

Coding for Variability: Writing Macros that Accept a Variable Number of Parameters

Michael P.D. Bramley, Fairfield, OH

ABSTRACT

This paper describes the components of a SAS MACRO catalog that allows SAS programmers to implement SAS MACROS with a varying number of parameters using the PARMBUFF option. Two methods of extracting parameters from such a macro invocation are examined along with their benefits and drawbacks, along with how they are implemented in the catalog.

In addition to using a variable number of parameters in the SAS System, this paper illustrates MACRO programming techniques, such as, the use of the %SYSFUNC, %SYSEVALF and other macro functions. Finally, a solution to a specific problem is provided as an example to provide readers with a springboard for creating their own macros of this type. This paper does not assume any advanced SAS MACRO knowledge on the part of the reader.

INTRODUCTION

There are many computing problems where it is impossible to know in advance how many parameters will be passed to a function during execution or impractical to impose an arbitrary limit or structure on those parameters. For these types of problems, programmers have often relied on parameter arrays or the language implementation to handle a variable number of parameters in the function declaration.

MACRO arrays are the most common approach used in SAS. This approach usually provides somewhat immediate relief of the problem (usually after extensive debugging). However, this approach does not lend itself well to problems requiring more than 20 lines of code. Moreover, the long-term effects of this approach have been known to include increased hair loss, an exponential increase in caffeine intake, and in rare cases, a sudden change in career path by SAS programmers confronted with more ampersands on one page than they normally see in their entire life.

ANOTHER SOLUTION

In addition to parameter arrays, some languages (such as C and S) implement the ellipsis (triple dot) syntax in a function declaration to indicate that an unknown number and type of parameters should be allowed during execution [1], [2]. The SAS MACRO language also allows programmers to define macros that accept a variable number of parameters. This feature is activated by including the PARMBUFF (or PBUFF) option in the macro declaration. When SAS compiles a macro that has been declared as:

```
%Macro Name < (parameter-list) > / PBuff ;
...
macro code
...
%Mend <Name> ;
```

SAS automatically includes code to generate a local macro variable named SysPBuff whose value equals the parameter list at invocation enclosed by round brackets [3]. For example, suppose the following SAS code is executed:

```
01 %Macro Test1( Parm1, Parm2= ) / PBuff ;
02   %Put Parm1=<&Parm1> ;
03   %Put Parm2=<&Parm1> ;
04   %Put SysPBuff=<&SysPBuff> ;
```

```
05 %Mend Test1 ;
```

```
06 %Test1(Work, Option=Debug, Parm2=List) ;
```

The log will contain the following output from the macro:

```
Parm1=<Work>
Parm2=<List>
SysPBuff=<(Work, Option=Debug, Parm2=List)>
```

This example demonstrates behaviour by SAS that is worthy of note. It appears that the SAS MACRO declaration rules remain the same, but the macro invocation rules do not. That is, positional parameters must precede keyword parameters in the declaration. In the macro's invocation, this rule seems to be relaxed, allowing more parameters to be specified than originally declared. The example illustrates these points: the macro invocation at line 6 includes a keyword parameter that was not in the original macro declaration.

Moreover, if the above macro had originally been declared without a parameter list:

```
%Macro Test1 / PBuff ;
```

then its invocation may gleefully include any number, order, or mix of positional and/or keyword parameters. Thus, the invocation rules appear to have been relaxed even further.

According to SAS technical support, no rules have been relaxed: it is simply that the PBUFF option views these extraneous "parameters" as text. When a macro is invoked, SAS creates and populates the parameters based on the macro's declaration. However, since the above alternate declaration contains no parameters, SAS skips this action, and everything between the brackets is treated as text to be placed in the automatic SysPBuff macro variable.

This is an extremely important "feature" that SAS programmers must be critically aware of when writing and using macros with a variable number of parameters. If the macro declaration does not include the PBuff option, such an invocation would cause SAS to print the error message "keyword parameter OPTION not defined" and then fail to invoke the macro. Moreover, within the macro, the macro variable Option does not exist (assuming it is not defined globally). Finally, if the macro contained a reference to it, SAS would print the warning message "Apparent symbolic reference OPTION not resolved" in the log (assuming the SERROR system option is set).

What this means is that programmers who write macros that accept a variable number of parameters must take extra pains to ensure that their work is sufficiently debugged, their documentation is complete (including published within their environment), and that their macro is smarter than the average macro. That is, their macro must (as a minimum) be able to perform the following:

- interrogate the parameter list,
- determine the number of parameters specified at invocation,
- determine the value of positional parameters at invocation,
- determine the keyword parameters specified at invocation, and
- create macro variables and set values based on the parameters specified at invocation.

The following sections of this paper demonstrate how to achieve the above requirements using the author's macro library.

INTERROGATING THE PARAMETER LIST

In general, it can be assumed that the SysPBuff macro variable will be of the form: < (< parameter1 <, parameter2,...>>) >. How a macro interrogates the parameter list then depends on what it is looking for: positional parameters, keyword parameters, or delimiters. In the case of positional parameters, the macro can easily employ the %QSCAN macro function to search the SysPBuff macro variable for commas (as delimiters to each parameter) and return the appropriate token. Such a methodology assumes that commas will not be contained in a parameter's value. Locating a keyword parameter is dealt with later.

The syntax for the %QSCAN macro function is %QSCAN(string, n <, delimiters>). (The reader should know that all macro functions beginning with a Q return quoted results and have corresponding macro functions whose names do not start with Q). This function scans the string for the delimiters and returns the nth token with all of the special characters and mnemonic operators masked, such as %, &, LE and NOT. That is, it returns the string between the n-1th and nth delimiters, where the start and end of the string are the 0th and n+1th delimiters, respectively. It will be necessary to specify the delimiter, as the ASCII default includes space, period, comma, and < (+ & !\$*); ^ - / % |.

In this context, the %QSCAN macro function returns the ith positional parameter that the macro is interested in. However, the macro must first strip off the brackets that SAS automatically adds to the macro variable SysPBuff. If this is not done, the unbalanced brackets may very well confuse SAS and cause the well-crafted program to crash and leave very nasty errors in the log, such as Stack Overflow. The routines in this article employ the user-written macro function named %Strip to perform this task.

A documented feature to be aware of is that the %QSCAN macro function treats two or more consecutive delimiters (i.e. commas) as a single delimiter. This feature can cause wrong results to be returned and can be avoided by converting instances of consecutive commas to comma-space-comma prior to invoking %QSCAN.

HOW MANY PARAMETERS WERE PASSED

One method that the macro can employ to determine the number of parameters at invocation is to remove everything from the parameter list except commas. The number of parameters then equals the length of the resulting string (the number of commas) plus unity. Unity must be added to count the first parameter. However, this method requires that the macro be prepared for any and all characters that may appear in a parameter list.

Taking the complement of the above method yields an even simpler solution: Compute the difference in the length of the parameter list and the parameter list without commas, plus unity. In this method, the macro does not have to account for any and all characters that may appear in a parameter list, except for commas. Moreover, this method results in the following quick solution:

```
01 %Macro NumParms(ParmList) ;
02   %If %Strip(&ParmList) NE %Then
03     %Eval(%Length(&ParmList) -
           %Length(%SysFunc(
             Compress(&ParmList,%Str(,)))
           + 1 ) ;
04   %Else
05     0 ;
06 %Mend NumParms ;

07 %Macro Ex1 / PBuff ;
08   %Put Number of Parameters=
```

```
<%NumParms(&SysPBuff)> ;
09 %Mend Ex1 ;

10 %Ex1(P1, P2, P3, P4, P5, P6, P7, P8) ;
```

Lines 1 through 6 declare the NumParms macro function, which first strips off the enclosing brackets, then returns the number of parameters that the calling macro was passed. The test at line 2 handles the condition of no parameters being passed. Lines 6 to 9 declare the Ex1 macro, which tests the NumParms macro function by including the PBuff option in its declaration and invoking the NumParms macro function with the automatic SAS macro variable SysPBuff as the parameter. Execution of this code results in the text "Number of Parameters=<8>" in the log.

POSITIONAL PARAMETER VALUES

The following macro function, GetPosParm, uses the %QSCAN macro function described earlier to perform the task of returning the specified positional parameter's value. The accompanying example and partial log demonstrate this approach:

```
01 %Macro GetPosParm(ParmNum, ParmList) ;
02   %Let List=%Strip(&ParmList) ;
03   %If %QSysEvalF(1 LE &ParmNum And
                   &ParmNum LE
                   %NumParms(&ParmList)) %Then
04     %QScan(%QSysFunc(
               Tranwrd(&List,%Str(,,),
               %Str(, ,)),&ParmNum, %Str(,))
05 %Mend GetPosParm ;

06 %Macro Ex2 / PBuff ;
07   %Put Parameter 2=<%GetPosParm(2,
                                   &SysPBuff)> ;
08 %Mend Ex2 ;

09 %Ex2(Parm1, Parm2, Parm3, Parm4, Parm5,
        Parm6, Parm7, Parm8) ;
```

Lines 1 through 5 declare the GetPosParm macro function, which first strips off the brackets from the parameter list created by SAS (line 2), then checks if the parameter number specified by ParmNum is valid (line 3). The %QSysEvalF macro function is used to perform this comparison as it uses floating point (not character or integral) values and correctly handles missing values. This macro function has two arguments: an arithmetic or logical expression and an optional conversion type. The result is always formatted using the BEST32. format. In this case, the expression is a logical one, the result of which is always Boolean, so no conversion is required. This method ensures that a test such as, 3. LE 11, returns the correct result. If SAS encounters a decimal place in an expression, then it evaluates the expression using the default character system and not numerically. Moreover, it is important to write logical expressions of the form X < Y < Z as X < Y AND Y < Z, as SAS is apt to return incorrect results using the former expression.

If the test on line 3 is true, the macro returns the quoted value of the parameter specified by the ParmNum parameter (line 4). If the test is false, then a blank value is returned. Note that the first argument to %QSCAN (on line 4) is the result of a call to %QSysFunc, that calls the Tranwrd SAS DATASTEP function to convert two consecutive commas to comma-space-comma.

The %QSysFunc macro function allows programmers to call most SAS DATASTEP functions without having to code a DATASTEP. This eliminates run boundaries from macro code and opens up an arsenal of functions to the SAS MACRO language. The important point to remember in using this macro function is that there must be a %QSysFunc for each DATASTEP function invoked. Thus, calling function fun1 which in turn calls function fun2 must be coded as:

```
%QSysFunc(fun1(%QSysFunc(fun2(macro args))))).
```

Lines 6 to 8 declare the Ex2 macro (with the PBuff macro option) that invokes the GetPosParm macro function and prints the value of the second parameter passed to Ex2. The Ex2 macro is invoked on line 9 with eight parameters, yielding "Parameter 2=<Parm2>" in the log.

The GetPosParm macro function is quite simplistic: it only tests that it has been passed a valid parameter number. It assumes that the parameter list will be formatted as per the SysPBuff macro variable. Moreover, the call to the Tranwrd function at line 4 is limited in that it will not work as expected when the parameter list has more than two consecutive commas [4].

The GetPosParm and NumParms macro functions allow macros to quickly and easily determine the number of parameters passed at invocation and their values. The following example demonstrates:

```
01 %Macro Ex3 / PBuff ;
02   %Do I=1 %To %NumParms(&SysPBuff) ;
03     %Put Parameter &I =
04       <%GetPosParms(&I,&SysPBuff)> ;
05   %End ;
06 %Mend Ex3 ;

06 %Ex3(Parm1, Parm2=Jack Rabbit, Parm3) ;
```

The partial log appears as follows:

```
Parameter 1=<Parm1>
Parameter 2=<Parm2=Jack Rabbit>
Parameter 3=<Parm3>
```

The above output from the example reveals a fundamental flaw in the design thus far. The GetPosParm macro function fails to distinguish between keyword and positional parameters.

USING KEYWORD PARAMETERS

To distinguish a keyword parameter, it suffices to apply the SAS definition of a keyword parameter to the value returned by the GetPosParm macro function shown previously. That is, parse the text between the commas for the structure keyword=<value>, where keyword is a valid SAS MACRO variable name and <value> is blank or alphanumeric text.

The above definition initially leads to six macro functions: to test if a given parameter is a valid keyword, to return a list of the keywords in a parameter list, to test if a keyword is present in a parameter list, to return a valid keyword parameter name (or blank), and two to return the value associated with a given keyword. The first macro function, IsKeywordValid, simply tests if the given parameter meets the above definition and returns a Boolean value. The second macro function, GetKeywordList, returns a space delimited list of unique keywords found in the parameter list. The IsKeywordPresent macro function is the third entry which scans for the first parameter (a keyword) in the second parameter (a parameter list), returning one if the keyword was found, zero if not, or minus one if the keyword fails to meet the above definition.

The fourth macro function, GetKeyName, checks if the *i*th parameter in a parameter list is of the form keyword=<value>, and if so, returns the keyword found. If the keyword construct is not found, then a blank is returned. The code for GetKeyName follows:

```
01 %Macro GetKeyName(KeyNum, ParmList) ;
02   %Let Parm=%GetPosParm(&KeyNum,
```

```
    &ParmList) ;
03   %If "&Parm" NE "" And
    %Index(&Parm,=) %Then %Do ;
04     %Let Key = %UpCase(%Cmpres(
        %QSubStr(&Parm, 1,
        %Eval(%Index(&Parm,=)-1))) ;
05     %If %IsKeywordValid(&Key) %Then
06       &Key ;
07   %End ;
08 %Mend GetKeyName ;
```

The GetKeyName macro function requires that the invoking macro pass it the parameter number that it is interested in and the complete parameter list that was passed to it. GetKeyName first obtains the value of the parameter using the GetPosParm macro function and saves it in the macro Parm (line 2). It then tests if the contents of the Parm macro are not blank and if it contains an equal sign (line 3). Assuming this is true, the macro extracts the keyword token from the left-hand side of the equals sign in the Parm macro and assigns it to the macro Key (line 4). The macro function then tests if the value of the Key macro satisfies the definition of a SAS variable (line 5). If it does, then the value of Key is returned (line 6), otherwise a blank is returned.

The %UpCase and %QSubStr macro functions perform as per their DATASTEP counterparts. That is, they convert their argument to uppercase and return a sub-string, respectively. %Cmpres is a SAS supplied macro function (line 4) that trims leading and trailing blanks and compresses multiple blanks. The %Eval macro function (line 4) is similar to %SysEvalF, except that it is limited to integer arithmetic and does not have a conversion parameter.

The fifth macro function, GetKeyValue, complements the previous macro function by returning the value for the keyword parameter. If the parameter does not contain the keyword=<value> construct, then the macro function returns the value of the specified parameter from the parameter list. The code follows:

```
01 %Macro GetKeyValue(KeyNum, ParmList) ;
02   %Let Parm=%GetPosParm(&KeyNum,
    &ParmList) ;
03   %If %GetKeyName(&KeyNum, &ParmList) NE
    %Then
04     %QSubStr(&Parm,
        %Eval(%Index(&Parm,=)+1)) ;
05   %Else
06     &Parm ;
07 %Mend GetKeyValue ;
```

This macro function takes the same parameters as the GetKeyName macro function. GetKeyValue first saves the value of the parameter in the macro Parm (line 2). It then uses the GetKeyName macro function to verify that the specified parameter contains a keyword parameter (line 3), and if so, returns the right-hand side of the construct (line 4). If this is false, then all of the parameter value is returned (line 6). Note that this macro function always returns a value if the parameter value is not null. The reason for this is that it was implemented with the belief that it would be invoked conditionally upon knowing the existence of the keyword construct.

The above two macro functions are designed to work together in providing a SAS MACRO programmer with information regarding the parameter list at invocation. Use the GetKeyName macro function to return the left-hand side of the keyword=<value> construct, and the GetKeyValue macro function to return the right-hand side.

The following SAS code demonstrates how these macro functions might be used by SAS macros to extract keyword parameters and

their associated values from the parameter list:

```

01 %Macro Ex4 / PBuff ;
02   %Put SysPBuff=<&SysPBuff> ;
03   %Do I=1 %To %NumParms(&SysPBuff) ;
04     %Let KeyName = %GetKeyName(&I,
                                &SysPBuff) ;
05     %If "&KeyName" NE "" %Then
06       %Put KeyName=<&KeyName>
           KeyValue=<%GetKeyValue(&I,
                                &SysPBuff)> ;
07   %End ;
08 %Mend Ex4 ;

09 %Ex4(1V=Test, Parm2=Jack Rabbit,
       Option=Jack In The BeanStalk) ;

```

Execution of this code produces the following output in the log:

```

SysPBuff=<(1V=Test, Parm2=Jack Rabbit, Option=Jack In The
BeanStalk)>
KeyName=<PARM2> KeyValue=<Jack Rabbit>
KeyName=<OPTION> KeyValue=<Jack In The BeanStalk>

```

Line 3 of the Ex4 macro initiates a do-loop based on the number of parameters in the macro invocation of Ex4. For each iteration of the loop, the macro checks to see if the current parameter is a keyword parameter (lines 4-5). If so, the macro prints both the keyword and its value to the log.

Note that the Ex4 macro prints only two KeyName= lines to the log: one for the keyword PARM2 and another for the keyword OPTION. The first parameter, 1V=Test, is ignored as it fails to meet the SAS variable name definition (cannot begin with a number). It is important to recognize that the GetKeyName macro did not print an error or warning message when it encountered the ill-defined keyword, but rather returned a null value quietly (with no error message).

KEYWORD PARAMETERS BY NAME

One large drawback in the above method is that it retrieves keyword parameters and their values based on their position in the parameter list at invocation. If the macro had been declared with no parameters, as in the alternate definition of Test1 on page 1, then it would have to check each parameter for the presence of the given keyword, much like reading a file sequentially.

A more efficient method would be to query the parameter list for the presence of a specific keyword. If found, then invoke another macro function to return the value for that keyword. This method can be thought of as a random-access method and possesses advantages over the previous method.

The first part of this method has already been implemented as the IsKeywordPresent macro function described earlier. The final part of the method is a fairly detailed macro function that appears more complex than it really is.

```

01 %Macro GetKeywordValue(KeyWord, ParmList) ;
02   %If %IsKeywordPresent(&KeyWord, &ParmList)
       EQ 1 %Then %Do ;
03     %Let Parm = %Strip(&ParmList) ;
04     %Let Found = 0 ;
05     %Let Idx = 1 ;
06     %Do %While(Not &Found And &Idx GT 0) ;
07       %Let Idx =
           %Index(%Quote(%UpCase(&Parm)),
                 %UpCase(&KeyWord)) ;
08       %If &Idx %Then %Do ;
09         %Let Parm = %SubStr(&Parm,
                               %Eval(&Idx+%Length(&KeyWord))) ;

```

```

10       %Let Parm = %Quote(&Parm) ;
11       %If %Length(&Parm) %Then
12         %If "%SubStr(&Parm,1,1)" EQ
           "=" %Then %Do ;
13         %Let Idx =
           %Index(%QSubstr(&Parm,2),
                 %Str(,)) ;
14         %If Not &Idx %Then
15           %Let Idx = %Length(&Parm) ;
16         %Let Parm = %SubStr(&Parm, 2,
                               %Eval(&Idx-1)) ;
17         %Let Found = 1 ;
18       %End ;
19     %End ;
20   %End ;

21   %If &Found %Then
22     &Parm ;
23   %End ;
24 %Mend GetKeywordValue ;

```

The GetKeywordValue macro function requires two parameters: the keyword of interest and a parameter list to extract the keyword value. This macro first checks that the keyword parameter meets the SAS variable name definition and that it is present in the parameter list via the call to IsKeywordPresent (line 2). If so, it then removes the brackets from the parameter list and initializes two macro variables (lines 3-5). It then executes a do-loop conditional upon the values of the Found and Idx macro variables (line 6).

Within the do-loop, the macro calls the %Index function to obtain the position of the specified keyword in the parameter list. If found (line 8), the macro chops off the parameter list beginning at the position after the keyword (lines 9-10). The modified parameter list should now begin with an equal sign followed by the keyword's value and optionally, more parameters from the original parameter list.

If the parameter list begins with an equal sign (line 12), the macro assumes that it has found the specified keyword and value. It then strips off any trailing parameters and sets the Found macro variable to 1 (lines 13-19), causing the do-loop to terminate. If the first byte is not an equal sign, the macro returns to the top of the do-loop to test the condition. If the keyword cannot be located in the parameter list, the do-loop terminates. Upon exiting the do-loop, the macro tests the Found macro variable to determine if the keyword was located, and if so, the keyword's value is returned (lines 21-22). If the keyword is not found, the GetKeywordValue macro function returns a blank.

The following SAS code demonstrates the use of the above two macro functions:

```

01 %Macro Ex5 / PBuff ;
02   %If %IsKeywordPresent(Option, &SysPBuff)
       %Then
03     %Put Option =
           <%GetKeywordValue(Option,
                               &SysPBuff)> ;
04   %If %IsKeywordPresent(Parm2, &SysPBuff)
       %Then
05     %Put Parm2 =
           <%GetKeywordValue(Parm2,
                               &SysPBuff)> ;
06 %Mend ;

07 %Ex5(1V=Test,
       Parm2=Jack Rabbit,
       Option=The BeanStalk) ;

```

Lines 1 to 6 declare the simple Ex5 macro that accepts a variable number of parameters. Line 2 tests if the keyword parameter Option was specified at invocation, and if so, then prints the message "Option = <The Beanstalk>" in the log (line 3). Lines 4-5 perform a similar task for the keyword parameter Parm2, resulting in "Parm2 = <Jack Rabbit>" being printed in the log.

EXAMPLES

The following example takes its lead from the simple, yet powerful, C printf standard function, which prints its parameters using the format specification found in the first parameter. The SAS MACRO version does not require a format specification and prints in the log the name of each parameter passed (assumed to be a macro variable) and its corresponding value in angle brackets. The code is presented below along with a sample invocation and partial output:

```
01 %Macro Print / PBuff ;
02 %If %Strip(&SysPBuff) NE %Then
03   %Do I = 1 %To %NumParms(&SysPBuff) ;
04     %Let Parm = %GetPosParm(&I,
                             &SysPBuff) ;
05     %Put &Parm = <&&Parm> ;
06   %End ;
07 %Mend Print ;

08 %Print(P1,P2,P3,P4,P5,P6) ;

P1 = <value of P1>
P2 = <value of P2>
P3 = <value of P3>
P4 = <value of P4>
P5 = <value of P5>
P6 = <value of P6>
```

Line 2 of the Print macro tests if any parameters were passed: no action is performed in this case. If at least one parameter is passed, the code at lines 3-6 first obtains the name of the macro variable using the GetPosParm macro function (line 4), then puts the name and its value to the log. The simplicity of the Print macro is due to the fact that the GetPosParm macro function provides most of the functionality in a simple macro invocation. It is this very concept of encapsulating difficult or tedious routines with a functional name that allows programmers to build systems and helps to ensure the success of a language. All in all, this is a fairly simple routine that is capable of generating a lot of output.

A HARDER EXAMPLE

Imagine if you will a front-end that allows a user to choose a data set and variables from it in order to produce a multi-row, multi-column report of the user's design. For example, suppose a customer data set contains the following variables:

Variable	Type	Length	Format
Customer	Num	8	Z10.
FullName	Char	50	\$50.
Address	Char	50	\$50.
CityStateZip	Char	50	\$50.
Phone1	Num	8	Phone.
Phone2	Num	8	Phone.
Phone3	Num	8	Phone.
CreditLimit	Num	8	Dollar15.2
Balance	Num	8	Dollar15.2
CurrentDue	Num	8	Dollar15.2
Over30	Num	8	Dollar15.2
Over60	Num	8	Dollar15.2
Over90	Num	8	Dollar15.2
FinCharge	Num	8	Dollar15.2
LastPymt	Num	8	Dollar15.2
PymtDate	Num	8	Date9.

and the user requires a report listing all of the above information in a 6 column report where each customer's record requires 3 rows. The first row will contain the customer number, full name, phone 1, credit limit, balance over 30 days, and last payment amount. The second row will contain the balance due, address, phone 2, current balance due, balance over 60 days, and last payment date. The third row will contain (beginning in the second column) city/state/zip information, phone 3, finance charges due, and the balance over 90 days. The user wants the report columns to appear as below:

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
Customer	Name	Phone1	Credit	Over30	LastPymt
Balance	Address	Phone2	Current	Over60	PymtDate
	CityStateZip	Phone3	FinCharge	Over90	

Leaving aside important issues, such as report titles, headings and error checking, declaring the macros that create reports based on users' requests becomes simple. The first parameter describes the data set, while successive parameters dictate which variables are to occupy each column of the report, working across the page from left to right. The report generating macros and example follow:

```
01 %Macro BuildRow(Row) ;
02   %Do I=1 %To %NumCols ;
03     %Let Var = %Scan(&&Col&I,&Row) ;
04     %Let Fmt = %Scan(&&Fmt&I,&Row,
                      %Str( )) ;

05     %If "&Var" NE "" %Then
06       Col&I = Put(&Var, &Fmt) ;
07     %Then
08       Col&I = " " ;
09   ;
10 %End ;
11 %Mend BuildRow ;

12 %Macro Report(DataSet) / PBuff ;
13   /* Initialization */
14   %Let FID = %SysFunc(Open(&DataSet)) ;
15   %Let NumCols = %Eval(-1 +
                        %NumParms(&SysPBuff)) ;
16   %Let NumRows = 0 ;

17   %Do I = 1 %To %NumCols ;
18     /* Get Column variables into Col_i */
19     %Local Col&I Fmt&I ;
20     %Let Col&I = %GetKeywordValue(Col&I,
                                    &SysPBuff) ;
21     /* Load formats for each variable */
22     %Let Var = %Scan(&&Col&I,1) ;
23     %Let J = 1 ;
24     %Do %While(&Var NE ) ;
25       %Let Fmt&I = &&Fmt&I
26       %SysFunc(VarFmt(&FID,
                       %SysFunc(VarNum(&FID,&Var)))) ;
27       %Let J = %Eval(&J + 1) ;
28       %Let Var = %Scan(&&Col&I,&J) ;
29     %End ; /* while */

30   %Let NumRows = %SysFunc(Max(&J-1,
                                &NumRows)) ;

31 %End ; /* while */

32 %Let FID = %SysFunc(Close(&FID)) ;

33 Data _Report_ ;
34   Set &DataSet ;
```

```

/* Length & Keep stmts omitted */

31      %Do Row = 1 %To &NumRows ;
32          %BuildRow(&Row) ;
33          Output ;
34      %End ;
35  Run ;
36 %Mend ;

37 %Report(Customer,
    Col1=Customer Balance,
    Col2=FullName Address CityStateZip,
    Col3=Phone1 Phone2 Phone3,
    Col4=CreditLimit CurrentDue FinCharge,
    Col5=Over30 Over60 Over90,
    Col6=LastPymt PymtDate
) ;

```

The BuildRow macro declared on lines 1-11 creates the necessary assignment statements to populate the columns of the specified row of the report. This is accomplished by executing a do-loop to extract the variable name and format information for each column and using the DATASTEP put function to create a formatted character variable for use in the report.

The Report macro declared on line 12-36 accepts a varying number of parameters to build a data set that meets the user's specifications. This macro accomplishes this goal in two major steps: extracting the variable information and executing a DATASTEP to create a data set for reporting.

Macro initialization occurs at lines 13-15, where it opens the data set to access variable information and sets the number of columns and rows. Lines 16-27 execute two do-loops that create macro variables named COL1-COLn and FMT1-FMTn. The COLi macro variables contain a space-delimited list of data set variables for the ith column. For each column, the inner do-loop at lines 21-25 uses the VarNum and VarFmt File I/O functions to extract the formatting information from the data set, building each FMTi macro variable [5]. After this block of code has executed, the macro variables COL1-COL6 and FMT1-FMT6 will be populated with space-delimited variable names and formats. This yields the following from the example (line 37):

```

COL1=<Customer Balance>
COL2=<FullName Address CityStateZip>
COL3=<Phone1 Phone2 Phone3>
COL4=<CreditLimit CurrentDue FinCharge>
COL5=<Over30 Over60 Over90>
COL6=<LastPymt PymtDate>

FMT1=<Z10. Dollar15.2>
FMT2=<$50. $50. $50.>
FMT3=<Phone. Phone. Phone.>
FMT4=<Dollar15.2 Dollar15.2 Dollar15.2>
FMT5=<Dollar15.2 Dollar15.2 Dollar15.2>
FMT6=<Dollar15.2 Date9.>

```

Now that all of the requisite variable information has been collected, lines 29-35 build a data set in the shape of the desired report. The heart of this step is the do-loop at lines 31-34, invoking the BuildRow macro for each logical row of the report. When this block of code has completed, it will have generated the DATASTEP statements to populate each column of each logical row of the report. The following shows the generated statements for the first logical row of the report:

```

Col1 = Put(Customer, Z10.) ;
Col2 = Put(FullName, $50.) ;
Col3 = Put(Phone1, Phone.) ;
Col4 = Put(CreditLimit, Dollar15.2) ;

```

```

Col5 = Put(Over30, Dollar15.2) ;
Col6 = Put>LastPymt, Dollar15.2) ;
Output ;
...

```

After this last step has completed, all that remains is to feed the data set to Proc REPORT or PRINT to physically produce the report in any of the formats supported by the Output Delivery System. Note that this macro is designed for multi-row, multi-column reports. If the report had a single row for each observation, then a simple Proc REPORT or PRINT would suffice.

CONCLUSION

The SAS MACRO facility can be a powerful tool in the hands of an experienced SAS programmer. This article demonstrates how that facility can be made even more so by opening it up to that class of problems that require a variable number of parameters to solve. This article shows that SAS programmers can use the SAS MACRO facility to write macros that accept a variable number of parameters, similar to other powerful languages, such as C/C++ and S-PLUS. Moreover, this power can be realized without using macro arrays and the increased complexity they bring about due to their requirement for multiple dereferencing of macro variables.

The contents of a user-written macro catalog have been presented which support macros that accept a variable number of parameters and expedite the act of interrogating and/or retrieving positional and keyword parameters not in the macro declaration. The routines in the user-written macro catalog may be used for positional or keyword parameters.

Short examples have been used to illustrate the uses of each of the components of this macro catalog. A SAS MACRO version of the C printf function is illustrated. A larger example is presented that ties the pieces of the macro catalog together with the aim of building a user-defined report.

Finally, this article demonstrates various SAS MACRO programming techniques and SAS MACRO functions that make it possible for the programmer to complete tasks without resorting to the DATASTEP and/or SAS procedures.

REFERENCES

- [1] *The C Programming Language*, B. Kernighan & D. Ritchie, Prentice-Hall, Toronto, 1988. p 155.
- [2] *S-PLUS Programmer's Guide*, Data Analysis Products Division, MathSoft, Seattle, 1997. p 81.
- [3] *SAS Language: Reference*, Version 6, First Edition, SAS Institute Inc., Cary, NC, 1990, p. 970.
- [4] This is by design and not due to a flaw in the Tranwrd function.
- [5] *Combining Pattern-Matching and File I/O Functions: A SAS® Macro To Generate A Unique Variable Name List*, M.P.D. Bramley, SUGI 27 Proceedings, 2002, Paper 37.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Michael P.D. Bramley
 223 Martha Lane
 Fairfield, OH 45014
 513.829.8485
 mbramley@att.net

Copies of the macro catalog described in this article are available from the author upon request via email.