THE SAS SUPERVISOR AND SET/MERGE PROCESSING

Alman M. Zeid, ARC Professional Services Group, Inc.

I. INTRODUCTION

This tutorial discusses the functions of the SAS Supervisor as they pertain to the execution of SAS set and merge statements and is a combination of two papers presented in the Advanced Tutorial session of SUGI 13. There will be a general overview of the SAS Supervisor followed by advanced examples of set and merge processing. The six coded examples will focus on the activities of the Supervisor during data step execution.

II. SAS SUPERVISOR

The function of the SAS Supervisor can be categorized as follows:

- Compile SAS Source Code, and
- Execute Resulting Machine Code

When a SAS DATA Step program is written, the DATA Step "module" must be integrated within the structure of the SAS System. This integration is done by the SAS Supervisor. Gaining a more complete understanding of what the Supervisor does and how our "program" is controlled by it is crucial to using the SAS System most effectively.

There are distinct compile and execute steps for all SAS jobs. This fact is not readily apparent since in a single program, the SAS Supervisor handles the compile and execution (including linkages-editing) steps of a SAS job. 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/PROC step is compiled and then executed; this is then followed by the compilation and the execution of the next DATA/PROC step, etc. The SAS Supervisor controls this process.

One of the features of the SAS System is the fact that the execution of one step can affect the compilation of another. Furthermore, compile errors in later steps do not prevent earlier steps from executing. Distinguishing compile time activities from execution time activities is crucial in programming in SAS or any other language.

The SAS programmer has tools that allows him or her to take full advantage of the compile/execute structure for SAS jobs.

For example, through the use of the Macro Language, the programmer has control over the sequence of DATA/PROC steps seen by the Supervisor and the statements contained within each step. The following sections discuss the actions of the SAS Supervisor during compilation and execution of a DATA Step.

III. COMPILe TIME ACTIVITIES

During the compilation of a DATA Step, the Supervisor creates both permanent and transient (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 SAS data set (the data is added to the data set at execution time). The transient entities include a variety of buffers, flags and work areas which, at execution time, control the creation of the desired output. The following is a partial list of the more important actions taken by the SAS Supervisor during the compilation of a DATA Step:

- Syntax scan;
- Translation from SAS source code to machine language object code;
- Definition of input and output files including variable names, their locations and attributes;
- Specification of variables to be written to the output SAS data set;
- Creation of the Program Data Vector;
- Specification of variables which are to be initialized to missing by the SAS Supervisor between executions of the DATA Step and during read operations; and
- Creation of a variety of "flag variables" which are used by the Supervisor at execution time.

The last three actions in the above list will be discussed in the following subsections.

A. Creation of the Program Data Vector

The Program Data Vector (PDV) is a buffer which includes all variables referenced either explicitly or implicitly in the DATA Step. It is used at execution time as the location where the working values of variables are stored as they are processed by the DATA Step "program". The PDV is created at compile time by the SAS Supervisor. Variables are added to
the PDV 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 PDV:

1. A variable is added to the PDV by its first occurrence (explicit or implicit) in the SAS source statements.

2. DROP and KEEP statements and output data set name parameters are ignored for the purpose of adding variables to the PDV.

3. The SAS automatic variables (_N_ and _ERROR_) are always added.

4. Variables can be implicitly referenced and thus added to the PDV through SET, MERGE, or UPDATE statements. Variables referenced in this way are added to the PDV when the "read" statement is encountered at compile time, regardless of whether or not the statement is ever executed. The DROP and KEEP data set name parameters used on an input data set affect which variables are added to the PDV from that data set.

B. Initialization to missing values

Initializing variables to missing takes place based on the value of the ITMV (Initialize To Missing Vector). The SAS Supervisor creates a buffer containing ITMVs. There is a one-to-one correspondence between PDV variables and ITMVs. The value of each vector is defined at compile time and cannot be changed at execution time. These vectors can take one of the following three possible values:

**Y** means initialize to missing between each execution of the DATA Step.

**N** means do not initialize to missing.

**R** means that the read operation (i.e., SET, MERGE, or UPDATE) will perform the initialization to missing process. This value is only used when multiple data sets are being read.

All variables which are referenced in SET, MERGE or UPDATE statements will have ITMV values set to "N", or "R" according to the following rules:

1. ITMV values are set to "N" for variables read from a single SAS data set with the SET statement.

2. Where two or more data sets are read with a SET, MERGE or UPDATE statement, ITMV values for the variables from those data sets are set to "R".

C. Process Control Flags

In addition to the above buffers or vectors, other flag variables are created during the compile phase of a DATA Step program. The Data Step Failed Flag (DSFF) and the End Data Step Flag (EDSF) are created at compile time; their values are supplied at execution time. The Output Statement Present Flag (OSPF) is created and supplied a value at compile time. OSPF is set to "Y" if there is any output statement present in the DATA Step program, otherwise it is set to "N". The DSFF, EDSF and OSPF are used by the SAS Supervisor during the execution phase to control DATA Step processing. The purpose of these flags will become clear in the discussion of execution time activities.

IV. EXECUTION TIME ACTIVITIES

Once the DATA Step has been successfully compiled and all of the above described buffers and flags have been created, the execution phase of the DATA Step can begin. This is illustrated by the simple Execution Time Flow Diagram in Figure 1. The SAS DATA Step can be viewed as a subroutine which is executed repeatedly by the SAS Supervisor until there is no more input data. In a typical SAS job, the Supervisor does the following:

1. Initialization of variables in the PDV to missing.

2. Execution ("calling") of the DATA Step program.

3. Outputting or copying values of variables in the PDV to the output SAS data set.

4. Repeating steps 1-3 until the input data source is exhausted.

The details of what happens during the execution of the DATA Step program (the second step in FIGURE 1) is controlled by the SAS code within the DATA step.

A. Ceasing Data Step Execution

On referring to Figure 1, the question arises as to how the SAS Supervisor knows when to stop executing the DATA Step program. The more Detailed Execution Time Flow Diagram given in Figure 2 is a more accurate representation of execution time processing, which can be described
as follows:

1. During the INITIALIZATION phase, set the values of DSFF and EDSF to "N".

2. Execute the DATA Step program, statement by statement.
   2.1 When executing the read operation (for this "generic"
      case, assume a simple SET or INPUT statement) call a
      Supervisor routine to:
      2.1.1 Determine if there is more input data.
      2.1.2 If no more data, set DSFF and EDSF to "Y" and skip the
           rest of the DATA Step program, returning control
           to the SAS Supervisor.
      2.1.3 Otherwise, copy the variables from the input data set to
           the PDV, set the values of any appropriate special vari-
           ables and return control to the next executable DATA Step
           statement immediately follow-

2. If OSPF="N" and DSFF="N" then
   execute the OUTPUT Routine.

4. If EDSF="Y" then end the DATA Step
   and proceed to the next DATA or
   PROC step. Otherwise, repeat the
   above steps.

FIGURE 1. Execution Time Flow

FIGURE 2. Detailed Execution Time Flow
In writing a SAS DATA Step program, it is crucial to keep in mind the statements that return control to the Supervisor and how they impact the values of the DSFF and EDSF flags. Execution of the following statements all cause an immediate return to the SAS Supervisor with the indicated values for the flags:

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>DSFF</th>
<th>EDSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DELETE</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IF false &lt;expression&gt;</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>RETURN</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>STOP</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Failed read operation</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

(i.e. INPUT, SET, MERGE or UPDATE)

B. Read Operation Details

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

- Call a SAS Supervisor routine to initialize selected variables in the PDV to missing.
- Copy variable values from one or more SAS data sets to the 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. These rules also control the values of the following variables:

IN=, END=, FIRST=, and LAST.

The remainder of this paper will focus on how the programmer can utilize this understanding of the SAS Supervisor to control the processing of data within the DATA step. Specifically, the examples will reinforce the concepts with respect to the SET and MERGE statements.

V. Coding Examples:

In the following sections each of these topics will be covered:

A) Every SET statement in a data step activates its own READ pointer.
B) The FIRSTOBS option on a SET statement can be used to perform a LOOK AHEAD MERGE.
C) The IN = variables can be reset by the programmer.
D) The POINT option on a SET statement can be used to read a specific observation, or group of observations.
E) The NOBS option on a SET statement is available for use at compile time.
F) A JOIN, or ALL COMBINATIONS MERGE can be achieved by creating POINTER data sets and using SET with the POINT option.

A. EACH SET STATEMENT ACTIVATES ONE READ POINTER

Problem: A program is required to update an employee master file, with all current employees, using a transaction data set of new employees. The employee ID, which is the observation number, is used by other applications as a pointer to access records directly. Therefore, when an employee leaves the company, that employee ID needs to be used again.

```
MASTER FILE
EMPNO EMPNAME DEPT GROSS
1 JONES 901 95000
2 903 29700
3 SMITH 902 47500
4 902 35900
5 BROWN 901 17275
```

```
NEW EMPLOYEE FILE
EMPNAME DEPT GROSS
TAYLOR 903 32600
JOHNSON 904 41900
GREEN 901 39200
HARRIS 902 21800
```

Solution: Figure A.1 contains the two SAS data sets, MASTER and NEWEMPS. Figure A.2 shows the SAS code used to solve this problem. The MASTER data set is read and if the employee name is missing, the NEWEMPS data set is read. When the end of the MASTER data set is reached, the rest of the NEWEMPS data set will be read until it is empty. One physical SET statement is used to read NEWEMPS, activating only one read pointer, thereby reading the NEWEMPS data set sequentially. As seen in Figure A.2 this goal can be achieved through the use of a LINK - RETURN. The NEWEMPS data set is read once from the first observation to the last, as desired. The output can be seen in Figure A.3.

The two read operations in the data step are inside DO LOOPS; therefore the WARNING message 'Program stopped due to
looping' would appear in the program log. To prevent this WARNING message, an explicit STOP is coded which will set the EDSF to 'Y' and stop the DATA step. Without the STOP statement, the SAS Supervisor would execute the DATA step again, perform no read operation, and end the step and generate the message.

FIGURE A.3 shows the NEWMASTR data set. The OUTPUT statements are required in these two DO LOOPS, otherwise the last record to be read would be the only record output to the NEWMASTR data set. By default, records are output when control is returned to the SAS Supervisor and in this case the values in the PIV would be those of the last record. This example illustrates two major points. Each SET statement activates one read pointer and is treated by the SAS Supervisor as ‘separate but equal’. To force a data set to be read sequentially when a read operation is desired in multiple locations within a program, only one physical SET statement can be present.

B. LOOK AHEAD MERGE WITH FIRSTOBS OPTION

Problem: The owner of a small florist shop would like to see a report showing the difference in income from month to month. The report should show the difference in income between February and January and then the difference between March and February, etc. The input data can be seen in FIGURE B.1.

FIGURE B.2 shows that it is necessary to remove the first observation from the data set (the January record) and "combine" that data with the data from the second observation (February) in the same execution of the DATA step. The first execution of the DATA step reads observations one and two; the second execution reads observations two and three; etc. This is accomplished by using the FIRSTOBS option on the SET statement. This option tells the SAS Supervisor where to begin reading the data set. Note that the variables in one of the SET statements must be RENAMED to prevent the values from being overwritten in the Program-Data Vector. The resulting data set 'COMPARE' can be seen in FIGURE B.3. Again there are two read pointers activated to the same data set and each will read the same data set independently. The read pointer associated with the second SET statement will reach the end of the data set first because the FIRSTOBS option forced the read pointer to start with the second observation. Therefore, the End of Data Step Flag will be set to 'Y' when the second SET statement fails the read operation. Also, note that the two SET

SALES DATA SET

MONTH  DATA SET  TOTAL

JAN  5000
FEB  17000
MAR  9000
APR  13000
MAY  20000
JUNE 15000
JUL  8000
AUG  8000
SEP  11000
OCT  16000
NOV 12000
DEC 14000

FIGURE B.1
statements act like a one to one merge and combine the two records into one observation in the new data set COM dissolution.

```
DATA COM Dissolution (KEEP = MONTH DIFFER);
  SET SALES (DROP = MONTH)
      (RENAME = (TOTAL = LASTTOT));
  SET SALES (FIRSTOBS = 2);
  DIFFER = TOTAL - LASTTOT;
RUN;
```

**FIGURE B.2**

```
COMPARE DATA SET
MONTH DIFFER
FEB 12000
MAR -6000
APR 4000
MAY 7000
JUNE -10000
JUL 9
AUG 3000
OCT 5000
NOV -4000
DEC 2000
```

**FIGURE B.3**

With a merge, however, the read operation would not fail until both data sets are empty.

C. RESETTING OF IN = VARIABLES.

Problem: A national department store chain keeps the information about the products sold in each department in separate SAS data sets. A report is to be generated that combines all the data sets to show the products carried in each department during each month. **FIGURE C.1** shows some of the input data from two departments.

Solution: **FIGURE C.2** shows the SAS code used to solve this problem. By examining the input data in **FIGURE C.1**, it is apparent that this is a many to many merge. It is necessary to set either PRODUCT or CPRODUCT to missing when the input data are retained rather than read from the respective data set to prevent the repetition of PRODUCT "potting soil" in February and CPRODUCT "sandals" in March. This objective can be accomplished by using IN = variables for both the input data sets and setting

```
DATA REPORT;
  MERGE GARDEN (IN = INGARD)
       CLOTHING (IN = INCLOTH)
      (RENAME = (PRODUCT -CPRODUCT));
      BY MONTH;
      IF INGARD = 0 THEN PRODUCT = "";
      IF INCLOTH = 0 THEN CPRODUCT = "";
      INGARD = 0;
      INCLOTH = 0;
RUN;
PROC PRINT DATA = REPORT;
  BY MONTH;
  ID MONTH;
  TITLE 'REPORT DATA';
  LABEL PRODUCT = "GARDEN"
      CPRODUCT = "CLOTHING";
RUN;
```

**FIGURE C.2**

**FIGURE C.3**

the IN = variables, unconditionally, to zero for every execution of the DATA step. **FIGURE C.3** shows the output of this program.
The IN = variables are initialized to 0 at the beginning of the DATA step. When input data comes from the associated data set, the IN = variable is set to 1 and its value retained until the next BY group. The important point to remember is that because the input data is read with a MERGE statement the IN = variable is not reinitialized to 0 until the BY group changes. The program must take control of the IN = variables by resetting them to 0 for every record, to determine if an observation from the input data set was read or retained. Then, the IN = variables only have a value of 1 when an input record is actually read from the associated data set. The IN = variables are in the Program Data Vector and thus are available for the programmer to use, although they are not output to the new data set.

D. POINT OPTION ON A SET STATEMENT TO READ A SPECIFIC OBSERVATION

Problem: To aid in budget planning for the upcoming year a report must be generated that shows the percentage of the annual budget that was spent for each month during the previous year. FIGURE 0.1 shows the input data set with the MONTH, AMOUNT and a cumulative, yearly to date total called CUM. To compute the annual percentage, the value CUM from the last record (the annual total), must be available for every execution of the DATA step.

<table>
<thead>
<tr>
<th>EXPENSES DATA SET</th>
<th>MONTH</th>
<th>AMOUNT</th>
<th>CUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN</td>
<td>5000</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>JAN</td>
<td>7000</td>
<td>12000</td>
<td></td>
</tr>
<tr>
<td>JAN</td>
<td>9000</td>
<td>21000</td>
<td></td>
</tr>
<tr>
<td>FEB</td>
<td>13000</td>
<td>34000</td>
<td></td>
</tr>
<tr>
<td>FEB</td>
<td>20000</td>
<td>64000</td>
<td></td>
</tr>
<tr>
<td>FEB</td>
<td>19000</td>
<td>73000</td>
<td></td>
</tr>
<tr>
<td>FEB</td>
<td>8000</td>
<td>81000</td>
<td></td>
</tr>
<tr>
<td>FEB</td>
<td>8000</td>
<td>89000</td>
<td></td>
</tr>
<tr>
<td>FEB</td>
<td>21000</td>
<td>110000</td>
<td></td>
</tr>
<tr>
<td>MAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>11000</td>
<td>590000</td>
<td></td>
</tr>
</tbody>
</table>

Solution: The code in FIGURE D.2 demonstrates how this can be achieved. On the first execution of the DATA step only, a SET statement is executed using the POINT and NOBS options to read the LAST record. The value of NOBS is set in the Program Data Vector at compile time. Thus, when this SET statement executes, the number of observations in the data set is available in the PDV as the value of NUM_OBS and the POINT option can be used to point specifically to the last record. Notice also that the variable CUM is RENAMED to ANNUAL. Therefore, the variable ANNUAL is present in the Program Data Vector and is never reinitialized to missing because it was read with a SET statement that is never executed again.

<table>
<thead>
<tr>
<th>MONTHLY DATA SET</th>
<th>EXPMONTH</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN</td>
<td>3.55%</td>
<td></td>
</tr>
<tr>
<td>MAR</td>
<td>5.92%</td>
<td></td>
</tr>
<tr>
<td>APR</td>
<td>6.95%</td>
<td></td>
</tr>
<tr>
<td>MAY</td>
<td>13.15%</td>
<td></td>
</tr>
<tr>
<td>JUNE</td>
<td>12.50%</td>
<td></td>
</tr>
<tr>
<td>JUL</td>
<td>5.26%</td>
<td></td>
</tr>
<tr>
<td>AUG</td>
<td>5.26%</td>
<td></td>
</tr>
<tr>
<td>SEP</td>
<td>7.24%</td>
<td></td>
</tr>
<tr>
<td>OCT</td>
<td>6.49%</td>
<td></td>
</tr>
<tr>
<td>NOV</td>
<td>7.89%</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>9.21%</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE D.1

FIGURE D.2

FIGURE D.3
Thus, ANNUAL is made available for every record and the monthly percentage can easily be calculated. Note also that the POINT option on the first SET statement precludes the use of a BY statement, therefore FIRST or LAST processing must be simulated with the LAG function. FIGURE D.3 shows the final report and the desired results.

E. THE USE OF THE NOBS OPTION AT COMPILE TIME.

Problem: Every evening a program is to run to update a master file. There is extensive editing code to verify that the records to be modified, added or deleted are valid. A complex update data step should be executed only if the editing step produced valid records.

Solution: FIGURE E.1 shows the macro used to solve this problem. The NOBS variable is added to the PDV and initialized at COMPILE time. The null DATA step creates a macro variable (&N_OBS) with its value set to the number of observations in the GOODRECS data set. At COMPILE time the value of the data step variable, _N_OBS, is initialized in the PDV. At execution time the SET statement will never execute but the program still creates the macro variable &N_OBS with the SYMPUT function. The update step is only executed if there are valid records in GOODRECS data set.

F. JOIN - ALL COMBINATIONS MERGE.

Problem: A company has a benefit program that provides employees with improved benefits as the employee is promoted. As can be seen in FIGURE F.1, an employee in CATEGORY 1 has four potential benefits while an employee in CATEGORY 5 has only two potential benefits. In the EMPLOYEE data set, FIGURE F.2, each employee has a CATEGORY associated with their position. It is necessary to produce a report that will show each employee the value of their specific benefits for the previous year.

<table>
<thead>
<tr>
<th>BENEFITS DATA SET</th>
<th>EMPLOYEE DATA SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS CATEGORY BENEFIT VALUE</td>
<td>EMPNAME CATEGORY HOURLY</td>
</tr>
<tr>
<td>1 1 DENTAL 200.00</td>
<td>MANN 1 25.00</td>
</tr>
<tr>
<td>2 1 FULL MEDICAL 1000.00</td>
<td>SMITHSON 2 23.00</td>
</tr>
<tr>
<td>3 1 BONUS 2000.00</td>
<td>JAMES 2 21.00</td>
</tr>
<tr>
<td>4 1 4 WKS VACA 4000.00</td>
<td>HOUSEMAN 3 20.00</td>
</tr>
<tr>
<td>5 2 FULL MEDICAL 1000.00</td>
<td>GARRETT 3 19.00</td>
</tr>
<tr>
<td>6 2 BONUS 1500.00</td>
<td>ROBBINS 4 20.00</td>
</tr>
<tr>
<td>7 2 3 WKS VACA 2760.00</td>
<td>STILES 4 18.00</td>
</tr>
<tr>
<td>8 3 FULL MEDICAL 1000.00</td>
<td>HOWELL 4 16.00</td>
</tr>
<tr>
<td>9 3 3 WKS VACA 2400.00</td>
<td>DUBOIS 4 15.00</td>
</tr>
<tr>
<td>10 4 HALF MEDICAL 500.00</td>
<td>JENKINS 5 12.00</td>
</tr>
<tr>
<td>11 4 3 WKS VACA 2400.00</td>
<td>PAYNE 5 10.00</td>
</tr>
<tr>
<td>12 5 2 WKS VACA 1500.00</td>
<td></td>
</tr>
</tbody>
</table>

Solution: A simple merge would work in this case if there were one benefit or one employee for each category. If there are multiple benefits for any category or multiple employee records for any category, however, the merge would not combine the BENEFITS and EMPLOYEE data sets in the desired manner. To produce the desired output, a JOIN or ALL COMBINATIONS (for the BY group) MERGE must be performed. This is accomplished by creating a pointer data set for the BENEFITS data set. This pointer data set contains one record for each value of CATEGORY and two variables whose values

```bash
%MACRO UPDAILY;
  DATA GOODRECS;
    ( edit code )
  RUN;
  /% records to process?? %/
  DATA _NULL_;
  IF 0 THEN SET GOODRECS;
  POINT = _N_ NOBS = _N_OBS;
  CALL SYMPUT ( ’N_OBS’ ,
      PUT ( _N_OBS ,5.));
  STOP;
  RUN;
  %IF &N_OBS > 0 %THEN
  XDO; /% exec if records %/
  DATA MASTER;
    ( update code )
  RUN;
  %END;
  %MEND UPDAILY;
```

FIGURE E.1
are the first and last observation numbers for the given CATEGORY in the BENEFITS data set. FIGURE F.3 shows the SAS code used to create the PNTRBEN data set. FIGURE E.4 shows the resulting data set.

CREATE POINTER DATA SET FOR THE BENEFITS DATA SET
DATA PNTRBEN (KEEP = STARTB STOPB CATEGORY);
RETAIN STARTB;
SET BENEFITS;
BY CATEGORY;
IF FIRST.CATEGORY THEN STARTB = \_;
IF LAST.CATEGORY THEN DO;
STOPB = \_;
OUTPUT;
END;
RUN;

FIGURE F.3

The pointer data set PNTRBEN will be used in reading all the benefit records for each CATEGORY from the BENEFITS data set.

FIGURE F.4

The pointer data set PNTRBEN will be used in reading all the benefit records for each CATEGORY from the BENEFITS data set.

FIGURE F.5

The SAS Supervisor controls the default actions taken in SAS DATA step processing. By gaining a more complete understanding of what the SAS Supervisor is doing at both compile and execution time, SAS programmers can make more informed decisions as to what they can let the SAS Supervisor do and what they should control themselves. Taking

VI. CONCLUSION

The SAS Supervisor controls the default actions taken in SAS DATA step processing. By gaining a more complete understanding of what the SAS Supervisor is doing at both compile and execution time, SAS programmers can make more informed decisions as to what they can let the SAS Supervisor do and what they should control themselves. Taking regardless of the application while the actual implementation is application specific.
control of the default activity of the SAS Supervisor was demonstrated with several programming examples focusing on SET and MERGE processing.

ACKNOWLEDGMENTS:
The input and assistance of the following people was greatly appreciated:
Ian Whitlock, Katie Hubbell and Kita Garrido.

References:


* SAS is a registered trademark of SAS Institute, Inc.

The author can be contacted at:

SOFTWARE APPLICATIONS AND TRAINING OPERATION
ARC PROFESSIONAL SERVICES GROUP
SUITE 501
501 INDIANA AVENUE, N.W.
WASHINGTON, D.C. 20004

TELEPHONE: (202) 737-2666