ABSTRACT
The SAS Macro Facility is a powerful tool for any SAS programmer. In the pharmaceutical industry we often find ourselves performing a similar task multiple times and macros are developed to repeat the task with ease. Some macros are complex in nature, whether it is a statistics macro with multiple options or a macro that has been generalized for re-use. They can often have a long daunting list of macro parameters. In some organizations macro libraries are developed to provide a common toolkit for performing statistical analysis and reporting in support of clinical trials. Often in the process, once a macro reaches the point of having many parameters, the usability of the macro suffers. The use of variable-length list parameters complicates the parameterization considerably. Also, the parameterization is usually not consistent across macros developed over the years. The goal of this paper is to explore the use of the PARMBUFF macro option to create a flexible, condensed, user-friendly and SAS-like syntax for calling a macro. With such an approach, an organization can define a common syntax across all macros and create a more user-friendly and consistent library of macros.

INTRODUCTION
We are all familiar with the power of the SAS Macros Facility and any mature SAS programming organization in the industry will likely have several reusable macros. Often, however, these macros that have been generalized for broad re-use become difficult to use. The macro parameters can be lengthy and rarely will these be consistent across macros used in the organization. Also, to generalize a macro over time the number of macro parameters often grows to accommodate varying conditions. Wouldn’t it be nice to have a common, user-friendly syntax across all of our macros without having to re-code all of them? Once we are not constrained by the built-in parameterization syntax of the Macro Facility, the possibilities open up for some interesting options.

CONCEPT
That’s where a little ingenuity mixed with PARMBUFF comes in to play. The general concept is to develop a new macro called MPROC that will provide a user-friendly interface to our existing macros. This is achieved by capturing everything between the open and close parentheses passed to MPROC, examining and parsing the content, and mapping it into a call to the existing macro. With a well-defined syntax for this content defined up-front, we can understand the “language” from the user and map it to the “language” of our macros.

Here we consider mimicking the common SAS syntax for many SAS Procedures as our user-friendly interface to our macro environment. The syntax consistent among SAS procedures is familiar to most programmers and would eliminate having to learn a new syntax. A typical SAS procedure will have the following generic syntax structure:

```
PROC PROCNAME OPTIONS;
VAR VARLIST;
BY BYVARLIST;
  (additional statements)
RUN;
```

Our objective is to replicate this form of syntax for a macro. There are a few constraints we have to work within. A macro must be called preceded with the % sign, and all parameters must be enclosed within parentheses, such as

```
%MYM MACRO(myparameters)
```

We can, however, break away from the constraints of the two types of macro parameters, namely the keyword parameter and positional parameter, by utilizing the macro option PARMBUFF. With this option, a system macro variable named SYSPBUFF is automatically assigned the value of the entire string of parameters, including the parentheses. The option is activated in the macro definition code on the %MACRO statement:

```
%Macro MYMACRO(myparameters)/PARMBUFF;
```
In theory, we can scan and parse the SYSPBUFF content to interpret the syntax and translate to macro variables that will be referenced in our macro code, just as a macro parameter would have been. We could then allow for a SAS-like syntax for our macros such as:

```sas
%Mymacro (options;
  var varlist;
  by byvarlist;
  (additional statements)
)
```

We will create a new macro called MPROC that will interface to all existing macros. The first word in the custom parameter list will be the requested macro followed by options and further statements. This way, all existing macros can remain unmodified and we have an interface to them that will translate SAS-like syntax to each macro’s parameters.

```sas
%MPROC (mymacro options;
  var varlist;
  by byvarlist;
  (additional statements)
)
```

The advantages are obvious:
- Provides a familiar syntax for macros
- Easily supports parameters with variable-length lists
- Reduces learning curve for using macros
- Improves readability of code
- Introduces a consistent syntax among macros and for SAS-wide programming

**HOW IT WORKS**

Parsing string values is nothing new to an experienced programmer. It is a fairly common task among most programming languages and many have functions that facilitate parsing tasks. In SAS, we have several such functions available to us, but there may be others that you will need to develop yourself. You may find yourself using a macro specific to your syntax to get next statement or word or item from the SYSPBUFF variable.

**ESSENTIAL FUNCTIONS**

In order to scan and parse the SYSPBUFF there are some essential SAS functions that will be used (see *SAS Macro Language: Reference, Macro Language Elements – Macro Functions*).

- `%scan` – search for a word specified by its number.
- `%substr` – produce a substring of a character string.
- `%index` – returns the position of the first character of a string.
- `%sysfunc` – execute SAS functions.

**DELIMITERS**

The key to defining a custom syntax, or to mimic the SAS PROC syntax, is to define delimiters. You will need a delimiter to easily parse out each statement and within statements the pre-defined elements of that statement.

In SAS, the statement delimiter is a semicolon. An item delimiter in a list of values is usually a space. Options are often specified either in the PROC statement or within parentheses for other statements. Sometimes special characters are used in the syntax to identify or delimit values. These are the rules we will follow in our custom macro parameterization syntax in order to resemble the SAS syntax.

**PARAMETER MAPPING**

The second key to our approach is a mapping definition. We wish to map SAS-like syntax to our existing macro parameters. For example, our macro may have a parameter that accepts a variable list to operate on called VARLIST. The mapping information will map the common SAS statement VAR to our macro variable VARLIST. This allows us to maintain existing macros and simply create an interface to them.

The mapping will be defined in an INFORMAT and a lookup will be performed using the INPUTC function. To achieve this lookup, we will uniquely identify each SAS-like syntax item and map them to our macro parameters. A common INFORMAT and a macro-specific INFORMAT will be defined to create a hierarchical structure: if not found in macro-specific list, then look in the common list. Here is an example of a key map INFORMAT.
%macro keymap;
  proc format;
  invalue $km_common
    "opt.data"="ds="
    "opt.out"="out="
    "opt.overall"="overall=1"
    "stmt.var"="vars="
    "stmt.var.strow"="startrows="
    "stmt.var.vals"="vals="
    "stmt.by"="byvar="
    "stmt.by.vals"="byvals="
  ;
  invalue $km_mystatmac
    "opt.stacked"="statsonerow=0"
    "opt.noparens"="parens=0"
    "opt.stats"="stats="
    other=[$km_common20.]
  ;
  run;
%mend;

In this example, in addition to the common keys, there is a list of mappings for a macro called MYSTATMAC. The first
level of the syntax keyword is the type of keyword, either statement or macro-line option (similar to PROC statement
options). The second level is the keyword itself and any levels below that are options within that keyword. Also,
options can either be a single word or a word with values specified (similar to SAS options NOPRINT or
LINESIZE=132, respectively). As you can see with the option keyword STACKED, we can map that to a macro
variable and value used in our macro MYSTATMAC. The existing MYSTATMAC macro is already defined with a
default value of 1 for the STATSONEROW parameter.

BEFORE AND AFTER EXAMPLE
For this MYSTATMAC example, the impact of our SAS-like custom parameter syntax can be seen by examining the
existing MYSTATMAC parameter list and what it would look like using the new syntax.

Before:
%mystatmac(ds=,out=,pt=subjid,vars=,vals=,startrows=,byvar=,byvals=,bycols=,
           overall=0,overallcol=,where=,n_in_hdr=,n_in_rows=,exact_ci=,n_in_cols=,
           byrowvar=,byrowvals=,byrowcol=,roverall=0,byrowfmt=,
           cumpcts=,n_kwrd=1,stats=N PCT,statsonerow=1,parens=1);

After:
%mproc (mystatmac {options};
  var var1(strow= vals= );
  by var(vals= cols= );
  byrow var(vals= col= fmt= );
)

STEPS IN THE PROCESS
To interface from the custom macro parameter syntax to our existing macro parameters, the main MPROC macro
goes through the following steps:

1. Load key map informats
2. Read SYSPBUFF, strip the open and close parentheses, and count how many statements to process
3. For each statement, parse the statement into two temporary macro arrays, one with the keyword identifier
   and the other for the value specified by the user
4. Translate the incoming syntax to a macro call with parameters populated

The main MPROC macro looks like this:

%macro mproc/pbuff;
  %* LOAD SUPPORTING MACROS *;
  %mproc_meta
  %mproc_parse
The %parse_read supporting macro will read the input and store the content in temporary macro arrays. One array stores the custom parameter identifier (for example, opt.data) and the other array stores the parameter values.

The %parse_write macro will translate these and produce the macro call to our existing macro (MYSTATMAC in this example).

Because of limited space here, we will only look at one section of %parse_read that will parse the first statement containing the name of the macro and any macro-line options. Here is the portion of the code:

```plaintext
%macro parse_read;
  %*Step 1: recognize proc;
  %let _stmt=%get_stmt(1);
  %let _proc=%lowcase(%get_word(1));
  %put NOTE: MPROC &_proc recognized.;

  %*Options and Statements are put in to temporary Macro arrays *
  *to be translated later by parse_write *;
  %*Step 2: read options;
  %let _nopt=%eval(%length(&_stmt)-
    %length(%sysfunc(compress(&_stmt,%str( )))) );
  %do _i=1 %to &_nopt;
    %let _optfull=%get_word(&_i+1);
    %global _opt&_i _optval&_i;
    %let _opt&_i=opt.%scan(&_optfull,1,%str(=));
    %if %index(&_optfull,%str(=)) %then
      %let _optval&_i=%substr(&_optfull,%index(&_optfull,%str(=))+1);
    %else %let _optval&_i=;
  %end;

  %*Step 3: parse remaining statements, if any;
  %do _i=2 %to &_nstmt;
    .
  %end;
  %mend;
```

We return to our MYSTATMAC example. An actual call to MYSTATMAC without using the new MPROC interface looks like this:

```plaintext
%mystatmac(ds=db.d_safsum,out=stat2, vars=aged gendern racen, vals=1|2!. 1|2 1|2!,
  startrows=9 14 19, byvar=trtcd, byvals=1 2, bycols=2 3, overall=1, n_in_hdr=-2)
```

With the MPROC macro, the same execution would be called with this syntax:

```plaintext
%mproc (mystatmac data=db.d_safsum out=stat2 overall nrow=-2;
  Var aged(strow=9 vals=1|2!.) gendern(strow=14 vals=1|2) racen(strow=19 vals=1|2!);
  By trtcd(cols=2 3 vals=1|2);
)```
While the length of the parameters may not be shortened a great deal, the grouping and common syntax provide for a better interface that is more user-friendly.

**EXAMPLE OF PARSING THE MACRO-LINE OPTIONS**

For this example, the parse_read macro above will produce the following two arrays of values:

<table>
<thead>
<tr>
<th>#</th>
<th>_opt#</th>
<th>_optval#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>opt.data</td>
<td>db.d_safsum</td>
</tr>
<tr>
<td>2</td>
<td>opt.out</td>
<td>stat2</td>
</tr>
<tr>
<td>3</td>
<td>opt.overall</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>opt.nrow</td>
<td>-2</td>
</tr>
</tbody>
</table>

Later, the parse_write macro will perform the informat lookup to translate these to our existing MYSTATMAC parameters. The resulting parameter list is stored in a macro array which will be used in the last step to write out the final call to the existing macro MYSTATMAC. The macro code to do this translation is:

```
%macro parse_write;
  %keymap  .
  %let _nparout=0;
  %do _i=1 %to &_nopt;
    %let _tmp=;
    %put &&_opt&_i;
    %let _tmp=%sysfunc(inputc(&&_opt&_i, $km_&_proc..));
    %if %length(&&_optval&_i) %then %let _tmp=&_tmp.&&_optval&_i;
    %let _nparout=%eval(&_nparout + 1);
    %let _parout&_nparout = &_tmp;
  %end;
%end;
```

and the result is as follows:

<table>
<thead>
<tr>
<th>#</th>
<th>_opt#</th>
<th>_optval#</th>
<th>Resulting parameter list, _parout#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>opt.data</td>
<td>db.d_safsum</td>
<td>ds=db.d_safsum</td>
</tr>
<tr>
<td>2</td>
<td>opt.out</td>
<td>stat2</td>
<td>out=stat2</td>
</tr>
<tr>
<td>3</td>
<td>opt.overall</td>
<td></td>
<td>overall=1</td>
</tr>
<tr>
<td>4</td>
<td>opt.nrow</td>
<td>-2</td>
<td>n_in_hdr=-2</td>
</tr>
</tbody>
</table>

The macro parsing and writing functions become somewhat complex and should also include error-checking and capturing, but this paper at least demonstrates the concept and main components to making this work.

**A MACRO CALLING A MACRO**

The final step is for the MPROC macro to assemble and execute the call to the requested macro using the translated parameters. Again, we will not demonstrate the entire process due to limited space here, but we will show how this is done for the macro-line options.

The trick here is to have the MPROC macro dynamically build a call to another macro and then execute it. This simple example shows that it can be done, just as you would expect it to be.

```%macro test(a=);
  %put &a;
%mend;
%macro callit;
  %let mname=test;
  %let parm=a;
  %let parmval=123;
  %&mname(&parm=&parmval)
%mend;```
The result in the LOG window is:

123

You can easily build and execute the MYSTATMAC call with a simple %do loop that iterates through the macro array _parout containing the mapped parameters and values as follows:

```sas
%let _strout=&_proc.(;
%do _i=1 %to &_nparout;
  %let _strout=&strout.&&_parout&_i;
  %if &_i ^= &_nparout %then %let _strout=&_strout.,;
%end;
%let _strout=&strout.)
&_strout
```

At the end of the loop, the _strout macro variable contains the name of the macro requested, an open parenthesis, all keyword parameters with their respective values as specified in the user's call to MPROC, and a close parenthesis. The final line will execute the generated macro call. If your existing macro has positional parameters, then you will need to adjust this approach.

CONCLUSION

The use of PARMBUFF can be utilized as a means to define a custom parameterization syntax for your macros. Here we have shown an example of a macro called MPROC that provides a syntax interface to existing macros. MPROC along with the supporting informats of key mapping definitions can serve as the interface to any number of existing macros thus providing the opportunity to standardize the macro syntax for your library of macros. You can customize any syntax you choose to be expected within the parentheses for MPROC and translate that to varying parameters of existing or new macros. Here we have explored the feasibility of providing a SAS-like syntax, but this method can be used to create any common syntax of your choice.

The business goal or application of this technique is to make an organization's macro library more user-friendly with a shorter learning curve. This is accomplished by standardizing as much as possible the parameterization syntax across all macros. By choosing the SAS syntax as our model, we provide a familiar syntax for programmers that they can readily read, understand, and begin using. In many cases, the parameterization for a macro can be significantly condensed since we are using a flexible syntax model rather than being limited to only positional or keyword macro parameters. The overall net experience to the programmer improves and the organization's investments over time into their macro library have a greater return.

ACKNOWLEDGEMENT

I would like to thank Celgene Corporation for their interest and support in exploring this concept.

CONTACT INFORMATION

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

Kyle McBride
Instat Services
227 Longwood Avenue
Chatham, NJ 07928
Work Phone: 973-210-4106
Email: kyle.mcbride@instatservices.com
Web: www.instatservices.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.