MACROS: TIPS, TECHNIQUES, AND EXAMPLES

Andrew M. Traldi, Advanced Quantitative Solutions, Inc., Alpharetta, 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=Jul2001;
…
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 dataset 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;
208  else do;
209     put 'Hello';
210  end;
211  run;

%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, it is never passed to the SAS compiler. This is an example of conditional execution vs. conditional compilation.

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 the resulting SASLOG:

```
1    %global var1;
2    %let var1=hello;
3    %let var2=world;
4    %put ** in open code var1=&var1
var2=&var2 **;
** in open code var1=hello var2=world **
5
6    %macro test;
7    %put ** in test: var1=&var1
var2=&var2 var3=&var3 **;
8    %mend test;
9    %test
WARNING: Apparent symbolic reference VAR3
not resolved.
** in test: var1=hello var2=world
var3=&var3 **
10
11   %macro test2;
12   %local var2;
13   %let var1=hi;
14   %let var2=universe;
15   %let var3=hello, world;
16   %put ** in test2: var1=&var1
var2=&var2 var3=&var3 **;
17   %test;
18   %mend test2;
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 (open code or a macro) 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 can be assigned values with the `%let` statement. However, there are many instances where we need to reference SAS datasets for the values that we want. The `CALL SYMPUT` statement is 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`:

```
%global mth yr;
%let mth=7;
%let yr=2002;

data _null_;
length test mthname $ 16;
* These statements are equivalent;
month=mdy(&mth,1,&yr);
put month= date9.;
month=mdy(SYMGET('mth'),1,SYMGET('yr'));
put month= date9.;
mthname=put(month,monname9.);
put mthname=;
* Now load into a macro variable;
CALL SYMPUT('mthname',mthname);
* Use SYMGET to get value;
test=SYMGET('mthname');
put test=;
test="&mthname";
put test=;
run;

%put ** &mthname **;
```

Note how we integrated the macro variable into a regular SAS statement, first by resolving it directly and then by using the SYMGET function. It is important to remember that macro variables that are set using CALL SYMPUT are not available to be resolved directly by the macro compiler until *after* the DATA step is finished; this distinction can be seen in the resulting SASLOG:

```
1     %global month year;
2     %let mth=7;
3     %let yr=2002;
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).
      11:11   11:29
month=01JUL2002
month=01JUL2002
mthname=July
test=July
test=&mthname
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds
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 month and year contained numeric values, SAS has to perform a character to numeric conversion when the SYMGET function is used. This is because the macro compiler treats all macro variables as text strings, even when they are valid numeric values[1].

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

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

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 the macro.

Another method of setting macro variables is to use PROC SQL. For example, suppose we want a list of all numeric variables in a dataset. Here is an example using PROC CONTENTS and PROC SQL.

```
33    proc contents data=sasuser.admit
noprint out=_cont_;
34    run;

NOTE: The data set WORK._CONT_ has 9
observations and 40 variables.
NOTE: PROCEDURE CONTENTS used:
      real time           0.01 seconds
      cpu time            0.01 seconds
35
36    proc sql noprint;
37    select name into: numeric_vars
separated by ' '
38    from _cont_
39    where type eq 1;
40    quit;
NOTE: PROCEDURE SQL used:
      real time           0.00 seconds
      cpu time            0.00 seconds
41
42    %put &numeric_vars;
Age Date Fee Height Weight
```

PARAMETERS

The macro language allows for passing of parameters in much the same way as other programming languages; those of you who develop SAS/AF applications or have used PASCAL or FORTRAN are probably used to passing parameters to functions and subroutines.

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=name customer
amount,dsnopt=%str(where=(state='GA')),by
=state,title=GA sales, options = noobs);

11   %macro smart_print
12
(dsn=_LAST_,title=,by=,id=,var=,dsnopt=,o
ptions=);
13   %* print the specified dataset,
using the specified variables
14      in the BY, ID, and VAR statements
and included options;
15
16   %if &by ne %then %let by=BY &by;
17   %if &id ne %then %let id=ID &id;
18   %if &var ne %then %let var=VAR &var;
19   %if %quote(dsnopt) ne %quote() %then
         %let dsnopt=%str ( (&dsnopt) );
20
21   TITLE "&title ";
22   PROC PRINT &options DATA=&dsn
&dsnopt;
23   &by;
24   &id;
25   &var;
26   RUN;
27   %mend smart_print;
28   options mprint;
29   %smart_print(dsn=sales,var=name
customer amount,dsnopt=%str(where=(state
=
29 ! 'GA')),by=state,
30     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 name customer
amount;
MPRINT(SMART_PRINT):   RUN;

NOTE: There were 100 observations read
from the data set WORK.SALES.
      WHERE state='GA';
NOTE: PROCEDURE PRINT used:
      real time           0.31 seconds
      cpu time            0.03 seconds
```

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,,name
customer,%str(where=(state='GA')),noobs);
```

Here, 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 which will have 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 datasets:

```
%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);
```

Here is the resulting SASLOG:

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

NOTE: There were 21 observations read
from the data set SASUSER.ADMIT.

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

```
NOTE: There were 8 observations read from
the data set SASUSER.COMPANY.

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

NOTE: There were 10 observations read
from the data set SASUSER.CREDIT.
```

This allows for printing a different number of datasets 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 go through the passed parameter and process each dataset.

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; I will go over a few of the more common ones here.

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 **;

Here are the results:

** 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 not 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;
```

It is the text string 1+1, not the text string 2. This might cause some unexpected results, for example:

```
%macro _uhoh(start);
%local j;
%let j=&start + 1;
%if &j eq 2 %then %do;
   …
%end;
%mend _uhoh;
%_uhoh(1);
```

When this macro is invoked, the `%if &j eq 2 %then %do` loop will not be executed by the macro compiler. Instead, the statement `%let j=%eval(&start+1);` should be used. This instructs the macro compiler to perform an arithmetic operation and put the result in the macro variable j. This only works for the basic arithmetic functions and only integer results are produced (for example, the statement `%let j=%eval(6/4);` will put the value 1 – not 1.5 – in the macro variable j).

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 mark 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.

Use the `%*` rather than the `*` comment statements, understanding the difference between them:

`%*` is a **macro compiler comment** – the macro compiler will ignore the statement and it will not print in the SASLOG

`*` 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 treats the * comment as a statement, and therefore will produce an error when it comes to the %else statement. Incidentally, this can happen in base SAS as well – note that the following will cause an error:

```
data one;
…
if x=1 then do;
 …
end ;;
else do;
 …
end;
run;
```

In most cases, a double semi-colon does not matter to the SAS compiler, but here it expects the else statement to immediately follow the end of the do-loop, and the extra semi-colon causes it to compile an additional statement, producing an error.

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 in this paper) generates a PROC PRINT, 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 a PROC PRINT:

```
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

Here is 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 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 before ;

%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);
%* determine if an external file exists
   if so, return the date and date/time
   of file (based on directory);

%global fexist _fdate _fdatetime;
%local rc;
%let _fdate=.;
%let _fdatetime=.;

%mprint;
%if &_debug eq Y %then %str(options
mprint notes;);
%else %str(options nomprint nonotes;);

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

%if &fexist eq yes %then %do;
   * get creation date of ext. file ;
   data _null_;
   rc=system("ls -l &fname > _dir.txt");
   call symput('rc',(put(rc,6.0));
   run;

   %if %eval(&rc) eq 0 %then %do;
     data _null_;
     length dummy1-dummy5 $ 18
             timeyearc $ 5 mthc $ 3;
     infile "_dir.txt";
     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 ;
         yr=year(today());
         time=input(timeyearc,time5.2);
      nd;
     else do;
          * previous year, time=11:59 pm;
          time=input('23:59',time5.2);
          yr=timeyearc;
     end;
     fdate=mdy(mth,date,yr);
     ftime=fdate * 60 * 60 * 24 + time;
     call symput('_fdate',put(fdate,5.));
     call symput('_fdatetime',
          compress(put(ftime,best18.)));
     run;

  %end;
%end;

options &mp &_notes;
%mend fexist;

* This program will read transactional
files for each market and create datasets
for reporting.  It will update a dataset
indicating which markets are available. ;
```

```
%let month=jan2003;    * desired month;

* 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;

* This has a dummy input line and reads
in one variable - a real transactional
file would have more variables;

%let mkt1=ATL;
%let mkt2=MIA;
%let mkt3=ORL;
%let mkt4=JAX;
%let mkt5=LEX;
%let mkts=5;

libname out 'billing/sasdata';

%macro _read;
%local i filenme read;

%do i=1 %to &mkts;
  %local complete&i;
  %let filenme=
       billing/&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);
  %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";
      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_;
      file print;
      put "dataset for &&mkt&i /"
          "&month already exists";
      run;
    %end;
    %else %do;
      %* read new transactional file;
      data out.&&mkt&i.._&month;
      infile "&filenme" end=last;
      transdate=&_fdatetime;
      dsndate=&dsndate;
      if last then do;
         file print;
         put "updating dsn " dsndate
              datetime16. /
              "with &&mkt&i file "
              transdate datetime16.;
      end;
      input i;
      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);
%end;
data out.&month._markets;
length market complete $ 3;
%do i=1 %to &mkts;
   market="&&mkt&i";
   complete="&&completed&i";
   output;
%end;
run;

proc print data=out.&month._markets;
title "Markets that are now available for
&month";
run;
%mend _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 dataset name `out.&month._markets` to indicate to stop resolving the macro variable name at that point.

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

```
MACROGEN(FEXIST):   options nomprint
nonotes;
MPRINT(_READ):   data _null_;
MPRINT(_READ):   if 1370908641.23525 le
1370908260 then call symput('read', 'Y');
MPRINT(_READ):   else call symput('read',
'N');
MPRINT(_READ):   run;

NOTE: DATA statement used:
      real time           0.00 seconds
      cpu time            0.00 seconds

MPRINT(_READ):   data _null_;
MPRINT(_READ):   file print;
MPRINT(_READ):   put "dataset for ATL /"
"jan2003 already exists";
MPRINT(_READ):   run;

MPRINT(FEXIST):  ;
```

```
MACROGEN(FEXIST):   options nomprint
nonotes
ERROR: Physical file does not exist,
/sasdata/shared/home/traldia/billing/jan2
003_ORL.txt.
MPRINT(FEXIST):  ;
MPRINT(_READ):   data _null_;
MPRINT(_READ):   file print;
MPRINT(_READ):   put "ORL file not
available";
MPRINT(_READ):   run;

MPRINT(_READ):   data out.JAX_jan2003;
MPRINT(_READ):   infile
"billing/jan2003_JAX.txt" end=last;
MPRINT(_READ):   input i;
MPRINT(_READ):   if last then do;
MPRINT(_READ):   file print;
MPRINT(_READ):   put "reading file for
JAX";
MPRINT(_READ):   end;
MPRINT(_READ):   run;

NOTE: The data set OUT.JAX_JAN2003 has
10000 observations and 1 variables.
NOTE: The DATASTEP printed page 36.

MPRINT(_READ):   data _null_;
MPRINT(_READ):   if 1370908641.29471 le
1370908740 then call symput('read', 'Y');
MPRINT(_READ):   else call symput('read',
'N');
MPRINT(_READ):   run;

MPRINT(_READ):   data out.LEX_jan2003;
MPRINT(_READ):   infile
"billing/jan2003_LEX.txt" end=last;
MPRINT(_READ):   transdate=1370908740;
MPRINT(_READ):
dsndate=1370908641.29471;
MPRINT(_READ):   if last then do;
MPRINT(_READ):   file print;
MPRINT(_READ):   put "updating dsn "
dsndate datetime16. / " with LEX file "
transdate
datetime16.;
MPRINT(_READ):   end;
MPRINT(_READ):   input i;
MPRINT(_READ):   run;

NOTE: 10000 records were read from the
infile "billing/jan2003_LEX.txt".
      The minimum record length was 1.
      The maximum record length was 5.
NOTE: The data set OUT.LEX_JAN2003 has
10000 observations and 3 variables.
NOTE: The DATASTEP printed page 37.

MPRINT(_READ): data out.jan2003_markets;
MPRINT(_READ): length market complete $
3;
MPRINT(_READ): market="ATL";
MPRINT(_READ): complete="YES";
MPRINT(_READ): output;
MPRINT(_READ): market="MIA";
MPRINT(_READ): complete="YES";
MPRINT(_READ): output;
MPRINT(_READ): market="ORL";
MPRINT(_READ): complete="NO";
MPRINT(_READ): output;
MPRINT(_READ): market="JAX";
MPRINT(_READ): complete="YES";
MPRINT(_READ): output;
MPRINT(_READ): market="LEX";
MPRINT(_READ): complete="YES";
MPRINT(_READ): output;
MPRINT(_READ): run;

MPRINT(_READ): proc print
data=out.jan2003_markets;
MPRINT(_READ): title "Markets that are
now available for jan2003";
MPRINT(_READ): run;

NOTE: There were 5 observations read from
the data set OUT.JAN2003_MARKETS.
```

OUTPUT

```
dataset for ATL /jan2003 already exists

dataset for MIA /jan2003 already exists

ORL file not available

reading file for JAX
updating dsn 10JUN03:23:57:21
with LEX file 10JUN03:23:59:00

Obs    market    complete

1      ATL        YES
2      MIA        YES
3      ORL        NO
4      JAX        YES
5      LEX        YES
```

This is an example of the type of application 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.

CONTACT INFORMATION

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

Andrew M. Traldi
Advanced Quantitative Solutions, Inc.
5825 Culler Court
Alpharetta, GA 30005
770-418-9167
atraldi@bellsouth.net