Format as Macro Parameter: A Design Pattern for Writing Generic Macro Applications with Lean and Stable Interfaces

Lei Zhang and Kenton Wride, Celgene Corporation, Summit, NJ

ABSTRACT
This paper describes the Format as Macro Parameter (or FAMP) design pattern for SAS programming. The purpose of this pattern is to enhance a macro application with fewer explicit parameters and separate the complex macro parameter values from the client programs that use them. This is done by encapsulating a group of macro parameters with a SAS format as composite parameter. A SAS format like a lightweight container has three important features among others. First, it can store a list of keyword/value pairs which can be organized into a hierarchically structured dictionary via the SAS nested format mechanism. Second, any entry value in the format can be accessed by the Base function putc() with appropriate format name and keyword as arguments. Moreover, within the macro implementation, any format-encapsulated macro parameter can be retrieved by wrapping the legendary macro function %sysfunc() around the putc(). Third, all the actual formats as parameters can be shared and managed separately in a central persistent place (that is, the format library), and referenced by any client programs. By replacing sets of ordinary macro parameters with a couple of format parameters, a SAS macro developer can significantly reduce the number of explicit parameters a macro application has to declare, and at the same time, extend its functionality, and shorten the pieces of invocation code in the client programs, therefore the macro application becomes simpler to use, easier to maintain, and eventually more flexible and powerful.

INTENT
Enhance a macro application with fewer explicit parameters and separate the complex macro parameter values from the client programs that use them.

ALSO KNOWN AS
Pass-by-Format

EXAMPLE
Since the SAS System is a powerful and versatile data analysis and presentation platform, the macro applications (or simply macros) to be developed often come up with a complex programming interface with a long list of parameters. Consider a typical report macro, for example, which is used by a large number of analysis programs to create various RTF clinical tables in a company. The macro is required to create tables with following structure:

Title 1: Company name
Title 2: Study name
Title 3: Table ID
Title 4: Main Title
Title 5: Subtitle (optional)
Table body
Footnote 1-5: Footnotes (optional)

From the above, we know that the macro should be able to create a table with up to five titles and footnotes when it is invoked. One way to implement this macro is to treat all titles and footnotes as macro parameters. List 1 is a mocked-up macro based on this idea, which we call %Report.

```sas
%Macro Report(
RPTDSN= /* Report dataset */
,Vars= /* Report variables */
,Title1= /* Company title */
)
List 1: the mocked up %Report Macro

The real-life report macro in clinical data analysis will have much more parameters than the above one, but here we just use this simplified version to illustrate the problems a macro developer often encounters. List 2 is four sample analysis programs that use the macro to generate tables for patient demographics, lab test and adverse events respectively in a clinical study:

/* Client Program - DEMOG.SAS */
.
%Report(
   RPTDSN=Demog,
   VARS=PT AGE Gender,
   Title1= XYZ Corporation,
   Title2= Protocol 8888,
   Title3= Table ID: 16.1,
   Title4= Demographic,
   Footnote1= The table is created by DEMOG.SAS,
   Footnote2= Report Creation Date: &SysDate,
   OutFile=C:\temp\demog.rtf )
/* Client Program - LAB.SAS */
.
%Report(
   RPTDSN=LAB,
   VARS=PT Type Test Value Flag,
   Title1= XYZ Corporation,
   Title2= Protocol 8888,
   Title3= Table ID: 16.5,
   Title4= Lab Test,
   Title5= Hematology,
   Footnote1= The table is created by LAB.SAS,
   Footnote2= Report Creation Date: &SysDate,
   OUTFILE=C:\temp\lab.rtf
   )
/* Client Program - AE1.SAS */
.
%Report(
   RPTDSN=AE1,
   VARS= PT Socname PTName AESev AEStart AEEnd,
   Title1= XYZ Corporation,
   Title2= Protocol 8888,
   Title3= Table ID: 16.8.1,
   Title4= Adverse Event During Double-Blind Period,
   Footnote1= The table is created by AE.SAS,
   Footnote2= Report Creation Date: &SysDate,
   OUTFILE=C:\temp\AE1.rtf
   )
/* Client Program - AE2.SAS */
.
%Report(
   RPTDSN=AE2,
   VARS= PT Socname PTName AESev AEStart AEEnd,
   Title1= XYZ Corporation,
   Title2= Protocol 8888,
   Title3= Table ID: 16.8.2,
   Title4= Adverse Event During Open Label Period,
   Footnote1= The table is created by AE.SAS,
If you take a deep look at the %Report macro and its client programs, you may have following observations:

1. %Report macro explicitly declares all 14 parameters it needs, 10 of which are lengthy cosmetic parameters for table titles and footnotes.
2. The title and footnote parameters defined in the macro have a linear relationship between them. They should be arranged in proper order.
3. The expressions or values given to the parameter Title1= and Title2= in the calling code are identical among the client programs, but these two titles have to be copied and pasted into every client program.
4. Some parameter values provided in different client programs are mutually dependent. For example, changing the table ID (provided by Title3=) in AE1.SAS may consequently result in changing the table ID in AE2.SAS, since the two AE tables are to be listed in sequence in a clinical study report.
5. Some parameter values or expressions follow exactly the same pattern in every client program. For example, the footnotes provided by parameter Footnote1= are same across all the client programs except the different program name embedded in the footnotes.

Based on the above observations, you might think of using global macro variables for titles and footnotes, but you will soon find this approach is a poor programming practice. It will introduce more troubles than serving as a different solution [1]. As the number of global macro variables increase, heavily using global macro variables will cause lots of undesired adverse side effects and increased coupling between unrelated programs. For example, before you call the %Report macro, you have to initialize all the related global macro variables properly; after the invocation, you have to remember to clean up the global macro variables in case they are unexpectedly affect the programs running after that.

Faced with the problems of long parameter list and global macro variables, you may be tempted to consider using a macro [2] or an external file [3] to store and pass a large amount of parameters to a macro application. But you may find this heavyweight approach overkills, takes time and effort to implement, and often ends up with little uses in various settings.

**CONTEXT**

You are developing a macro application that has a long list of parameters with complex macro parameter values.

**PROBLEM**

A macro application with a complex long list of parameters is usually a pain in the neck for both macro developers and users. For a macro developer, managing and documenting those parameters could be a big headache. If she or he wants to extend the macros with additional parameters, many client programs have to be modified accordingly because of the ripple effect. For a macro user, using a long list of parameters often means a tough test of memory or an engagement in guesswork. Calls to such a macro are often either cryptic or cumbersome. Besides, many different client programs repeatedly call this type of macros with similar or identical sets of parameter values; therefore, if a user wants to continuously tune such parameter values, changing them across many client programs could become a true torment.

In particular you may want to address the following forces:

- You want to reduce the number of parameters that a macro application has to declare explicitly, but you don’t want to either reduce its functionality, or use global macro variables.
- You want to extend a macro application in the future with yet unknown parameters, but will keep the existing macro calls unaffected.
- You want to avoid copying identical or similar macro parameter values again and again within or across many client programs.
- You want to continuously tweak the sets of interdependent parameters passed to a macro all together and at once without affecting the programs that issue the macro calls.

**SOLUTION**

To solve those issues in a macro application, you can use a SAS format as a lightweight container to hold several related macro parameters together, and treat the SAS format as a composite parameter. A SAS format, like a mini dictionary, can be used to store actual parameter/value pairs (or keyword/value pairs) and associate one or more parameters with each other and with the format itself. With format as parameter, the %Report macro can be recast by replacing the sets of title and footnote parameters with two SAS format parameters respectively. The enhanced macro interface can thus be declared as follows.
The titles and footnotes used in the four client programs can be defined with a PROC FORMAT program, as shown in List 3.

```
Libname library 'C:\FormatLIB';
PROC FORMAT Library=library;
Value $ DEMOG_TITLE_FMT
    "TITLE1"="XYZ Corporation"
    "TITLE2"="Protocol: 8888"
    "TITLE3"="Table ID: 16.1"
    "TITLE4"="Demographics"
    OTHER=".";
Value $ DEMOG_FOOTNOTE_FMT
    "FOOTNOTE1"='The table is created by DEMOG.SAS'
    "FOOTNOTE2"='Report Creation Date: &SYSDATE'
    OTHER=".";
Value $ LAB_TITLE_FMT . . .
Value $ LAB_FOOTNOTE_FMT . . .
Value $ AE1_TITLE_FMT . . .
Value $ AE1_FOOTNOTE_FMT . . .
Value $ AE2_TITLE_FMT . . .
Value $ AE2_FOOTNOTE_FMT . . .
 Run;
```

List 3: The Proc FORMAT program that defines formats for both titles and footnotes

For example, in order to create a DEMOG table, a user can call new %REPORT as follows:

```
/* Create Demog dataset */
.
%REPORT(
    RPTDSN=DEMOG,
    VARS=PT AGE GENDER,
    TITLE1FMT=$DEMOG_TITLE_FMT,
    FOOTNOTE_FMT=$DEMOG_FOOTNOTE_FMT,
    OUTFILE=C:\TEMP\DEMOG.RTF)
```

List 4: Client program DEMOG.SAS that uses the new %Report

It is obvious that code for creating tables with the new %REPORT macro becomes shorter and simpler because of much reduced parameter list.

One of the key questions you may have with this approach is how to conveniently retrieve titles and footnotes encapsulated in a format parameter within a macro implementation. This can be done by using the combination of famous macro function %sysfunc() and Base function putc(). For example, to retrieve title 1 from the format $DEMOG_TITLE_FMT, you can use the following macro statements by specifying TITLE1 as a lookup key for the format.

```
%local TITLE1;
%let TITLE1 = %sysfunc(putc(TITLE1, $TITLE_FMT_DEMOG.));
```

With the help of SAS format, a macro developer will be able to significantly reduce the number of explicit parameters that a macro has to declare, and at the same to allow a caller to pass arbitrary amount of external information to the macro; what is more, a macro user obtains a persistent mechanism (that is, a format library) that can store and manage the complex parameter values in a central place, independently of its various client programs.

**APPLICABILITY**

Use this pattern when

- You want to reduce the number of explicit parameters declared in a macro while extending its features and functionality; and
- You want to store and manage related or dependent parameter values in a central persistent place and separate them from calling programs; and
- You want to fine-tune parameter values used by client programs without changing the pieces of invocation code in the client programs; and
You want a stable macro programming interface that won’t change even when you add more new parameters to the macro later on.

**STRUCTURE**

Using FAMP pattern to develop a macro will result in a set of programs and formats with following structure:

**PARTICIPANTS AND COLLABORATION**

- **A format program**
  
  A format program that uses PROC FORMAT to create a format library, which is a central persistent place containing a family of the formats that client programs use to call the macro. The format program only has to be run once during the initialization. It can be independently modified and re-run to update the format library when necessary.

- **A format library**
  
  A central persistent place that stores the formats (or informats) containing various sets of parameter values used by client programs. The format library is created by the format program.

- **A macro**
  
  Declare the format parameters with other parameters in the macro declaration; specify the format structure it can handle; and retrieve the parameter values passed by formats when it is called.

- **A family of client programs**
  
  Invoke the macro with the appropriate parameter formats stored in the format library, together with other actual parameters.

**CONSEQUENCES**

The FAMP pattern offers the following benefits:

1. It lets you create a very lean interface for parameter passing to a macro. By encapsulating subsets of parameters with SAS formats, you can significantly reduce the number of explicit parameters declared in the macro without sacrificing its functionality, hence making the macro more generic, more easy to understand and use, and eventually more powerful.

2. It lets you provide a stable programming interface to a macro, which often ends up with more parameters as it evolves. Although adding more parameters to a macro makes the macro more powerful and flexible, it also make it harder to use for users who don't need to customize it. FAMP can keep unchanged the declared macro parameter list that that is enough for most users while adding new encapsulated parameters through the existing format parameters. Only users who need more customizability will have to look into and use the parameters hidden in the format parameters.

3. It lets you tweak the complex parameter values stored in a format all together in one place without affecting the macro calls in the client programs. Macro parameter values usually stay close with the macro invocation code, but when you have sets of interrelated parameter values that are shared by many different programs, and frequently subject to change, you will find they
are sources of serious coding and maintaining problems. With FAMP, complex and interrelated parameter values are referenced through the names of formats that are stored and managed separately in a format library; therefore making any changes on them will not affect the corresponding pieces of the invocation code.

4. It lets you not only consolidate complex parameter values in a single persistent place but also organize them hierarchically. The SAS nested format mechanism allows a new format implementation to be defined in terms of the existing formats by reusing, overwriting, and extending the ones already defined.

5. It lets you use generic parameter expressions and manipulate them dynamically. Any SAS format or informat entry can contain macro expressions or macro variables as placeholders (like the &Sysdate and %sysget(SAS_EXECFILENAME) in the next implementation section). When a macro expression is enclosed with single quotation marks (') in a format, it can be dynamically substituted with actual macro values every time the entry is extracted from the format during the macro execution.

6. It reduces the learning curve necessary to successfully leverage a macro application, minimizes the duplicated logic and data in the client programs, and makes the invocation code shorter, clearer and less vulnerable to the changes.

The pattern has following drawbacks:

1. It brings in an extra level of indirection that may add to the complexity of the macro design and implementation, and you have to create a separate file to handle the parameters encapsulated in formats.

2. It requires the macro developer to write additional documents for the format-encapsulated parameters, and a macro user may have to learn how to create the right formats, though it is quite easy to do even for an entry-level SAS programmer.

3. It may cause format inflation. A macro user may end up with defining a plethora of formats for the possible macro calls if he or she doesn’t carefully design and construct format parameters using the nesting format mechanism.

4. It may end up with a lot of dangling formats that are either not used or abandoned by macro users. Besides, without careful attention, format parameters may become dumping grounds, littered with obsolete or abandoned parameters once used.

IMPLEMENTATION

Implementation considerations regarding the macro

1. Wrapping proper subsets of parameters with formats. Although you can pack all the parameters that a macro has with a single format, it is most likely not the way you want, because the format parameter will bring in another layer of indirection. If your macro has a long list of parameters, you should consider packaging the subsets of parameters with formats that are often used together, changed together, or contain the same or similar values across client programs.

2. Retrieving a parameter from a format. When you declare a format parameter in your macro, you have to write code to retrieve the parameters stored in the format within a macro implementation. Since there are two types of SAS formats, i.e. SAS format (defined with value statement) and SAS informat (defined with invalue statement). If you use a SAS format as a macro parameter, use the following macro statement to obtain an encapsulated parameter from the format with the parameter name as a key word

```plaintext
%Local hidden_parameter;
%Let hidden_parameter=%Sysfunc(putc(PARAMETER, FORMAT.));
```

If you use a SAS informat, use the following macro statement to obtain an encapsulated parameter from the informat.

```plaintext
%Local hidden_parameter;
%Let hidden_parameter=%Sysfunc(inputc(PARAMETER, INFORMAT.));
```

Note that if the parameter values contain special characters, such as %, &, you may have to use %QSysfunc() instead of %Sysfunc() to obtain the right contents. Besides, the parameter name, or keyword in the base function inputc() and putc() is case sensitive.

3. Retrieve an array of parameters from a format. If a group of parameters wrapped in a format have a linear relationship and are properly indexed, you may access them with a loop statement like one as follows:

```plaintext
%Do idx = 1 %to %10;
  %local Hidden_Parameter&IDX;
  %let Hidden_Parameter&IDX =%Sysfunc(putc(PARAMETER&IDX, FORMAT.));
%End;
```

4. Providing a default format for a format parameter. When a macro format parameter is defined, it is better to create a default format for it in the PROC Format program. Here are the reasons for doing so: first, a client program doesn't have to create the format parameter each time the default value is sufficient; second, it will show what kind of format that a macro user should provide if he or she has to create a new one as parameter for his or her special use; third, the default format can be commonly referenced by many client programs; finally, a macro user can define a new format by reusing many entries in the default format.
5. **Documenting the format parameters.** Since a format parameter introduces another level of indirection, the extra documents should be provided to describe how the format should be customized, the meaning of each parameter wrapped in the format, and examples of how to use it in various situations.

Implementation considerations regarding the formats that are used as macro parameters:

1. **Creating a new format from existing formats.** Basically, the format (or informat) used in the FAMP pattern is a simple dictionary of keyword/value pairs. To facilitate the format creation and reduce the redundancies within formats, it is best to create a new format with following three approaches through the SAS nested format mechanism:
   - **Reuse** – the new format directly uses the keyword/value pairs defined in the existing formats, or its parent formats.
   - **Overriding** – the new format replace one or more keyword/value pairs that it has inherited from its parent formats.
   - **Extension** - the new format can add new keyword/value pairs apart from the existing pairs from its parent formats.

2. **Using macro variables or macro expressions in format entries.** You can use macro variables or even macro expressions in format entries. If you enclose a macro expression with double quotes (“”) in a format entry, the macro expressions in the entry will be resolved immediately when the Proc FORMAT is compiled and executed; if you enclose a macro expression with single quotes (‘’) in a format entry, the macro expressions will not be resolved until the entry value is retrieved at the macro execution time. This dynamic resolution of single-quoted macro values or expressions makes defining parameter values very flexible and powerful. You can even use macro control and loop statements within a format entry and resolve them with macro values generated during the macro execution time. However, you should be very careful not to have the program logic slip in the macro parameter values, or expressions, because doing so will contradict the intent of this pattern.

3. **Supporting exception handling.** If a requested parameter name or keyword does not match any entry in an actual format parameter provided, a runtime macro error may occur. Therefore, in order for a macro to catch the errors in format parameters during the macro execution time, you can always have OTHER="." entry defined at the end of a format (or informat). Therefore, when a specified parameter cannot be retrieved from the specified format, a dot (.) will be returned, which makes the macro aware that either the specified parameter value cannot be found, or it is the end of parameter retrieval.

**EXAMPLE RESOLVED**

List 5 is an complete enhanced version of %Report macro implemented with FAMP pattern. Two format parameters are introduced to replace five-title parameter and five-footnote parameters; therefore, the total number of macro parameters is reduced to 5 from 14 in the original version.

```plaintext
/* The enhanced %Report macro using FAMP Pattern */
%MACRO REPORT(
   RPTDSN= /* REPORT DATASET */
   ,VARS= /* REPORT VARIABLES */
   ,TITLEFMT= /* TITLE INFORMAT */
   ,FOOTNOTEFMT= /* FOOTNOTE INFORMAT */
   ,OUTFILE= /* OUTPUT REPORT FILE */
   );
%local idx titlex footnotex;
ODS Listing close;
ODS RTF NOTOC_DATA File="&OutFile" style=minimal;
%let idx=1;
%let Titlex=%qsysfunc(putc(TITLE&IDX, &TitleFMT));
%Do %While(&Titlex NE %str(.));
   Title&idx %unquote(&Titlex);
   %let idx = %eval(&idx +1);
   %let Titlex=%qsysfunc(putc(TITLE&IDX, &TitleFMT));
%End;
%let idx=1;
%let Footnotex= %qsysfunc(putc(FOOTNOTE&IDX, &FootnoteFMT));
%Do %While(&Footnotex NE %str(.));
   Footnote&idx %unquote(&Footnotex);
   %let idx = %eval(&idx +1);
   %let Footnotex=%qsysfunc(putc(FOOTNOTE&IDX, &FootnoteFMT));
%End;
Proc PRINT DATA=&RPTDSN WIDTH=Minimum;
   VAR &VARS;
```

All titles and footnotes used by the client programs are collected and defined in a separate Proc Format program file as follows:

Libname library 'C:\FormatLib';
Proc FORMAT library=library;
VALUE $ Default_TITLE_FMT
   "TITLE1"="XYZ Corporation"
   "TITLE2"="Protocol: 8888"
   "TITLE3"="Table ID: xx.x.x"
   "TITLE4"="Main Title"
   Other="."
;
VALUE $ DEMOG_TITLE_FMT
   "TITLE3"="Table ID: 16.1"
   "TITLE4"="Demographics"
   OTHER={$Default_Title_FMT256.}
;
VALUE $ LAB_TITLE_FMT
   "TITLE3"="Table ID: 16.5"
   "TITLE4"="Lab Test"
   OTHER={$Default_Title_FMT256.}
;
VALUE $ AE1_TITLE_FMT
   "TITLE3"="Table ID: 16.8.1"
   "TITLE4"="Adverse Event"
   "TITLE5"="During Double-Blind Period"
   OTHER={$Default_Title_FMT256.}
;
VALUE $ AE2_TITLE_FMT
   "TITLE3"="Table ID: 16.8.2"
   "TITLE5"="During Open Label Period"
   OTHER={$AE1_Title_FMT256.}
;
VALUE $ GENERIC_FOOTNOTE_FMT
   "FOOTNOTE1"='The table is created by %sysget(&SAS_EXECFILENAME)'
   "FOOTNOTE2"='Report Creation Date: &SysDate'
   OTHER="."
;
Run;

As you may see, format $Demog_Title_FMT, $Lab_Title_FMT, and $AE_Title_FMT are constructed by extending and overwriting parent format $Default_Title_FMT so that the title 1 and title 2 can be shared by all the programs in a study, which makes those child format definitions much simpler. $Generic_Footnote_FMT is a generic format that includes macro variables, and expressions so that it will dynamically provides a uniform footnote for all the tables generated.

With FAMP pattern, all four client programs showed in List 7 are greatly simplified:

/* DEMOG.SAS */
Libname library 'C:\FormatLib';
...
%REPORT(
   RPTDSN=DEMOG,
   VARS=PT AGE GENDER,
   TITLEINFMT=$ DEMOG_TITLE_FMT,
   FOOTNOTEINFMT=$GENERIC_FOOTNOTE_FMT,
   OUTFILE=C:\TEMP\DEMOG.RTF
)
/* LAB.SAS */
...
%REPORT(
   RPTDSN=LAB,
   VARS=PT TYPE TEST VALUE FLAG,
   TITLEINFMT=$LAB_TITLE_FMT,
   FOOTNOTEINFMT=$GENERIC_FOOTNOTE_FMT,
   OUTFILE=C:\TEMP\LAB.RTF
)
/* AE1.SAS*/
/* AE2.SAS*/

List 7: The revised client program DEMOG.SAS, LAB.SAS, AE1.SAS and AE2.SAS that use the enhanced %Report macro

KNOWN USES

The FAMP pattern is used by Celgene Biostatistics & Data Management group to develop and enhance standard macros. One example is a general purpose listing macro that can create different styles of listings with various titles and footnotes. By using FAMP pattern, many property parameters previously explicitly declared in the macro are replaced by a single format parameter, making both the macro and the listing programs much easier to understand and use.
Many macros adopt the FAMP pattern to provide more powerful functions with fewer parameters. The pattern is especially useful to the development of those standard macro applications with a long list of secondary or cosmetic parameters, such as table macros, graphical macros, and even macros for regular statistical analyses and data manipulations.

RELATED PATTERNS
Vararg [4] is another way to shorten macro parameter list by treating multiple parameters as one parameter. However, Vararg’s purpose is to develop a macro that can handle an indefinite number of parameters. The macro using Vararg pattern does not know in advance how many parameters will be passed through. It only knows that the parameters to be received are all of the same form, and they can be treated in a uniform way. In contrast, a macro using FAMP pattern does know how many parameters, both explicit and implicit, will be passed when you design and implement the macro. The FAMP pattern reduces the number of parameters a macro declares explicitly, but not the number of parameters that will actually be passed. Besides, FAMP pattern focuses on providing a central persistent repository for families of actual parameters that have common base values and structures used by many different client programs, and for other complex parameter values that a user wants to define only once and separately.

Vararg pattern can be used in conjunction with the FAMP pattern. This is because a format parameter can turn a group of ordinary macro parameters into a consistent form, which makes it easy to convert an ordinary macro into a vararg macro.

REFERENCES

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact authors at:
Lei Zhang and Kenton Wride
Celgene Corporation.
86 Morris Avenue
Summit, NJ 07901
Phone: (908) 673-9000

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.