

# The Use and Abuse of the Program Data Vector

Jim Johnson, Covance Periapproval Services Inc., Radnor, PA

## ABSTRACT

Have you ever wondered why SAS does the things it does or why your programs “get away with” the things that they do or why SAS would not do what you wanted it to? A key operational component of SAS is the program data vector. Without it SAS would not function, as we know it. With knowledge of the program data vector, programmers can better understand how SAS works. This paper will help you understand how the program data vector works, how data steps use it, and how you can exploit, manipulate, and trick it.

## INTRODUCTION

Imagine a pile of fifty cinder blocks, each weighing about forty pounds. If you were asked to move the pile of cinder blocks from one room to another room fifty feet down the hall, you can imagine the amount of effort and the number of trips that will be necessary to accomplish the task. Now imagine a pile of fifty bricks, each weighing about five pounds. The amount of effort needed to move those bricks down the hall to another room is less, because you can take more units per trip, and therefore fewer trips. Finally, imagine a pile of fifty pennies. Obviously the effort required to move them to the end of the hall would be almost nil. The same is true for the function of the SAS System. The more you make the system do, the harder it will be and the longer it will take. If, however, you can reduce your cinder blocks to bricks, you can help SAS do its job better and faster. Knowing how the program data vector works, and how to use it efficiently can go a long way toward reducing your cinder blocks to bricks. If every time you write SAS code you imagine cinder blocks and bricks and pennies, you can become a better programmer.

All examples provided in this paper utilize SAS version 8.2 in Microsoft Windows NT®.

## PART 1: WHAT THE PROGRAM DATA VECTOR IS AND HOW IT WORKS

In this section the program data vector will be defined and described. Examples of how the program data vector is designed to work in the data step and some of the tools that can be used to change the contents of the program data vector will be discussed.

## THE PROGRAM DATA VECTOR

The SAS Language Reference, Version 8 defines the program data vector as: “A logical area in memory where SAS builds a data set, one observation at a time. When a program executes, SAS reads values from the input buffer or creates them by executing SAS language statements.

The data values are assigned to the appropriate variables in the program data vector. From here, SAS writes the values to a SAS data set as a single observation.”

Stated another way: The program data vector is a storage place in memory that contains all of the variables encountered by the data step. The order of the variables in the program data vector is determined by when they are encountered. These variables may have indicators flagging them to be kept or dropped or renamed. When the program runs, the program data vector contains the observation currently being processed. At the end of the step, the data is output according to the drop, keep, or rename instructions encountered in the program.

The program data vector also contains two automatic variables: `_N_` and `_ERROR_`. The `_N_` variable is the number of times the data step has executed. This is not necessarily the number of observations in any given data set. The `_ERROR_` variable indicates if an error has occurred in the data step. The value will be 0 if no errors occurred. It will be 1 if one or more errors occurred.

Other variables, referred to as temporary variables, can also appear on the program data vector. Automatic variables are on the program data vector automatically, you need not do anything to create them and they do not get written to the output data set. Temporary variables are created through the use of certain SAS options and statements, and like automatic variables, are not written to the output data set. Some common temporary variables include:

- *first.BY-variable* and *last.BY-variable*: These temporary variables always appear in pairs, one pair for each BY-variable on a BY statement. They have the value of 1 or 0 if the condition is true or false respectively.
- *IN=variable*: *IN=* is a data set option that indicates whether a particular data set has contributed to the current observation. The user specifies a variable with the option that will have the value of 1 if the data set contributed to the observation or 0 if it did not.
- *END=variable*: *END=* is an option to the SET statement that indicates the end of the input data has been reached. The user defines a variable that will have the value of 1 if the end of the data has been reached, 0 if it has not. Only one *END=* can be specified on the SET statement. If multiple data sets are given in the SET statement, the *END=* variable will be set to 1 only when the last observation of the last data set has been read.

Automatic and temporary variables are not written to the output data set, they cannot be kept, dropped, or renamed. User defined variables can be assigned the value of automatic variables if it is necessary to have their values in the output data set.

The program data vector holds all variables, user defined, automatic, and temporary, from the data sets, input

sources, or created during the execution of the data step. The program data vector is used to populate the output data sets. All variables in the output data sets are in the program data vector, but not all variables on the program data vector are written to the output data sets.

## DROP / KEEP / WHERE / RENAME

Use of DROP, KEEP, WHERE, and RENAME will affect the program data vector and can have a profound effect on the operation of SAS. It is important to understand when and where to use these operations. Use them in the wrong place and they will have little effect on the program data vector and efficiency.

Efficient use of the program data vector requires knowing the difference between a **SAS statement** and a **SAS data set option** and when each takes effect in the data step.

**SAS STATEMENTS** appear in data step code. The action of SAS statements occur more globally within the data step, at different times, and may have different effects than their SAS data set option counterpart. The KEEP, DROP, RENAME and WHERE statements are informational declarative statements that provide information about the program data vector and SAS data sets and take effect at compilation.

**SAS DATA SET OPTIONS** allow you to specify actions on specific SAS data sets as the data are read or written. These options include keeping, dropping, and renaming variables and subsetting data. Data set options are specified in parentheses following a SAS data set name, and the option name is followed by an equal sign and option details. Below are examples of SAS data set options in bold.

```
data demog(keep=pt age sex);
set source(rename=(dob=birthday));
proc print data=final(where=(complete=1));
```

Some of the differences between SAS statements and SAS data set options are:

1. Where they are specified
  - SAS statements appear anywhere in the data step code.
  - Data set options appear only in the DATA statement or input statements such as SET, MERGE, or UPDATE.
2. How globally they apply
  - SAS statements apply to all data sets created in the data step.
  - Data set options apply only to the data set to which they are attached.
3. When they take effect
  - The SAS statements DROP, KEEP, and RENAME take effect as the observation is written to the output data set. The WHERE statement takes effect as the data is read from the input data set(s).
  - Data set options can take effect as the data is written to the data set if specified on an output data set, or as the data is read if specified on an input data set.

## DROP STATEMENT

- The DROP statement indicates which variables in the program data vector should NOT be written to the output SAS data set(s).
- The DROP statement takes effect as the observation is written to the output data set(s).
- One DROP statement applies to all output data sets in the data step and can be used anywhere in the data step.
- Multiple DROP statements can be used in a data step. All will apply to the output data set(s).
- All variables listed on the DROP statement are on the program data vector and available for use in the current data step until the observation is output.

## DROP= DATA SET OPTION

- The DROP= data set option used on an input data set specifies the variables not to be READ from the data set to the program data vector.
- When used with an output data set it lists the variables on the program data vector that are not to be WRITTEN to the output data set.
- Data set options must be specified for each data set to which they apply.
- Only one DROP= data set option can be used with any one data set.

## KEEP STATEMENT

- The KEEP statement indicates which variables in the program data vector should be written to the output SAS data set(s).
- The KEEP statement takes effect as the observation is written to the output data set(s).
- One KEEP statement applies to all output data sets in the data step and can be used anywhere in the data step.
- Multiple KEEP statements can be used in a data step. All will apply to the output data set(s).
- Variables **not** listed in the KEEP statement are on the program data vector and available for use in the current data step until the observation is output.

## KEEP= DATA SET OPTION

- The KEEP= data set option used on an input data set lists those variables to be READ from the data set to the program data vector.
- When used with an output data set it specifies variables to be WRITTEN from the program data vector to the output data set.
- All variables are on the program data vector and available for use in the current data step until the observation is output.
- Data set options must be specified for each data set to which they apply.
- Only one KEEP= data set option can be used with any one data set.

If both DROP and KEEP are used in the same data step, the DROP will take effect first. If no DROP or KEEP are specified, all variables except automatic and temporary variables in the program data vector will be written to the data set(s).

The order the DROP and KEEP statements or data set options appear does not matter.

### EXAMPLE 1

```
data one;
  a = 1; b = 2;
  output;
run;

data two;
  set one(drop=a keep=a);
run;
```

NOTE: There were 1 observations read from the data set WORK.ONE.

NOTE: The data set WORK.TWO has 1 observations and **0 variables**.

The output data set above contains zero variables because on input the DROP= eliminated the variable A and the KEEP= effectively dropped everything except the variable A causing SAS to read zero variables from data set ONE.

Example 2 uses STATEMENTS instead of DATA SET OPTIONS.

### EXAMPLE 2

```
data two;
  set one;
  drop a;
  keep a;
run;
```

WARNING: The variable a in the DROP, KEEP, or RENAME list has never been referenced.

NOTE: There were 1 observations read from the data set WORK.ONE.

NOTE: The data set WORK.TWO has 1 observations and **0 variables**.

The effects of the DROP and KEEP *statements* occur as the observation is written to the output data set. The WARNING message is generated because the variable A was dropped by the DROP statement before it could be kept by the KEEP statement. The same message would occur if data set options are used on the output data set instead of the input data set in example 1.

### EXAMPLE 3

```
data one;
  a = 1; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output; /** duplicate **/
  a = 3; b = 2; c = 3; output;
run;
```

```
data two;
  set one(drop=a);
  if a = 1;
run;
```

NOTE: Variable a is uninitialized.

NOTE: There were 4 observations read from the data set WORK.ONE.

NOTE: The data set WORK.TWO has **0 observations** and 3 variables.

No observations are in data set TWO because variable A was dropped and is no longer available for use in the subsetting IF statement. Because it was dropped using a data set option on the input data set, it was never available to the program data vector, thus the uninitialized note was produced.

### WHERE STATEMENT

- The WHERE statement provides a subset of observations in the data set.
- The WHERE statement takes effect as data is read from the input data set(s). If an observation does not meet the condition in the WHERE statement, it will not be read. Therefore, the observation will never reach the program data vector.
- One WHERE statement applies to all input data sets in the step.
- Multiple WHERE statements can be used in a data step. All will apply to the input data set(s) as if written as one statement with AND operators between them.

### WHERE= DATA SET OPTION

- The WHERE= data set option used on an input data set subsets the data as it is READ from the data set. Observations that do not meet the where condition are never read, therefore never reach the program data vector.
- The WHERE= data set option used on an output data set subsets the data as it is WRITTEN to the output data set. The observations will be in the program data vector and will be written to the output data set if the condition is met.
- Data set options must be specified for each data set to which they apply.
- Only one WHERE= data set option can be used with any one data set.

### WHERE STATEMENT USED WITH WHERE DATA SET OPTION

If the WHERE= data set option is used on an input data set and the WHERE statement is also used in the program code, the data set option will be applied to the data set, and the WHERE statement will be ignored. Input data sets not specifying a WHERE= data set option will be subject to the WHERE statement.

```
data one;
  a = 1; b = 3; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output; /** duplicate **/
  a = 3; b = 2; c = 3; output;
  a = 3; b = 3; c = 3; output;
run;
```

Example 4 uses the WHERE data set option used on input with WHERE statement in the program code.

#### EXAMPLE 4

```
data two;
  set one(where=(a=1));
  where b=2;
```

WARNING: The WHERE statement cannot be applied to a data set on the last SET/MERGE/UPDATE/MODIFY statement. Either the data sets listed failed with open errors or they already specify a WHERE data set option.

```
run;
```

NOTE: There were 1 observations read from the data set WORK.ONE.

WHERE a=1;

NOTE: The data set WORK.TWO has 1 observations and 3 variables.

In example 5, the data set ONE is set twice, once with the WHERE= data set option, once without. SAS applies the WHERE= data set option to one input data set, but not to the other. The WHERE statement applies only to the first input data set. As a result, the final output data set contains 4 observations, one of which has the value of 3 for the variable B, which appears to conflict with the WHERE statement.

#### EXAMPLE 5

```
data two;
  set one
    one(where=(a=1));
  where b=2;
run;
```

NOTE: There were 3 observations read from the data set WORK.ONE.

WHERE b=2;

NOTE: There were 1 observations read from the data set WORK.ONE.

WHERE a=1;

NOTE: The data set WORK.TWO has 4 observations and 3 variables.

Obs	a	b	c
1	1	3	3
2	2	2	3
3	2	2	3
4	3	2	3

#### WHERE VERSUS SUBSETTING IF

Example 6 uses a WHERE statement and only one record is read. Example 7 uses a subsetting IF statement and all four records from the input data set are read.

#### EXAMPLE 6

Using a WHERE statement

```
data one;
  a = 1; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output; /** duplicate **/
  a = 3; b = 2; c = 3; output;
run;
```

```
data two;
  set one;
  where a=1;
run;
```

NOTE: There were **1 observations** read from the data set WORK.ONE.

WHERE a=1;

NOTE: The data set WORK.TWO has 1 observations and 3 variables.

#### EXAMPLE 7

Using a subsetting IF statement in place of WHERE statement

```
data two;
  set one;
  if a=1;
run;
```

NOTE: There were **4 observations** read from the data set WORK.ONE.

NOTE: The data set WORK.TWO has 1 observations and 3 variables.

#### SUBSETTING IF STATEMENT AND WHERE STATEMENT USED WITH END= DATA SET OPTION.

Example 8 uses the WHERE statement. Only two records are read from the input data set. Since the data step only sees two records, the END= option recognizes the last observation has been read and the 'LAST OBS REACHED' message is written to the log.

#### EXAMPLE 8

```
data a;
  a = 1; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output;
  a = 2; b = 2; c = 3; output; /** duplicate **/
  a = 3; b = 2; c = 3; output;
  a = 3; b = 3; c = 3; output;
run;
```

```
data b;
  set a end=last;
  where a=2;
  if last then put "LAST OBS REACHED";
run;
```

LAST OBS REACHED

NOTE: There were 2 observations read from the data set WORK.A.

WHERE a=2;

NOTE: The data set WORK.B has 2 observations and 3 variables.

In example 9, a subsetting IF statement is used in place of the WHERE statement. As a result, the last observation never reaches the "if last then" statement and the 'LAST OBS REACHED' message is not written to the log.

## EXAMPLE 9

```
data b;
  set a end=last;
  if a = 2;
  if last then put "LAST OBS REACHED";
run;
```

NOTE: There were 5 observations read from the data set WORK.A.

NOTE: The data set WORK.B has 2 observations and 3 variables.

Different positioning of the subsetting IF statement would produce different results, but could also further reduce efficiency if the data step were to contain more program statements.

## WHERE STATEMENT VERSUS SUBSETTING IF STATEMENT

- The WHERE statement is executed as observations are read. Observations not meeting the WHERE condition are never added to the program data vector.
- The subsetting IF is executed after the observations reach the program data vector.
- When the WHERE statement is used with a BY statement, the WHERE statement is executed first.
- When a subsetting IF is used with a BY statement, the BY statement is processed first.
- When the WHERE statement is used with a MERGE statement, the WHERE statement is executed first.
- When a subsetting IF statement is used with a MERGE statement, the MERGE is executed first.

## RENAME STATEMENT

- The RENAME statement specifies variables to be renamed and their new names.
- The RENAME statement takes effect as the observation is written to the output data set.
- One RENAME statement applies to all output data sets in the data step and can be used anywhere in the data step.
- Multiple RENAME statements can be used in a data step. All will apply to the output data set(s).
- The program data vector contains the variables with their OLD names, therefore, programs should continue to use the old variable names for processing in the current data step.

## RENAME= DATA SET OPTION

- The RENAME= data set option used on an input data set will rename variables to the new name as the variables are READ from the data set to the program data vector. The program should use the NEW variable names for the processing in the current data step.
- The RENAME= data set option used on an output data set will rename variables to the new name as the observation is being WRITTEN to the output data set. The program should use the OLD variable names for the processing in the current data step.
- Data set options must be specified for each data set to which they apply.
- Only one RENAME= data set option can be used with any one data set.

## HIERARCHY

The order of execution of these operations is:

1. DROP
2. KEEP
3. RENAME

If statements are combined with data set options, the hierarchy is the same within the effective timing described above.

In example 10 several SAS statements and data set options have been combined to demonstrate timing and hierarchy.

## EXAMPLE 10

```
data one;
  a = 1; b = 3; c = 3; output;
  a = 2; b = 1; c = 3; output;
  a = 2; b = 2; c = 3; output; /** a duplicate observation **/
  a = 3; b = 2; c = 3; output;
  a = 3; b = 3; c = 3; output;
run;
```

- 1) data two(keep=new\_code rank a b rename=(rank=index));
- 2) set one(rename=(c=code1)) end=last;
- 3) by b;
- 4) drop b;
- 5) rank = b;
- 6) where a = 2;
- 7) rename code1=new\_code;
- 8) keep code1 rank a b;
- 9) put \_all\_;
- 10) run;

WARNING: The variable b in the DROP, KEEP, or RENAME list has never been referenced.

WARNING: The variable b in the DROP, KEEP, or RENAME list has never been referenced.

```
last=0 a=2 b=1 code1=3 FIRST.b=1 LAST.b=1 rank=1
_ERROR_=0 _N_=1
```

```
last=1 a=2 b=2 code1=3 FIRST.b=1 LAST.b=1 rank=2
_ERROR_=0 _N_=2
```

NOTE: There were 2 observations read from the data set WORK.ONE.

WHERE a=2;

NOTE: The data set WORK.TWO has 2 observations and 3 variables.

```
proc print data=two;
run;
```

NOTE: There were 2 observations read from the data set WORK.TWO.

Obs	a	new_code	index
1	2	3	1
2	2	3	2

The RENAME= data set option on the input data set ONE in line number 2 is the first operation to take effect, renaming the variable C to CODE1. The variable C does not exist on the program data vector because the variable was changed on input.

The next operation is to read data from the input data set ONE. The WHERE statement in line number 6 will function at this time. Only records meeting the condition will be read into the program data vector.

Two executable statements exist in this data step.

- Line number 5: the variable RANK is populated with the contents of the variable B. The variable RANK will take the type and length attributes of variable B.
- Line number 9: writes the contents of the program data vector to the log.

The put statement in line 9 shows that the program data vector contains the following variables:

- LAST (from the END= option to the SET statement);
- A, B, and CODE1 (formerly named C) from the input data set ONE;
- FIRST.B and LAST.B (from the BY statement);
- RANK (from the assignment statement in line 5);
- \_ERROR\_ and \_N\_ (automatic variables)

Note the value of the temporary variable LAST is 0 (false) for the first observation output to the log, and 1 (true) for the second. The second output observation is the third observation in the data set ONE, not the last. This observation receives the value 1 indicating it is the last observation in the data set because of the WHERE statement on line 6. Only two of the original 5 observations in the data set ONE met the condition in the WHERE statement, thus the second of the two was the last in the data set and receives the indicator setting LAST to 1.

The DROP, RENAME, and KEEP *statements* in lines 4, 7, and 8 will be performed as the observation is written to the output data set in the following order:

- The DROP statement will prevent the variable B from being written to the output data set. The variable B is still on the program data vector and available for use and is being used in the assignment statement in line 5.
- The KEEP statement will write to the output data set only the variables listed. The variable B is on the KEEP statement as well as the DROP statement. The DROP executes first. When the KEEP executes, the first WARNING message is generated because the variable B no longer exists to be written.
- The RENAME statement will change the variable name CODE1 to NEW\_CODE. The RENAME occurs after the KEEP statement, therefore the KEEP statement uses the old name.

As the observation is written to the output data set, additional instructions are provided by the KEEP= and RENAME= data set options.

- The KEEP= data set option executes first keeping only the variables NEW\_CODE, RANK, A, and B. Since B has been previously dropped a second WARNING message is generated.
- The RENAME= data set option then changes the variable name RANK to INDEX.

The PROC PRINT shows all variables in the output data set. The output data set contains three variables: A,

NEW\_CODE, and INDEX. The temporary variables LAST, FIRST.B and LAST.B and the automatic variables \_ERROR\_ and \_N\_ were not written to the output data set. The order of the variables in the output data set is determined by when they were encountered in the data step.

- A came in first from the input data set;
- B was dropped;
- C was renamed to CODE1, then renamed to NEW\_CODE;
- RANK was created using an assignment statement, then renamed to INDEX.

Therefore, the final order of the variables is A, NEW\_CODE, and RANK.

## RETAIN

The RETAIN statement specifies variables whose values are **not set to missing** at the beginning of each iteration of the DATA step, causing SAS to hold the value from one iteration of the DATA step to the next. Only variables NOT in the input data sets can be retained. When multiple data sets are SET together, all non-retained variables in the program data vector are set to missing when subsequent data sets are encountered. Example 11 illustrates a number of different scenarios that are explained in detail below.

### EXAMPLE 11

```
data one;
  a = 1; b = 1; c=5;output;
  a = 1; b = .; c=5;output;
  a = 1; b = .; c=5;output;
run;

data two;
  d = 0; e = 0; output;
  d = 1; e = 0; output;
  d = 2; e = 0; output;
run;

data three;
  set one
    two;
  retain a 5 b;
  if _n_ = 1 then f = 0;
  a + 1;    /** in retain stmt **/
  b = b + 1; /** in retain stmt **/
  e + 1;   /** sum stmt, auto retain **/
  f = f + 1; /** NOT retained **/
  g + 1;   /** sum stmt, auto retain **/
run;
```

Obs	a	b	c	d	e	f	g
1	2	2	5	.	1	1	1
2	2	.	5	.	2	.	2
3	2	.	5	.	3	.	3
4	1	.	.	0	1	.	4
5	2	.	.	1	1	.	5
6	3	.	.	2	1	.	6

**Variable A** – input from data set ONE; variable is retained; value is calculated. At observation 4, the value is reset to missing, the sum statement adds one and a new value is output. The RETAIN, both explicit and implicit have no effect on the first three observations because the value comes from the input data set. When the second input data set is encountered, the value is set to missing and the RETAIN becomes effective.

**Variable B** – input from data set ONE; variable is retained; value is calculated. Though the variable is explicitly retained, the value is not retained from the first observation because the variable comes from the input data set.

**Variable C** – input from data set ONE. At observation 4 the variable is reset to missing when the second input data set is encountered.

**Variable D** – input from data set TWO. The variable exists in the first three observations, though it has a missing value. The variable is introduced to the program data vector at observation 4.

**Variable E** – input from data set TWO; implicitly retained by the sum statement; value is calculated. The sum statement calculates the value on observation 1: missing (input value) + 1 = 1. At the second observation the value is incremented by one, and again in the third observation. At observation 4, the value is set to 0 by the second input data set. The value is then incremented and output. At observation 5 the value is again set to 0 by the second input data set, incremented, and output.

**Variable F** – value is initialized when `_N_` is 1; value is calculated. The variable is not on an input data set, and it is not retained, therefore, at observation 2, it is reset to missing. The statement used to increment the variable F is not a sum statement, therefore, when the value is reset to missing at observation 2, the result of the addition is missing.

**Variable G** – value is calculated; implicitly retained by the sum statement. The value is not reset to missing at observation 4 because the value is retained and is not on an input data set.

#### SUMMARY OF SOME OF THE RULES OF RETAIN

- Retaining a variable on an input data set has no effect. Their values will be set by the input data set.
- At the start of each iteration of the data step, non-retained variables created in a data step are set to missing.
- When subsequent data sets are encountered in the SET statement, all non-retained variables are set to missing.
- Variables created with a sum statement are automatically retained.

If a variable name is specified **only** in the RETAIN statement and no initial value is specified, the variable is **not** written to the data set, and a note stating that the variable is uninitialized is written to the SAS log.

In example 12 below, variable A is retained with an initial value of missing. Variables B and C are retained with no default values. Of the three variables in the data step, only two are written to the output data set. Variable C is not written because it has no value, nor have its attributes been assigned.

#### EXAMPLE 12

```
data one;
  retain a . b c;
  format b 8.3;
run;
```

NOTE: Variable b is uninitialized.

NOTE: Variable c is uninitialized.

NOTE: The data set WORK.ONE has 1 observations and 3 variables.

Obs	a	b
1	.	.

## PART 2: HOW TO USE, ABUSE, MANIPULATE, AND EXPLOIT THE PROGRAM DATA VECTOR

This section will show how to take advantage of the program data vector's strengths and weaknesses, better understand why SAS does what it does, and some ways to get it to do things you want it to do.

### CHANGING LENGTH OF EXISTING VARIABLE

#### EXAMPLE 13

```
data one;
  var1 = 'abc';
run;
```

```
data two;
  set one;
  attrib var1 length=$5;
```

WARNING: Length of character variable has already been set. Use the LENGTH statement as the very first statement in the DATA STEP to declare the length of a character variable.

```
run;
```

NOTE: There were 1 observations read from the data set WORK.ONE.

NOTE: The data set WORK.TWO has 1 observations and 1 variables.

The LENGTH (or ATTRIB) statement should be appear before the SET statement so that the variable with its new attributes enters the program data vector before the SET statement. The program data vector variable attributes will initially come from the first statement. Subsequent statements can augment the variable attributes.

#### EXAMPLE 14

```
data two;
  attrib var1 length=$5;
  set one;
run;
```

NOTE: There were 1 observations read from the data set WORK.ONE.

NOTE: The data set WORK.TWO has 1 observations and 1 variables.

---

## MISSING AS A DATA STEP KEYWORD

Some programmers use MISSING as a data step keyword. No documentation exists describing the existence of MISSING as a data step keyword. MISSING is supported by PROC SQL, but not by the data step, at least not explicitly.

In example 15 the program checks to see if the variable DATE is null.

### EXAMPLE 15

```
data one;
  date = '04may2003'd; output;
  date = '05may2003'd; output;
  date = '06may2003'd; output;
  date = . ; output;
  date = '07may2003'd; output;
run;

data two;
  set one;
  if date = missing;
run;
```

NOTE: Variable missing is uninitialized.  
NOTE: There were 5 observations read from the data set WORK.ONE.  
NOTE: The data set WORK.TWO has 1 observations and 2 variables.

The expected results are received in spite of the NOTE indicating that the variable MISSING is uninitialized. The variable MISSING is uninitialized which ultimately gives it a missing value. The uninitialized variable MISSING is then compared to data set variables to locate null values. This works nicely, but for all the wrong reasons.

When the variable MISSING is created in the program data vector it takes the type and length attributes of the variable it is being compared to. In the example above, MISSING becomes a numeric variable of length 8 with a value of missing.

In example 16, the character variable TEXT is being checked for missing values. In this case the variable MISSING will take on the type and length attributes of the variable TEXT: character type, length of 3.

### EXAMPLE 16

```
data one;
  text = 'abc'; output;
  text = 'def'; output;
  text = ""; output;
  text = 'ghi'; output;
run;

data two;
  set one;
  if text = missing;
run;
```

NOTE: Variable missing is uninitialized.  
NOTE: There were 4 observations read from the data set WORK.ONE.  
NOTE: The data set WORK.TWO has 1 observations and 2 variables.

There is nothing particular about the variable name MISSING, it could just as well have been any other user defined variable name such as NULLTEXT, ALPHA, or TESTER.

---

## RETAINED OR NOT?

Given the data set below, VAR2 needs to have the value of 1 for every occurrence of X in VAR1 and 2 for every occurrence of Z.

<u>var1</u>	<u>var2</u>
X	.
Z	.
Z	.
X	.
X	.

```
data one;
  var1 = 'X'; var2 = .; output;
  var1 = 'Z'; var2 = .; output;
  var1 = 'Z'; var2 = .; output;
  var1 = 'X'; var2 = .; output;
  var1 = 'X'; var2 = .; output;
run;
```

```
proc sort data=one;
  by var1;
run;
```

```
data two;
  set one;
  by var1;
  if first.var1 then var2 + 1;
  put var1= var2=;
run;
```

```
var1=X var2=1
var1=X var2=.
var1=X var2=.
var1=Z var2=1
var1=Z var2=.
```

The first attempt looks like a RETAIN statement is necessary ... but, isn't the sum statement automatically retained? Well, lets try adding an explicit RETAIN statement:

```
data two;
  retain var2;
  set one;
  by var1;
  if first.var1 then var2 + 1;
  put var1= var2=;
run;
```

```
var1=X var2=1
var1=X var2=.
var1=X var2=.
var1=Z var2=1
var1=Z var2=.
```

There is no change after adding an explicit RETAIN. Why doesn't the RETAIN retain? The problem comes because the variable VAR2 is on the input data set. Retained or not, the variable VAR2 is reset by the SET statement. One solution is to use a DROP= data set option to remove the variable VAR2 on input.

```
data two;
  set one(drop=var2);
  by var1;
  if first.var1 then var2 + 1;
  put var1= var2=;
run;
```

```
var1=X var2=1
var1=X var2=1
var1=X var2=1
var1=Z var2=2
var1=Z var2=2
```

---

## OVERWRITTEN VARIABLE ATTRIBUTES

The two data sets below both contain the variable NAME, but with different attributes. When they are SET together some attributes are changed.

```
data one;
  name = 'Mary';
run;

data two;
  label name="Subject name";
  name = 'Scott';
run;

data all;
  set one
      two;
  put name =;
run;

name=Mary
name=Scot
```

The length attribute from data set ONE overrides the length attribute of the variable in data set TWO resulting in truncated values.

When the data set ALL is printed using the following PROC PRINT, the statement above seems to be contradicted because the variable label from data set TWO is used.

```
proc print data=all label;
run;
```

	Subject
Obs	name
1	Mary
2	Scot

The program data vector operates with a "first come, first served" action. If a variable name is encountered multiple times with differing attributes, the first non-blank variable attribute encountered is used to set the attributes in the program data vector.

When data set ONE is read the program data vector sets up the variable NAME with a length of \$4 and a null label. When data set TWO is encountered, the length of NAME is different, but the length has already been set in the program data vector. The variable NAME from data set TWO has a label assigned to it. Since no label yet exists on the program data vector for this variable, the label from data set TWO is used.

---

## CHANGING A VARIABLE'S ATTRIBUTES WITHOUT CHANGING ITS NAME IN ONE DATA STEP

This technique is very simple given the features described in this paper. In the example below the input data set DEMOG contains the character variable AGE. Using a combination of the DROP= and RENAME= data set options, the variable AGE is changed from character to numeric in one data step.

```
data demog;
  age = '25';

data demog(rename=(num_age=age)
  drop=age);
  set demog;
  num_age = input(age,8.);
```

This works because the order the data set options are processed is DROP=, KEEP=, RENAME=. Thus the original variable AGE is dropped, then the numeric variable NUM\_AGE is renamed to AGE as the data is written to the output data set.

---

## TEMPLATES

Templates can be used to get what you want from a data step. This is an efficient technique, but care needs to be used to manipulate the program data vector appropriately. As indicated earlier in this paper, the first non-missing attributes are used in the program data vector. This technique works best if subsequent data sets have no attributes like FORMAT or INFORMAT. If they do contain attributes undefined in the template, those attributes will come through in the final data set.

Create a template data set, with zero observations, that define the variables and attributes desired in the final data.

```

data demog;
  attrib pt length=8 label='subject number' format=z6.;
  attrib sex length=$6 label='subject gender';
  attrib age length=3 label='subject age';
  attrib race length=8 label='ethnic origin' format=race.;
  if _n_ = 0;
run;

```

NOTE: Variable pt is uninitialized.  
NOTE: Variable sex is uninitialized.  
NOTE: Variable age is uninitialized.  
NOTE: Variable race is uninitialized.  
NOTE: The data set WORK.DEMOG has **0 observations**  
and 4 variables.

#### The statement

```
if _n_ = 0;
```

allows the data set to be created with zero observations. The notes referring to uninitialized variables are expected since no values are assigned to the variables being created.

```

data garbage;
  pt = 101;
  sex = 'male';
  age = 21;
  dob = '21mar1985'd;
  addte = '10jan2003'd;
  moddte = '15jan2003'd;
  race = 2;
  output;
run;

```

NOTE: The data set WORK.GARBAGE has 1 observations  
and 7 variables.

The data set GARBAGE contains variables used in data processing. Many intermediate and unnecessary variables exist. The desired variables do not have the required attributes. There are no labels, some lengths are wrong, and no formats are assigned.

Using the template created earlier in a SET statement will set the desired attributes. The template data set must be referenced first, which will load its variable attributes into the program data vector. When the second data set is encountered, the values will be "forced" into the attributes already defined in the program data vector.

```

data final;
  set demog /** TEMPLATE, zero observations **/
  garbage; /** data set with all sorts of junk **/
run;

```

NOTE: There were 0 observations read from the data set WORK.DEMOG.  
NOTE: There were 1 observations read from the data set WORK.GARBAGE.  
NOTE: The data set WORK.FINAL has 1 observations and 7 variables.

The data set FINAL has more variables than defined in the template DEMOG. Add a KEEP or DROP statement to limit the variables in the FINAL data set.

## CONCLUSION

Knowing that all the variables read are present, whether ultimately kept, dropped, or renamed, and knowing when and how the program data vector processes the variables it encounters provides the programmer with valuable information to understand why SAS does what it does, and how it can be manipulated to enhance the operation of the program.

With this knowledge, and the simple tools provided by SAS, the programmer can change cinder block into bricks, and even bricks into pennies. This goes a long way toward efficient programming.

What has been discussed in this paper is quite literally the tip of the iceberg. A good working knowledge of the program data vector can only be obtained through years of programming experience, experimentation, and trial and error. Readers are encouraged to research and experiment further on their own.

## REFERENCES

SAS Institute Inc. (February 2000) *SAS OnlineDoc@, Version 8*, Cary, NC: SAS Institute Inc.

## TRADEMARKS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute, Inc. in the USA and other countries.

## CONTACT INFORMATION

Jim Johnson  
Principal Scientific Programmer  
Covance Periapproval Services, Inc.  
One Radnor Corporate Center, Suite 300  
Radnor, PA 19087  
jb.johnson@covance.com