A SAS® Programmer's View of the of the SAS Supervisor

Ian Whitlock, Westat Inc.

Abstract

This tutorial answers questions like:

- As a DATA step programmer, what do I need to know about the SAS supervisor and why?
- How does the SAS supervisor process DATA step code?
- How does a SAS MERGE work?
- What about engines, indexing, and views?
- What happens when my DATA step code contains macro variables?
- What if my DATA step invokes a macro or is contained in a macro?

For many years Donald Henderson or one of his colleagues gave a tutorial about the SAS Supervisor (SUGI 8, and 12 - 17). This tutorial builds on those earlier articles and adds the experience of the author.

Introduction

A basic SAS program consists of DATA steps and procedure steps. In both cases machine code is executed. The main difference is that the machine code already exists for procedures so that only the parsing of a little control information is needed. The DATA step language provides the flexibility needed to read and manipulate arbitrary data structures in ad hoc ways.

The SAS supervisor manages step processing. We will be mainly interested in how it manages the DATA step, but first we will take a quick look at the consequences of having steps. Finally we will take a quick look at how the macro facility fits in.

Why should one be interested in how the supervisor works? Such knowledge gives you better control over the language, i.e. the ability to make a program do exactly what you want because you have a better idea about what is happening and when it is happening.

Step Processing

At a very early point in the development of the SAS System, a fundamental decision was made to compile and execute each step before the next is considered. This shrewd decision has significant consequences for the SAS programmer:

1. Each step is an executable image optimized for fast execution.
2. The executable image is by default thrown away; hence the source code is faithful to the action.
3. Information is basically passed from one step to another via files; hence SAS is primarily an I/O bound language. This means the programmer should, consistent with clarity, minimize the number of steps in the program design.
4. Almost by definition, SAS is very modular, amenable to structured programming techniques and data flow diagrams.
5. Since costly early steps may execute only to have the program fail on a mistake in a later step, the programmer must bear the responsibility for adequate testing.
6. Since later steps need not even exist when earlier steps are executed, the earlier steps can determine what program is needed and then write it. Consequently even batch programs can have an interactive flavor to them.

A common request on the software ballot is to provide true syntax checking. Although SI has said they will provide it, I don't see how they can without changing the nature of the language. Consider the step:

```
data w ;
  retain x 'abc' ;
  set data1 ;
run ;
```

This code is syntactically correct if DATA1 does not contain a numeric variable X, but it is incorrect otherwise. Usually one would expect a syntax checker to catch errors in variable type. Now suppose that the step creating DATA1 was included from code written by a previous step. How can SAS possibly have the needed information without executing far enough to create the needed code? This is one example where the power of the SAS system design makes it hard to implement what looks like a standard requirement for programming languages.

The Basic DATA Step Loop

Each DATA step requiring I/O (really we mean input) is built with an implied loop. Hence the common structure for a DATA step is initialization, read a record, do something with the data, and write out the transformed data. The automatic variable _N_ counts iterations of this implied loop. It is always available in a DATA step. Typically _N_ is used in code where variables must be initialized. For example:

```
data _null_ ;
  if _n_= 1 then set stats;
  set sales ;
```

```
  ....
run ;
```

Perhaps the first question to ask is, "What stops the looping process and where does it stop?" By default the process stops when an input request cannot be satisfied; hence a DATA step typically stops with a SET, MERGE, UPDATE, MODIFY or INPUT statement. Not on the last record, but the next time around the loop, when the statement is to execute and it cannot. For the programmer this means the best place to test for end-of-file is not after the I/O statement, but before it! When the test for end-of-file is placed after the reading statement, there is a danger that some subsetting code will cause the test to never be executed. The following code works even when the REGION on the last record is not "WEST".

```
data _null_ ;
  if eof then put total= ;
  set sales end = eof ;
  if region = 'WEST' ;
  total + sales ;
run ;
```

The STOP and ABORT statements can be used to explicitly halt a DATA step, either because some condition is reached or because the normal default cannot operate. For example, when one uses a SET statement with the POINT option, the end-of-file is never set, consequently one must take explicit action or depend on the step stopping at some other I/O statement. The DO-loop below would execute over and over until it runs out of space or time without the crucial STOP statement. Since input is done in every iteration of the implied loop, there is no reason to stop the implied loop.

```
data sample ( drop = i ) ;
 do i = 1 to 100 ;
   ptr = int(ranuni(0)*nobs+1);
   set universe point = ptr
       nobs = nobs ;
 end ;
 stop ;
run ;
```

On the other hand no STOP is needed below, since the DATA step will stop when it runs out of records from LOOKUP.

```
data found ;
  set lookup ;
  /* ptr is variable in lookup */
  set answers point = ptr ;
run ;
```

What if there is no input? Then processing simply stops at the bottom of the step as if there were no implied loop. You might try to trick the compiler with

```
data ;
  if 0 then set dataset ;
run ;
```

The looping operation is set up because of the SET statement. Since no input statement is executed, it cannot run out of data, but the step will stop at the bottom of the first iteration of the loop with the message

NOTE: DATA STEP stopped due to looping.

DROP, KEEP, LABEL, and RENAME

Now let's look more carefully at the compile process and in particular at compile time directives. These are non-executable statements that tell the compiler how to set up something. DROP, KEEP, LABEL, and RENAME all give instructions about the output SAS data sets. They do not give information about the variables during execution of the DATA step. They may occur anywhere in the step without changing anything that is done. The drop and keep variables may be repeated. Why is this important? A macro to generate DATA step statements may need to create and drop variables. The macro may be invoked several times because the repetition of the drop statement doesn't matter.

If a variable is both kept and dropped, the drop takes precedence, and a warning is issued. If the same variable is renamed or labeled more than once, the last one rules. When a rename output data set option is used with a RENAME statement for the same variable, the RENAME statement is done first and the variable is no longer available for renaming. You can test your understanding with the following code.

```
/* 1 */ data w (rename=(z=q));
/* 2 */   x = 1 ;
/* 3 */   y = 'a' ;
/* 4 */   rename x = u ;
/* 5 */   rename x = y ;
/* 6 */   rename y = z ;
/* 7 */ run ;
/* 8 */ proc print data = w ;
/* 9 */ run ;
```

The resulting values are Y = 1 and Q = 'a'. X is renamed to Y since line 5 comes after line 4. Y is renamed first to Z by line 6 and then later to Q by the output data set option in line 1. Note that the compiler applies the output data set option RENAME last even though it occurs first in the physical order of the code. Any KEEP and DROP options refer to the name before use of the RENAME option, but after application of RENAME statements.

Note that lines 5 and 6 imply that

```
rename x = y y = x ;
```

should interchange the two variable names. It does in a DATA step, but it results in an error message in PROC DATASETS saying that Y and X already exist.

How might one make use of this information? A common question on SAS-L is "I have character variable X on a SAS data set but I want it to be numeric with the same name. How can this be accomplished?" Assume W is a data set with a character variable X which has numeral values. One possibility is

```
data w ( drop = temp ) ;
   set w ;
   temp = input (x, best12.) ;
   rename x = temp ;
```

```
   rename temp = x ;
run ;
```

Usually DATA step code, using pure data set options, is shorter, simpler and clearer, since the moment of action is spelled out.

```
data w ( drop = temp ) ;
   set w (rename=( x=temp )) ;
   x = input ( temp, best12. );
run ;
```

The Order of Variables

The first explicit occurrence of a variable is important, because with the exceptions of DROP, KEEP, RENAME, LABEL, and possibly RETAIN, this occurrence will determine whether the variable is character or numeric. When is the length of a variable determined? Character variables are treated differently from numeric because numeric variables are always manipulated as 8 byte floating point variables during execution of the DATA step.

Since numeric lengths are not applicable to the DATA step processing (only the output data set), numeric lengths are determined by the last LENGTH or ATTRIB statement to assign a length. An ARRAY statement can also assign numeric lengths, but only when there is no relevant LENGTH or ATTRIB statement overruling it.

On the other hand, character data lengths are determined by the first statement that allows specification of a length. In particular assignments, formats, and informats all determine character lengths in addition to the explicit length specifiers LENGTH and ATTRIB. For character variables the length specified in an ARRAY statement takes precedent when it comes first. Any attempt to change a character variables length results in a warning.

Even FORMAT statements can determine the length of character variables when they occur first. For example,

```
format c $myfmt. ;
```

will determine that C has the length of the longest label in $MYFMT. This does not make too much sense; hence it is best to avoid having a FORMAT statement announce that a variable will be character.

When a variable is known to be character and the length cannot be determined, a default is chosen. Hence, the length of character data is always determined by the first statement determining that the variable is character. A default length 8 or 200 is usually chosen when no better information is available. For example, list input causes character variables of length of 8 and

```
c = symget ( 'macvar' ) ;
```

determines a length of 200. Unfortunately one cannot depend on reason because the INFILE option FILENAME= defaults to a length of 8. Some functions determine other lengths, but not necessarily in a consistent fashion. For example

```
c = substr ( char , 1 , 1 ) ;
```

will cause C to have the same length as CHAR, but

```
c = put ( char , $agecat1. ) ;
```

will cause C to have a length of 1. The difference is explained by the fact that the second argument of the PUT function is code and known to determine a length of 1 at compile time. Had the "1" not been present, then the length would be determined by the longest label of the format, consistent with the format determination of length given above..

On the other hand, the third argument of SUBSTR is a DATA step value, in principle could be any length up to the length of CHAR at execution time. Thus the third argument of SUBSTR cannot control the length even when a constant.

One might expect that

```
call vname ( var , name ) ;
```

will determine NAME as a character variable, but that is not true. It will be numeric by default. Probably function and subroutine argument types never determine the type of the corresponding variable.

In review, it is impossible to determine that a variable is character without at the same time knowing its memory size requirements. This means that the compiler knows how to arrange storage for a variable as soon as it knows whether the variable is numeric or character. Moreover, this determination can be made by the first mention of a variable (with the exception of DROP, KEEP, RENAME, LABEL and sometimes RETAIN).

As soon as a variable is determined to be character or numeric, it can be assigned a position in the logical Program Data Vector (PDV). This order is important because it determines the order in which variables are stored on output data sets. It is also important within the DATA step when using the notation *"firstvar -- lastvar"* to avoid writing a long list of variable names. In particular, this is the default order that many procedures use.

The other compile time directives ARRAY, ATTRIB, FORMAT, INFORMAT, LENGTH, and sometimes RETAIN all allow one to distinguish numeric from character, hence they all can place a variable in the logical PDV. Their position in the code may not be important for their intended purpose, but it can be crucial in determining the logical PDV. (Prior to Version 6, the PDV was actually a contiguous area in memory where the variables were stored during DATA step manipulation. It still plays an important role as a logical tool for organizing what happens during DATA step processing.) All references below to the PDV are really the logical PDV.

RETAIN is a peculiar exception in that it may not provide the ability to determine the data type, but it can still determine the position in the PDV. For example,

```
retain var /* no value */ ;
```

does not determine whether VAR is numeric or character. But when it appears before other statements referencing the variable VAR, (other than the standard exceptions DROP, KEEP, RENAME, and LABEL), it does determine the position of VAR in the PDV, whenever VAR has any position in the PDV. If VAR does not appear anywhere else in a statement in the step, other than the standard exceptions, then VAR will not be in the PDV, and hence not on any output data sets. These facts can be demonstrated with the code:

```
data w ;
  retain x ;
  y = 1 ;
  put _all_ ;
run ;
```

Now X is not written to the SAS log and it does not appear in the contents of the data set W. On the other hand, if VAR does appear elsewhere in the DATA step, then the first occurrence determines the variable type, but not its position in the PDV. This can be seen by assigning X a value (character or numeric) just before the PUT statement in the above DATA step code.

Another common question on SAS-L is how can one reorder the variables of a SAS data set. From the knowledge above you can see the simplest answer is

```
data out_data_set ;
  retain order_you_want ;
  set input_data_set ;
run ;
```

Although SET, MERGE, UPDATE, and MODIFY statements may not explicitly name any variable they do implicitly name all variables in the associated SAS data sets, which are not ruled out by a DROP= or KEEP= data set option. (Note that this means each SAS data set named in a DATA step must exist and be available to the DATA step compiler.) Thus for a complete determination of the order in the logical PDV one must also consider the placement in the DATA step of the above I/O statements.

Note that DROP= and KEEP= options are preferable to DROP and KEEP statements precisely because they control what goes into the PDV instead of what is transferred to output buffers. A smaller PDV means a faster executing DATA step, and less chance of a variable conflict in the current or later DATA steps.

Finally we must consider the SAS statements that ask for the system creation of variables (usually known as automatic variables). These variables are placed in the logical PDV at the point the DATA step compiler encounters the statement, unless they have been mentioned before in another statement. For example,

```
set dataset end = eof ;
by id ;
```

requests three automatic variables EOF, FIRST.ID, and LAST.ID. Each will be added to the PDV as the compiler reads the corresponding statement. In the above example, EOF will come first, followed by the variables on DATASET, and then FIRST.ID and LAST.ID. The variables _ERROR_ and _N_ are always added at the end of the PDV. You should now be able to read any SAS data step and write the variables in the PDV in the correct order. How can you be sure of the order on the PDV? Add

```
put _all_ ;
```

to your DATA step. (The keyword _ALL_ in this case refers to the PDV at the time of execution after it is completed, not as it exists at this point in compilation. In contrast, RETAIN _ALL_ and ARRAY A (*) _ALL_ refer to the PDV at their point of compilation.) The variables are always written in PDV order at the point that the PUT statement executes. Thus you can also get their values at any point during execution after the first initialization set up by the supervisor. Some of the variables may not be missing because they were initialized during compile time.

An important example is given by

```
data _null_ ;
  call symput ( 'nobs',
        left(put(nobs,best12.))) ;
  stop ;
  set test nobs = nobs ;
run ;
```

It is correct to refer to the variable NOBS in the CALL statement because is was assigned at compile time when the DATA step compiler read the SET statement and looked at the directory of TEST. There is a STOP in front of the SET statement because we do not wish to read any observations from TEST. The purpose of this step is to create a macro variable NOBS holding the number of observations in TEST. (This code will not produce a correct result when observations have been "deleted" from TEST without physically removing them, thus the interest in the code is more theoretical than practical. Today the problem is better solved with PROC SQL.)

Initialization of Variables

Prior to Version 6 the supervisor did the initialization to missing at the top of each iteration of the DATA step, and it was relatively slow because the variables were processed as stored in the PDV. With Version 6 the variables are actually stored in four separate blocks - character versus numeric, need to initialize versus no need. Now the supervisor generates code to initialize a block at a time and it is part of the execution module as is the implied loop. Hence the looping and initialization are very fast and there is little need on efficiency grounds for programmers to avoid either.

We still have not determined which variables are initialized once at the beginning of the DATA step and which are initialized to missing at the beginning of every loop of the step. Automatic variables, variables from SAS data sets, and variables that appear in RETAIN statements are initialized once. Some variables are initialized to non-missing values. For example,

- _N_ is set to 1,
- _ERROR_ is set to 0,

- the END= variable is set to 1 if the file is empty and 0 otherwise,
- the NOBS= variable is set to the number of records in the file when this number is known,
- the LAST. and FIRST. variables created because of the BY statement are set to 1, and 0 otherwise,
- variables which appear in a RETAIN statement assigning values are initialized to their corresponding value, and
- the remaining user variables are initialized to missing.

If a variable is not automatic, does not come from a SAS data set, and does not appear in a RETAIN statement, then it will be initialized to missing at the beginning of each loop of the DATA step. There is one exception the statement

```
RETAIN ;
```

anywhere in the DATA step means that only the first initialization will be done. The RETAIN statement is often confusing because it really means initialize once; hence, values are retained until changed; it does not mean that values are constant. Automatic variables can be automatically changed during a loop of the DATA step. For example,

- _ERROR_ is set to 1 when an execution time error occurs,
- FIRST. variables are set to 1 when the relevant BY-group begins and 0 on the next record, and
- the END= variable is set to 1 when the SET, the last record is executed.

For completeness temporary arrays should be discussed. Consider

```
array a (10000) _temporary_ ;
```

This statement creates 10,000 contiguous numeric storage units in yet another area for variables. They are not part of the logical PDV and they have no names. They are initialized once to missing unless assigned in the ARRAY statement. One can reference the elements only with array notation, which is very fast compared with the ordinary SAS arrays because the storage is contiguous. Since they have no names, they do not go on any output file.

In Version 6, the BY statement is local to the previous SET, MERGE, MODIFY, or UPDATE. This change is important for DATA steps like the following.

```
data totpop ;
  if _n_ = 1 then set totals ;
  merge pop1 pop2
  by state ;
  pctpop = pop / totalpop ;
run ;
```

In Version 5 execution of this code would result in the message that STATE is not on the data set TOTALS. (In a similar manner, WHERE statements are local to the previous input statement in Version 6.

Now let's look at a simple merge in detail using the logical PDV as a tool to understand what is happening at key points in the DATA step.

Consider data sets ONE and TWO:

ONE	
K	A
1	x
2	y
4	z

TWO	
K	B
1	r
3	s
4	t
4	u
4	v

and the code:

```
data merge ;
 before:  put 'before: ' _all_;
   merge one two ;
   by k ;
   if b = 'u' then a = '?' ;
 after:  put 'after: ' _all_ ;
run ;
```

K	A	B	FIRST.K	LAST.K	ERROR	N
.			1	1	0	1
1	x	r	1	1	0	1
1	x	r	1	1	0	2
2	y		1	1	0	2
2	y		1	1	0	3
3		s	1	1	0	3
3		s	1	1	0	4
4	z	t	1	0	0	4
4	z	t	1	0	0	5
4	?	u	0	0	0	5

before
after
before
after
before
after
before
after
before
after

4	?	u	0	0	0	6	before
4	?	v	0	1	0	6	after
4	?	v	0	1	0	7	before

PDV - Time Line for the above DATA step.

produces the PDV time line shown on the previous page. Each row corresponds to the state of the PDV at the time the PUT statement shown on the right is executed. Hence there are two rows for each iteration of the loop except the last one, which ends with the MERGE statement.

Note the missing values corresponding to the variables coming from the data sets in the first row. This is due to the one time initialization of retained variables. Note the 1's for FIRST.K and LAST.K in the first row are because a BY statement is present not because K=1 uniquely in both data sets. Also note that both the SAS data set variables and the automatic variables FIRST.K and LAST.K are retained. For example, the first row with _N_ = 4 still has LAST.K = 1. It is not changed until the SET statement is processed. In contrast _ERROR_ and _N_ are assigned at the beginning of each iteration of the loop. One could easily test this fact by incrementing _N_ and _ERROR_. For example:

```
data _null_ ;
  put 'TOP: _all_ ;
  input ;
  _n_ + 7 ;
  _error + (-1) ;
cards ;
1
2
3
;
```

Are _N_ and _ERROR_ retained? I expect so, but it doesn't matter because these values are assigned at the top of the loop.

Returning to the previous example, note that A='?' in the last two rows. This is not because the A changed values (B is no longer equal to 'u'), but because it stayed the same after it changed. It is important to realize that records from B are read into the PDV only once in each BY-group. The value persists because it is retained, not because it is refreshed. This is a common trap that occurs when one tries to change values in the single record per key data set based on a value in the multiple records per key data set. The solution is to rename A, say to SAVEA and then drop it. Since SAVEA is retained and not changed, one now finds that A returns to the original value after modification.

```
data merge ( drop = savea ) ;
 before:  put 'before: ' _all_;
   merge one ( rename = (a = savea) )
         two ;
   by k ;
   if b = 'u' then a = '?' ;
   else a = savea ;
 after:  put 'after: ' _all_ ;
run ;
```

What happens when both sets ONE and TWO also contain a variable, say X, which is not part of the BY-group? If the value of X is contributed by both sets then the one on the right wins the first time and it depends on which set has multiple records after that. Let's assume as above that ONE contains at most one record per BY-group and TWO may contain multiple records per BY-group. For

```
merge one two ;
by k ;
```

TWO contributes the value for X unless there are no records in TWO matching the BY-group. Now reverse the positions of ONE and TWO.

```
merge two one ;
by k ;
```

Here ONE contributes the value of X on the first record of each BY- group for which it has a match, but TWO contributes the value of X for any remaining records of the BY-group. Since it is very rare that one need actually have a common variable not in BY-list, I would suggest that one should use KEEP= or DROP= options to eliminate the possibility. (If it seems necessary to have a common variable not in the BY-list, then probably UPDATE or MODIFY is more appropriate than MERGE.) When this advice is followed, the code is more stable and clearer, so it is worth the extra effort.

When are the IN= variables set in a merge? They are set every time a record is read from the corresponding file. But one should remember that records are read only once. When a one-to-many merge is performed, the IN= variables for the singleton records is assigned once at the beginning of each BY group and retained. In a many-to-many merge (rather unusual) one often wants to know that two new corresponding records came in to the PDV. This can be done by resetting the IN= variables.

```
data pairs oddballs ;
  many1 = 0 ;
  many2 = 0 ;
  merge many1 ( in = many1 )
        many2 ( in = many2 ) ;
  by partkey ;
  if many1 and many2 then
    output pairs ;
  else
    output oddballs ;
run ;
```

When first encountered, one is often surprised that the user can change an automatic variable. Actually automatic variables are not used by the system; they are

only for the user and may be modified in any way the user chooses. As hinted in a previous example, one can even change _N_, but of course at the top of each implied loop the system will set it equal to the system's counter. The most dramatic example that I know of is

```
data check ;
  set something ;
  If _n_ = 1 then first.x = 5 ;
  put first.x= ;
run ;
```

Note there is no BY statement, and X need be not a variable in SOMETHING. Even without a BY statement FIRST.X is an automatic variable, hence it will be retained and dropped. It was initialized to 0 because there was no BY statement, changed to 5 in the third line, and it remains that way because the variable is retained.

I/O Engines

In Version 6 SAS data is accessed via an I/O engine. The engine is chosen by a part of the SAS supervisor called the I/O engine supervisor. This means that the I/O is no longer handled directly by the SAS supervisor. For you it means that many different structures can be read as if they were SAS data sets, since the appropriate engine knows how to make the data appear in SAS format. For the Institute it means that they can more easily change the underlying structure of SAS data sets.

Indexes

SAS data sets may be indexed by one or a combination of SAS data set variables, by using either the INDEX CREATE statement in PROC DATASETS, or the CREATE INDEX statement in PROC SQL, or the (output) data set option INDEX. A special index file is created giving index values and the locations of each value. The value/location pairs are stored in a B-tree structure that enables the engine to perform a binary search for the associated record. When the observations are to be read as indexed data, the index engine is chosen by the supervisor.

The request for using an index may be explicit as in

```
set mydata key= index / unique;
```

or it may be implicit when a WHERE or BY statement is used. For the implicit cases the choice algorithm is quite complex.

An index is eligible for BY processing when

1. There is no NOTSORTED or DESCENDING in the BY statement.

2. An initial list of variable(s) in the BY statement agrees with the complete list variable(s) in a composite (or regular) index and that index was not created with the NOMISS option.

An index is eligible for WHERE processing when

1. The WHERE statement can be broken into two parts such that the first part consists of conditions of the form

   variable op constant

   or

   constant op variable

   joined by AND's where the set of variables matches an initial list of variables in the index, and, either the two parts can be joined by an AND, or the second part is empty. For example, an index on X could be used with the WHERE statement

   ```
   where x > 1 and y = 2 ;
   ```

   since the second part is joined by an AND. On the other hand, the WHERE statement below could not make use of the index on X.

   ```
   where x > 1 or y =2 ;
   ```

2. The conditions on the variables in the first part of the WHERE do not involve missing values when the index was made with the NOMISS option. For example, the WHERE statement

   ```
   where x < 1
   ```

   could not involve an index including X with the NOMISS option, since missing is a potential value for X that would not be read using the index.

The eligible set of indices is reduced as follows:

- If both BY and WHERE statements are present form the intersection of the two eligible sets. If non-empty this is the eligible set, otherwise take the set corresponding to the BY statement.

- If there is only a BY or a WHERE statement, then the eligible set of indices is the eligible set for that statement.

The index is now chosen as follows:

1. If there is only one eligible index, use it.

2. If a BY statement is present, use the index with the most variables.

3. Otherwise, use the index which will select the smallest subset of data assuming a uniform distribution of values.

The competition between BY and WHERE is interesting. Consider

```
options msglevel=i ;
data w ( index = ( x y z ) ) ;
  do x = 1 to 100 ;
    do y = 1 to 5 ;
      do z = 1 to 3 ;
        r = ranuni ( 0 ) ;
```

```
        output ;
      end ;
    end ;
  end ;
run ;

data w2 ;
  set w ( sortedby = x y z) ;
  by x y z ;
  where x = 2 and y = 1 ;
run ;
```

Here there are competing choices. The data set must be read with BY variables X Y Z. The SAS supervisor knows that an efficient direct sequential read is possible because of the SORTEDBY option. In fact without the WHERE statement this would be the choice of the supervisor. But with the WHERE statement present it chooses the X index because of the more efficient subsetting. Of course it will not be more efficient when most of the file has X = 2, but the choice is based on a uniform distribution of X values. Can the choice hurt other than efficiency? No. If the data set were really sorted by X Y and Z when the index was made then Y and Z will appear in order when using the X index. If the data set was not in order when the index was made then both the index and a sequential read would fail.

You have no explicit way to force the use of a particular index, but you can find out which index is chosen by using the MSGLEVEL option.

```
options msglevel = i ;
```

How can indexing hurt efficiency? First one must pay the price of extra I/O to read the index file, so reading the full file with an index will be slower that reading it sequentially without the index. Secondly I/O is performed on pages which usually contain several records (sometimes many). In sequential reading only one physical I/O is performed for each page. When the file is read using an index, then a new physical I/O must be performed every time the index file indicates that the next record is located on a different page. Hence the same page may be read many times when an index is used. So why should use indexes? Because they can drastically speed up subsetting when the subset is a small portion of the file. As an extreme case look at what happens when all the wanted records reside on one page. Now only one physical I/O is needed beyond those need to locate the page. The system knows that it has found all the required records. On the other hand without the index a sequential search through the entire file is required to do the subsetting.

Views

Another consequence of I/O engines is the ability to construct SAS views. A view is instructions to construct SAS data. This idea started with PROC SQL where it is important to be able to define the data elements once, but provide different views for different uses. Version 6.07 introduced the concept a data view. For example, suppose PERM.SALES has one record per year holding monthly sales amount for each salesman and you want to print out the information in annual terms.

```
data sales / view = sales ;
   set perm.sales ;
   annual = sum(of mon1-mon12);
run ;

Proc print data = sales ;
   var salerep year annual ;
run ;
```

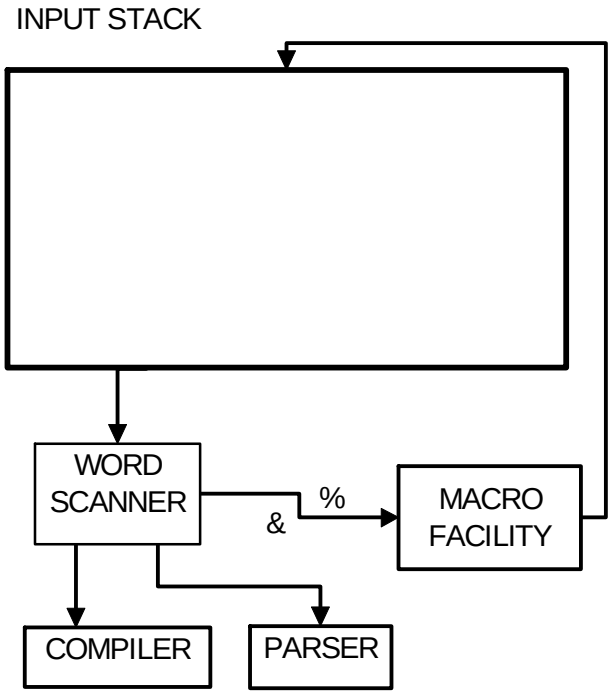
The DATA step does not read PERM.SALES and it does not create a data set SALES. Instead it creates instructions for making SAS data, a user engine. When the PROC PRINT executes the I/O supervisor chooses the SALES view to generate data. During the execution of PROC PRINT PERM.SALES is read and the data created. One very practical use of views is in sorting wide data sets where only a few variables are required. One would think

```
proc sort
  data = wide ( keep= varlist )
  out = narrow ;
   by id ;
run ;
```

would be sufficient. But due to a design bug in PROC SORT, the KEEP option is active only after the data has been sorted. Hence one can often save time, space, and money by creating a view to feed the right set of variables to PROC SORT.

The Macro Facility

How does the macro facility fit into the SAS supervisor's job? The SAS supervisor must parse the code from the input stack into tokens and send these tokens to the appropriate subsystem. Let's call the manager of this part of the process the word scanner (WS). The diagram below shows the three subsystems - the DATA step compiler, the procedure parser, and the macro facility.



Without the macro facility the code in tokenized form goes to either the DATA step compiler or procedure parser. To allow macro code, the WS had to learn one new trick - recognize tokens containing a %-sign or &-sign and pass these tokens to the macro facility. In addition it had to learn to take orders from the macro facility some of the time in the same way that it took orders from the DATA step compiler and procedure parser.

Let's look at the creation and use of a macro variable. Assume we are processing data collected state by state and writing programs to work on these data. We might have

```
%let state = TX ;
data &state ;
  set lib.&state ;
  /* more code */
run ;
```

First the WS picks up "%let" from the input stack of code. Since it begins with a %-sign it goes to the macro facility. The %LET-handler now takes over and asks for a token; hence "state" goes to the macro facility even though it has no "%" or "&". The process continues to the semicolon; then the variable name STATE is stored away with its value TX. Since the %LET handler finished its task when it received the semicolon, no one is asking for tokens. WS gets the "data" token on its own. Since the token was not requested by either the DATA step compiler or the procedure parser, it is the free key word "data" indicating the beginning of a DATA step; hence, it goes to the DATA step compiler, which now asks for tokens. The next token, "&state", is not given directly to the DATA step compiler since it begins with an &-sign. "&State" goes to macro facility which looks up STATE and finds TX. This token is then put back in the input stack to be found as the next token by WS. Thus when WS goes to get the next token, it is TX and this is given to the DATA step compiler. After the semicolon the DATA statement is compiled, and the DATA step compiler asks for more tokens. Now "set" goes to the DATA step compiler, and "lib.&state" to the macro facility, where it is resolved to LIB.TX and put back on the input stack. Now "lib.TX" is found and passed to the DATA step compiler. The semicolon ends the SET statement and the DATA step compiler does its thing as described earlier. And so on, until the step boundary (either a RUN statement or an initial keyword "DATA" or "PROC") At the step boundary the SAS supervisor executes the module.

The important thing to realize here is that the DATA step compiler never sees macro code and knows nothing about it. On the other hand the macro facility knows nothing about SAS code or how to compile it. Finally the word scanner had to learn very little more to make the whole process work. It is one of the best examples, that I know of, demonstrating how to break up a complex process to be managed by three separate little managers who know how to interact, but know nothing of the other's business.

How does a macro fit into this process? Using our previous example consider the following code.

```
%macro edits ( state = NJ ) ;
  data edited.&state ;
     set raw.&state ;
     %if &state = TX %then
     %do ;
         /* special TX edits */
     %end ;
     /* more code */
  run ;
%mend  edits ;

%edits ( state = TX )
```

When WS encounters the token "%macro", the token goes to the macro facility, which hands it to the macro compiler. Now the macro compiler asks for tokens and compiles until it runs into the semicolon on the %MEND statement. This means everything is stored away in a form for quick processing by the macro facility at a later time. No resolution of macro variables occurs at this time and no macro instructions are actually performed; the code is merely prepared for later use.

After the macro compilation WS encounters "%edits" and sends it to the macro facility which now calls up the compiled macro. The tokens "(", "state", "=", "TX", and ")" cause the compiler to store the parameter, STATE, as TX instead of NJ and to dump tokens from the macro compiler into the input stack. Now "Data" goes to the DATA step compiler as before. "Edited.&state" goes back to the macro facility for resolution. The resolved form is dumped by in the input stack to be found by WS and given to the DATA step compiler, since the macro facility is not asking for tokens. The semicolon goes to the DATA step compiler, for the same reason. The process continues in this manner.

When the %IF instruction is encountered by WS and sent to the macro facility, it will test whether &STATE resolves to TX or not. In our case it does, so the macro facility will then dump the special edits for Texas into the input stack for processing by WS. The process continues until the RUN statement is encountered. At this point the DATA step is finished compiling, thus it is executed by the SAS supervisor. Had there been more code in the macro after the RUN statement, the WS would have found this code and continued to distribute it to either the DATA step compiler, the procedure parser, or the macro facility as appropriate.

A common problem for macros that manage one or more steps is the specification of the input data set. SAS usually assumes the last created data set by default. How can we make use of the above knowledge to make the macro behave the same way? Consider

```
%macro steps ( data = &syslast ) ;
   .....
%mend  steps ;
```

At macro compile time, when the %MACRO statement is read, the value of &SYSLAST is irrelevant since macro variables are not resolved at macro compile time. The default value is the expression '&SYSLAST', not the resolved form. Now when the macro is invoked with

```
%steps ( )
```

the parameter DATA is assigned the default, &SYSLAST, and now it is resolved. Hence, DATA will have the value named by the last data set created before the macro invocation.

The technique is handy, but one must be careful. Often one would like the default for a parameter to be the resolution of some global macro variable whose value is assigned at the beginning of the program. Suppose we make the mistake of naming the parameter and the global variable by the same name, say DATA.

```
%macro steps ( data = &DATA ) ;
   .....
```

```
%mend  steps ;
```

At macro compile time there is no problem, but when the macro is invoked with the default, there is a big problem. How should &DATA be resolved? Since DATA is a parameter, it is local; hence &DATA does not refer to the global variable DATA. Now the macro facility is stuck with the conundrum, "Find the value of DATA by looking at the value of DATA", which results in an error message.

Conclusion

You have probably learned more than you wanted, but it should help to make you a better SAS programmer. You should now have a better sense of the four basic times

1. SAS compile time
2. SAS execution time
3. Macro compile time
4. Macro execution time

and what the SAS supervisor is doing in each of them.

Prior to Version 6 the SAS supervisor played a much bigger role during the execution of the DATA step. In Version 6 during DATA step execution it only handles the LIST statement and the dump tripped by _ERROR_ = 1. Both are done at the bottom of the implied loop and not where they occur in the code.

The author can be contacted by mail at

Westat Inc.
1650 Research Boulevard
Rockville, MD 20850-3129

or by E-mail at

whitloi1@westat.com

References

Donald J. Henderson, (1983), “The SAS Supervisor,” *Proceedings of the Eighth Annual SAS Users Group International Conference*, 924-931.

Tom Miron, (1996), “The Secret Life of the DATA Step,” *Proceedings of the Twenty-First Annual SAS Users Group International Conference*, 170-177

Mary G. Rabb, Donald J. Henderson, and Jeffry A. Polzin, (1992) “The SAS System Supervisor - A Version 6 Update, ” *Proceedings of the Seventeenth Annual SAS Users Group International Conference*, 190-197.