Paper 149

SAS® MACROS
BEYOND THE BASICS

JAY A. JAFFE

So you got some %LET statements, some &symbols, and maybe some %MACROs that help you do your work? Let's go ahead and leverage the full power of the SAS Macro facility. After you complete this workshop, you will understand the mysteries of &&double and &&&triple ampersands, macro functions, the SYMPUT call you've always wanted to use, Macro Language advances in SAS 6.11/6.09E and Version 7, and more. We'll show practical, real-world examples features we discuss, so you will grasp why and when to use Macro Language.

A WEE TEST

This paper is for those with at least a little macro experience. If you can predict what the following code will do, then continue.

```
%LET MOTHER=ma;
%MACRO WHATHAP(PET,VEHIKLE);
DATA _NULL_;
PUT
"My &vehikle&mother ran over my &pet&mother!";
RUN;
%MEND WHATHAP;
%WHATHAP(dog, kar)
```

THE SAS MACRO FACILITY

Most modern computer programming languages offer a feature called preprocessing, the general activity of which is to take one set of character stream data and convert it to another set of character stream data according to well-specified rules. Raw source code is first pumped through the language scanner, where sections of it are syntactically identified as grist for the preprocessor and shunted there for action, rather than to the language compiler. The preprocessor component of the SAS System is called the macro processor.[1] The "macro facility" refers to the macro processor together with the coding language supplied for the preprocessor (the Macro Language) and the features for using preprocessor variables (macro variables).

The coding syntax provided for you to use the macro facility is quite rich compared with the preprocesing features provided with most computer languages. It may be considered, truly, a language in itself, with statements, functions, assignment, and flow control.

In its context — that is, in SAS source code — the SAS System recognizes Macro Language (and macro variables) by particular syntactic tokens, namely the ampersand (&) and the percentsign (%) which prefixed to a name[2] does normally indicate that some segment of code is not to go directly in for SAS processing (e.g., by the DATA Step compiler, a PROC parser, command processor, options tables, etc.), but instead is to take a journey through the macro processor.

The Macro Facility and SAS jobs

Let's take a look[3] at this interaction between the macro processor and other activities of the SAS System as it attacks your source code job. Armed with this understanding, you will be much better equipped to correctly use

[1] In common SAS parlance we refer to macro "processing", although what is going on is preprocessing. I use the terms "preprocessing" and "macro processing", or "preprocessor" and "macro processor", interchangeably.

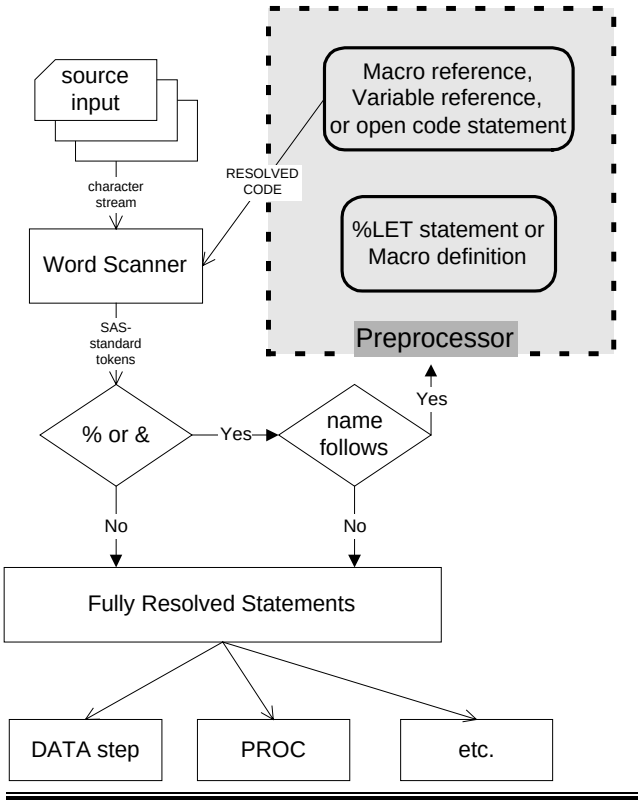
[2] A SAS ***name*** is composed of a string of characters consisting of letters, numerals, and/or underscores (_), the first character of which may not be a numeral.

[3] For greater detail on this and other topics covered in this Tutorial, you should get one of the few complete manuals produced by SAS Institute for the latest Version 6 release, **SAS Macro Language: Reference,** not to be confused with the SAS Guide to Macro Processing, which you already may have but which was put out with the first Version 6 release (6.06). Do get the new one, which represents macro processing as of release 6.11 (6.09E on MVS) .

advanced features of macro variables and Macro Language.

Figure 1 is a simplified logical illustration of what happens to your source code after the SAS System gets hold of it. The word scanner (a tokenizing parser) retrieves a stream of input and chops it into tokens (the smallest chunks information syntactically meaningful to the SAS languages). Normally, these tokens are then handled by some component of the SAS System responsible for code compilation and execution (DATA or PROC steps, SCL language blocks, etc.), or for maintenance of the dynamic SAS environment (via global statements such as OPTIONS, FMTSEARCH, etc., or via session commands).

Figure 1: Macro Language Processing



When, however, the macro processor kicks in, this flow of information from the input stack to these various action components is interrupted and placed on hold. Instead, the preprocessor interprets the macro symbols — ie, the names following the ampersands or percentsigns — and determines what to do with them. The ultimate result of this becomes some sort of substituted code, which the preprocessor then releases back to the wordscanner for passage through to the various components of the SAS system. Having had its way with the original source segment, the preprocessor determines what the SAS System "sees" as source code outside and after preprocessing.

Macro definition and invocation

Preprocessing need not always result in the immediate expression of resolved code to other SAS System components. The macro facility provides a mechanism for saving this until later. In Figure 1, this is illustrated by separating the activities of the preprocessor into two groups, those concerned with defining macros or macro variables, and those concerned with expressing the results of macros or the contents of macro variables.

A SAS macro is nothing more nor less than a compilation of Macro Language and free text stored by the macro facility for later use. A macro is defined by placing such language and text within the bounds of the %MACRO and %MEND statements; the %MACRO statement itself is inspected for parameters and options, which themselves are subject to macro processing along with the remainder of the Macro Language and free text when the macro is invoked. What you are doing when you send "%MACRO ...; [...] %MEND;" code to the SAS System is asking that the system compile the Macro Language therein, and save (remember) both its procedural commands and the textual content also given in the macro definition. This is how we might verbalize the instructions implicit in the "Wee Test" macro definition listed above to some generation of robot:

> Define for later use a macro that understands the two symbols "PET" and "VEHIKLE". When I subsequently call upon this macro by its name, and to tell it a pet and a vehicle, in that order, I want it to transmit back to the wordscanner for further processing the following, in this order and entirely:
>
> The constant text {DATA _NULL_; PUT "My }
>
> The value I told the macro of the symbol VEHIKLE
>
> The value of the independent symbol MOTHER
>
> The constant text { ran over my }

```
The value I told the macro of the symbol PET
The value of the independent symbol MOTHER
The constant text {!"; RUN; }
```

The macro thus defined in the location WHATHAP we then invoked — caused to execute — with the line

```
%WHATHAP(dog, kar)
```

At invocation, the macro facility retrieves the contents of the macro and executes the instructions therein.

Macro Variables

In SAS jargon, a macro variable refers to a named storage location, and the value of a macro variable the data content of the named location. Assigning a value to a macro variable populates its location[4] with a value, and the up-to-the-millisecond record of names and values of all allocated locations is called the symbol table.

With Release 6.09E, 6.10 and higher of the SAS System, you can use a special form of %PUT statement directly to write the content of the symbol table to the SAS log. This is useful both for debugging and for self-education. We'll do this once below, but ***as an exercise:*** Code the statement {%PUT _ALL_;} at the head of any of your SAS programs (can't hurt the program), and, if the program contains macros and macro variables — use the Wee Test macro WHATHAP if you like — some {%PUT _USER_;}s here and there both outside and inside macro definitions (also can't hurt). Give it a run, and look at the log.

Assignment of macro variable values

Look back at the Wee Test, and consider the statement

```
%LET MOTHER=ma;
```

This statement occurs in open code — that is to say, outside a macro definition — and the macro variable MOTHER is created and given the value {ma} then and there relative to the remainder of the source code. Having been created, its value will remain constant unless and until acted upon by an outside force: another opencode %LET statement, or a dynamic assignment.

Dynamic assignment means that the value of a macro variable is determined not by face-evident source code. One form results when the preprocessor gets involved, as when we code

```
%LET GMOTHER=grand&mother;
```

after first assigning MOTHER=ma, or when the %LET statement contains macro functions or invocations, or when it is coded inside a macro definition. Dynamic assignment can be othewise obtained. The SYMPUT call function, coded in a DATA step program, will cause a macro variable to be (created and) populated when it executes at runtime. Macro variables can also be assigned in SCL programs, again through the SYMPUT function call.

Automatic variables

Some of macro variables are defined not with %LET statements or other directions from the user program, but are offered up for use by the SAS System itself, and almost all of their names begin with "SYS", with which you should not begin any of your own macro variable names. We'll use a few of them later below; you can see 'em all by coding {%PUT _ALL_;} or {%PUT _AUTOMATIC_;}.

MACRO LANGUAGE I :
EXPRESSIONS; FUNCTIONS; %IF CONDITIONS

The %LET statement, used to assign text to macro variables, is structured like any assignment statement: A variable name, the assignment operator (=), and an expression. And an expression in Macro Language may consist, like any expression, of variables, constants, operators, and functions.

The variables are, of course, macro variables. The macro constants are in all cases character strings. These may be treated as numerics for arithmetic evaluation, where allowed (see the %EVAL and %SYSEVALF functions, below).

Macro Language operators include especially the equality and inequality comparisons as well as the logicals (AND, OR,

[4] If the name of the variable upon assignment was not earlier used in the program, the location is first allocated, associated with the name, and then populated.

NOT), and in some circumstances the arithmetics (again, see %EVAL). The syntax used is the same as those used in the DATA step and other SAS sublanguages. No concatenation operator is needed.

Of particular interest here are the functions, which allow you to perform tasks not readily accomplished by other operations. Macro Language functions are *not* the same as those of the DATA step, and their names, like statement names, begin with '%'. We will examine some functions in this presentation.

String Quoting

Perhaps the first Macro Language function anybody uses is the **%STR** function. This is used to quote a character string so not to get special characters confused. Consider the simple SAS step

```
PROC PRINT DATA=A; RUN;
```

and suppose you want to assign this as a string to a macro variable WHATPROC. You cannot write

```
%LET WHATPROC=PROC PRINT DATA=A;RUN;;
```

for there are embedded semicolons. If you write that, then &WHATPROC will be {PROC PRINT DATA=A}, and a RUN and a null[5] statement are passed through to open code.

We therefore must *quote* the string. Macro quoting means to mask special characters such that the macro processor does not care about them. Thus:

```
%LET WHATPROC=%STR(PROC PRINT DATA=A;RUN;);
```

Seeing this, the preprocessor stops caring about anything that's in the function argument (ie within the parentheses following its name), and just stores the argument as constant text.

The argument string may contain macro "trigger" tokens (& or %), in which case the Macro Language is resolved prior to the assignment, thus the job:

```
%LET MOTOR=HARLEY;
%LET CYCLE= %STR(PUT &MOTOR DAVIDSON;);
DATA _NULL_; &CYCLE   RUN;
```

results in {HARLEY DAVIDSON} written to the log. Should you wish to use the '&' or '%' characters within a constant string, you quote it with the **%NRSTR** function.

```
%LET STORE=%NRSTR(A&P);
```

stores the value {A&P}, as is literally. The macro processor does not attempt to find a macro variable, P, to resolve.

%IF ... %THEN and Comparison

Available within the confines of a macro definition are the %IF...%THEN and %ELSE statements, as well as %DO...%END groups of various forms. These are used very much like the analogous statements in DATA Step Language, and as such, make use of logical equality comparisons, thus

```
%IF &STORE=%STR(A&P) %THEN
   %LET COAST=EAST;
%ELSE LET COAST=OTHER;
```

assigns the value {EAST} to the macro variable coast if the value of the macro variable STORE is {A&P}, and assigns the value {OTHER} otherwise.

Not only the macro assignment statement, but other statements can be executed conditionally, as can macro text, thus

```
%IF &P=YES %THEN %STR(PROC PRINT;);
```

is perfectly fine, because %STR(...), though not a statement, resolves to text. A group of statements and text can be conditioned, thus with a %DO group:

```
%IF &SEX=M %THEN %DO;
   macro statements and text ...
%END;
```

Only if the macro variable SEX has the value {M} will the body of the definition between %DO and %END be expressed for further processing. For the value returned by a comparison operation is 1 (true) or 0 (false), as also in DATA Step Language or SCL. Equality and inequality comparisons take place between the resolved value of an expression on the left of the operator with the value of the one on the right. Thus if we had written

```
%IF P=YES %THEN %STR(PROC PRINT;);
```

we'd never get a PROC PRINT, because {P} never equals {YES}, though the resolution of {&P} might.

Arithmetic evaluation

Observe that the comparison {&P=YES} has

[5] The null statement consists of a lone semicolon.

a character string {YES} on the right. Unlike the DATA Step Language, the string need not be enquoted (in fact if it is, it's a different string). There are also numeric comparisons in Macro Language, but the rules about when a string will be interpreted as a number differ from the DATA Step Language.

In the Macro Language, expression {1.0=1.0} is true, but {1=1.0} is false, because the latter two are string and not numeric comparisons, and the character string "1" is not the same as "1.0"; {1<1.0} is instead true. Importantly, the expression {1+1} is also evaluated as a string. But you can force an arithmetic evaluation of an expression containing integers by use of the **%EVAL** function, thus:

```
%LET SUM=1+1;        * SUM is "1+1";
```

but

```
%LET SUM=%EVAL(1+1);     * SUM is "2";
```

An implicit %EVAL is performed in the macro %IF statement, thus:

```
%IF 2=1+1 %THEN ...; * expression is true;
```

Beginning with SAS release 6.11, the **%SYSEVALF** function is available, which allows real arithmetic, that is on non-integral numbers. Thus

```
%SYSEVALF(1.5+2.5)
```

equals 4. You can convert the result of an evaluation to an integer, or to the highest (ceil) or lowest (floor) closest integer, by passing a second argument to the function, thus

```
%SYSEVALF(1.5+2.6 , INTEGER)
```

is also 4, but

```
%SYSEVALF(1.5+2.6 , CEIL)
```

is 5.

String Functions

The Macro Language provides several functions that allow you to inspect and manipulate strings. Each of these has a DATA Step Language analog, though not all things you can do in the DATA step can you do in Macro Language. What you can do is return a substring ...

```
%SUBSTR(YES,1,1)          is Y
```

determine argument length ...

```
%LENGTH(THISARG)          is 7
```

convert case ...

```
%UPCASE(Lower)            is LOWER
```

find a character ...

```
%INDEX(HELLO,E)           is 2
```

and find a word ...

```
%SCAN(THIS MACRO,2)       is MACRO
```

These illustrations notwithstanding, of course you would normally not use text constants in practice, but rather would examine the results of a macro expression. For example, you may have user input about something, input that could have been written "YES" or "Y", or "NO" or "N". Assign this to variable ANSWER, then

```
%IF  %SUBSTR(&ANSWER,1,1)=Y %THEN . . .;
```

Executing SAS System Functions

With release 6.11 and higher of the SAS System, the majority of system functions that hitherto were available only to the executing DATA step or SCL program are now available to the preprocessor. Both internal functions and CALL routines are available. In the Macro Language, you access internal SAS functions with **%SYSFUNC**, and CALL routines with **%SYSCALL**.

%SYSFUNC is used in function format, and its single argument is itself a function, a SAS function as it would be written in DATA Step Language, except that quoted-string function arguments are not quoted in Macro Language. As in the DATA step, the result of %SYSFUNC preprocessing is some value, which may then be assigned, compared, etc.

Example 1: %SYSFUNC

In a system I developed for Kaiser Permanente, an enduser might choose a month of warehoused data for further analysis by the system. She specifies the month in the form *yyyymm*, eg to get February 1998 the user would select 199803. The system stores this in a macro variable imaginatively named M.

The problem is to convert these into SAS dates representing the first and the last days in the month, which will be used to establish a date range. Here is how it was done:

```
%LET _SDT=%SYSFUNC(INPUTN(&M.01,YYMMDD8.));
%LET _EDT=
   %EVAL(%SYSFUNC(INTNX(MONTH,&S_DT,1))-1);
```

Note that the constant {MONTH} passed to the

INTNX function[6] is not placed in quotemarks when the function is used by %SYSFUNC.

Like the corresponding CALL statement in DATA Step Language, %SYSCALL is a Macro Language statement, not a function. The %SYSCALL statement otherwise takes the same form that the corresponding CALL statement would take, which depends on the routine in question.

VARIABLE RESOLUTION

Look back at Figure 1 and you will see that the macro processor does not throw anything directly forward to SAS tasks, but back through the wordscanner. The jargon for this activity is *rescanning* — taking resolved macro invocations or variables and seeing if there's any more macro trigger tokens "&" or "%".[7]

With rescanning, the resolved values of a macro or macro variable might be further resolved, because if the first resolution did contain macro triggers the preprocessor will get involved again. Consider the code segment

```
%LET NAME=JEFF;
%MACRO WHO;
   %PUT &NAME;
%MEND;  %WHO
```

that of course results in JEFF, not "&NAME", being written to the log. Internally, the preprocessor first resolves the macro and comes up with {%PUT &NAME;}. Then it proceeds to resolve &NAME to JEFF, and only then to use the %PUT statement to write to the log.

Rescanning can happen an arbitrary number of times, stopping when there are no more macro triggers in the resolved value.

Indirect Reference

Add to the facts of rescanning the fact that when two ampersands are placed together, the first rescan resolves them to one; further interpretation is left the next rescan. Then add the fact that scanning proceeds from left to right, and macro language is resolved when it is encountered. Thus we will find that

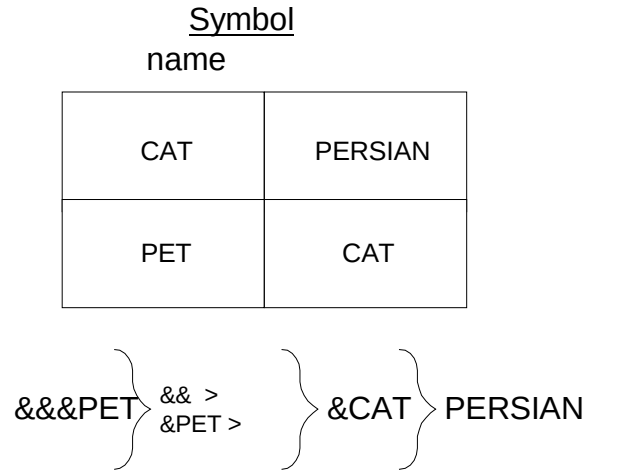
```
%LET CAT=PERSIAN;
%LET PET=CAT;
%PUT MY &PET IS A &&&PET;
```

results in

```
MY CAT IS A PERSIAN
```

being written to the SAS log. The operation is illustrated in Figure 2: On the first scan, &PET is resolved to CAT, && to &, and &PET again to CAT. Then the resulting macro variable reference from the first rescan — ie &CAT — is resolved to yield the value PERSIAN.

Figure 2: Indirect Reference via Rescanning



I call this indirect reference: the desired result is stored in a macro variable whose name is in turn stored in another macro variable, which name must first be retrieved before the desired result can be retrieved. We'll see an application of this below.

If you follow this right along, it becomes simple to predict the results[8] of other numbers of ampersands:

```
&&PET  >  &PET  >  CAT
&&&&PET >  & &PET > &PET > CAT
&&&&&PET > && &PET > &CAT > PERSIAN
```

Delayed Reference

Suppose we have three macro variables MV1, MV2, and MV3, containing the values ABLE,

[6] What, you don't know INTNX? You better get publication TR-P222, *Changes and Enhancements to the SAS System, Release 6.07*. Really, you want this book.

[7] You can prevent rescanning using certain quoting functions, including the %NRSTR function, that do pass the results through to other SAS tasks, bypassing the macro facility. What's in a name? NRSTR = No Rescan STR.

[8] or you can cheat by turning the SAS system option SYMBOLGEN on, submitting a bunch of %PUT statements, and inspecting the SAS log.

BAKER, and CHARLIE respectively. Suppose further we need to know the value of one of these variables, but we don't know which variable unless we know the value of another variable we'll call WHICH, which may be 1, 2, or 3; assume {%LET WHICH=2;} for now.

OK, now we have another kind of indirect reference, or so we think when we write

```
%PUT The codeword is &MV&WHICH;
```

But this will fail, because MV is not one of our macro variables. MV2 is, but not MV, and on first scan the preprocessor can resolve WHICH to 2, but not MV. As you may know, if a symbol cannot be resolved, the literal value including its ampersands or percentsigns will be passed through to the SAS task, with a notation in the SAS log to the effect that the symbol cannot be resolved. Thus the line of code above generates such a log complaint about alleged variable MV, and the %PUT statement results in {The codeword is &MV2}.

Aha, we think, we should write

```
%PUT The codeword is &&&MV&WHICH;
```

to capture the indirectness of the reference. This, indeed, will succeed in yielding {The codeword is BAKER}, but the reason it succeeds is because on first scan the first two ampersands were reduced to one, yielding

```
%PUT The codeword is &&MV2;
```

which could be further resolved to

```
%PUT The codeword is &MV2;
```

and finally to

```
%PUT The codeword is BAKER;
```

. Here the same result would be achieved by coding only two ampersands in the first place; the third was actually superfluous. The form {&&macvarA&macvarB} is common in delayed reference.

Example 2: Delayed Reference

In later Version 6 releases and in Version 7, you can access the results of the most recent LIBNAME or FILENAME statement via the automatic macro variables SYSLIBRC and SYSFILRC; if nonzero, then the allocation did fail. The following macro can be used to abort the job if a dynamic allocation fails:

```
%MACRO _CHK(T,CODE); %* T: LIB OR FIL;
  %IF &&SYS&T.RC %THEN %DO;
    DATA _NULL_; ABORT RETURN &CODE; RUN;
  %END;
%MEND;
```

If we execute a LIBNAME statement, and then

```
%_CHK(LIB)
```

the preprocessor evaluates {&&SYS&T.RC} as follows:

Pass 1: & SYS LIB RC

Pass 2: the value of the &SYSLIBRC variable

which, if nonzero, causes the job to abort and return the LIBNAME's return code to the operating system.

Note the period (.) within the original expression. The period serves as an explicit in-code delimiter for macro variable names. It is used in cases where the end of the name would not otherwise be known, that is, when the end of the name is immediately followed by a letter, numeral, or underscore and not by a blank or a special character. In this case, omitting the "." as in {&&SYS&TRC} would cause the preprocessor to complain on first pass that there is no such macro variable "TRC" in the symbol table.

MACRO LANGUAGE II :

FLOW OF CONTROL; ITERATION

The DATA Step Language provides mechanisms for directing flow of control, that is, the sequencing of executable statments other than in straightforward code order. For this purpose, besides the IF ... THEN ... [ELSE ...] construction and simple DO groups, there is the GOTO statement and forms of iterative DO group.

The Macro Language provides similar statements. The %GOTO statement transfers control to a labeled portion of the macro, which like a DATA step label must begin at a statement boundary (ie, the first non-whitespace immediately after a semicolon) and is suffixed with a colon to identify it as a label. The macro label, however, also has a percentsign prefix:

```
%GOTO THERE;   . . .  %THERE:  . . .
```

The Macro Language also provides analogs to the DO *variable* = *start* TO *end*, DO WHILE *expression*, and DO UNTIL *expression* groups. The iterative %DO group can be used when there is a need to preprocess a set of statements an arbitrary but defined number of times.

Example 3: Using Iterative %DO Groups

In the background of the enduser information delivery system introduced in Example 1, the user can select more than one month of data by specifying two *yyyymm* selections. Not by coincidence, the data mart monthly snapshots are named as follows:

exact.same.prefix.Myyyymm

in order to provide some interesting possibilities.

First, let's construct a macro to dynamically allocate all the monthly datasets that we like. Assume that the global variable _SDT has been created by processes similar to those illustrated in Example 1, and that another variable _MTHS has been created to contain a the number of months of data desired. Now we write the _LIBS macro:

```
1 %MACRO _LIBS(LPFX=ML,CLEAR=N,
      DISP=SHR, DPFX=exact.same.prefix.M);
2  %DO I=1 %TO &_MTHS;
3  %LET D=
 %EVAL(%SYSFUNC(INTNX(MONTH,&_SDT,&I))-1);
4  %LET Y=%SYSFUNC(PUTN(&D,YEAR4.));
5  %LET M=%SYSFUNC
    (PUTN(%SYSFUNC(PUTN(&D,MONTH2.)),Z2.));
6 %IF &CLEAR=Y %THEN LIBNAME
    &LPFX&I CLEAR %STR(;);
7 %ELSE LIBNAME &LPFX&I
      "&DPFX&Y&M" DISP=&DISP %STR(;);
8 %END;
9 %MEND;
```

Line 2 begins (and line 8 ends) an iterative %DO group. The iteration variable is a macro variable named I. This begins with a value of 1 and goes sequentially to the value of _MTHS where we have stored the integer equal to the number of months in the desired time range. At line 3, we obtain a date within the current month -- the last day of the month is fine -- by using the INTNX function with the MONTH option on the start date &_SDT, with the number of months ahead given by the iteration variable, yielding the SAS date value representing the first day of that month, subtracting 1 from which yields the last day of the previous month. Now at lines 4 and 5 we reconstruct *yyyy* and *mm*, by using the YEAR and MONTH functions; at line 5, we also use the PUTN function to format the month with leading zeroes. Used at line 7, "&Y&M" then gives the *yyyymm* representation used in this system.

Assuming we don't override the libref prefix defined on the %MACRO statement, the libref &LPFX&I resolves on the first %DO pass to ML1, next pass to ML2, and so on, and if we let (for example) _SDT contain a SAS date value in March 1998, and _MTHS contain 3, invoking the macro with %_LIBS( ) yields:

```
   LIBNAME ML1 exact.same.prefix.M199803;
   LIBNAME ML2 exact.same.prefix.M199804;
```

While we're on a roll, let's see how we may use these librefs in concatenating the several monthly datasets:

```
%MACRO _SET(SASDATA,LPFX=ML,KEEP=,DROP=);
 %GLOBAL _MTHS;
 %LOCAL I;
 %DO I=1 %TO &_MTHS;
   &LPFX&I..&SASDATA
     %IF X&KEEP NE X %THEN (KEEP=&KEEP);
     %ELSE %IF X&DROP NE X
        %THEN (DROP=&DROP);
 %END; %MEND;
```

If after defining and invoking _LIBS as shown, and defining _SET as shown, we now write

```
   DATA EXAMPLE;
      SET %_SET(MDATA,KEEP=NAME AGE SEX);
   RUN;
```

what will be seen after macro processing is all done is

```
   DATA EXAMPLE;
      SET
        ML1.MDATA(KEEP=NAME AGE SEX)
        ML2.MDATA(KEEP=NAME AGE SEX)
        ML3.MDATA(KEEP=NAME AGE SEX)
       ;
   RUN;
```

Some fun!

DATA STEP INVOLVEMENT

Figure 1 didn't tell the whole story, exactly, of how macro processing intersects with other SAS system tasks, in particular SAS job steps.

As you may know, the SAS System typically does its work in a sequence of compile-execute, compile-execute cycles. A DATA or PROC step begins with the DATA or PROC statement, and ends (properly) with the RUN (for some PROCs, QUIT) statement, or if RUN is not coded (bad form!) when the next DATA or PROC step is encountered.

The part of the story not told by Figure 1 is when in the first place tokens get transferred to the word processor. This is done during the compile phases of SAS processing, and is suspended if a step is actually executing.

The SYMPUT routine

Armed with this knowledge, we are in a position to properly understand DATA step / preprocessor intercommunication. We begin with the SYMPUT call routine, which gives us a mechanism whereby data known only to the executing DATA step can be placed into the macro variable symbol table for use later in the program. The form of the call is

CALL SYMPUT(*expression1*,*expression2*);

where *expression2* can be any SAS expression — which itself could be the result of preprocessing, I might add — while *expression1* must resolve to a valid macro variable name. In practice, *expression1* is often a character literal that explicitly names the macro variable, as in the following example.

Example 4: Using SYMPUT

We desire to set the value of a macro variable named ZERO_ROW to Y if there are no rows in a given dataset, and N if there are. After invoking the macro, the variable ZERO_ROW may then be used later in the program for various purposes, for example to determine whether other job steps should be run.

```
%MACRO ZEROROW(INDATA);
  %GLOBAL ZERO_ROW;
PROC CONTENTS DATA = &INDATA
     NOPRINT  OUT = ZEROROW;
RUN;
DATA _NULL_;
  SET ZEROROW;
  IF _N_ = 1 THEN DO;
    IF NOBS = 0 THEN DO;
      CALL SYMPUT('ZERO_ROW','Y');
    END;
    ELSE DO;
      CALL SYMPUT('ZERO_ROW','N');
    END;
    STOP;
  END;
RUN;
%MEND ZEROROW;
```

Note that if the variable ZERO_ROW does not exist in the global symbol table[9], invocation of %ZEROROW will cause its creation.

With regard to Example 4, please note that ZERO_ROW is not populated with a macro variable until the DATA _NULL_ step runs. If the RUN statement were omitted and the programmer tried to use &ZERO_ROW immediately after invoking the macro, the intent would fail.[10]

The SYMGET function

As the SYMPUT routine can place values in the macro variable symbol table, so the SYMGET function can retrieve them. In practice, the SYMGET routine comes into its own when the value of the variable of interest is set not by some method foreseen by the program, but by some external process, eg in a SAS session affected by an SCL program.

Example 5: Symbols Back and Forth

In the context of another system operative at Kaiser Permanente, a certain selection of items is to be made available if the enduser has access to confidential psychiatric data, but not otherwise. My colleague Berwick, faced with this problem, wrote something similar to the following, based on expecting a value Y or N for the macro variable _PSYC:

```
IF SYMGET('_PSYC') = 'N'
   THEN CALL SYMPUT('GETRSRTP',
     "('AF','AG','BP','CI','CJ','AR')");
ELSE CALL SYMPUT('GETRSRTP', ' ') ;
```

Used later in the program, &GETRSRTP will return a parenthesized set of comma-separated string literals as shown, but only if the value of &_PSYC was N when this executed. Note that the 'N' is quoted in the comparison: This is a DATA step, remember, not macro language.

The RESOLVE function

RESOLVE provides a mechanism for returning the result of Macro Language resolution within the context of a DATA step.

Both macros and macro variables can be RESOLVEd; you can only SYMGET macro variables. But also, if the resolved value of a macro variable contains other macro triggers, SYMGET will not attempt further resolution, whereas RESOLVE will. For SYMGET simply plunks in a value from the macro symbol table,

[9] I have avoided discussing the global versus local symbol tables. The topic of macro variable reference environments, will unfortunately not fit within this modest presentation. Consult my textbook **Mastering the SAS System** (as well as the SAS macro language manual) for this and other esoterica, but wait to buy the fully-updated third edition of my book, coming Spring 1999 from SAS Institute.

[10] There are special cases where you may want to leave a DATA step "unfinished" at the end of a macro definition, but this involves macro reference environments, which we're not getting into.

while RESOLVE actually involves preprocessor functionality. Thus macro invocations can be resolved, as can any macro language expression acceptable in open code, and the rescanning feature is available.

The EXECUTE routine

There are times when you would like the executing DATA step to control the very execution of macros or, indeed, of other SAS code. A powerful feature of the SAS System, which was available under Version 5, went away for awhile (along with RESOLVE) under Version 6, and came back again in later Version 6 releases and Version 7, is CALL EXECUTE.

A CALL EXECUTE statement takes as its argument an expression that, when resolved, is submitted to the wordscanner *after* the DATA step has run, just as if the argument to CALL EXECUTE had in fact been placed in the source code directly after the DATA step. Using this routine helps avoid cumbersome write-a-sourcefile-then-%INCLUDE-it constructions.

Example 6: The Executioner

Let's return to the monthly data snapshots created for an enduser information system. Due to a hindsight-induced change in specification, it was necessary to modify the monthly extracts for calendar year 1997 and the first half of 1998.

The code for the modification was placed in a macro named FIX (for that's what it did), which took a single positional parameter; as you'll see, you do *not* have to know what that parameter was named to appreciate this example, but you do have to know it is expected to have a value in our famous *yyyymm* form.

Having coded the macro, here's how I applied it to each month in 1997 and the first half of 1998:

```
DATA _NULL_;
DO YEAR='1997'; DO MONTH=1 TO 12;
  CALL EXECUTE
 ('%FIX('||YEAR||PUT(MONTH,Z2.)||')');
END; END;
DO YEAR='1998'; DO MONTH=1 TO 6;
CALL EXECUTE
 ('%FIX('||YEAR||PUT(MONTH,Z2.)||')');
END; END;
RUN;
```

Observe the structure of the CALL EXECUTE statement. The expression is broken down as follows:

a hardcoded literal {%FIX(}

concatenated with the value of the variable YEAR

concatenated with the month number left-zero-filled

concatenated with hardcoded literal {);}

Do you see it? After the execution of this DATA step, the wordscanner is given

%FIX(199701)

%FIX(199702)

• • •

%FIX(199806)

JAY A. JAFFE, PHD

1012 BERMUDA DRIVE

CONCORD, CA 94518

(925) 689-6657

J.J@IX.NETCOM.COM