Paper 157-2010

Understanding the SAS® DATA Step and the Program Data Vector

Steven First, Systems Seminar Consultants, Madison, WI

ABSTRACT

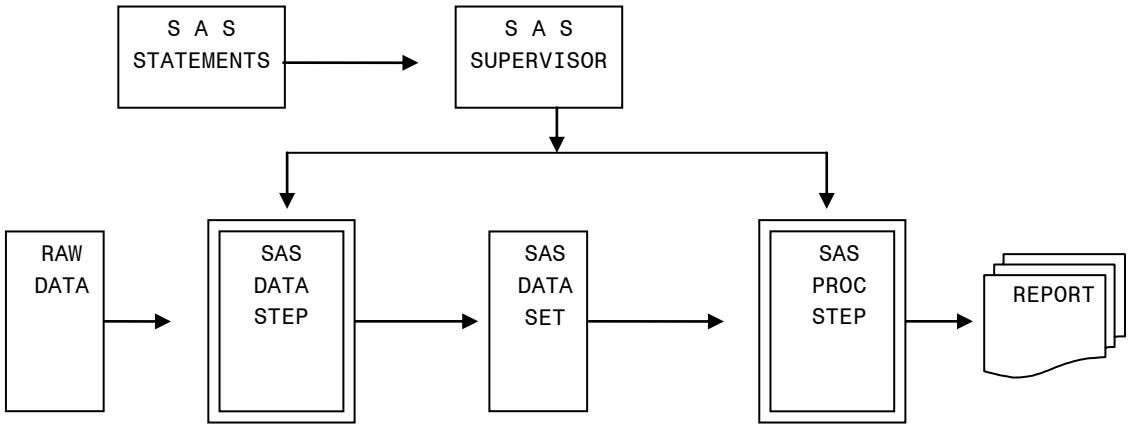
The SAS system is made up of two major components: SAS PROCs and the SAS DATA Step. The DATA Step provides an excellent, full fledged programming language that allows programs to read and write almost any type of data value, convert and calculate new data, control looping and much, much more. In many ways, the design of the DATA step along with its powerful statements, is what makes the SAS language so popular. This paper will address how the DATA step fits with the rest of the SAS System, DATA step assumptions and defaults, internal structures such as buffers, and the Program Data Vector. It will also look at major DATA step features such as compiler and executable statements.

INTRODUCTION

The SAS system's origins are in the 1960's and 1970's when A. J. Barr, James Goodnight, John Sall, and others developed the beginnings of the SAS system. Some of the concepts in the design include "self defining files", a system of default assumptions, procedures for commonly used routines, and a data handling step that would evolve into the SAS DATA step. The DATA step, in my opinion originally had an extremely simple, yet elegant design that continues today along with more than 30 years of enhancements.

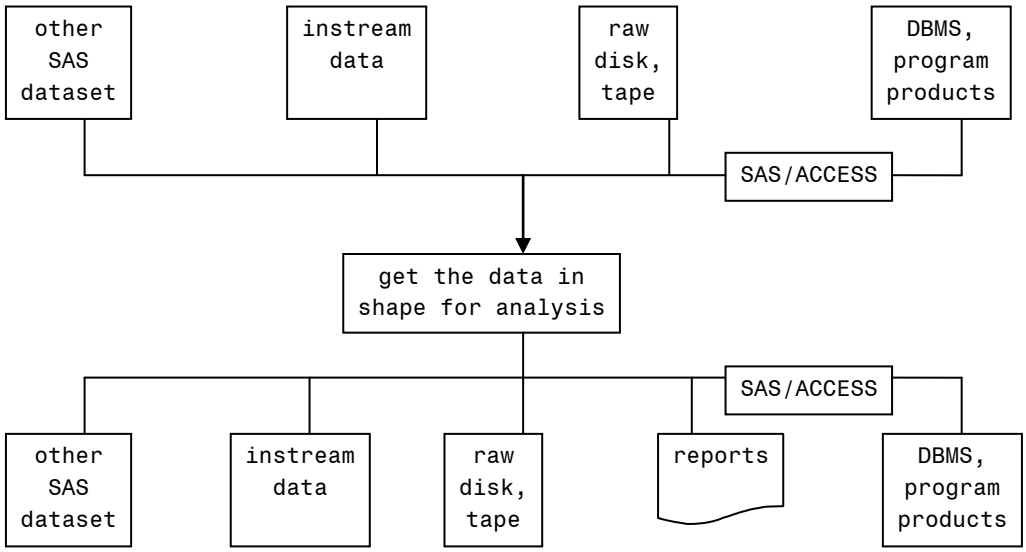
SAS STEPS

In general, when we invoke the SAS system, we first run a DATA step to get our data in shape and then use a SAS PROC to analyze our well defined SAS data set and print our results.



PURPOSE OF THE DATA STEP

The DATA step's function is in general to "get the data in shape" for later PROCs and DATA steps. SAS PROCs can only read SAS datasets, but we might have some other type of file to process. By definition, a SAS dataset has a built in descriptor that keeps track of names and attributes of each of the dataset's columns, so that later steps don't have to remember as many details. In the DATA step, we don't always have well defined data, and the DATA step gives us the power to read and write virtually any kind of file and do calculations and computations on a single row of data. It has a very powerful data handling language to accomplish the above. The definition of a "raw file" for this paper is any file that is not a SAS data set and is not an intelligent file such as a RDMS.



DEFAULT ASSUMPTIONS

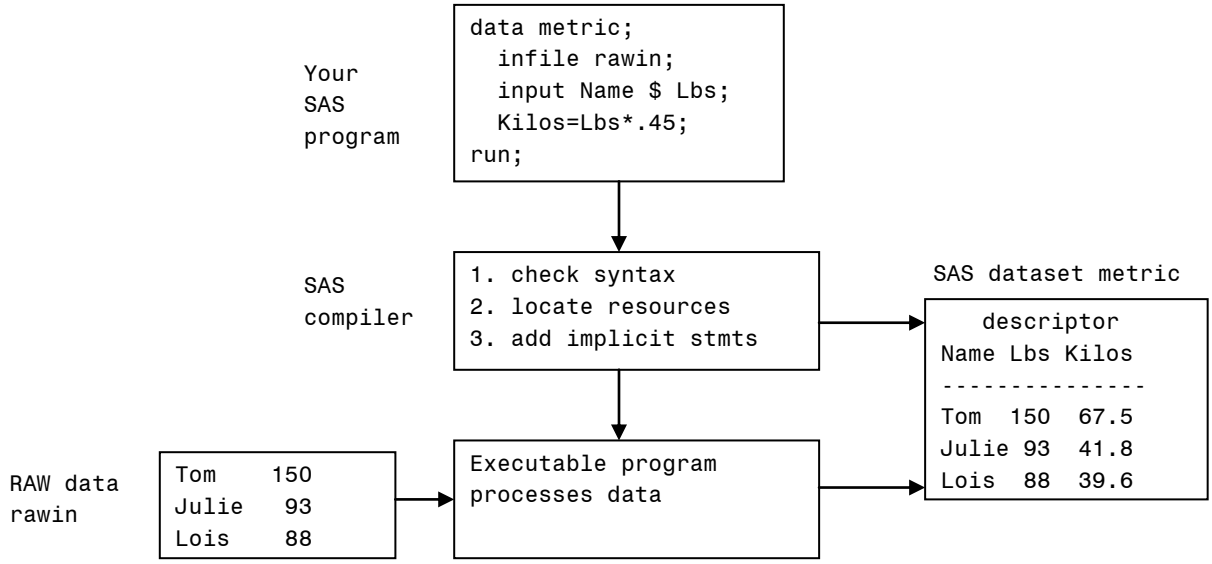
As a computer science major I studied and used many different programming languages, and early on, I was intrigued by the cleverness and common sense that the DATA step provided. Many tedious programming tasks were eliminated through the use of default assumptions, but the system still provided a means to override those defaults when necessary. Our job as a DATA step programmer then is different from those using other languages. In many ways, our tasks are understanding the defaults and knowing how to work with them and override them as necessary.

Examples of such defaults are:

- Handling the compile and execution naming and storage details for the programmer
- A dataset descriptor that makes SAS datasets “self defining”
- Generating data set names if omitted
- When reading a data set, assume to read the most recently created dataset if not specified
- Processing all the rows and columns in a file
- Automatically opening and closing of files
- Automatically controlling data initialization, DATA step looping, data set output, and end of file checking
- Automatically defining storage areas for each variable referenced without need to predefine them
- A default length of 8 was assumed for all variables
- An assumption that a variable is numeric if not specified
- LIST input assumed that data values would be separated by blanks rather than specifying exact columns
- SUBSETTING IF statements which imply to continue processing if a condition is true, else delete the observation
- When no comparison is made in an IF statement, assume to be checking for 1 (true)
- Abreviated sum statements. (Ex. Salestot+sales)

COMPILING A DATA STEP

As in many computer languages, the DATA step is first processed by a compiler and later, the compiled program is then executed. There are definitions for compiled languages, interpreted languages, and more, but it is probably safe to call the SAS system a hybrid language with features from other languages and obvious unique features. It is sometimes more difficult to separate compile versus execution events with SAS, since DATA steps are almost always compiled and executed immediately. Other compiled languages have a very discrete compile step that is done once, with separate executions done later. In any case, as in most languages, the DATA step compiler examines SAS statements for syntax and data structures and then generates an executable program. Different from other languages however, the SAS compiler checks for the existence of resources and also makes assumptions that it "inserts" into the source code.



DATA STRUCTURES

If the DATA step's function is to "get the data in shape", it needs data structures to hold that data as it is processed. Again, all computer languages need to address this, and though each language may name the structures differently, there is a lot of similarity in the way most languages store data.

RAW FILE BUFFERS

If the DATA step is going to read "raw" or non-SAS data, a memory buffer is needed to temporarily hold at least one input record at a time. There are also times when multiple lines of input can be held in buffers, and this allows the program to logically read later rows before earlier ones. If the DATA step is *writing* a raw file, then similar buffers are needed for each output file that is created. It should be noted that a buffer contains the complete input and output record, regardless of whether the INPUT statement reads all of the columns. When reading SAS datasets and RDMS (which usually appear as SAS datasets), there is no need for raw buffers as the files are already in "shape".

LOGICAL PROGRAM DATA VECTOR (PDV)

The DATA step refines data, and as such, a second memory area is needed for:

- Inputting and input formatting (informatting) desired variables
- Revising existing values
- Computing new variables
- System indicators and flags

This second area in memory is called the Logical Program Data Vector (PDV). Again, many languages have a similar working area. For example, COBOL calls this area Working Storages. All variables referenced in the DATA step will be automatically defined in the PDV by the compiler, using characteristics from the first reference of a

variable. That is, if the following statement is the first time AGEMO is used in the DATA step, AGEMO will be defined using the same characteristics as AGE, which is numeric in this case.

```
agemo=age/12;
```

When the compiler processes the DATA step, it needs to define a slot for each variable referenced in the program. These PDV slots will be defined in the order referenced in the program, and each variable has the following attributes:

- Relative variable number
- Position in the dataset
- Name
- Data type
- Length in bytes
- Informat
- Format
- Variable label
- Flags to indicate dropping and retaining of variables

The concept of *logical* PDV is used because RETAINed variables, which are DATA step variables that are not automatically initialized on each DATA step pass, are stored separately from the non-retained variables. This segregation allows the DATA step to clear all non-retained variables with just a few instructions. Even though the variables are not stored contiguously, we can *logically* consider them as contiguous.

DATA TYPES AND CONVERSION

Another feature of PDV variables is that they contain only two types of data values: numeric or character. Hundreds of different data types can be read or written via the PDV, but in the PDV, every character value is stored as a native EBCDIC or ASCII value with length between 1 and at least 32767, with numerics stored as double precision floating point values with length between 3 and 8 bytes. Storing only two data types greatly simplifies things for SAS datasets and moves the complication of converting different data types (packed, binary, etc.) to the INPUT and PUT statement along with appropriate INFORMATS and FORMATS. The choice of floating point for numbers with a length of 8 allows for storage of very large numbers without overflow, though floating point does have minor mathematical issues of its own.

PSEUDO VARIABLES

There are several special variables that the compiler creates that do not get added to the output file, thus the name *pseudo*. One variable called _N_ contains the number of times the DATA step has looped. Another _ERROR_ is set to 0 if there were no input errors, otherwise 1. Several others can be requested by the programmer to indicate when the end of the file is reached, beginning and ending of by groups, access to system control blocks, etc. Many of these variables are switches with values of 0 and 1, and others contain longer character values. These pseudo variables can be referenced by the DATA step, but they are dropped and don't end up on our final dataset.

OUTPUT DATASETS

The final structures needed by the DATA step can be a raw file out, in which case, output buffers will receive the results of FILE and PUT statements. These structures act in exactly the reverse of what INFILE and INPUT do, but this time the values are being converted *to* a raw file. Though DATA steps usually build *SAS* datasets, raw files can be extremely useful for passing data to other programs, and this makes the DATA step a very versatile utility.

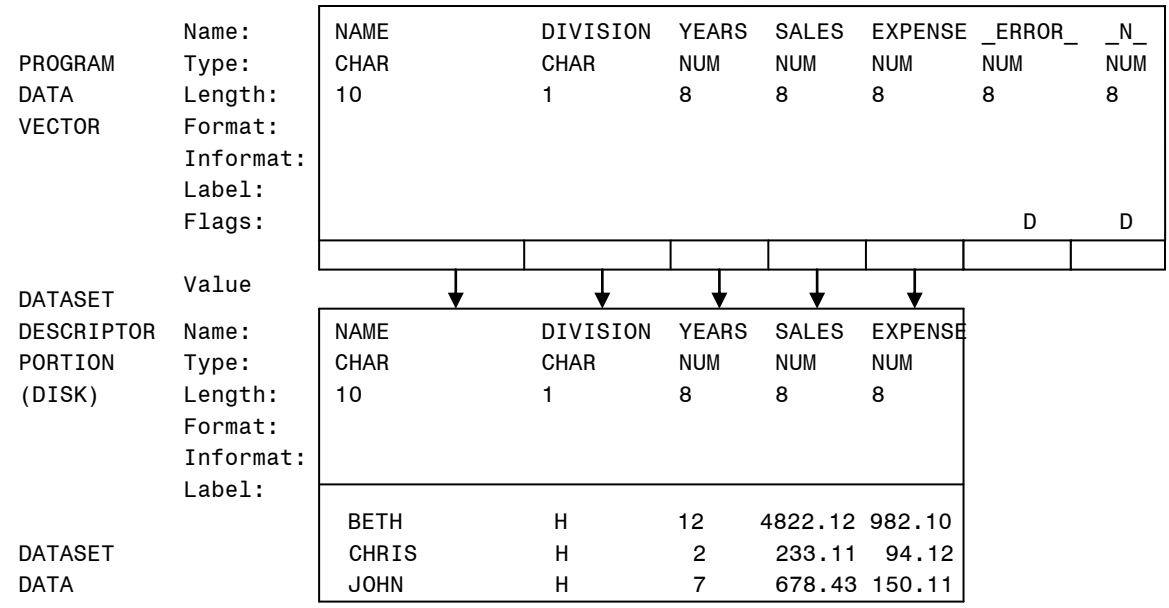
As stated above, most DATA steps produce SAS data sets. This is a much simpler operation for the programmer because SAS automatically builds the structures needed and outputs the record at DATA step return. In addition, all variables except dropped and pseudo variables will be included in the output data set. Basically a copy of the PDV is written to the SAS dataset descriptor that is stored with the SAS file along with all the values. This descriptor gives later steps all the information needed about the dataset and allows the programmer to concentrate on results rather than file layouts, data types, etc. This descriptor can be easily displayed by PROC CONTENTS or through displaying SAS dictionary tables. It also can serve as dataset documentation.

A TYPICAL SAS JOB

Read a raw file and create a SAS data file.

```
                                  12345678901234567890123456789012345
                 input buffer    | BETH       H  12  4822.12   982.10
                                 | CHRIS      H   2   233.11    94.12
                                 | JOHN       H   7   678.43   150.11
```

```
data softsale;
  infile rawin;
  input name $1-10 division $12 years 15-16 sales 19-25 expense 27-34;
run;
```



QUESTIONS ABOUT THE DATA STEP

If traditional programming experience is applied this DATA step, questions might be:

- Where are the opens?
- Where do we write out records?
- What is looping?
- When does the program stop?

ASSUMPTIONS MADE IN THE DATA STEP

The answers to the above are that the design of the DATA step fits well with the following scenarios which apply to *most* DATA steps:

- Input files are read starting with the first record and continuing until the end of the file.
- To eliminate errors and programmer work, all values from a previous record should be cleared before processing a new record.
- All variables referenced will be included on the output file.
- All records will be included on the resulting output file.
- All files should be opened at the beginning and closed at the end of the program step.
- Data file definitions should be passed automatically from one step to another.
- Most programs should not continue to loop if no data is read in a previous pass.

To accommodate *most* programs with the least amount of work, the DATA step has made use of the SAS dataset descriptor along with many assumptions made during DATA step processing. The SAS compiler makes the following assumption and inserts code to do the following:

- A DATA step will be entered at the top, and statements will be executed in sequence downward.
- Immediately upon entry, a check is made whether the previous step read any records from any file. If not, the DATA step is stopped with a looping message.
- All values from non-SAS files are cleared before executing any statements.
- If any reading statement would read a record after end of file, the step stops.
- If the program reaches the last statement in the step, or if a RETURN statement is executed, the current PDV contents (all columns for each row) is output to the SAS data set being built.
- A branch is executed to go to the top and enter the DATA step for another pass.

Another way of thinking about it would be as if the compiler inserted the following bold italicized code along with many other statements into our program.

```
data softsale;
if no input last time thru then stop
initialize PDV
infile rawin;
  if at EOF then stop
  input Name      $1-10
        Division  $12
        Years      15-16
        Sales      19-25
        Expense    28-34
        State     $36-37;
  output to SAS Dataset
  goto top of DATA step
run;
```

Not only do these inserted statements save us work, but they make our DATA steps virtually infinite-loop proof.

OVERRIDING DATA STEP ASSUMPTIONS

Of course, not all programs fit into the scenario above. If we know about the assumptions, how can we alter the behavior of the program?

The DATA step is a full featured programming language and provides many statements that alter the process above.

- The FIRSTOBS= and OBS= system options alter the program to begin and end logically after the first record and before the last record respectively.
- The RETAIN compiler statement instructs the step to *not* initialize variables.
- The STOP statement, usually used with an IF statement, stops the step before end of file is reached.
- The DELETE and subsetting IF statements exit the DATA step early and thus never reach the implied OUTPUT at the bottom of the step. The result is that fewer records than were read are written to the output dataset.
- RETURN exits the DATA step early but *does* output to the SAS file.
- DROP and KEEP statements and dataset options exclude or include variables in the final dataset.
- GOTO and various DO groups alter the looping path for the program.

RETAINING DATA VALUES

While initialization of all new variables works well for most cases, there are times when variables should not be cleared even though the DATA step has been exited and re-entered. The RETAIN statement specifies that the DATA step should not initialize those listed variables. RETAIN can also set an initial value, and if it references a variable for the first time, it will give the compiler length information. Since RETAIN essentially tells the compiler to set a flag that tells the execution phase to never clear the field, it really doesn't matter where we code RETAIN in the DATA step. It should be noted that any variable read with SET, MERGE, or UPDATE is considered to be retained.

Example: Read a file with CITY in the first row, RETAIN the value, discard the first row, but include CITY on all observations.

```
1234567890123456789012345678901234
 MADISON
BETH      H  12  4822.12   982.10
CHRIS     H   2   233.11    94.12
JOHN      H   7   678.43   150.11
```

```
data softsale;
  infile rawin missover;
  if _N_ = 1 then
    do;
         input city $2-8;
         delete;
    end;
  input name $1-10 division $12 years 15-16 sales 19-25 expense 27-34;
  retain city;
run;
```

PROGRAM DATA VECTOR

Name:	NAME	DIVISION	YEARS	SALES	EXPENSE	CITY	_ERROR_	_N_
Type:	CHAR	CHAR	NUM	NUM	NUM	CHAR	NUM	NUM
Length:	10	1	8	8	8	7	8	8
Format:								
Informat:								
Label:								
Flags:						R	D	D
Value						MADISON		

DATASET DESCRIPTOR PORTION (DISK)

Name:	NAME	DIVISION	YEARS	SALES	EXPENSE	CITY
Type:	CHAR	CHAR	NUM	NUM	NUM	CHAR
Length:	10	1	8	8	8	7
Format:						
Informat:						
Label:						

DATASET DATA PORTION

BETH	H	12	4822.12	982.10	MADISON
CHRIS	H	2	233.11	94.12	MADISON
JOHN	H	7	678.43	150.11	MADISON

STOPPING EARLY

Though most DATA steps start with the first line of data and continue until all records are read, this behavior can be overridden when necessary. As mentioned earlier, system options and dataset options can be set to start late and end early. A simple IF statement with STOP could be used to halt early or whenever some specified event occurs.

Example: Stop after 50 records.

```
data softsale;
  infile rawin;
  if _N_ = > 50 then stop;
  input name $1-10 division $12 years 15-16 sales 19-25 expense 27-34;
run;
```

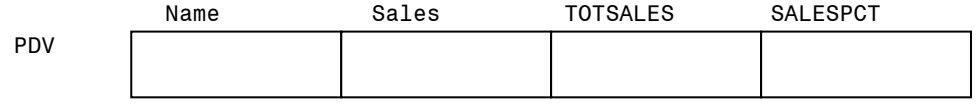
STOPPING LATE

A step may need to continue even though it has read the last record. An example is when calculating a percentage of total by reading a file twice. In order to accomplish this, the program needs to read the input file once and add up totsales, then pass thru it again to read the individual sales and do the calculation. By using the END= on the SET statement, we name a pseudo variable (EOF) that can be checked to test for the last record. Because we don't want to read past the last record with the first SET, it is necessary to do our own looping and to only execute the first SET when the program begins. The second SET statement has an independent pointer and reads the same records a second time, and the step will stop when the second SET tries to read a record past end of file.

```
DATA PCTDS;
 IF _N_ = 1 THEN
     DO UNTIL(EOF);
        SET CONCAT(KEEP=NAME SALES)
           END=EOF;
        TOTSALES+SALES;
     END;
 SET CONCAT(KEEP=NAME SALES);
 SALESPCT=(SALES*100)/TOTSALES;
RUN;
```

CONCAT DATASET

OBS	NAME	YEARS	SALES
1	BETH	12	4822.12
2	CHRIS	2	233.11
3	JOHN	7	678.43
4	MARK	5	298.12
5	ANDREW	24	1762.11
6	BENJAMIN	3	201.11
7	JANET	1	98.11



OUTPUTTING AND DELETING OBSERVATIONS

Of course, many programs will not want to include exactly every input row of data into the SAS output file. We may want to leave some unwanted records behind, or we might want to actually output more records than were read. Remember that the default action is to output the current row of data if our DATA step reaches the implied OUTPUT statement at the bottom of the DATA step. There are several statements that will force the program to avoid the OUTPUT statement and thus not include the row.

- DELETE (usually after an IF) instructs the DATA step to leave the DATA step and not OUTPUT.
- False cases of Subsetting IF statements also exit the step without OUTPUTing.
- RETURN exits the step but does OUTPUT.
- An explicit OUTPUT statement OUTPUTs when executed, but no longer does the implied OUTPUT at the bottom of the step.

Depending on your viewpoint, you may prefer those positive statements such as Subsetting IF, OUTPUT etc. to select wanted rows, or you might take a negative statement such as DELETE to filter unwanted rows.

It should be noted that the WHERE statement can also filter rows if the input is a SAS file. However, this action takes place outside of the DATA step, and the DATA step only sees rows that pass WHERE conditions.

Example: Only output records with more than 4000 in Sales.

```
data softsale;
if no input last time thru then stop
initialize PDV
infile rawin;
  if at EOF then stop
  input Name      $1-10
        Division  $12
        Years      15-16
        Sales      19-25
        Expense    28-34
        State     $36-37;
  If sales > 4000;
  If sales not gt 4000 then goto top of DATA step
  output to SAS Dataset
  goto top of DATA step
run;
```

SUM STATEMENTS

A very common task is to accumulate variables. That is, to add the values of a column from several observations and create a new variable. In most programming languages this would be coded such as:

```
totsales=totsales+sales;
```

This type of statement would not work correctly in a DATA step for 3 reasons:

- Totsales would be initialized to missing where 0 may work better in this case
- The value of totsales is not retained.
- If any value of sales is missing, propagation will set totsales to missing.

Example: Count the employees and sum their hours (incorrectly).

```
data timecard;
  infile rawin;
  input Name $ Hours;
  Ktr=ktr+1;
  Hourstot=hourstot+hours;
run;
proc print data=timecard;
  title 'SOFTCO PAYROLL';
run;
```

File RAWIN

```
JOE    40
PETE   20
STEVE   .
TOM    35
```

SOFTCO PAYROLL

OBS	Name	Hours	Ktr	Hourstot
1	JOE	40	.	.
2	PETE	20	.	.
3	STEVE	.	.	.
4	TOM	35	.	.

The above problems could be eliminated by using RETAIN and the SUM function along with the coded statement.

```
data timecard;
 infile rawin;
 input Name $ Hours;
 ktr=ktr+1;
 hourstot=sum(hourstot,hours);
 retain Ktr Hourstot 0;
run;
proc print data=timecard;
 title 'SOFTCO PAYROLL'; run;
```

File RAWIN

```
JOE    40
PETE   20
STEVE   .
TOM    35
```

	Name	Hours	Ktr	Hourstot
flags			RM	RM
PDV			0	0

SOFTCO PAYROLL

OBS	Name	Hours	Ktr	Hourstot
1	JOE	40	1	40
2	PETE	20	2	60
3	STEVE	.	3	60
4	TOM	35	4	95

SAS provides a more abbreviated statement called the SUM statement. The example below works as was intended with the statement above, with 3 differences:

- The left most variable is initialized to 0 rather than missing.
- The variable is retained.
- Any missing values are ignored.

Example: totsales+sales;

```
data timecard;
 infile rawin;
 input Name $ Hours;
 Ktr+1;
 Hourstot+hours; run;
proc print data=timecard;
 title 'SOFTCO PAYROLL'; run;
```

File RAWIN

```
JOE    40
PETE   20
STEVE   .
TOM    35
```

	Name	Hours	Ktr	Hourstot
flags			RM	RM
PDV			0	0

SOFTCO PAYROLL

OBS	Name	Hours	Ktr	Hourstot
1	JOE	40	1	40
2	PETE	20	2	60
3	STEVE	.	3	60
4	TOM	35	4	95

COMPILER INSTRUCTIONS

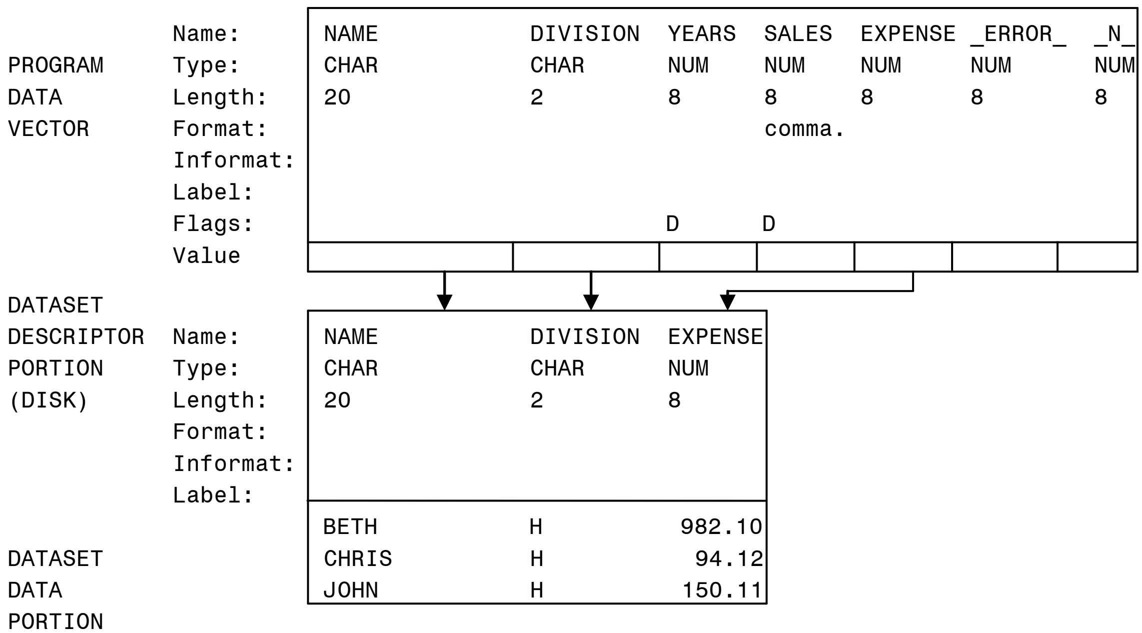
There are a series of statements that you can code that instruct the compiler to alter attributes of variables in the DATA step. In general, these are declarative statements that can be coded in any order and should be coded out the way of logical statements where the order is important. These statements allow the program to be very explicit in the definition of SAS structures.

Examples of these statements are:

- LENGTH statement to set a variables internal length
- INFORMAT to set input format
- FORMAT to set output display format
- LABEL to define a variable label
- ATTRIB to define any or all of the above in one statement
- DROP to indicate which variables are to be left behind on SAS file
- KEEP to indicate which variables to include on the SAS file
- RETAIN to set initial values and instruct SAS to never clear

```
1234567890123456789012345678901234
BETH       H  12  4822.12   982.10
CHRIS      H   2   233.11    94.12
JOHN       H   7   678.43   150.11
```

```
data softsale;
  infile rawin;
  length name $20;
  attr   division length=$2;
  format sales comma10.2;
  input name $1-10 division $12 years 15-16 sales 19-25
        expense 27-34;
  drop sales years;
  run;
```



DEBUGGING FEATURES

There is an interactive DATA step debugger available for stepping through a DATA step and displaying and breaking at various points. That system is an excellent well documented tool and is beyond the scope of this paper. There are also some simple DATA step statements that can be used to display data to help with debugging:

- The LIST statement can display the input buffer from the most recent INPUT.
- The FILE LOG with PUT can display any text and variable from the PDV in the SAS log.
- The PUTLOG statement can also display text to the SAS log.
- PROC CONTENTS and PROC PRINT/REPORT can be used to display the dataset descriptor and data values of the final dataset.

By putting the LIST and PUT/PUTLOG statements at strategic points in the DATA step, the program can display data before any statement that does not appear to be working correctly and show the flow of data as it is being refined. In many ways, this is the simplest and best way to debug your DATA step.

Example: The program below selects 0 records. Which statement is causing the problem?

```
                                  12345678901234567890123456789012345
BETH       H  12  4822.12   982.10
CHRIS      H   2   233.11    94.12
JOHN       H   7   678.43   150.11
```

```
data softsale;
  infile rawin missover;
  input name $1-10 division $12 years 15-16 sales 19-25 expense 27-34;
  if sales > 4000;
  if division='h';
run;


NOTE: The data set WORK.SOFTSALE has 0 observations and 5 variables.
```

Now use PUTLOG to display messages and values around our IF statements.

```
                                  12345678901234567890123456789012345
BETH       H  12  4822.12   982.10
CHRIS      H   2   233.11    94.12
JOHN       H   7   678.43   150.11
```

```
data softsale;
  infile rawin missover;
  input name $1-10 division $12 years 15-16 sales 19-25 expense 27-34;
  putlog '$$$ before sales if ' _n_= sales= division=;
  if sales > 4000;
  putlog '$$$ before division if ' _n_= sales= division=;
  if division='h';
run;


$$$ before sales if _N_=1 sales=4822.12 division=H
$$$ before division if _N_=1 sales=4822.12 division=H
$$$ before sales if _N_=2 sales=233.11 division=H
$$$ before sales if _N_=3 sales=678.43 division=H
NOTE: The data set WORK.SOFTSALE has 0 observations and 5 variables.
```

Looking at the partial log shows that the program passes the first IF but doesn't pass the second IF for BETH. It must be something in the second IF that is incorrect. (lower case 'h') We can correct the program and rerun.

OTHER DATA STEP STATEMENTS

The SAS DATA step has many, many more statements that can read, write, and process data in almost any form, including over 500 DATA step functions, interfaces to the SAS macro system, and much more. There has been much written about those features, and it is beyond the scope of this paper to try to cover them all. It is fair to say though, that the DATA step is an extremely versatile and full featured programming language.

OTHER TOPICS

As powerful and well designed as the DATA step is, it is different from other languages, and some might argue that a more standardized language such as SQL might be more auditable and more desirable in some cases. Terrible DATA step code can also be written, and it is important that good design and adequate documentation be included with the SAS code to make a well written program. PROC SQL is also a great tool that adds that language's

features to our SAS job. In any case, SAS programs can be well written programs that can be used for everything from one time programs to full fledged production programs.

CONCLUSIONS

The SAS DATA step is an excellent programming language with unique features and extremely versatile features.

CONTACT INFORMATION

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

Name	Steven First, President
Enterprise	Systems Seminar Consultants
Address	2997 Yarmouth Greenway Drive
City, State ZIP	Madison, WI 53711
Work Phone:	608 278-9964
Fax:	608 278-0065
E-mail:	sfirst@sys-seminar.com
Web:	www.sys-seminar.com