Paper 015-30

Automatic Parameter Checking

Greg Silva, Biogen Idec, Maidenhead, UK

ABSTRACT

Much of applications development involving the SAS system involves macros. The most useful are parameter driven. By giving the user choices, the power of the application can be greatly extended. The problem for the developer is keeping track of the parameters and their acceptable values. This paper will show that by using standards parts of the SAS system and developing standard naming conventions, the developer can be relieved of the burden of checking parameters.

INTRODUCTION

As small pieces of SAS code evolve into large and complex SAS macros, the developer is tasked with providing the required functionality, as well as protecting the users from themselves. No matter how efficient or 'cool' a macro is, if it produces the incorrect results, it's not very useful. The developer has control of what is written in the macro, but is at the mercy of the user when it comes to parameters.

The SAS system has provides you, the developer, with tools that can help ease the burden of macro parameter checking. This paper will show that by using the PARMBUFF option, &SYSPBUFF macro variable and developing standard naming conventions, you can automate most of the parameter checking of macros.

THE ISSUES

When it comes down to it, writing macros that use parameters can have a lot of issues. Some macros have a few parameters. Some macros may have a lot of parameters. Parameters can be added. Parameters can be removed. Parameters can be changed. The bottom line when developing macros for any system wide use is: parameters need to be checked.

Macro parameters should be checked before any "real" processing begins. If you decide that everyone knows that the parameter "patients" means "provide the macro with the number of patients to analyze," then you need to be sure that it is well documented. Even when the macro is well documented the user may enter some incorrect value, so the parameter needs checking.

The obvious answer is to check each parameter in the macro before starting the "real" processing. This allows the user to get instant feedback if there is a problem, as well as insuring that there will be no results generated based on incorrect parameter values.

AN ANSWER

The following is an example of a simple way to check parameters in a macro.

```
%macro blah(Protocol=,
            Default =5,
            Test    = );
   %if (%substr(&Protocol,1,1) ^= A) %then %do ;
       %put ERROR: Invalid protocol number!;
       endsas ;
   %end ;
   %if (0 < &Default or &Default > 10 ) %then %do ;
      %put !!! Default value is incorrect !!! ;
       endsas ;
   %end ;
                    .
                    .
                    .
%mend;
```

SOME PROBLEMS

You may already see some problems with this solution. A simple problem is the inconsistent error message. This is easy to see in this example, but is harder to find as multiple developers develop multiple macros and then combine them into a system. Not a fatal error, but an annoyance for the users.

A bigger problem for the user is that he or she has to remember all of the parameters when using the macro. “Is the Default parameter a whole number?” “In the other macro I used, the protocol parameter expected yes or no for a value. What do I use here?” Documentation and training are great answers to these questions, but users sometimes do not have one or the other readily available.

From the developer’s perspective, there are larger problems with future development and maintenance. For each parameter added, checks need to be in place to protect the code from the user. If you have checks on parameters throughout the code, then the removal of a parameter becomes a real problem.

Perhaps the biggest problem, and the hardest to track, is if a parameter changes from a character to a numeric value, or from a yes/no Boolean to a real number. Not only do you need to re-code to accommodate the change, the users need new training and documentation for this.

A STANDARD

One of the easiest ways to educate developers and users to the meaning of a variable is to have some standard for the variable names. With the advent of version 7 of SAS, the macro variable name lengths have been extended to 32 characters. Some think this just adds more to the typing task, but it also allows for more standard variable names.

In C, Visual Basic and other languages a standard has emerged for naming variables. It is known as Hungarian Notation. It simply adds a prefix to a variable name that identifies what” type” of variable it is. Some possibilities include:

is	= YES/NO
chr	= Character
num	= Numeric

By incorporating a standard naming convention to your macro parameters, both the users and developers will have a better understanding of what is expected by each parameter. The following is a simple example.

```
%macro combine(chrProtocolName      A-123,
               isDummyRun           =no,
               chrSubjectIDRange    =,
               numSubjects          =,
               isWithinCenter       =no,
               numCenters           =,
               numCenterSubjects    =,
               numTreatments        =,
               isStratified         =no,
               numList              =1);
```

This is not the whole solution. This only helps you to see what type of value should be passed through each parameter. The bigger issue is how do you, as a developer, check these parameters in a way that is efficient and does not become burdensome.

AN OPTION AND A MACRO VARIABLE

My philosophy about SAS programming is: “Let the computer and the language do the repetitive work.” The SAS system provides an option and an automatic macro variable (PARMBUFF and &SYSPBUFF, respectively) that can help you to check macro parameters. This combination allows for the capture of macro parameters passed to a macro.

Assume that you have this macro definition with the PARMBUFF option added to the definition:

```
%macro blah(chrParm   =,
            numDefault=100,
            numParm   = ) /parmbuff;
                    .
                    .
```

```
                    .
%mend;
```

If it is called with these parameters,

```
%blah(chrParm = testing,
      numParm = 3.1.1579)
```

then the macro variable &SYSPBUFF will resolve as follows:

```
%put syspbuff=&syspbuff ;

syspbuff=(chrParm=testing, numParm=3.1.1579)
```

You can see that SAS provides a macro string that contains any parameter passed to the macro. Note that any default values that are not changed are not included in the string. The SAS system is providing you with all the information that you need to easily incorporate a standard set of parameter checks in all macros being developed.

A SOLUTION

Putting all of the proceeding together, you can create a single parameter checking macro that requires no additional effort on the your part. By applying standard prefixes to parameters and taking advantage of the feature provided by the SAS system, a simple macro can be written to automate the checking of macro parameters.

The following is a parameter checking macro (check_parameters) that can easily be extended as more standard "types" of parameters are developed. Each "type" of parameter will require a standard error checking macro, and each of these would be called in this macro.

The macro parameters are the parameter list provided by the resolution of the &SYSPBUFF macro variable, and the name of the macro whose parameters are being checked. This is used to provide a message in the log that can be useful for the user when debugging.

```
%macro check_parameters(lstParameters, chrMacroName);
   %let errorz = 0;
   %let _done  = 0;
   %let _count = 1;
```

The macro reads in a string through the lstParameters parameter. A '$$=$$' is added to show the end of the list of parameters. This is a simple way to know when you are done parsing through a string (or macro variable). The two characters removed from the length are the parentheses that enclose the list of parameters.

```
   %let length      = %eval(%sysfunc(length(&lstParameters)) -2) ;
   %let parameters  = %sysfunc(substr(&lstParameters, 2, &length))%str(, $$=$$) ;
```

The string is parsed through to get the name-value combinations for parameter = value.

```
   %do %until(&_done) ;
      %let parm_name  = %scan(%nrquote(&parameters), &_count, %nrquote(,)) ;
      %let parm_value = %scan(&parm_name, 1, %str(=)) ;

      %if (&parm_value = %nrquote($$)) %then %let _done = 1 ;
```

This section of the code checks the parameter prefix to see if the parameter is of a certain type. Based on the prefix, the parameter is checked by the appropriate type checking macro.

```
      %else %do ;
         %if       %substr(&parm_ value, 1, 2) = is  %then
            %yesno_parameter(&parm_value) ;

          %else %if %substr(&parm_ value, 1, 3) = chr %then
             %char_parameter(&parm_value) ;

      %end ;
```

```
      %** Increment the parameter count. ;
      %let _count = %eval(&_count + 1) ;

   %end ;
```

Each parameter checking macro will return an error message about that “type” of macro parameter. At the end of all of the checking, a single standard message is used inform you that there is a problem with one or more parameters. At this point you know all of the parameters that have invalid values, as well as the macro that used them.

```
   %if (&errorz > 0) %then %do ;
      %put ----- There were errors found in parameters defined for
                 &chrMacroName.. The application will stop. ;
   %end ;
%mend;
```

So in the end, you have a self-documented macro call that is not much more verbose then old cryptic eight character or less parameter names.

```
%macro combine(chrProtocolName          =Test,
               isDummyRun               =no,
               chrSubjectIDRange        =,
               numTreatments            =,
               allocation_of_treatments =,
               isStratified             =no,
               numList                  =1) / parmbuff;
```

The only addition needed to check all parameters with standard prefixes is this simple macro.

```
%*** All parameter checking here... ;
   %check_parameters(&syspbuff, Combine);
                     .
                     .
                     .
%mend ;
```

The only effort on the developer’s part is the second parameter that identified the macro whose parameters need to be checked. Adding this call to all new macros will allow for consistent parameter checks to be run on any new macro, regardless of the user or developer.

CONCLUSION

By developing and using standard naming conventions on macro parameters, you have a better understanding of the system through the types of data expected in each macro call. Incorporating standard components of the SAS system, including PARMBUFF and &SYSPBUFF, will allow you to easily add standard checks on parameters to all new macros that you are writing.

From a developer’s perspective, this means less code to write for parameter checking and less code to validate. The biggest advantage for everyone is a better understanding of what goes into a macro, so that there is less chance of erroneous values being generated by the macro.

CONTACT INFORMATION

Comments and questions are appreciated. I anticipate my phone number will be changing within the year, so please contact me via email.

Greg Silva
Biogen Idec
Thames House
Foundation Park
Maidenhead, SL6 3UD, UK
Greg.silva@biogenidec.com