Paper 055-29

Better Clay Builds Better Bricks:

Some Simple Suggestions To Writing Professional Macros

Michael P. D. Bramley, Kendle International, Cincinnati, OH

ABSTRACT

Interested in taking your SAS® macros to a new level? This tongue-in-check article describes a collection of tips, techniques, and macro functions that represent the author's unorthodox approach to software: it should never be provided on an "AS IS" basis.

The offerings in this article range from the basic of knowing exactly what the user expects and wants to the more esoteric, yet highly useful, principle of providing cogent feedback and error messages. This article promotes an integrated philosophy of professionalism for SAS system designers and programmers.

This article's target audience is anyone who has, does, or ever will, design or write a macro. Regardless of how basic or omniscient the macro's function or the final task to be accomplished, this author believes that the following principles will result in improved code and happier users.

INTRODUCTION

Every SAS programmer is eventually faced with the exponentially aging stress of having to debug a macro that went insane due to an undetected and erroneous parameter. Perplexed is an apt term to describe the programmer faced with the innocent user's query, "What does this macro do?". Sympathy is poured out to the ill-fated programmers who struggle for days trying to figure out why someone's code that ran a million times before is suddenly broken, only to discover that a macro is unexpectedly populating a macro variable outside of the macro's scope.

These are but a few of the obstacles that SAS programmers face on a daily basis. This article presents an integrated philosophy of SAS/MACRO design and writing to meet these challenges head on. Moreover, it provides SAS programmers with actual tools that they can utilise immediately.

1. DON'T EXPOSE YOURSELF

The most important thing a macro can ever do is keep its variables to itself (unless specifically designed otherwise). This leads to the important concept known as scope. In SAS, a macro variable's scope is either local or global, depending upon which symbol table SAS finds the macro variable's name in. A symbol table is nothing more than an internal table that SAS maintains to keep track of macro variables. During a SAS session there is one global symbol table and a local symbol table for each macro invoked. Global macro variables exist for the duration of the SAS session and can be read from or written to anywhere—even inside a macro. Local macro variables, on the other hand, live only inside the macro they are declared in and exist only during that macro's invocation. Any macro invoked by the declaring macro also has full access to these locally declared macro variables.

Macro programmers should make it a mission to ensure that the macro variables used inside their macros are local. SAS employs the following rules to determine the scope of macro variables.

1. Macro variables declared with a %LOCAL statement are local; those declared with a %GLOBAL statement are global.
2. Macro parameters, that is, arguments to macros, and the SYSPBUFF macro variable are *always* local.
3. Creating macro variables with SYMPUT can be your downfall: it places macro variables in the most local nonempty symbol table. This is often not where you expect them to appear.
4. CALL SYMDEL and %SYMDEL only delete macro variables from the global symbol table.
5. Within a macro, %LET and %DO do not automatically create a local macro variable. If a macro variable of the same name exists in an outer symbol table, then that (probably incorrect) macro variable is used.

The lesson to learn from this list is that the best way to ensure that the macro variables are local (excluding parameters) is to list all of them on a %LOCAL statement right after the macro's declaration.

2. PROTECT THE ENVIRONMENT

Macros (or more accurately, macro programmers) occasionally feel the need to change the setting of a SAS system option in order to hide the details of their process or simply ensure that output is in the desired format. However,

some users may not appreciate that a macro has, for example, turned macro printing off, when they had specifically set that option on.

To avoid aggravating users (who may know where the author lives), it is advised that macro programmers first save the setting of the option they intend to modify, with the caveat that their macro restore the setting prior to exiting. This can easily be accomplished with the GETOPTION function and the SYSFUNC macro function. The GETOPTION function returns the setting of the option passed as the first parameter, with optional parameters for units of measurement and keyword formatting. The SYSFUNC macro function is the bridge that allows macro code to execute datastep functions. For example, the following code saves the system option that controls whether macro generated SAS code is included in the log and the number of lines per page in the macro variables **MPRINT** and **PGSIZE**, respectively:

```
%Let MPrint   = %SysFunc( GetOption( MPRINT, KeyWord ) ) ;
%Let PGSize = %SysFunc( GetOption( PS, KeyWord ) ) ;
```

The nice feature of the GETOPTION function is that, by using the KEYWORD option, it always returns a value that is directly usable in a SAS OPTIONS statement. Continuing with the above example, GETOPTION returns either MPRINT or NOMPRINT in the first statement and PS=60 in the second statement, assuming this value has been previously set. Thus, the following statement is all that is required to restore these options prior to exiting a macro:

```
%Options &MPrint &PGSize ;
```

If the macro needs to save / restore many SAS options, an alternate and more efficient method is to use the OPTSAVE and OPTLOAD procedures. These procedures save / restore most of the options to / from either a SAS registry or a SAS data set. The default is to use the SAS data set SASUSER.MYOPTS. These procedures do not save / restore options requiring a password, options that can only be specified at startup, and the ARMAGENT, ARMFORMAT, ARMLOC, ARMSUBSYS, AWSDEF, FONTALIAS, and STIMER options.

The following code saves and restores most of the SAS options to and from the data set named MyOptions in the SASUSER library:

```
Proc OptSave Out = SASUser.MyOptions ;
Run ;

Proc OptLoad Data = SASUSer.MyOptions ;
Run ;
```

Note that while the documentation for OPTSAVE indicates that the user specifies the data set name with a DATA= option, in fact, it requires an OUT= option. Moreover, if the data set already exists, it will be overwritten without warning.

Finally, there are a pair of DMS commands that save / restore SAS options during interactive sessions. Their syntax is DMOPTSAVE <*location*> and DMOPTLOAD <*location*>, where location is either a quoted registry key or data set name (with the default currently set to SASUSER.MYOPTS). The drawbacks to these commands are the lack of an option to stop the DMOPTSAVE command from overwriting an existing key or data set as well as an option to have DMOPTLOAD delete the key or data set after successfully loading the options. These options could be implemented quite naturally in the OPTSAVE and OPTLOAD procedures.

Thus, SAS software provides three separate ways to save / restore the system options, which give rise to methods to check an option's setting. Henceforth, there is no excuse for macro programmers to change SAS options and not restore them prior to exiting. Failure to do so is either arrogance or laziness.

3. FORGOTTEN FOOTNOTES

Saving and restoring system options can be extended to system titles and footnotes. Unfortunately, there are no functions to get or set titles or footnotes. Perhaps, this is an item that may be better addressed in a SASware ballot. Until that time, programmers must use the SQL dictionary tables to save / restore titles and footnotes.

The following code uses the SQL procedure to save the titles and footnotes to the SAS data set WORK.MYTITLES.

```
Proc SQL ;
   Create Table Work.MyTitles As
     Select *
     From Dictionary.Titles
   ;
Quit ;
```

This code creates a data set with the columns TYPE, NUMBER, and TEXT. The TYPE column is one byte long and equals "T" if the observation is a title or "F" if it is a footnote. The NUMBER column contains the title or footnote number and the TEXT column equals the value of the title or footnote (up to 256 bytes). Note that if there are no defined titles and footnotes, the resulting data set will have zero observations.

A simple datastep can be used to restore titles and footnotes once the macro has finished its task. The following code demonstrates:

```
Title ;
Footnote ;

Data _Null_ ;
   Set Work.MyTitles ;
   Length EType $11 ;

   If Type EQ 'F' Then
      EType = 'Footnote' ;
   Else
      EType = 'Title' ;

   EType = CatS(EType, Number) ;

   Call Execute(EType||Trim(Text)||';' ) ;
Run ;
```

For each observation in the Work.MyTitles data set, the above datastep initialises the **ETYPE** variable to either the string "FOOTNOTE" or "TITLE". The SAS9 CATS function concatenates its arguments by first automatically converting numeric values to character with the BEST. format and trimming leading and trailing blanks. Thus, the **ETYPE** variable will be set to "TITLE" or "FOOTNOTE", with the appropriate number appended. The text of the title or footnote is then appended to this variable and the result passed to the EXECUTE function to queue it for execution. This function places its parameter in the compiler's queue to execute, just as if it had been written to the SAS program file originally. After the datastep has processed the observations in the data set, all of the title and / or footnote statements will be executed.

The title and footnote statements prior to the datastep ensure that all titles and footnotes are cleared in case the user has previously issued a title / footnote statement without text. This may not be the most elegant solution, but it is an efficient method of saving and restoring titles and footnote.

4. IT'S YOUR MESS – YOU CLEAN IT UP!

This point is closely related to the previous sections. Macros often feel the need to reach out to the world via temporary datasets or external files. In order to accomplish this, a file or library reference may be necessary. When the macro exits, these file or library references remain active in the SAS metadata, and the datasets or external files associated with them occupy space on the operating system's file system. What's even worse is that the macro could change a user's file or library reference instead of creating one that is temporary and unique.

The FILENAME and LIBNAME functions allow two calling forms: to assign and to de-assign a file or library reference. These calling features provide macros the capability of creating references on the fly and removing them from the SAS metadata prior to exiting. To create a file or library reference, call the requisite function with two parameters: a reference name and a path, as follows:

```
%Let RC = %SysFunc( FileName( FileRef, File ) ) ;
%Let RC = %SysFunc( LibName( LibRef, Path ) ) ;
```

The macro variable **RC** (return code) will contain a value of zero if the call was successful, non-zero otherwise. To remove the file or library references, call the function with only the reference name as a parameter:

```
%Let RC = %SysFunc( FileName( FileRef ) ) ;
%Let RC = %SysFunc( LibName( LibRef ) ) ;
```

The FILENAME function has the wonderful feature of automatically generating a unique temporary file reference when called in macro and the first parameter is the name of an empty macro variable. In this case, the FILENAME function creates a unique file reference of the form #LNkkkkk, where kkkkk is a zero-filled integer that SAS maintains internally, and places this value in the macro variable. In the first example, if **FILEREF** was a macro variable with no value, then calling the FILENAME function would populate it with a unique file reference. The second example would de-assign the file reference and set the macro variable to blank.

While the LIBNAME function does not possess this capability, the proposal has been made to SAS Institute to extend this functionality to it. In the meantime, macro programmers can still create and use library reference names that are not likely to be used by others. Prefixing and suffixing library reference names with one or more underscores is an excellent technique of avoiding users' library references. However, this technique is by no means bullet-proof. To test if a library reference is currently active, check the name to the PATHNAME function. If the PATHNAME function returns a blank, then the library reference is not active and can be used.

Be advised that the PATHNAME function can return misleading results if there is a file reference and a library reference of the same name. For such cases, it has an optional second parameter that takes either the value L or F to force it to look for a library or file reference, respectively. The default is to look for the latter. It is this author's belief that not including this parameter is a guaranteed future free ticket to debug hell.

SAS software also provides methods to remove temporary external files and data sets. Use the FDELETE function to remove those external files (or empty directories) a macro creates by calling the function with the sole parameter of the file reference for the external file:

```
%Let RC = %SysFunc( FDelete( FileRef ) ) ;
```

FDELETE returns a zero if it was successful, non-zero otherwise. This function must be called prior to removing the file reference from the SAS metadata via the FILENAME function.

To delete temporary data sets, there are many possibilities. These include using the DELETEITEM command, the DELETE statement in the DATASETS procedure, or the DROP TABLE statement in the SQL procedure. The DELETEITEM command can only be used in interactive sessions and requires that the data set name be fully specified, that is, *libname.member.type*. Unfortunately, all of these methods add a RUN-boundary to the macro. The SASware ballot will hopefully include the addition of a library member deletion function, so ensure you vote for it.

Given the above SAS functions, commands, and procedures, there is no reason for a macro to modify SAS metadata and not clean up after it has finished its task. Moreover, these same SAS functions can be used to ensure there is no collision between the metadata the macro requires and those already created by the user.

5. THE CASE FOR RETURN CODES

Macro code is usually tested under controlled conditions before being let loose to wreak havoc on the world. In the shop, the code always works because it executes under pristine conditions. Given this false sense of security, it can be easy for macro programmers to forget to test the results of functions and/or commands. Missing just one check can result in a poorly behaved macro and one very frustrated and upset user.

SAS functions and commands always return information on the status of their execution, be it directly or indirectly (there are some exceptions). Such return codes should always be mined for their invaluable information to ensure that the macro executes successfully.

For example, the SAS I/O functions FOPEN and DOPEN, which open an external file or directory, return a file identifier for future use or zero [1]. If zero is returned, then the SYSMSG function can be used to display the text of the error message, indicating why the function failed (this applies to all SAS I/O functions). In addition, there are automatic macro variables such as SYSRC, SYSFILRC, and SYSLIBRC that contain the status of the last X (or %SYSEXEC) command, FILENAME or LIBNAME function, respectively. Testing these allows a macro to determine if it should continue normally or abort. Macro programmers should comb their code mercilessly to ensure that they check return codes at every instance.

SAS is quite good at ensuring that functions and external activities return a status to the calling procedure. This makes return codes the arbiters of truth for macro programmers. It is folly to assume that a function will always

execute successfully simply because it has done so before. "Past performance is not an indication of future results" is a mantra for investors and programmers alike.

6. FORESIGHT REQUIRES PLANNING

There is nothing worse than the stabbing pain one feels when a user emails to complain that a macro failed. That extra pain is the twisting of the knife when one realises that the macro in question does not contain any debug facility.

Debugging facilities can be quite simple or extremely complex: the choice should depend on the subject of the macro. Either way, they should always be included in any sufficiently large or complex macro. Perhaps, the best medium between these two extremes is to add a keyword parameter say, VERBOSE=Y|N or DEBUG=Y|N (with the default set to N). Then, add macro code that conditionally prints vital data to the log or listing, such as, all macro variable and parameter values, and the intermittent values in a complex section of code. Whatever can be used as breadcrumbs on the problem-finding trail should be written to the log. However, simply dumping anything and everything possible to the log is not a useful debug facility. Nor does the output have to make sense to the user—they are not going to debug the code. However, guideposts in the form of comments can certainly aid the programmer in finding what they are looking for. While formatting is nice, the effort should be relative to the benefit.

Finally, do not place any weight in the myth that adding many tests via the %IF statement will significantly add to the macro's run-time. Testing *two hundred* %IF statements in a macro increases run-time by approximately *one one-hundredth* of a second, while testing *two thousand* %IF statements adds approximately *eight one-hundredths of a second* [2]. Clearly, such increases in run-time are more than acceptable to both the developer and the user.

The main point of this section is that macro programmers should include time for adding debugging code to the development cycle. Once a macro is released into the wild, it will travel uncharted waters and face many dangers. By planning for the macro's failure in advance, macro programmers are possibly saving themselves much aggravation (not to mention the user's lost productivity).

7. DO YOU, AH, DOCUMENT?

A macro programmer's most dreaded task is writing documentation for a macro. Once written, this document can easily become obsolete while the macro undergoes revision in its early stages of life.

To avoid some of this drudgery, it might be more appropriate to imbed the documentation in the macro. That is, write a macro that generates the documentation-on-demand. This coupling brings the documentation into the coding phase, and macro programmers may begin to abstain from stream-of-consciousness coding and actually come to enjoy performing this function. Alright, that last part is probably a hard sell, but the users will love the fact that they can generate documentation in a format they choose at will.

This documentation-on-demand can be achieved by having the macro recognise an invocation of the form

```
%MyMacro( HELP ) ;
```

as a request to produce the documentation. This can be extended quite naturally to the form

```
%MyMacro(HELP-PDF) ;
```

or

```
%MyMacro(HELP-RTF) ;
```

where the PDF and RTF suffixes indicate which ODS destination the user prefers. This ability must be advertised: The author can inform end-users in a standard introductory document, while the macro should notify users every time it is invoked by printing a banner message in the log. It probably wouldn't hurt to include the macro's invocation form that produced the documentation, eg: %MyMacro(HELP-PDF), in the footnote of the documentation.

A natural extension of this is to use the final macro to generate the official documentation for the macro using an ODS destination, and place that file on the web. Thus, the macro author can, with a little bit of extra work, provide the users with multiple access points to the documentation.

This is a simple, yet effective idea to ensure that version specific documentation is always at the user's fingertips. Moreover, it helps to convey the shop's professional attitude and desire to educate users.

8. NEVER TRUST A USER

Macro programmers should begin each line of code with the tenet that users spend their days inventing new and evil ways to break their exquisitely crafted code. While this may not put users in a very favourable light, it does place the onus on programmers to be diligent and write code that assumes nothing.

Macro programmers who base their code on this belief realise that they must interrogate every parameter to ensure that it contains the correct type of data and that it is logically valid. Relieving this high level of paranoia is easy with the following guidelines.

Begin by checking the contents of all parameters by employing the SAS9 functions NOTALPHA, NOTALNUM, NOTDIGIT, and NOTNAME, which test for alphabetic, alphanumeric, digits, and SAS variable names, respectively. These functions take a character parameter and an optional parameter indicating the position to start testing at, and return zero or the position of an offending character in the first parameter. Prior to SAS9, these functions can be implemented using the VERIFY function.

These functions enable a macro to catch users' mistakes (innocent or otherwise), such as when a macro parameter **Parm1** contains the value $Test and a SAS name is expected. For example, the code snippet below verifies that the macro variable **StartNum** contains only an integral value, and failing this, writes an error to the log and sets an error flag:

```
%If %SysFunc( NotDigit( &StartNum ) ) %Then %Do ;
    %Put ERROR: Only integral values allowed in parameter StartNum (&StartNum). ;
    %Let Error = 1 ;
%End ;
```

The reasoning behind the error flag is the belief that every macro can (and should) test all of its parameters prior to performing the actual designated task. Once all of the testing is done, the macro tests the error flag and either continues normally or executes the %RETURN macro element (which jumps to the end of the macro). Setting an error flag—instead of terminating a macro as soon as the offending parameter is discovered—allows all of the parameters to be tested and provides complete feedback to the user. Thus, if k parameters contain erroneous settings, the user only has to invoke the macro once to find all of the errors, instead of iteratively correcting one parameter and invoking the macro.

An alternative to the above strict testing is to have the macro set the parameter to a default value when an invalid value is detected. This is demonstrated in the following code snippit:

```
%If %SysFunc( NotDigit( &StartNum ) ) %Then %Do ;
    %Put WARNING: Parameter StartNum contains a non-integral value (&StartNum). ;
    %Put WARNING: StartNum will be set to the default of 1. ;
    %Let StartNum = 1 ;
%End ;
```

Overriding the user's parameters makes sense for the parameters that do not seriously affect the macro's ultimate task, such as page numbering. When no reasonable default exists or can be decided upon, then it is best to flag the condition as an error.

Macro programmers can also increase their return on investment by testing the logical correctness of parameters. Not only does SAS provide functions to test for the existence of external files and SAS data sets, it also provides an arsenal of SAS I/O functions that interrogate a SAS data set to determine if it has any observations, variables, indices, etc. Compare the following macro code snippet:

```
%Let FID  = %SysFunc( Open( SASHelp.Class ) ) ;
%Let NObs = %SysFunc( AttrN( &FID,NOBS ) ) ;
%Let RC   = %SysFunc( Close( &FID ) ) ;
```

to the datastep version:

```
Data Null ;
   Call SymPut( 'NObs', CatS( NObs ) ) ;
   Stop ;
   Set SASHelp.Class NObs = NObs ;
Run ;
```

The first code snippet opens the SAS data set SASHELP.CLASS, sets the macro variable NOBS to the number of observations in the data set, then closes it. The ATTRN function returns information on numeric properties of a data set, while ATTRC returns information on character properties of a data set. These two functions make up the who's who of data set properties, allowing a macro to query 39 properties of a SAS data set.

The second code snippet achieves the same task, but requires a great deal more overhead: SAS first compiles the datastep, and in so doing, sets up the automatic variables, buffers, initialisation, and error trapping code that programmers have come to rely on. Then the datastep is executed. Then terminated. This code could entail a fair amount of overhead and time if executed even ten times and/or written inefficiently.

In contrast, the first code snippet only uses the macro compiler and the %SYSFUNC function to create a datastep wrapper for the function calls. This code executes significantly faster than the second and without creating a run-boundary. The datastep version is prone to failure if the data set cannot be opened, whereas, the first code snippit can test the result of the OPEN function and conditionally execute code. Moreover, if the macro needs other properties from the SAS data set, then the datastep version must add nested function calls for each property or break them into multiple lines of code. Furthermore, if the datastep version needs information from macro variables, then code must be added to extract that data for every instance. Overall, the datastep code is inefficient and harder to maintain.

Another check to perform is to verify that user-specified variable names actually exist in a data set and are of the correct type. This type of parameter validation is almost trivial and has been available since SAS 6.12. The functions with the VAR prefix extract variable properties from a data set, such as name, position, format, type, length, and label. For instance, the following code checks that the user-specified variable in the macro parameter VName exists and is numeric:

```
%Let FID  = %SysFunc( Open( SASHelp.Class ) ) ;
%Let VPos = %SysFunc( VarNum( &FID, &VName ) ) ;

%If &VPos %Then %Do ;
    %If %SysFunc( VarType( &FID,&VPos ) ) NE N %Then
        %Put &VName is not numeric. ;
%End ;
%Else
    %Put &VName does not exist in data set. ;

%Let RC   = %SysFunc( Close( &FID ) ) ;
```

The SAS I/O VARNUM function returns either the position of the variable or zero if the variable does not exist in the data set. The VARTYPE function returns the variable type as either C or N (or blank if the variable position is out of range). The above code can be easily modified and placed in a macro function that returns either a zero if the variable does not exist or is of the wrong type, or a one to indicate that the variable exists and is the correct type.

Implementing the above techniques in macros greatly reduces the probability that the macro will attempt to perform actions on parameters with nonsense values. Moreover, the combined use of the VARXXX functions permits macros to detect a user's mistakes and avoid potentially crippling errors.

9. NO SUBSTITUTES ALLOWED

While the previous section stressed the importance of not letting users hijack code into realms of the unknown, it is also important to treat users gently. This includes making their life easier by cutting down on repetition or typing and providing them with short cuts.

Library and file references offer perhaps the greatest opportunity for providing simple yet highly appreciated short cuts. For those macros that require the user to specify a path as a parameter, add code that checks if the parameter is a valid SAS name, and if so, attempt to resolve it as a library reference via the PATHNAME function.

For those macros that require a file name, ensure that the code will also accept a file reference. If the file reference cannot be used directly within the macro, then add code similar to the above library reference check to resolve the file reference and use that instead.

Differentiating between concatenated and simple library references is fairly straightforward: a concatenated library reference is a list of single-quoted pathnames enclosed in parentheses. Provided that some simple guidelines are followed, the following macro function handles any library references seamlessly:

```
00 %Macro _GetPathName( Lib, PNum ) ;
01    %Local P ;

02    %If "&PNum" EQ "" OR %SysFunc( NotDigit( &PNum ) ) %Then
03        %Let PNum = 1 ;

04    %If Not %SysFunc( NotName( &Lib ) ) %Then %Do ;
05        %Let Path = %SysFunc( PathName( &Lib, L ) ) ;

06    %If %Length( &Path ) %Then
07        %If "%SubStr( &Path, 1, 1 )" EQ "(" %Then %Do ;
08            %Let Path = %QSysFunc( TranWrd( &Path, %Str( %'), %Str(%') ) ) ;
09            %Let Path = %Scan( &Path, &PNum, ()%Str(%') ) ;
10        %End ;
11    %End ;

12    &Path
13 %Mend _GetPathName ;
```

Lines 00-01 declare the macro function with the parameters library name and path number, and the local macro variable P, while lines 02-03 check that the path number parameter is valid. If the library name parameter is a valid SAS V7 name (line 04), the macro then attempts to resolve the pathname(s) the library reference points to in the macro variable P (line 05). If the library reference begins with a parenthesis, it is concatenated (line 07), and thus, demands special handling. First, all occurrences of space-single-quote are converted to single-quotes (line 08), and the desired pathname is extracted from the list with the %SCAN function by specifying the delimiters are parentheses and a single-quote. Note that the _GETPATHNAME macro function takes advantage of the fact that %SCAN treats multiple delimiters as a single delimiter.

The _GETPATHNAME macro functions works as long as a simple guideline is obeyed: pathnames cannot include single-quotes or parentheses. If this rule is applied, then it is easy to extract each path in the concatenated library reference.

Allowing users to specify parameters using SAS short-form is always a big plus. The SAS I/O VARNAME function returns the position of the variable name or zero if the variable does not exist in the data set. Marrying this function with VARNUM can spruce up a macro to make it appear smarter than the average bear. For example, sometimes the double-hyphen form of the SAS variable list is not allowed. The following code allows a user to specify this form and expand it into the macro variable LIST:

```
%Local List FID FromVar ToVar I ;
%Let FID = %SysFunc( Open( &DataSet ) ) ;

%Let FromVar = %Scan( &VarList, 1 ) ;
%Let ToVar   = %Scan( &VarList, 2 ) ;

%Do I = %SysFunc( VarNum( &FID, &FromVar ) ) %To %SysFunc( VarNum( &FID, &ToVar ) )
;
    %Let List = &List %SysFunc( VarName( &FID, &I ) ) ;
%End ;

%Let FID = %SysFunc( Close( &DataSet ) ) ;
```

The above code works by using the %SCAN function to isolate the starting and ending variable names. These variable names are then converted to variable positions in the data set in the DO-loop. The single line inside the DO-loop builds a variable list by appending the variable name at each variable position. All error-checking has been stripped from the code in order to focus on methodology.

With the above ideas, it is also quite easy to write a macro function that implements the SAS colon operator on variable names, that is, collects all variable names of same prefix into a list. It takes a little more ingenuity to write a macro function that expands a singly-hyphenated variable list of the form XXXaaaa – XXXbbbb, where XXX is a common prefix and aaaa and bbbb are numbers, with aaaa = bbbb. All in all, adding this type of functionality to macros elevates them from simply smart to sublimely superior.

10. WHY SANTA KEEPS LISTS

Connected with the previous section's emphasis on making things easier for the user, sometimes it makes sense to design a macro around a central function, and have it execute that task iteratively on each of the items in a user-supplied space-delimited list. This way, the user does not have to repeatedly execute the macro for each item.

This list could be an expanded variable name list, a list of data sets, a list of external files matching a pattern, or whatever is required to get the job done. Each item (or token) in the list is a self-contained item capable of being acted on by the central task of the macro. The central task is the backbone of the macro, and everything else is fancy window dressing to make the user's life simpler.

The following template can be employed as part of a shell macro (don't forget the error checking) that invokes the macro that performs the central task on each token in the list. This is one of those simple concepts in theory, that in practice appears so magical, aka Santa-esque.

First, define the rudimentary macro that performs the central task with the single parameter of **TOKEN**. All of the remaining macro variables shall be inherited from the invoking macro, and thus, planning is required.

```
00 %Macro Action( Token ) ;
     ...
01 %Mend Action ;

02 %Macro DOIT(DataSet, List) ;
03    %Local I Token ;
04    %Let I = 1 ;
05    %Let Token = %Scan( &List, &I ) ;
06    %Do %While( &Token NE ) ;
07        %Action( &Token ) ;
08        %Let I     = %Eval( &I + 1 ) ;
09        %Let Token = %Scan( &List, &I ) ;
10    %End ;
11 %Mend DOIT ;
```

The first two lines of the above code are dedicated to declaring and defining the central function that is invoked iteratively for each item in the list (via **TOKEN**), whatever it contains. Lines 02-04 declare the macro template with the requisite parameters and initialise the counter I. In this case, the macro ACTION is to perform some task on each variable name (stored in **TOKEN**) in the **LIST** that is found in the data set described by DataSet.

Line 05 extracts the first token from the **LIST**. Lines 06-10 are the core of the macro that loop through each token in order in the **LIST** until all tokens have been processed. Line 07 invokes that macro that performs the central task, say running some kind of analyses. Lines 08-09 increment the counter for the next token and extract that token from the **LIST**. The Do-loop then iterates.

When the counter attains a value higher than the number of tokens in the **LIST**, %SCAN returns a blank. At this point the Do-loop test is negative and the Do-loop exits. All of the tokens in the **LIST** have been processed by the macro ACTION, and the user is sufficiently impressed. After all, isn't that what this is all about?

CONCLUSION

This article has humourously presented ten simple yet powerful concepts / suggestions fashion that can go a long way towards making SAS macros easier to write, update, and more importantly, utilize. While some suggestions, such as cleaning up behind you, takes us back to our childhood days, this author has been frustrated by real life commercial SAS macros that rebel against such common sense ideas.

It is believed that other suggestions, such as allowing the use of substitutions and document-on-demand, are both novel and fairly simple to implement. Other suggestions dig deeper into SAS and provide the macro programmer with a greater sense of control over the SAS environment and thereby, the ability to do more with macros. It is hoped that this article will be accepted in this vein with the solitary goal of improving software and improving the software development cycle.

Unfortunately, there are another twelve suggestions that, due to time and space constraints, did not make it into this article. That, of course, just means that future articles must be written to fill this void. Hopefully, the wait will not be long.

REFERENCES

[1] For more on SAS FILE I/O Functions, refer to this author's article *Combining Pattern-Matching And File I/O Functions: A SAS® Macro To Generate A Unique Variable Name List* in SUGI 27 Proceedings.

[2] Tested on a 1.4GHz personal computer using SAS9.

ACKNOWLEDGMENTS

The author would like to thank SAS Technical Support for their help in answering questions and providing insight in The SAS System.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Please feel free to contact the author at:

Michael Bramley
1200 Carew Tower
441 Vine Street
Cincinnati, OH 45202
W: 513-345-1528
M: 513-315-3756
Bramley.michaelp@kendle.com

Cha Gheill !!