Paper CC13

The Care and Feeding of SAS Macro Program Parameters

Don Boudreaux, PhD, SAS Institute Inc., Austin, TX.

ABSTRACT

Most parameterized macros that I have either written or reviewed have taken a great deal of care to correctly perform the primary programming task that they were originally designed for. In retrospect however, I would propose that most also do not spent the same effort in checking the values passed through their parameters. To help remedy this situation, a number of typical parameter value processing and validation examples are presented in this paper. The examples include checking for null values, numeric values, integer values, case specific character values, existing file names, existing libnames, valid dataset names, and validity of the syntax of SAS statements passed as parameter input. Suggestions are also made to help the macro program developer communicate the requirements of the parameters to the user.

INTRODUCTION

SAS programmers generally understand the concept of data documentation and data validation in data and proc step processing. I would suggest that communicating parameter specifications and checking parameter values in macros need to be seen as another form of data which demands the same level of concern and programming effort. Now, this is not to say that there are not arguments in defense of a low level of parameter processing. Try the argument that the effort would result in a macro program that would involve an increase in processing time and resource utilization. It sounds reasonable, but in reality this is very weak. Try the argument that a high level of parameter processing would increase the complexity of the code. The macro program would take longer to develop and be harder to maintain. This is a better argument, but it is still not very convincing. Try the argument that the macro is only being used by its creator who fully understands the parameters and would not code beyond their intended values. Not a great argument, but it would have to be considered the most reasonable. Unfortunately, these arguments lead to macro programs that may only serve a limited audience. A macro that does not properly document and effectively filter its parameter values is unlikely to be easily used by anyone beyond the original author. I would argue that as a macro is distributed beyond the original author it would have to be either very simple or very intuitive to limit its chances to be improperly used. Time is also a factor that can effect the use of macro programs. At a later point in time, a macro that is undocumented might not even be easily used by the original author. The bottom line is that a macro program that has an important function should properly document its purpose and parameters. It should also carefully screen and process any values passed to its parameters.

COMMUNICATING PARAMETER SPECIFICATIONS

The initial task in dealing with macro parameters is documenting the parameters and communicating the requirements to the user. There are several techniques that are typically used to attempt this. These include comment statements, macro-level comment statements, writing messages to datatsets, put statements, and macro-level put statements. For discussion purposes, consider a macro program that allows a user to work with a named macro parameter called CHAR that is intended to only contain a character value of size 1. If we limit our discussion to only communicating information about that single parameter, starting with comment statements, it would be very easy to enter the following into our macro program:

```
* CHAR= a character value of size 1 is required. ;
```

Now, this is a simple and informative comment. Unfortunately, unless the user has access to the macro program code or executes the macro with the MPRINT option on, this statement will not be seen by the users. Even if MPRINT is specified, it is even possible that the statement could be in a section of code that is conditionally executed and would not be presented to the user. The second technique, using a macro-level comment, is also simple to code:

```
%* CHAR= a character value of size 1 is required. ;
```

Unfortunately, this is even more restrictive than using a simple comment statement. The macro-level comment is only available to a user who has access to the macro code! Remember, the MPRINT option only displays code in the log that is sent to the compiler by the macro facility. A third strategy, that is only rarely used but that can surface information outside of the log, is writing messages to a dataset. Consider the following:

```
data work.msg_to_user ;
   note="CHAR= a character value of size 1 is required." ;
run ;

proc print data=work.msg_to_user noobs ;
   var note ;
run ;
```

Within the data step you have the ability to manipulate the messages in any way, print the messages to the log, write the messages to files, or surface them to output with subsequent steps. In the PRINT procedure example shown here you also have the ability to use titles, footnotes, options, and ODS to further customize the resulting presentation of the information. A fourth strategy, really just a variation off of the third, would be to just use the data step with PUT statements. The coding can be a bit simpler than the third strategy and you will lose any presentation enhancements that would have been available with the further procedure processing. But, you retain all of the advantages of data step processing.

```
data _null_ ;
   put "CHAR= a character value of size 1 is required." ;
run ;
```

In either strategy that employs the data step, the put will write text to the log by default. The additional use of a FILE statement will allow you to easily redirect this to a text file. And, the use of PUT and PUTLOG statements would allow you to write to a text file and the log within the same data step.
The fifth strategy, and the last one I'll discuss, is the use of the macro-level put statement. The code for this statement would look like the following:

```
%put CHAR= a character value of size 1 is required. ;
```

This is a macro level statement that always writes to the log. It does not require quotes around the value to be written (unless you wish to see the quotes) and, like other macro level statements, it does not need to be coded within a data step. Using macro-level puts has actually become my preferred way of providing documentation within macro programs I write. Although, I also add in the initial text of NOTE, WARNING, or ERROR to provide color coding to my messages. By default, NOTE generates log messages

in blue, WARNING generates log messages in green, and ERROR give you red. In addition, I also add in a text snippet that identifies the macro being used. The following code shows a number of macro-level puts that are part of a program that I wrote to do some testing on character values. In the example I communicate the purpose of the macro and information about each specific macro parameter using macro-level PUT statements:

```
%macro testMACRO(
   CHAR= ,
   MIN=a ,
   MAX=e ) ;
%put NOTE <testMACRO> ... purpose: ;
%put NOTE:<testMACRO> Check if a single lowercase ;
%put NOTE:<testMACRO> character is within a range. ;
%put NOTE:<testMACRO> The default range is a to e. ;
%put NOTE <testMACRO> ... parameter(s): ;
%put NOTE:<testMACRO> CHAR= single character to check ;
%put NOTE:<testMACRO> MIN=  range minimum value [default: a] ;
%put NOTE:<testMACRO> MAX=  range maximum value [default: e] ;
...
```

FILTER AND/OR VALIDATE - NUMERIC VALUES

Beyond communicating the requirements of the macro parameters is the task of coding to enforce those requirements. Lets start with a check for null values (shown within a macro-level if...then structure) that involve a macro parameter named &n. The question here would be to catch if a macro parameter passed with a null value or was the default null and the value never changed by the user? Notice the between the equals operator EQ and the macro-level %then there is nothing.

```
%* check if macro variable n is null ;
%if &n EQ %then %do ;
   %put ... ;
%end ;
```

Not generally applicable to macro parameters, that would always exits unless they were explicitly deleted, the following uses the %symexist function to check if a macro variable exists.

```
%* check if macro variable n exists ;
%if %symexist(n) EQ 1 %then %do ;
   %put ... ;
%end ;
```

The fact that SAS primarily only has only two data types has always made the language easy to learn and quick to code. However, it does put an increased burden on the programmer to check specific types of data values. Lets look at a few numeric examples starting with integers. Note the use of a Perl regular expression and the %sysfunc function with an embedded prxmatch. This is going to show up in many of the following examples. The regular expression is looking to only consider values that have a leading plus or minus sign followed by one or more numeric digits. Other typical SAS numeric values, such as floating point numbers and scientific notation are not "caught" by this matching.

```
%* check for an integer ;
%let regex = '^ *[\+\-]?\d+ *$' ;
%if %sysfunc(prxmatch(&regex,&n)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

If you want to further restrict the example to only allow a positive integer, remove the minus sign from the regular expression and include a check on zero :

```
%* check for positive integer ;
%let regex = '^ *\+?\d+ *$' ;
%if
   %sysfunc(prxmatch(&regex,&n)) EQ 1 AND
   &n NE 0
%then %do ;
   %put ... ;
%end ;
```

Going the other way, a check on any integer or floating point number (but not scientific notation) would only require a slight modification of the regular expression. Notice the regular expression's check beyond the plus or minus sign and initial integer digits for a possible decimal point and following digits:

```
%* check for any 'simple' number ;
%let regex = '^ *[\+\-]?\d+(\.?\d+)? *$' ;
%if %sysfunc(prxmatch(&regex,&n)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

Checking for a value within a given set of values is also common. Starting with a set of size one to check on a single given known value (in this case a 3):

```
%* check on a single set numeric value ;
%if &n EQ 3 %then %do ;
   %put ... ;
%end ;
```

Next, lets check for a value within the set: 1, 2, or 3. Again, note the use of a Perl regular expression. This might seem a bit complex, but given that the IN operator does not have a macro-level equivalent , it actually is a fairly easy piece of code.

```
%* integer within a list ;
%let regex = '^ *[123] *$' ;
%if %sysfunc(prxmatch(&regex,&n)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

Numbers are also commonly checked to exist within a range. The range is set here with two macro variables: min and max. In real code they would most likely be either hard coded constants or other parameters. The strategy is to check that we have a number and it is within a given range.

```
%* check for a number within a given range ;
%let min = 1 ;
%let max = 5 ;
%let regex = '^ *[\+\-]?\d+(\.?\d+)? *$' ;
%if
   (%sysfunc(prxmatch(&regex,&n)) EQ 1) AND
   (&n GE &min AND &n LE &max)
%then %do ;
   %put ... ;
%end ;
```

A number of the code segments used in this paper have used the macro level function %sysfunc. Do be aware that this function requires a data step function as an argument and will not evaluate an expression by itself. A way around this restriction is to encapsulate a numeric expression inside a sum function or a character value inside of a trim function.

FILTER AND/OR VALIDATE - CHARACTER VALUES

The first example is a 'standard' simple filter to keep the checking of the macro parameter values as simple as possible. Do not modify the same macro variable name if it would be used later in a situation where the case is important - like as a proper name or used in a title or footnote. Notice that we are dealing with a parameter named &c in these character examples.

```
%* set character case ;
%let c = %lowcase(&c) ;
```

If case becomes an issue and presetting the case is not applicable, it needs to be considered in your code. In the following example, what would normally be a very simple check for a single character, just becomes a bit more complex to consider either case.

```
%* check for a Y ;
%let regex = '^ *[Yy] *$' ;
%if %sysfunc(prxmatch(&regex,&c)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

In the next example, a regular expression is used to check for a single character value within a given set range. This is important as the use of GE and LE in the simple expression '%if (&c GE a) AND (&c LE d) %then ...' would return a true if 'c=a' or if 'c=cat' - oops! Unlike the regular expression, it does not consider the number of characters within &c in its checking.

```
%* check for a single character a to d ;
%let regex = '^ *[a-d] *$' ;
%if %sysfunc(prxmatch(&regex,&c)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

Regular expressions can also easily be used to check if you are dealing with a parameter that is a single word. In the following case insensitive check, the regular expression is looking for a blank delimited collection of characters.

```
%* check for a single word input ;
%let regex = '^ *[a-z_A-Z]+ *$' ;
%if %sysfunc(prxmatch(&regex,&c)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

The regular expression shown below would allow all case variations on a check for the word "yes".

```
%* check for the word YES ;
%let regex = '^ *[Yy][Ee][Ss] *$' ;
%if %sysfunc(prxmatch(&regex,&c)) EQ 1 %then %do ;
   %put ... ;
%end ;
```

The last character example is a check on a single word within a list of words. The regular expression is used to check that the parameter &c contains a single word and the %index function checks the list. Without a single word check, a two or three word sequence could be tested as true in the %index. Notice that the lower case value of the parameter is checked against a lower case list without modifying the original value within the macro variable &c.

```
%* check for a word in a list ;
%local word_list ;
%let word_list = the cat is blue ;

%local word ;
%let word = %lowcase(&c) ;

%let regex = '^ *[a-z]+ *$' ;
%if
   (%sysfunc(prxmatch(&regex,&word)) EQ 1) AND
   (%index(&word_list,&word) NE 0)
%then %do ;
   %put ... ;
%end ;
```

FILTER AND/OR VALIDATE - FILES, DATASETS, and LIBREFS

Not everything you need is a number or character value. In this example I am checking for the existence of an external file with a parameter named &f.

```
%* check for an existing file ;
%if (%sysfunc(fileexist(&f)) EQ 1) %then %do ;
   %put ... ;
%end ;
```

You may want to check first that &f is not null. The fileexist function will generate an error with a null value as an argument.

There is also a data step function that checks that a SAS dataset exists. The following macro code checks for a dataset named WORK.ds with the exist function. This function also needs to get a non-null value.

```
%* check for an existing SAS dataset ;
%if %sysfunc(exist(WORK.ds )) EQ 1 %then %do ;
   %put ... ;
%end ;
```

First files, then datasets, and now library references - this example uses the libref function to see if the macro variable &r contains a defined library reference.

```
%* check for an existing libname ;
%if %sysfunc(libref(&r)) EQ 0 %then %do ;
   %put ... ;
%end ;
```

FINAL EXAMPLE - CODE CHECK

The final example is a fully defined macro that will test a named parameter that contains SAS code. The test is to check that the code is free of syntax errors. The macro is named testMACRO and the triggers that it is intended to work against would look like the following:

```
%let mycode = %str( x=1 ; if x EQ 1 then y=2 ; ) ;
%testMACRO(code=mycode) ;

%let mycode = %str(z/\2 ;) ;
%testMACRO(code=mycode) ;
```

In the first case the global macro variable named &mycode contains a simple assignment statement and an if...then. The macro will confirm that the syntax is ok. The second trigger contains a statement that will not test as valid. The macro definition starts by naming the macro and defining a parameter. A number of %put statements communicate to the user the name of the macro, that the macro (testMACRO) is beginning execution, what its purpose is, and information about the parameter (CODE).

```
%macro testMACRO(CODE=) ;

%put NOTE:<testMACRO> ... macro * START ;
%put ;
%put NOTE:<testMACRO> ... purpose ;
%put NOTE:<testMACRO> to test data step code contained ;
%put NOTE:<testMACRO> within the macro parameter. ;
%put ;
%put NOTE:<testMACRO> ... parameters ;
%put NOTE:<testMACRO> CODE= value is the name of a global ;
%put NOTE:<testMACRO> macro variable containing SAS code. ;

%* internal macro variables ;
%local msg ;
%let msg = ;

%* check if CODE contains SAS code with errors ;
proc printto log=_null_ ;
run ;
```

```
data _null_ ;
   &&&code. ;
run ;
%if &syserr NE 0 %then %let msg = SAS_CODE_ERROR ;
proc printto log=log ;
run ;
%if &msg EQ %then %do ;
   %put NOTE:<testMACRO> SAS code is error free ;
   %put NOTE:<testMACRO> and can be used. ;
%end ;
%else %do ;
   %put WARNING:<testMACRO> SAS code syntax error! ;
   %put WARNING:<testMACRO> CODE= set to null value. ;
   %let CODE = ;
%end ;

%put ;
%put NOTE:<testMACRO> ... macro * STOP ;
%mend testMACRO ;
```

The processing strategy of the macro is to turn off the log messaging with a proc printto, execute the code within a data step, and test the results by looking at the &syserr automatic macro variable. Next the log messaging is restored with another proc printto and custom messages are written out. The macro concludes by send a final log message to the user to indicate that the macro is finished processing. A relatively simple example that shows how to not only check data step code that might get inserted into a macro, but also shows how to communicate the requirements of the macro to the user in an effective way. It did expand off of the just having the data step, the essential processing piece of this macro, but it will allow me to easily use or comfortably distribute this to other users.

CONCLUSION

Documenting and communicating the requirements of your macro parameters and filtering their values are extremely important to building effective macro programs. Your ability to distribute your macro and have others use it properly may depend on it. You might also be saving yourself at some point in the future when you try and use your own code.

REFERENCES

SAS Institute Inc. (2002-2006), SAS 9.1.3 Language Reference: Dictionary, 5th Edition, Vols 1-4, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), SAS Macro Language Course Notes, Book Code 59793 Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2008), SAS Programming 3: Advanced Techniques and Efficiencies Course Notes, Book Code E1221 Cary, NC: SAS Institute Inc.

SAS9 - Perl Regular Expression Tip Sheet http://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf

CONTACT INFORMATION

Please forward comments and questions to:

Don Boudreaux, Ph.D.
SAS Institute Inc.
11920 Wilson Parke Ave
Austin, TX 78726
Phone: 512.258.5171 Ext. 55265
E-mail: don.boudreaux@sas.com