ranx;
   quit;

%mend _loadmac;
```

To illustrate how this works, here is a very small data set `markets` that we will use as the source for extracting values into macro variables:

```
   market_               market_
   number  market         code     region

    01     New York       NYC       East
    02     Atlanta        ATL       East
    03     Miami          MIA       East
    04     Los Angeles    LAX       West
    05     Las Vegas      LV        West
    06     Dallas         DLS       West
    07     Chicago        CHI       West
```

```
08     Boston         BOS       East
09     Philadelphia   PHI       East
10     Washington     DC        East
```

These markets correspond to the 10 markets we used in the previous macro. Note that `market_number` is a numeric variable with a `Z2.` format. Now suppose we want to load all the market codes into a series of macro variables – the same thing we previously accomplished with the series of `%let` statements – we could use the following:

```
%_loadmac(dsn=markets,var=market_code,macrovar=mkt,numvals=mkts,series=Y);
```

Let's look at some pieces of the resulting SASLOG:

```
proc sql noprint;
select compress(format),
       case when type='num' then '1' else '2' end  into: fmt, :type
from dictionary.columns
where (compress(upcase(libname|| '.' || memname))="WORK.MARKETS") and
      upcase(name)=upcase("market_code");
quit;

* make copy of original dataset, using where to restrict obs ;
data data4982;
set markets;
run;

data _null_;
set data4982 end=last;
call symput("mkt" || compress(put(_n_,12.0)),market_code);
run;
```

Here are the values of the macro variables created:

```
%put * &mkts *;
* 10 *
%put * &mkt1 *;
* NYC *
%put * &mkt2 *;
* ATL *
%put * &&mkt&mkts *;
* DC  *
```

As another example, suppose we want a single macro variable with the values of all of the market names. We could use:

```
%_loadmac(dsn=markets,var=market,macrovar=mktnames)
```

This produces the macro variables `mktnames`, with the value:

```
New York Atlanta Miami Los Angeles Las Vegas Dallas Chicago Boston Philadelphia
Washington
```

Note that parsing this macro variable later might be a problem because some of the markets contain an embedded space. If we had specified `sep=|` in the macro invocation, we'd have:

```
New York|Atlanta|Miami|Los Angeles|Las Vegas|Dallas|Chicago|
Boston|Philadelphia|Washington
```

This is another example of having to use caution with macro variables; if we wanted to separate the values with a comma, we couldn't use `sep=,` because that would confuse the compiler – it would think that the comma was indicating that another parameter was following the `sep=` parameter. We would have to use `sep=%str(,)` instead.

Let's look at what happens when we invoke the macro with the following:

```
%_loadmac(dsn=markets,var=market_number, macrovar=mkt,series=Y,macfmt=4., numvals=
num_mkts, where=%str(region='East'),byvar=market);
```

This will use the same data set but will load the values of `market_number` instead of `market`, generate a series of macro variables using the prefix `mkt`, using a format of `4.`, return the number of values in the macro variable `num_mkts`, restrict the results to markets in the East region, and order the results by the value of `market`. Note that we could have specified these parameters in any order since they are keyword parameters.

```
%put * &num_mkts *;
%put * &mkt1 *;
%put * &mkt2 *;
%put * &&mkt&num_mkts *;

533   %put * &num_mkts *;
* 6 *
*    2 *
*    8 *
*   10 *
```

Now there are only 6 markets loaded and note how specifying a format of 4. has caused leading spaces in the `mkt` macro variables.

These are a few examples of the type of applications that can be written using the SAS macro language. Although there is some additional development time when first writing a program like this, it will often save time when implemented in a production-like environment where the program is run often. Additionally, the ability to conditionally generate certain `DATA` and `PROC` steps gives the application much more flexibility.

CONCLUSION

We have just scratched the surface of what can be done with macros. For the beginning user, they can be used to make programs more automated. Once we feel more comfortable with them and understand how powerful they are, we can use them in complex production applications.

REFERENCES

SAS Institute, Inc. (1990), *SAS Guide to Macro Processing, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Your comments and questions are encouraged. Please contact the author:

Andrew M. Traldi
Cingular Wireless, MKIS & Affiliate Marketing
12555 Cingular Way
Alpharetta, GA 30004
678-893-1325
andrew.traldi@cingular.com