The SAS® System Supervisor - A Version 6 Update

Merry G. Rabb, SAS Consulting Services Inc.
Donald J. Henderson, SAS Consulting Services Inc.
Jeffrey A. Polzin, SAS Institute Inc.

ABSTRACT

This tutorial updates a talk given at previous SUGIs that discussed the functions of the SAS® System supervisor during the compilation and execution of a SAS DATA step program. Under Version 6 of the SAS System, the SAS System supervisor provides interface services at a higher level than before, and its primary activity in this context is to determine whether a DATA or PROC step is being invoked. Once invoked, the DATA step controls its own environment during both compile and execution phases. This tutorial presents a logical model describing both the compile and execution time actions of the DATA step processor.

INTRODUCTION

There is a distinct compile step and execution step for each DATA or PROC step in a SAS job. The DATA and PROC steps are compiled and executed independently according to their sequence in the program. In particular, the first DATA or PROC step is compiled and then executed, followed by the compilation and execution of the next DATA or PROC step, and so on. This tutorial concentrates on DATA step processing. Within the DATA step, the activities of the SAS System can be categorized as follows:

- compiling SAS DATA step source code
- optimizing the executable image
- executing the resultant machine code.

Gaining a complete understanding of how the user's DATA step code is controlled by the DATA step processor is crucial to using the SAS System more effectively. This tutorial first discusses the changes in the processing environment for Release 6.06 of the SAS System. Next, the processing activities within each of the phases listed above are addressed.

RELEASE 6.06 ENVIRONMENT

The SAS System supervisor controls the overall processing of a SAS job. In Version 5, the supervisor had much more control over the details of DATA step processing than in Version 6. Under Version 6, the supervisor determines whether to invoke the DATA step or a procedure. Once the supervisor has determined that a DATA step is to be invoked, it provides basic services such as parsing the input SAS statements as directed by the DATA step processor. The actual processing of a DATA step program, however, is controlled by the DATA step processor.

The DATA step processor is responsible for checking the syntax of the DATA step code, creating the executable image, and invoking that executable image to produce the desired output.

Access to SAS data sets is now controlled by the I/O engine supervisor. The I/O engine supervisor is a significant change from Version 5 in how data are accessed. The Version 5 concept of a SAS data set has been replaced by the Version 6 SAS data model.

DATA Step Processor

The Version 6 DATA step processor manages the working storage areas used by the DATA step program. Under Version 5, a variety of work areas, buffers, and flags were created and used to control execution time processing. Under Version 6, much of the work that was previously done at execution time is handled through the generation of machine code specific to the functions being performed and is discussed in **OPTIMIZATION OF EXECUTABLE IMAGE** later in this paper. In this way, performance is improved since the logical evaluations are performed at compile time once, and not at execution time for every input observation. The structure of the internal work areas has also been changed substantially. In Version 5, the Program Data Vector (PDV) was maintained as a contiguous storage area for all the variables referenced in a DATA step program. In Version 6, the PDV no longer exists as a physical entity. Instead, a logical PDV whose storage is not contiguous and that is optimized to improve execution time performance is utilized.

I/O Engine Supervisor and the SAS Data Model

Data access under Release 6.06 is supported by a new concept: the SAS data model. There are three main components of the SAS data model: data storage, data access, and indexes. These are briefly summarized in the following sections. The reader should refer to Chapter 6, SAS Files," of *SAS Language: Reference, Version 6, First Edition* for details.

Data Storage

In Version 5 of the SAS System, SAS data sets were physical structures that stored the data values and the descriptor information in the same file. Release 6.06 Multiple Engine Architecture makes it possible to use SAS data sets in two forms. In one form, the data values and descriptor information are obtained from the same file. In the other form, only the information necessary to derive the descriptor information and data values is stored in the file that represents the SAS data set. The descriptor information and data values are derived from other sources using this information. Within a DATA step, all access to a SAS data set is handled by the I/O engine supervisor.

A SAS data set is called a SAS data file if it is implemented in a form that contains both the data values and the descriptor information. When you obtain a list of files in a SAS data library, SAS data files have the member type data.

A SAS data set is a SAS data view if it is implemented in a form that obtains the descriptor information or data values, or both, from other files. Only the information necessary to derive the descriptor information or data values is stored in the file of member type view.

Data Access

In Version 6, data are accessed by engines. Engines are an interface between the user's program and the data; they isolate the program from the physical details of how the data are stored. Engines

are sets of internal instructions the SAS System uses to read from and write to files. Engines open files, direct input/output operations, and gather descriptive information about files and their contents.

Indexes

SAS data files can be indexed by one or more variables, known as key variables. A SAS index contains the data values of the key variable or variables paired with location identifiers for the observations containing the values. The value/identifier pairs are ordered in a B-tree structure that enables the engine to perform a binary search by key values. Therefore, by using an index, the SAS System can quickly locate the observations associated with a data value or range of data values. SAS indexes are classified as regular or composite, according to the number of key variables whose values make up the index.

The index data structure can be logically represented as illustrated in **Figure 1**. For each index, the observation numbers for data with the specified values of the index variable or variables can be quickly identified. Note that our sample data have three indexes.

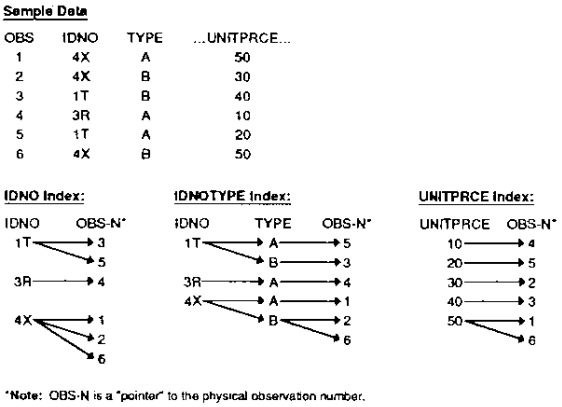

Figure 1 Index Data Structure

COMPILE TIME PROCESSING

During compilation, the DATA step directs the creation of both permanent and temporary (in that they "disappear" after the compilation or execution of the current DATA step) entities. The primary permanent entity is the directory or header portion of the output SAS data set, which the I/O engine supervisor creates at the direction of the DATA step (the data are added to the data set at execution time). The temporary entities include a variety of work areas that, at execution time, are used in the creation of the desired output. The following is a partial list of the more important actions taken during compilation of a DATA step:

- syntax scan
- translation from SAS source code to machine code
- definition of input and output files including variable names, their locations, and their attributes
- creation of the Logical Program Data Vector
- specification of variables to be written to the output SAS data set
- specification of variables that are to be initialized to missing between iterations of the DATA step and during read operations
- passing of information to the I/O engine supervisor, that determines the index to be used.

The last four actions in the above list are among the topics discussed in the following subsections.

Local Compile Time Statements

At compile time, two statements, BY and WHERE, affect how a program executes. These are local statements, meaning their location within the DATA step is critical to how they are interpreted. In Version 5 (under mainframe environments), the BY statement was global, meaning that it applied to any and all SET, MERGE, or UPDATE statements in the DATA step. In Version 6, the BY statement applies only to the last encountered SET, MERGE, or UPDATE statement. There are at least two implications from this behavior. First, it is no longer necessary for all the data sets referenced in every SET, MERGE, or UPDATE statement in the DATA step to contain the BY variables; only the last encountered SET, MERGE, or UPDATE statement must have data sets containing the BY variables. Second, a DATA step with a BY statement can also contain a SET statement with the POINT option as long as the BY statement follows a different SET, MERGE, or UPDATE statement. Like the BY statement, the WHERE statement is local and applies only to the last encountered SET, MERGE, or UPDATE statement.

Other Compile Time Statements

All of the following statements (nonexecutable or information statements) do all of their work at compile time:

- DROP
- KEEP
- LABEL
- RENAME
- RETAIN

Their location within the DATA step code is irrelevant; they can be placed anywhere within the DATA step program.

The following statements also do all of their work at compile time, but their placement is important since they will determine the attributes of any variable whose first reference is in one of these statements:

- ARRAY
- ATTRIB
- FORMAT/INFORMAT
- LENGTH

Creation of the Logical Program Data Vector

The Logical Program Data Vector (PDV) is a set of buffers that includes all variables referenced either explicitly or implicitly in the DATA step. It is created at compile time, then used at execution time as the location where the working values of variables are stored as they are processed by the DATA step program.

In Version 5 of the SAS System, the logical entity known as the PDV actually represented a contiguous data storage area. In Version 6, however, one area is used to store information on variable attributes and four other areas are set up to store data values for retained numeric, unretained numeric, retained character, and unretained character data. For the purpose of this tutorial, we use the term

Logical PDV to represent the functions performed by these memory segments.

Variables are added to the Logical PDV variable attribute area sequentially as they are encountered during the parsing and interpretation of SAS source statements. The following rules are used in defining the variables and their attributes to the Logical PDV:

1. A variable is added as it first occurs (explicit or implicit) in the SAS source statements.
2. DROP and KEEP statements and output data set name parameters are ignored for the purposes of adding variables to the Logical PDV.
3. The SAS automatic variables (for example, _N_ and_ ERROR_) are always added. The variable _I_ is added if the implicit form of the ARRAY statement is present in the DATA step program.
4. Variables can be implicitly referenced and thus added by SET, MERGE, or UPDATE statements. The DROP and KEEP data set name parameters used on an input data set affect which variables are added from that data set.
5. If a BY statement is specified for a SET, MERGE, or UPDATE statement, then a set of FIRST. and LAST. variables are added, corresponding to the variables in the BY statement.

The use of these rules is illustrated for a sample program in **Figure 2**.

Specification of Variables for Output

The specification of the list of variables to be copied from the Logical PDV to the output SAS data set is accomplished using the following rules:

1. All SAS special variables (for example, _N_, _ERROR_, END=, IN=, POINT=, FIRST. and LAST. variables, and implicit ARRAY indices) are dropped from the list of variables to be output.
2. If only DROP statements are present, all variables not listed in any DROP statement are kept.
3. If only KEEP statements are present, only variables listed in any KEEP statement are kept.
4. If both the DROP and KEEP statements are present, variables that appear in at least one KEEP statement and do not appear in any DROP statement are kept.
5. Variables referenced in DROP or KEEP statements but not defined in the Logical PDV generate error messages.
6. DROP= and KEEP= data set options apply logic similar to rules 2 through 4. However, variables referenced in a DROP= or KEEP= option but not present in the set of variables being processed do not generate error messages.

The use of these rules for the sample program is illustrated below for the sample program in **Figure 2**:

- Dropped special variables:
  - FIRST.IDNO
  - LAST.IDNO
  - FIRST.TYPE
  - LAST.TYPE
  - _ERROR_
  - _N_
- Dropped by the DROP statement:
  - UNITPRCE
  - QUANTITY
- Dropped from output data set SALES by the DROP= option:
  - DISCOUNT
  - SALESAMT
- Dropped from output data set DETAIL by the DROP= option:
  - TOTSALES
  - TOTDSCNT
- Final list of variables to be output for SALES:
  - TOTSALES
  - TOTDSCNT
  - IDNO
  - TYPE
- Final list of variables to be output for DETAIL:
  - IDNO
  - TYPE
  - SALESAMT
  - DISCOUNT

Initialization to Missing Values

The specification of the variables that are to be initialized to missing between each iteration of the DATA step program is accomplished when variables are assigned to one of four data storage areas. The four possible areas in which a variable can be stored are areas for unretained numeric, unretained character, retained numeric, and retained character variables. These assignments, like the specification of variables to be output, are defined at compile time and cannot be changed at execution time. The following categories of variables are assigned to retained storage areas:

1. all SAS special variables.
2. all variables listed in a RETAIN statement. If an array name is listed, it causes the list of variables that constitute the array to be retained. The following forces all variables into retained storage:

```
RETAIN;
```

3. all variables that are the accumulator variable (the variable to the left of the + sign) in a SUM statement.
4. all variables that are elements of an ARRAY statement, when the array name appears in the accumulator position

of a SUM statement. This is an enhancement from Version 5.

5. all variables read from data sets with a SET, MERGE, or UPDATE statement.

These rules are illustrated below for the sample program given in **Figure 2**:

- Retained numeric area:
  - _N_ (Rule 1)
  - _ERROR_ (Rule 1)
  - FIRST.IDNO (Rule 1)
  - LAST.IDNO (Rule 1)
  - FIRST.TYPE (Rule 1)
  - LAST.TYPE (Rule 1)
  - TOTSALES (Rules 2 and 3)
  - TOTDSCNT (Rules 2 and 3)
  - UNITPRCE (Rule 5)
  - QUANTITY (Rule 5)
- Unretained numeric area:
  - SALESAMT
  - DISCOUNT
- Retained character area:
  - IDNO (Rule 5)
  - TYPE (Rule 5)
- Unretained character area:
  - (none in this example)

Use of Indexes

The presence of a BY (without either the NOTSORTED or DESCENDING keywords) or a WHERE statement (or WHERE data set option) may result in the use of an index in accessing the data. The SAS programmer has no direct control over whether an index is to be used. If a BY statement dictates the use of an index, the data need not be sorted; use of an index forces the engine to retrieve the data sorted on the values of the index keys.

At compile time, the DATA step processor passes information on any BY statements and WHERE statements or options to the I/O engine supervisor, which considers whether to use an index in accessing the data. The I/O engine supervisor first determines which, if any, indexes are eligible for use by the BY statement and the WHERE statement or data set option.

The following rules determine the indexes eligible for use by the BY statement:

1. Exclude indexes constructed with the NOMISS option.
2. If a single BY variable was specified, and it matches a variable in a regular index or it matches the first variable in a composite key index, the index is eligible for use.
3. Otherwise if the BY statement contains more than one variable and the first one or more variables match an index, that index is eligible for use. If more than one index meets this criteria, the index with the most variables is used.

The following rules apply to the WHERE statement and data set option:

1. Exclude indexes constructed with the NOMISS option if the expression compares the index variables with a missing value, for example, UNITPRCE LE 10.
2. If one or more of the variables used in the expression has a simple index or is the first variable in a composite index, the index is potentially eligible. The index must select all observations that satisfy all conditions of the WHERE clause in order to be used.
3. A potentially eligible index is eligible if, based on a uniform distribution of values of the index variables, less than one third of the data satisfy the expression. This number is an approximation; it can vary depending on an internal costing algorithm. The index is used if the costing algorithm indicates that its use will be "cheaper" than sequential access. For example, if an index exists for UNITPRCE and the range of values is from 1 to 100, the index is eligible for

   ```
   WHERE UNITPRCE > 80;
   ```

   since it is expected that the index will select 20% of the data, but not for

   ```
   WHERE UNITPRCE > 40;
   ```

   since it is expected that the index will select 60% of the data.

Once index eligibility is determined, the following rules determine which index to use:

1. If both BY and WHERE are present, take the intersection to subset the eligible set.
2. If the intersection is empty, the eligible set contains the BY indexes.
3. If only one of the statements is present use its eligible indexes.
4. If more than one index remains
   a. If a BY statement is present, use the index with the most variables (largest value of *n*).
   b. Otherwise, when a WHERE statement is present, based on a uniform distribution of values, use the index that will select the smallest subset of data.

The use of these rules is illustrated below for the sample program:

Eligible BY:	IDNO IDNOTYPE
Eligible WHERE:	UNITPRCE
Intersection:	null
Index to Use:	IDNOTYPE

OPTIMIZATION OF EXECUTABLE IMAGE

The Version 6 DATA step compiler generates executable code differently from the Version 5 compiler. In general, the code generated by the DATA step compiler is more self contained, such that much less execution time intervention by the SAS DATA step processor is required. Functions that were previously handled by checking flags or looping through buffers at execution time are now incorporated into the generated executable code at compile time. Some examples of this optimization process are described in the following subsections.

Looping Control

During the DATA step code optimization, the compiler examines the code submitted by the programmer and generates the most efficient DATA step loop. The flow of the looping process is illustrated in **Figure 3**. If a LIST statement is issued or errors occur within DATA step code (that is, _ERROR_¬=0), control is transferred out of the DATA step code back to the DATA step execution supervisor for processing, and then control is returned to the DATA step code. Since the looping process is now entirely under the control of the DATA step, there is no longer an efficiency gain realized when all observations are read and written inside of an explicitly coded DO loop. The generated code produced by the compiler would be the same, whether or not a controlling DO loop were coded.

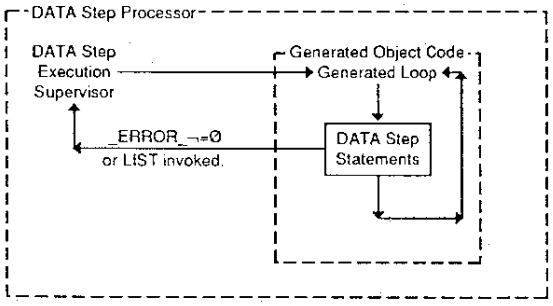

Figure 3 Compiler Generated DATA Step Loop

Initialize to Missing (ITM) Instruction

In Version 5, initialization of data values in the PDV to missing was handled by the SAS System supervisor, usually when control of processing was returned to the supervisor between passes of the DATA step program. In Version 6, the generated code includes an ITM (initialize to missing) instruction at the appropriate points in the code. In addition, since retained and unretained data values are stored in separate areas of the Logical PDV, execution of the ITM instruction does not require looping through all of the variables and checking to see which ones are retained and, therefore, should not be initialized to missing. Instead, whole blocks of adjacent memory, allocated for unretained variables, are initialized at once.

Handling of Explicit Versus Default Output

Incorporation of the appropriate output routine is now handled at compile time. In previous releases, an output statement present flag was created and set at compile time and then checked at execution time to determine whether or not a default output routine needed to be called at the end of each pass of the DATA step. In Version 6, the generated executable code will already contain the appropriate output routine, invoked at the appropriate point in the code, and no flag checking needs to be done at execution time.

Method of Identifying Next Observation to Read

The criteria by which the I/O engine supervisor determines whether or not an index is to be used was discussed in a previous section. Another action of the I/O engine supervisor at compile time is to set up a function pointer that indicates how the next observation is to be selected at execution time. One of four basic methods is identified based on information available at compile time.

If no index is to be used

1. and no WHERE expression is present
   - A sequential read is employed
2. and a WHERE expression is present
   - Do until the WHERE expression is true:
     1. Point to the next observation.
     2. Examine that observation to determine if the WHERE expression is true.

If an index is to be used

1. and no WHERE expression is present
   - Data are accessed using the index
2. and a WHERE expression is present
   - Do until the full WHERE expression is true:
     1. Do until the WHERE expression fragment using only indexed variables is met:
        - Point to the next observation based on the index.
        - Examine that observation to determine if the WHERE expression fragment is true.
     2. Examine the observation to determine if the remainder of the WHERE expression is true.

EXECUTION TIME ACTIVITIES

Once the DATA step has been successfully compiled and optimized, the execution phase of the DATA step can begin. In a typical SAS job, the logical flow of control is the iterative execution of machine code that:

1. initializes variables to missing.
2. executes the statements in the DATA step program.
3. outputs observations to a SAS data set, if not previously directed by the user; this is the default OUTPUT statement if no OUTPUT statement was compiled.

The details of what happens during the execution of the DATA step program (step 2 above) are controlled by the user in his or her SAS code. Steps 1 and 3 are handled directly by the generated machine code, which includes instructions to perform the initialization and output.

The diagram given in **Figure 4** presents the details of execution time processing:

1. During the INITIALIZATION phase, the generated machine code executes the ITM command to assign missing values to all variables stored in the unretained character and numeric portions of the Logical PDV.

2. The programming statements that compose the DATA step are executed, supplying values for the variables in the Logical PDV.

   a. When executing the read operation (for this generic case, assume a simple SET statement) the generated machine code invokes the I/O engine supervisor to select the next observation to read. The function pointer logic generated during the optimization of the executable image (discussed above) is used.

   b. If no more data, end the DATA step program, returning control to the SAS DATA step processor.

   c. Otherwise, copy the variables from the observation in the input data set to the Logical PDV, set the values of any appropriate special variables, and return control to the next executable DATA step statement immediately following the read operation statement.

3. At the end of the iteration of the DATA step, or whenever an OUTPUT statement is executed, the generated machine code instructs the I/O engine supervisor to copy the variables in the Logical PDV that are to be kept to the output SAS data set. If the COMPRESS option is on, adjacent repeating characters are compressed.

Read Operation Details

The SAS read operations SET and MERGE perform two general actions when executed:

- The ITM command is executed to initialize to missing the set of variables that will not be contributed by a read from the current data set. This operation is optimized such that in many cases nothing is performed.

- Copy variable values from one or more SAS data sets to the Logical PDV.

These actions are performed according to a set of rules, depending on which type of read operation is being performed and whether or not a BY statement is present.

The SET statement When a SET statement references more than one SAS data set, and no BY statement is present, the data sets listed in the SET statement are concatenated. The following actions are performed when the SET statement is executed:

1. Determine which data set is to be read and set IN= and END= variable values.

2. Identify the next observation to read.

3. If the SET statement will read from a different data set than was read on its last execution, the ITM command is executed to assign missing values for variables that will not be read from the current data set.

4. Copy the values of variables from the selected observation in the current data set to the Logical PDV, uncompressing the input if necessary.

When a SET statement referencing more than one SAS data set has a BY statement associated with it, the data sets listed in the SET statement are interleaved. The following actions are performed when the SET statement is executed:

1. Identify the next observation in each data set.

2. Select a data set to read by looking ahead to the values of the variables in the BY statement for the next observation in each data set. Set values for IN= and END= variables.

3. If the observation to be read is the first observation to be read for a new BY group, then do the following:

   a. Set the appropriate FIRST. variables to 1.

   b. Execute the ITM command to assign missing values to variables that will not be read from the selected data set.

4. If the SET statement will read from a different data set than was read on its last execution, regardless of whether the BY group changes, the ITM command is executed to assign missing values to variables that will not be read from the selected data set.

5. Copy variable values to the Logical PDV from the selected observation in the selected data set, uncompressing the input if necessary.

6. Identify the next observation in the selected data set.

7. Look ahead to the values of the variables in the BY statement for the next observation in each data set. If there are no more observations for this BY group, then set the appropriate LAST. variables to 1.

The MERGE statement When a MERGE statement with no BY statement is present, the observations in the data sets listed on the MERGE statement are merged one-to-one. The following actions are performed when the MERGE statement is executed:

1. Identify the next observation in each data set.

2. Copy variable values to the Logical PDV from the selected observation in the first data set listed in the MERGE statement, then the second data set, and so on, until all data sets have been read, uncompressing the input if necessary.

3. If end-of-file has been reached for a data set and no observation is read, the ITM command is executed to set missing values for variables unique to that data set.

4. Set IN= variables depending on which data sets are read.

When a MERGE statement with a BY statement is executed, the observations in the data sets listed are merged according to the values of the variables in the BY statement. The following actions are performed when the MERGE statement is executed:

1. Identify the next observation in each data set.

2. If all the observations to be read represent a new BY group, then do the following:

   a. Set the appropriate FIRST. variables to 1.

   b. Set all of the IN= variables to 0.

c. Execute the ITM command to assign missing values to variables that will not be read from the current data set.

3. For each data set listed in the MERGE statement having another observation for the current BY group, do the following:

   a. Set the appropriate IN= variable to 1.

   b. Copy variable values from the selected observation in the data set to the Logical PDV, uncompressing the input if necessary.

4. Identify the next observation to read for each data set and determine if any more observations are present for this BY group. If not, set the appropriate LAST. variable values to 1.

EXAMPLE PROGRAMS

Understanding the rules above can be helpful when writing or debugging a SAS program with complex DATA step code. The following sections discuss how the SAS programmer can take control from the SAS System supervisor within a DATA step, and how the rules governing the actions of the supervisor and of the read operations apply under those circumstances.

Reading SAS Data Sets within a Loop

The following program, in which the user has taken control away from the DATA step processor by placing the read operation statement inside a DO loop:

```
DATA TOSELL;
  DO UNTIL (LASTREC);
     SET INVSALES END=LASTREC;
     SALESAMT=UNITPRCE*QUANTITY;
     OUTPUT;
  END;
RUN;
```

will perform identically to this program:

```
DATA TOSELL;
  SET INVSALES;
  SALESAMT=UNITPRCE*QUANTITY;
RUN;
```

since during the optimization of the executable image, the DATA step processor created machine code that includes the automatic looping. Thus, unlike Version 5, in which the first program above used less resources, under Version 6 the relative performance of these two programs is the same.

The programs will perform differently if there are variables that are to be initialized to missing (that is, there are variables in the unretained storage areas). If the SAS program does its own looping, the generated machine code does not include an instruction to execute the ITM command, and so any variable that is set conditionally retains its value until it is reset explicitly by the DATA step program. If the DATA step processor generates the loop, the ITM command is executed at the appropriate time. Thus, the program need not be concerned with initialization. Note that statements that cause the current iteration of the DATA step loop to end (DELETE, RETURN, subsetting IF) will cause the issuance of the ITM command.

Conditional Execution of a Read Operation

Any SAS read operation can be executed conditionally. For example, suppose the programmer has a SAS data set with a single observation that contains a constant. This constant value is needed for each iteration of the DATA step. The SAS data set can be read by executing the SET statement only once, as shown below:

```
PROC MEANS DATA=SALES NOPRINT;
  VAR TOTSALES;
  OUTPUT OUT=OVERALL SUM=ALLSALES;
RUN;
DATA PERCENTS;
  SET SALES;
  IF _N_ = 1 THEN SET OVERALL;
  PERCENT = 100 * TOTSALES / ALLSALES;
RUN;
```

Since variables read from a SET statement are maintained in the retained storage area of the Logical PDV, they are not initialized to missing on subsequent iterations of the DATA step. Therefore a RETAIN statement listing the variables in the data set OVERALL is not necessary.

Referencing a Data Set at Compile Time

It is possible to reference SAS data sets during compilation exclusively. For example, suppose the programmer wants to add variables to the Logical PDV, even though these variables will not be read in this DATA step. This can be accomplished by executing the read operation based on a condition that will never be true. For example, the following program

```
DATA _NULL_;
  CALL SYMPUT('N_OBS',PUT(N_OBS,5.));
  STOP;
  SET TRANSALE NOBS=N_OBS;
RUN;
```

sets a macro variable whose value is the number of observations in data set TRANSALE. The SET statement is never executed because the IF 0 condition is never true. However, the value for the NOBS= variable (N_OBS) is supplied at compile time. The only executable statements in the DATA step are CALL SYMPUT and STOP.

CONCLUSION

By gaining a more complete understanding of what happens at compile and execution time, SAS programmers can make more informed decisions so that they can develop more flexible and efficient SAS programs.

The first two authors can be contacted at the following address:

SAS Consulting Services Inc.
Suite 330
1700 Rockville Pike
Rockville, MD 20852

Phone: (301) 881-8840

and the third author at

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

Phone: (919) 677-8000

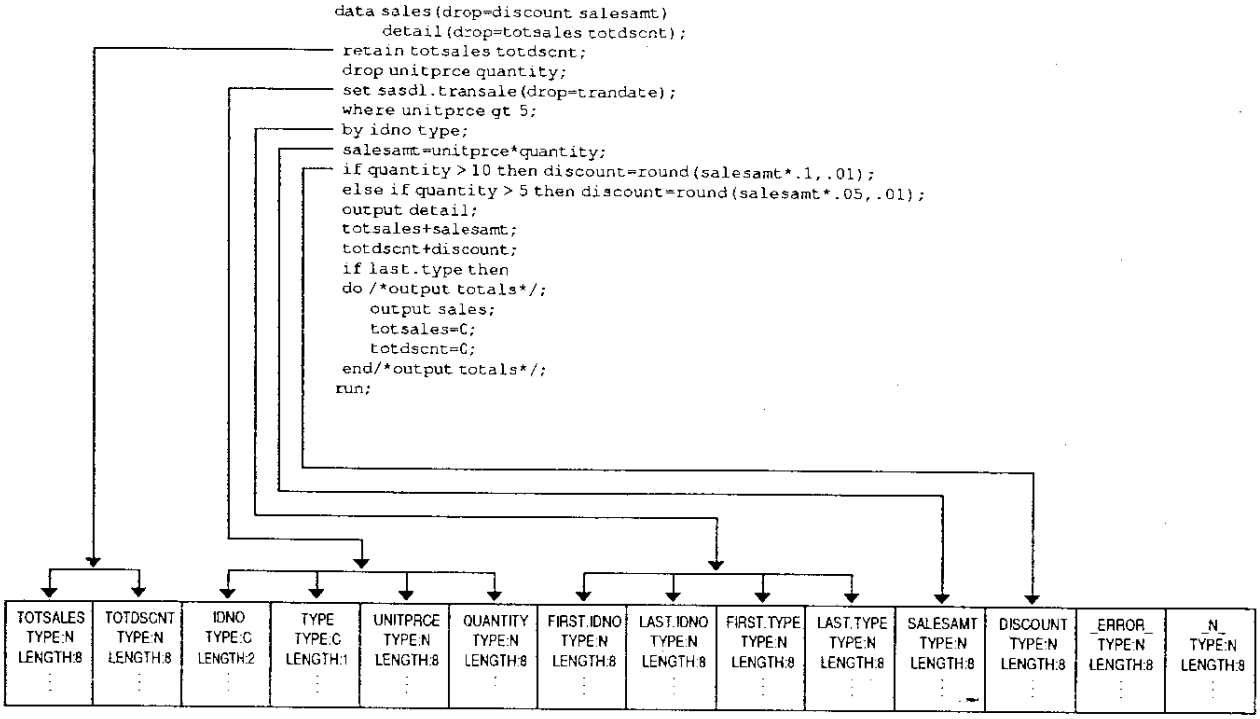

Figure 2 The Logical Program Data Vector

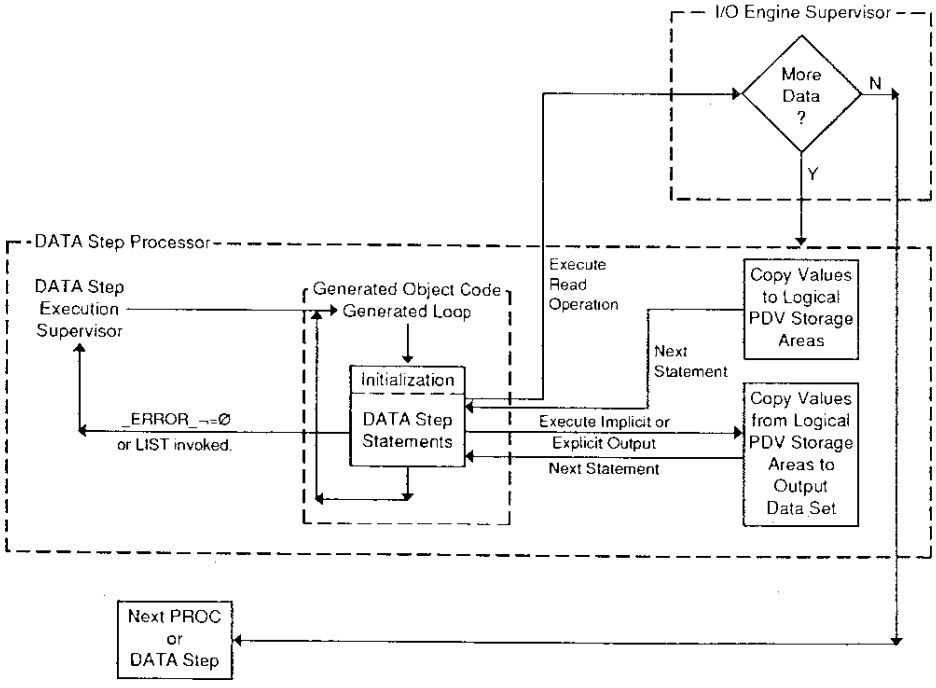

Figure 4 Execution Time Logic Flow