SAS® DATA Step – Compile, Execution, and the Program Data Vector

Dalia C. Kahane, Westat, Rockville, MD

ABSTRACT

The SAS® DATA Step is one of the primary methods for creating SAS data sets. To be a good SAS programmer it is essential that you understand the intricacies of the DATA Step because some tasks related to data manipulation and data set creation may only be done via the DATA Step (they cannot be done or are too complex to be done via SAS procedures.)

It is especially important to learn the rules governing the DATA Step Compile phase and Execution phase; as well as to understand the Program Data Vector (PDV), which spans across both phases and which determines what information is held in storage and how it changes within the DATA Step.

The Compile phase includes:

- compiling the SAS statements, including syntax checking
- creating an input buffer (if reading a raw input file), a Program Data Vector (PDV), and descriptor information

The Execution phase includes:

- counting the iterations of the DATA Step (beginning at the DATA statement)
- setting all variables to missing in the Program Data Vector and initializing automatic variables
- reading an input record (from raw file or SAS data set), if relevant
- executing additional manipulation or computation statements
- writing an output record into an output data set and looping back to the DATA statement

The Program Data Vector spans across the compile and execution phases:

- it is a logical area in memory where SAS builds a data set, one observation at a time
- contains current values for all variables
- maintains two automatic variables, _N_ and _ERROR_

If you know the rules that govern these important aspects of the DATA Step process, then you will be able to take control, correctly instruct SAS what to do, and utilize the automatic variables for your benefit.

INTRODUCTION

The SAS® documentation describes the DATA step as follows: “the primary method for creating a SAS data set with base SAS software. A DATA step is a group of SAS language statements that begin with a DATA statement and contains other programming statements that manipulate existing SAS data sets or create SAS data sets from raw data files.”[1]

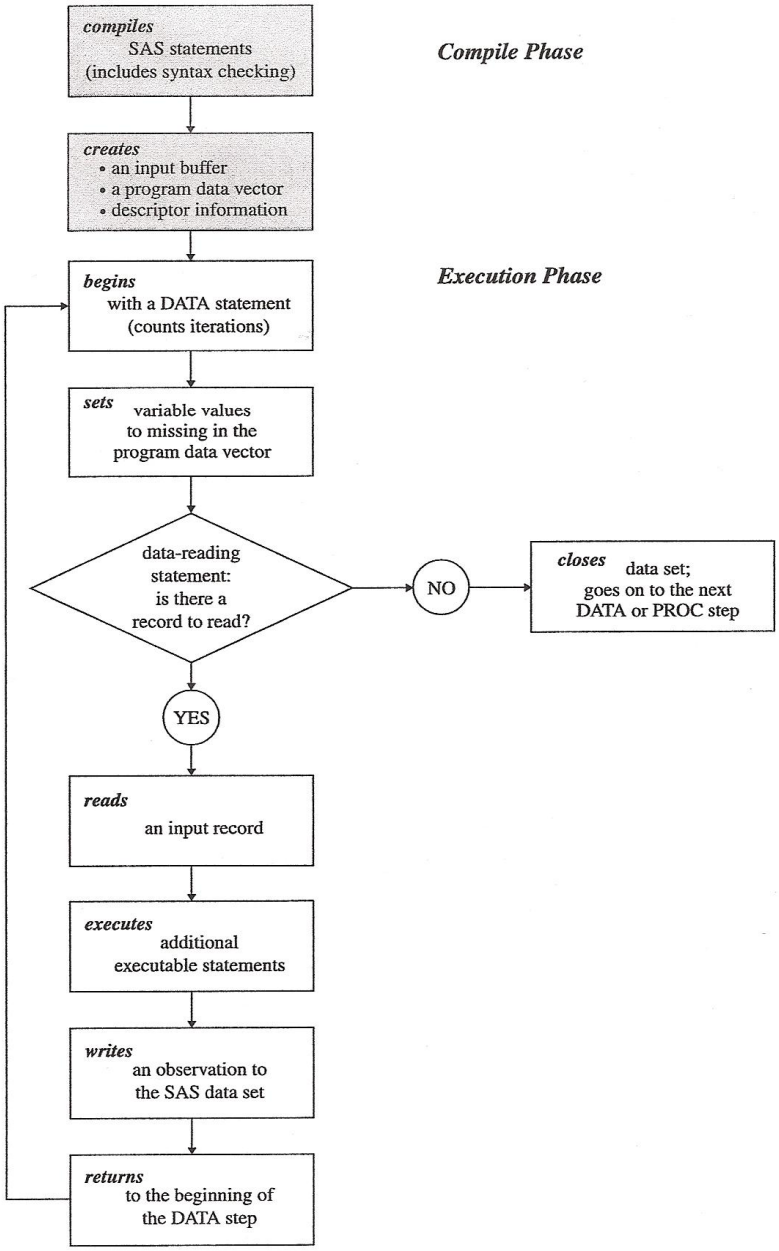
SAS processes the DATA Step in two phases.

- First, there is the compile phase: the code is read, checked for accuracy of syntax, the DATA Step environment is set up and machine code is created.
- Then, there is the execution phase: within the boundaries of a simple DATA Step, statements are executed in order; once per iteration of the step. There is an implied loop. The boundaries of a DATA Step are between the DATA statement at the “start” of the step and the implied or explicit RUN statement at the “end” of the step. The “end” of a simple DATA Step serves as an implied RETURN until the last observation of this DATA Step is processed.

At Compile time, the Program Data Vector (PDV) is initialized. It is the logical area in memory where SAS builds a data set, one observation at a time. When a program executes, SAS reads data values from the input buffer or from an existing data set or creates them by executing SAS language statements. The data values are assigned to the appropriate variables in the Program Data Vector and from there, SAS writes the values to an output data set as a single observation.

SAS provides the following flowchart to describe the processing of the DATA Step[1]

Figure 20.1 Flow of Action in the DATA Step



This paper describes the Compile and Execution phases as well as the PDV in detail. You will be shown what SAS does “behind the scenes” and will become aware of how to code your DATA Step in a way that is most effective for your needs.

COMPILE PHASE

As mentioned above, the first phase in DATA Step processing is the Compile phase. First, SAS does a syntax check of the code. If this check fails, errors will appear in the log, compilation will stop, and execution will not commence. Once past the syntax check, SAS performs the following tasks:

- automatically translates the statements into machine code to be executed later
- identifies the type and length of each new variable
- determines whether a type conversion is necessary for each subsequent reference to a variable
- creates input buffer if there is an INPUT statement for an external file
- creates the Program Data Vector
- creates descriptor information for data sets and variable attributes
- processes statements, which are limited to compile-time; these provide information to the compiler on how to set things up; in fact, they drive how things will be set up within the PDV; they include:
  DROP; KEEP; RENAME; RETAIN; WHERE; LABEL; LENGTH; FORMAT; ARRAY; BY; ATTRIB
- sets up some automatic variables, including:
  _N_, _ERROR_, END=, IN=, FIRST, LAST, POINT=

Compile Example – Syntax Errors Found

If the code submitted within a data step causes a syntax error to be generated, SAS terminates processing of the DATA Step, no execution is done, and SAS may even set the observation limit to zero beyond that point.

```
data example_x;
   x = “CAT”
   y = “DOG”;
run;
```

```
The log will present the following messages:
         data example_x;
              x = "CAT"
              y = "DOG";
              -
              22
ERROR 22-322: Syntax error, expecting one of the following: !, !!, &, *, **, +, -, /, ;, <, <=,
              <>, =, >, ><, >=, AND, EQ, GE, GT, IN, LE, LT, MAX, MIN, NE, NG, NL, NOTIN, OR, ^=,
              |, ||, ~=.

         run;

NOTE: Character values have been converted to numeric values at the places given by:
      (Line):(Column).
      75:16   76:16
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.EXAMPLE_X may be incomplete.  When this step was stopped there were 0
         observations and 2 variables.
```

Note that in order to check the code for syntax errors (and use minimum execution time) you may use the following statements:

```
%let testing=1;
%if &testing=1 %then %do; options obs=0; %end;
```

EXECUTION PHASE

During the Execution Phase, SAS performs the following functions, in this default order:

- begins with the DATA Statement and sets the _N_ variable to 1 (with each new iteration of the DATA Statement, the _N_ variable gets incremented by 1)
- sets variable values to missing in the Program Data Vector
- reads a data record into the input buffer with the INPUT statement (if reading a raw file)
- reads a SAS observation from a SAS data set into the PDV
  with one of the following statements: SET, MERGE, MODIFY or UPDATE
- executes programming statements found within the DATA Step
- writes an observation into the output data set(s), except in the case of “data _NULL_”
- returns to the DATA Statement
- all actions are repeated per iteration until end of input file (or input data set) is reached
- if the DATA Step includes no reading of input records, the Execution (by default) performs only one iteration

This default order of execution may be changed through the use of conditional IF-THEN-ELSE statements, DO Loops, LINK, RETURN, and GO TO statements, etc.

Simple Execution Examples – default and forced sequence

The first example shows a simple DATA Step, which follows the standard execution of statements and output of records. Here, one iteration produces one observation in the output data set:

```
data example_x1;
   x = “CAT”;
   y = “DOG”;
run;
```

```
The log presents the following message:
NOTE: The data set WORK.EXAMPLE_X1 has 1 observations and 2 variables.
```

The second example shows how a single iteration of the data step produces 5 observations in the output data set. The default order of execution was changed via the DO Loop:

```
data example_x2;
   do i = 1 to 5;
      x = "CAT";
      y = "DOG";
      output;
   end;
run;
```

```
The log presents the following message:
NOTE: The data set WORK.EXAMPLE_X2 has 5 observations and 3 variables.
```

THE PROGRAM DATA VECTOR

The program Data Vector (PDV) is a logical concept to help you think about the manipulation of variables in the DATA step. Prior to version 6 it was also a physical concept referring to the specific area in memory where DATA step variables were stored. In version 6 the memory structure was reorganized to speed up the system time but the PDV concept continues to be a good way for programmers to think about what is important in variable manipulation[2].

The PDV can be thought of as one long list of storage areas set aside for all named variables of the DATA Step[2] (these include user-defined, automatic and temporary variables). The order of the variables in the PDV is determined by when the variables appear in the code. The best way to "look behind the scenes" into the PDV at any point in the Execution phase is by utilizing the simple by powerful PUT _ALL_ statement.

In the following examples, the structure and contents of the PDV are shown as they would appear at the end of compile and at various points of the execution phase within the example DATA Step.

DATA STEP PROCESSING EXAMPLES

Example 1 – simple default – no reading of input data

```
data example1;
  put "after compile before execution: " _all_;
  x = "CAT";
  y = "DOG";
  x = "MOUSE";
  n = 5;
  k = n*2;
  put "at end of execution: " _all_;
run;
```

The Compile phase sets up the PDV as follows:

x	y	n	k	_N_	_ERROR_
		.	.	1	0

At the end of the DATA Step Execution the PDV appears as follows:

x	y	n	k	_N_	_ERROR_
MOU	DOG	5	10	1	0

One single record is output into the example1 data set. Only a single iteration of the Step is done.

Notes:

- the value of x has become MOU (neither CAT nor MOUSE) because the compiler set the length of the variable x to 3 when it first read that X="cat". An explicit LENGTH statement needs to be added at the start of the DATA Step if x should be stored in a longer string
- the output record includes all the variables except for the automatic _N_ and _ERROR_

```
The log presents the following messages due to the put _all_ statements:
after compile before execution: x=  y=  n=. k=. _ERROR_=0 _N_=1
at end of execution: x=MOU y=DOG n=5 k=10 _ERROR_=0 _N_=1
```

Example2 – reading of raw input data

Let's assume a simple data file "grades" to be input, as follows:

```
101     A     80     70
102     B           60
103     A     70
104     A     80     95
105     B     90     80
```

```
data example2;
   infile grades missover ;
   length StudentID 8 Class $5 English Math eng_cnt math_cnt average 8 ;
   input StudentID Class $ English  Math ;
   if English > . then eng_cnt=1 ; else eng_cnt=0 ;
   if Math > . then math_cnt=1 ; else math_cnt=0 ;
   class_cnt  = sum (eng_cnt, math_cnt) ;
   average = sum (English, Math) / class_cnt  ;
run;
```

The Compile phase sets up the Input Buffer and the PDV as follows:

Input Buffer – gets filled with input data in byte-order – only two input records shown as example

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	0	1					A					8	0					7	0										
1	0	2					B											6	0										

Program data Vector

StudentID	Class	English	Math	Eng_cnt	Math_cnt	Class_cnt	Average	_N_	_ERROR_
		.	.	.	.	.	.	1	0

This table shows how each record would look in the PDV before it is written out to the example2 SAS data set. Remember that the PDV only holds one record at a time and writes it to the output data set before a new record is built in the PDV.

StudentID	Class	English	Math	Eng_cnt	Math_cnt	Class_cnt	Average	_N_	_ERROR_
101	A	80	70	1	1	2	75	1	0
102	B	.	60	0	1	1	60	2	0
103	A	70	.	1	0	1	70	3	0
104	A	80	95	1	1	2	87.5	4	0
105	B	90	80	1	1	2	85	5	0

Example3 – reading of input data from SAS data set

```
data grades;
  StudentID=101; Class="A"; English=80; Math=70; output;
  StudentID=102; Class="B"; English=. ; Math=60; output;
  StudentID=103; Class="A"; English=70; Math=. ; output;
  StudentID=104; Class="A"; English=80; Math=95; output;
  StudentID=105; Class="B"; English=90; Math=80; output;
run;
```

```
data example3;
    set grades;
    length eng_cnt math_cnt average 8 ;
    if English > . then eng_cnt=1 ; else Eng_cnt=0 ;
    if Math > . then math_cnt=1 ;
    class_cnt  = sum (eng_cnt, math_cnt) ;
    average = sum (English, Math) / class_cnt  ;
run;
```

Here there is no Input Buffer. The Compile phase sets up the Program data Vector, as follows:

StudentID	Class	English	Math	Eng_cnt	Math_cnt	Class_cnt	Average	_N_	_ERROR_
		.	.	.	.	.	.	1	0

Here, the PDV at execution, before each record is written out to the output SAS data set:

StudentID	Class	English	Math	Eng_cnt	Math_cnt	Class_cnt	Average	_N_	_ERROR_
101	A	80	70	1	1	2	75	1	0
102	B	.	60	0	1	1	60	2	0
103	A	70	.	1	0	1	70	3	0
104	A	80	95	1	1	2	87.5	4	0
105	B	90	80	1	1	2	85	5	0

Note that the PDV at the end of the Compile phase and at the end of the Execution phase for example 3 looks exactly the same as described above for Example 2.

Example4 – reading of input data and processing of data with a BY statement

SAS allows you to process data using BY-processing for ordered data sets. This is very useful for calculating values based on BY-groups, creating selective output, etc. When BY-processing is requested, SAS sets up special automatic variables that reflect the status of each record relative to the BY variable(s).

Assuming the following sort:

```
proc sort data=example3 out=example4;
by class;
run;
```

The following statements:

```
set example4(keep= StudentID Class English Math) end=eof;
by class;
```

cause SAS to add the variables FIRST.CLASS and LAST.CLASS to the Program Data Vector after the variables in example4. As the set statement is executed the PDV includes the following:

StudentID	Class	English	Math	First.Class	Last.Class	_N_	_ERROR_
101	A	80	70	1	0	1	0
103	A	70	.	0	0	2	0
104	A	80	95	0	1	3	0
102	B	.	60	1	0	4	0
105	B	90	80	0	1	5	0

The significance of these variables means that you, as the SAS programmer, automatically know when a group begins and ends. This is useful information and gives you enough control to analyze data at every record.

There is no need to see all of the data at once to determine the group end points. You don't even need to look ahead or behind by one record. Consequently, many programming tasks are simplified. For example: you may split the data set into multiple sets, one per class; you may calculate summary statistics per group and overall; you may draw conclusions about the groups of data, sending out status records; etc.

The following example shows how the First.Class and Last.Class automatic variables are used to help calculate and output summary statistics:

```
data summary_stats(keep=class sumMath cntMath avgMath sumEnglish cntEnglish avgEnglish) ;
   length sumMath  cntMath  sumEnglish cntEnglish sumAllMath cntAllMath sumAllEnglish cntAllEnglish  8;
   if eof then do;
      class="T";
      avgMath=sumAllMath/cntAllMath;
      avgEnglish=sumAllEnglish/cntAllEnglish;
      put "at eof and before output: " _all_;
      output;
   end;
   set example4(keep= StudentID Class English Math)  end=eof;
   by class;
   retain sumMath  cntMath  sumEnglish cntEnglish sumAllMath cntAllMath sumAllEnglish cntAllEnglish ;
   /* calculate summary stats per class and output separately */
   if _n_=1 then do; sumAllMath=0; cntAllMath=0; sumAllEnglish=0; cntAllEnglish=0; end;
   if first.class then do;
     sumMath=0;  cntMath=0;
     sumEnglish=0; cntEnglish=0;
     put  "at First.Class: " _all_;
  end;
  if Math>. then do; sumMath+Math; cntMath+1; end;
  if English>. then do; sumEnglish+Math; cntEnglish+1; end;
```

```
    if last.class then do;
      avgMath=sumMath/cntMath;
      avgEnglish=sumEnglish/cntEnglish;
      sumAllMath+sumMath;
      sumAllEnglish+sumEnglish;
      cntAllMath+cntMath;
      cntAllEnglish+cntEnglish;
      put "at Last.Class and before output: " _all_;
      output;
    end;
  run;
```

The Log messages (due to the PUT _ALL_ statements) show the following:

```
at First.Class: sumMath=0 cntMath=0 sumEnglish=0 cntEnglish=0 sumAllMath=0 cntAllMath=0
sumAllEnglish=0 cntAllEnglish=0 eof=0 class=A avgMath=. avgEnglish=. StudentID=101 English=80
Math=70 FIRST.class=1 LAST.class=0 _ERROR_=0 _N_=1

at Last.Class and before output: sumMath=165 cntMath=2 sumEnglish=165 cntEnglish=3 sumAllMath=165
cntAllMath=2 sumAllEnglish=165 cntAllEnglish=3 eof=0 class=A avgMath=82.5 avgEnglish=55
StudentID=104 English=80 Math=95 FIRST.class=0 LAST.class=1 _ERROR_=0 _N_=3

at First.Class: sumMath=0 cntMath=0 sumEnglish=0 cntEnglish=0 sumAllMath=165 cntAllMath=2
sumAllEnglish=165 cntAllEnglish=3 eof=0 class=B avgMath=. avgEnglish=. StudentID=102 English=.
Math=60 FIRST.class=1 LAST.class=0 _ERROR_=0 _N_=4

at Last.Class and before output: sumMath=140 cntMath=2 sumEnglish=80 cntEnglish=1 sumAllMath=305
cntAllMath=4 sumAllEnglish=245 cntAllEnglish=4 eof=1 class=B avgMath=70 avgEnglish=80
StudentID=105 English=90 Math=80 FIRST.class=0 LAST.class=1 _ERROR_=0 _N_=5

at eof and before output: sumMath=140 cntMath=2 sumEnglish=80 cntEnglish=1 sumAllMath=305
cntAllMath=4 sumAllEnglish=245 cntAllEnglish=4 eof=1 class=T avgMath=76.25 avgEnglish=61.25
StudentID=105 English=90 Math=80 FIRST.class=0 LAST.class=1 _ERROR_=0 _N_=6

NOTE: There were 5 observations read from the data set WORK.EXAMPLE4.
NOTE: The data set WORK.SUMMARY_STATS has 3 observations and 7 variables.
```

Without the LENGTH statement, which appears just after the DATA STEP statement (and before the SET), the Log messages will show the variables in an undesired and confused order. For example:

```
at First.Class: eof=0 class=A avgMath=. cntAllMath=0 sumAllMath=0 avgEnglish=. cntAllEnglish=0
sumAllEnglish=0 StudentID=101 English=80 Math=70 FIRST.class=1 LAST.class=0 sumMath=0 cntMath=0
sumEnglish=0 cntEnglish=0 _ERROR_=0 _N_=1

at Last.Class and before output: eof=0 class=A avgMath=82.5 cntAllMath=2 sumAllMath=165
avgEnglish=55 cntAllEnglish=3 sumAllEnglish=165 StudentID=104 English=80 Math=95 FIRST.class=0
LAST.class=1 sumMath=165 cntMath=2 sumEnglish=165 cntEnglish=3 _ERROR_=0 _N_=3
```

BY-processing with multiple BY variables and with the merge of multiple data sets is much more complex. There are FIRST and LAST variables per BY variable. The PDV retains variables from multiple data sets following rules that take into account the order of the data sets, whether or not variables (other than those used in the BY-matching) are shared in the multiple data sets, etc. For a discussion of the PDV under SAS MERGE, please refer to the section "The MERGE and the Program Data Vector" in my paper "" referenced below.

CONCLUSION

SAS DATA Step processing is complex but follows clear rules which, if mastered, allow you take control over your programs and data. This paper describes some of these rules and gives examples that guide you in better understanding how and why the DATA Step behaves as it does and causes specific results to happen. Many other papers and the SAS on-line documentation offer additional insights A search of the SAS on-line documentation for key concepts mentioned here will lead you to additional sources which will further your understanding of how to cause SAS to do exactly what you want within the DATA Step.

NOTES

[1] Refer to the "SAS 9.1.3 Language: Reference: Concepts", chapter 20, DATA Step Processing

[2] Refer to Ian Whitlock's paper described in the references section below

REFERENCES

Howard, Neil (2003), *"How SAS Thinks or Why the DATA Step Does What It Does",* Proceedings of the 28th Annual SAS Users Group International Conference

Kahane, Dalia C (2011), *"*SAS® DATA Step Merge – A Powerful Tool*"*, Proceedings of the 2011 North East SAS Users Group Conference

SAS® 9.1.3 Language Reference: Concepts, Third Edition. 2005, SAS Institute Inc., Cary, NC, USA

Whitlock, Ian (2006), *"How to Think Through the SAS DATA Step"*, Proceedings of the 31st Annual SAS Users Group International Conference

Whitlock, Marianne (2007), *"The program Data Vector AS an Aid to DATA Step Reasoning"*, Proceedings of the 2007 North East SAS Users Group Conference

DISCLAIMER

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

ACKNOWLEDGEMENTS

I would like to thank Michael Raithel and Judy Walsh who reviewed my paper and provided comments and encouragement.

CONTACT INFORMATION

If you have any questions or comments about the paper, you can reach the author at:

Dalia Kahane, Ph.D.
Westat
1600 Research Blvd.
Rockville, MD 20850
Kahaned1@Westat.com