Paper 235

Programming Idioms Using the SET Statement

Jack E. Fuller, Trilogy Consulting Corporation, Kalamazoo, MI

ABSTRACT

While virtually every programmer of base SAS® uses the SET statement, surprisingly few programmers know what it actually does. Even fewer programmers use the full range of its potential at both compile and execution times. This paper provides both a basic introduction to the SET statement and a series of programming idioms for solving common and not so common problems using the SET statement.

INTRODUCTION

At first glance, the SET statement may appear to be a simplistic programming statement and quite unworthy of further analysis. However, I have found in the course of my career that many SAS programmers do not understand exactly what this command does. Additionally, many programmers do not possess within their repertoire a set of programming idioms that use the SET statement for solving recurring problems. A programming idiom “describe[s] how to solve implementation-specific problems in a programming language” (see Buschmann, *et. al.*, p. 346).

This paper explores the SET statement by providing idioms that can be used in the course of problem solving: first, it provides an introduction to the basic SET statement; second, it provides a set of basic idioms for using the features of the SETstatement; third, it provides several more advanced idioms for using the SET statement to solve specific problems.

THE SET STATMENT

Understanding the Program Data Vector (PDV)

The key to understanding how the SET statement works is to first understand what the PDV is and how it works. While a complete examination of the PDV is beyond the scope of this paper, an excellent primer can be found in "The SAS System Supervisor -- A Version 6 Update" by Henderson, *et al.* which contains the following definition of the PDV:

> *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 the course of their paper, Henderson, *et. al.* make several additional points which directly impact the techniques we will be using:

- The SAS system supervisor adds a variable to the PDV as it first occurs in the source code
- This logical PDV itself is broken into four logical areas as follows:

	Retained	***Unretained***
Numeric	X	X
Character	X	X

- All SAS special variables (including variables from the SET statement options: NOBS=, END=, POINT=) are placed into the appropriate **retained** area of the PDV; these variables are not written to output datasets
- Variables that are added by SET, MERGE, and UPDATE statements are placed into the appropriate ***retained*** area of the PDV

Understanding SET Statement Execution

In order to demonstrate what the SET statement does during execution, consider what will be written by the following example to the SAS log:

Understanding the Basic SET Statement

```
DATA whatever;
   DO some_var=1 TO 3;
      OUTPUT;
   END;
RUN;
```

```
DATA _null_;
   PUT some_var=;
   SET whatever;
   PUT some_var=;
RUN;
```

Log of "The Basic SET Statement"

```
1    SOME_VAR = .
2    SOME_VAR = 1
3    SOME_VAR = 1
4    SOME_VAR = 2
5    SOME_VAR = 2
6    SOME_VAR = 3
7    SOME_VAR = 3
```

1. While the PDV has been created by the time this output is generated, no observations have yet been loaded into the PDV from the dataset WHATEVER.

2. The first observation has been loaded.

3. Because variables placed into the PDV from a SET statement are placed into a retained area, the PDV still contains the value of SOME_VAR from the first observation in WHATEVER.

4. The second observation has now been loaded.

5. The PDV still contains the values from the second observation.

6. The third and final observation has now been loaded.

7. The PDV still contains the values from the third observation. Additionally, a datastep that contains a SET statement which is not using a random access option (i.e. POINT=, KEY=) ends when the SET statement is called and there are no more observations to be read from the dataset.

BASIC IDIOMS

Using Unexecuted SET Statements

Just as the last example demonstrated what occurs during execution, the following examples show what occurs during the compile phase of a datastep. By placing statements such that they are compiled but never executed, this example demonstrates how to add variable definitions from one dataset to another.

Add Variables from a Master Dataset

```
DATA whatever;
1  IF (0) THEN
2     SET master;
   SET whatever;
RUN;
```

1. This statement will always evaluate to false.

2. Since this statement will never execute, its only influence upon the datastep occurs during the compile phase of the datastep. In this case, it will add all of the variables in MASTER to WHATEVER. It will also override the attributes of variables in both datasets with the attributes from MASTER.

Another example of using a non-executing SET statement occurs when there is a need for one variable to hold the value of another.

Create a Variable That Has the Correct Dimensions to Hold Intermediate Values

```
1DATA whatever (DROP=_:);
   IF (0) THEN
2     SET whatever (KEEP=some_var
         RENAME=(some_var=_temp));
   SET whatever;
   /*  Place  code  here  using  _TEMP
*/

   _temp = some_var;
RUN;
```

1. Since we will be using the convention of prefixing intermediate variables with an underscore, the statement DROP=_: will drop all intermediate variables in one step. This naming convention also helps to prevent naming collisions between different datasets.

2. Since the variable TEMP comes from a SET statement, it will be placed in a retained area of the PDV. There is no need for a RETAIN statement. Also, if SOME_VAR is a character variable, TEMP will inherit the correct length so that no bytes are truncated.

Using the NOBS= Option

The NOBS= option is used to place the number of observations in the dataset into the referenced variable. Similar to previous examples in that SET statements are situated so that they are not actually executed, retrieving the number of observations in a dataset does not require that the SET statement actually execute. This is because the number of observations is placed into the appropriate variable at compile time.

Retrieve the Number of Observations into a Macro Variable

```
DATA _null _;
1  CALL SYMPUT ('nobs',
      COMPRESS (PUT (n_obs,
      best.)));
2  SET whatever (drop=_all obs=0)
      NOBS=n_obs;
RUN;
```

1. Before execution of the SET statement, the variable N_OBS has already been set to the correct value.

2. The statement *OBS=0* will cause this statement to never actually read any data into the PDV.

Another useful example using the NOBS= option in a SET statement occurs when the number of observations is required from a dataset other than the one which being currently processed.

Retrieve the Number of Observations from Another Dataset

```
DATA whatever;
   IF (0) THEN
1     SET other_ds (DROP=_all_)
         NOBS=n_obs;
   SET whatever;
   /* Place code here using N_OBS
*/

RUN;
```

1. This SET statement does not add any variables to WHATEVER due to the DROP=_ALL_ statement. However, it does correctly return the number of observations in OTHER_DS.

Using the END= Option

The END= option is used to set a flag when the last observation has been read from the dataset. A common mistake when using the END= option is to misplace its use to the end of the datastep; on the contrary, the normal case is to place the code which process the END= prior to the SET statement in order to handle the case when the dataset is empty.

Place END= Operations Before SET Statements

```
DATA whatever;
1  IF (at_end) then do;
      /* Place code here */
   END;
   SET whatever END=at_end;
RUN;
```

1. Even with an empty dataset, this code will always execute.

Using the POINT= Option

The SET statement contains two mutually exclusive options for random access: the POINT= and the KEY= options. The POINT= option allows a dataset to be accessed by observation number based upon the current value of the referenced variable. It is not so much poorly used as it is rarely used. One example of its use is when the need arises to reverse the observations in a dataset without the aid of using a PROC SORT.

Reverse the Observations in a Dataset

```
DATA whatever;
1  DO point=nobs TO 1 BY -1;
2     SET whatever POINT=point
         NOBS=nobs;
3     OUTPUT;
   END;
4  STOP;
RUN;
```

1. Loop through the observations in the dataset from the end to the beginning.

2. Retrieve the next observation by using the random access POINT= option.

3. Write out the PDV.

4. Since this example is using random instead of sequential access, the datastep must be explicitly stopped.

Using the KEY= Option

The KEY= option is used to perform table lookup on a dataset using the current values of the variables specified in the referenced index. For example, the following code example assumes a simple index on NAME is attached to the LOOK_UP dataset. At the time of the SET statement's execution, the NAME variable determines which observation is read from LOOK_UP. This example performs table lookup based upon the name of the numeric variable.

Performing Table Look-Up Based on the Names of Numeric Variables

```
DATA whatever (DROP=_:);
   SET whatever;
1  ARRAY _nums {*} _numeric_;
2  IF (0) THEN
      SET look_up (KEEP=name);
   END;
3  DO _index=1 TO DIM {_nums);
4     CALL VNAME (_nums {_index},
         name);
5     SET look_up (KEEP=name value)
         KEY=NAME / UNIQUE;
6     IF (_iorc_=0) THEN DO;
         /* Place code using */
         /*    VALUE here */
      END;
7     ELSE DO;
         _iorc_ = 0;
         _error_ = 0;
      END;
   END;
RUN;
```

1. Create an array with all of the numeric variables. If there were no numeric variables in WHATEVER, this statement would generate a runtime exception.

2. Ensure that the variable _NAME is defined with correct type and length.

3. Loop through the numeric variables.

4. Place the name of the variable referred to by _NUMS {_INDEX) into the variable NAME.

5. Based upon the value of the variable NAME, retrieve the next observation from the dataset LOOK_UP. The UNIQUE options is used to ensure that lookup always begins at the top of the LOOK_UP dataset.

6. If the lookup was successful, execute the appropriate code.

7. In this example, an unsuccessful lookup is not deemed to be a failure and the error conditions are simple reset

ADVANCED IDIOMS

Using the Interleaving BY Statement

Two statements directly affect the manner in which a SET statement executes: the BY and the WHERE statements. The BY statement interleaves datasets similarly to shuffling a deck of cards. One of the most useful applications of the interleaving SET statement is to merge datasets when the key values do not match exactly (i.e. a "fuzzy merge").

I first learned of this technique from the SUGI paper "Techniques for Interrelating and Reducing Data Bases Using SAS" by Davis, *et. al.*. Their example consisted of merging highway segment data (e.g. Highway 101 from marker 31 to marker 40) with accident data (e.g. highway 101 at marker 35). Highway data was keyed by highway name and starting marker number. Accident data was keyed by highway name and marker number. The following is an adaptation of their approach and joins segment data keyed by the start of the segment to point data.

A Data-Driven Fuzzy Merge

```
DATA whatever (DROP=_:);
1  SET segment (KEEP=key IN=in_seg)
         point;
      BY key;
2  IF (in_seg) THEN
      _seg_ptr + 1;
3  ELSE DO;
4     IF (_seg_ptr) THEN
         SET segment (DROP=key)
            POINT=_seg_ptr;
      /* Place code here */
      OUTPUT;
   END;
RUN;
```

1. Interleave the segment and the point data. As with a MERGE statement, the only variables in common should be the key variables.

2. If the current observation originates in the segment data, simply remember the current segment observation number.

3. If the current observation originates in the point data, write out the data.

4. Before writing out the data, add the current segment data back into the PDV.

Using Metadata Iteratively

Metadata consists of data about data. For instance, the SAS view SASUSER.VTABLE contains information about datasets and views in the SAS System®. The following code provides a data driven approach (using the SET statement, of course) to taking action based upon metadata.

Looping Through Metadata

```
%DO i=1 %TO &n_obs;

   DATA _null_;
1     _point = &i;
      SET metadata POINT=_point;
2     /* Set macro variables here
*/
      /*    with CALL SYMPUTs */
      STOP;
   RUN;

   /* Place code here that uses */
   /*    the metadata variables */
%END;
```

1. Assign the current observation number to the variable referenced by the POINT= option.

2. Set macro variables using the current row of metadata.

Using SCL to Simulate a SET Statement

Finally, instead of presenting another technique for using a SET statement to solve a certain problem, the following code simulates a SET statement. In fact, using a SET statement in its traditional role would simplify the code to the following:

```
DATA whatever / VIEW=whatever;
   SET whatever;
RUN;
```

The benefit of this added complexity comes from the fact that the timing of when the SAS System performs certain actions is now more rigorously defined. Actions can be taken before the SET statement actually begins its work. I have currently found two places to apply a simulated SET statement:

- This technique can be used to achieve push-down of a WHERE clause between a SAS datastep view and an SQL view. Simply define the SQL view of the form *where some_val = symget ('some_val')* and then set the value of SOME_VAL prior to opening the view with a *call symput ('some_val', 'whatever')*. In one instance, I was able to employ this technique to decrease the time needed to access a series of views (SQL view → ... → datastep view) by over a factor of ten.

- This technique can also be used to vary the dataset that is being accessed. For instance, the dataset to open could be found using table lookup with the KEY= option.

A Simulated SET Statement

```
DATA whatever (DROP=_:)
       / VIEW=whatever;
1   IF (0) then
       SET whatever;
2   LENGTH _char $1 _num 4;
3   ARRAY chars {*} _CHARACTER_;
4   ARRAY nums {*} _NUMERIC_;

    /* Place code here */

5   _dsid = OPEN ('whatever', 'i');
6   if (_dsid LE 0) then
       STOP;

    /* Loop thru the dataset */
7   _rc = FETCH (_dsid);
8   DO WHILE (_rc=0);

       /* Write character vars */
9      DO _i=1 TO DIM (_chars) - 1;
10        CALL VNAME (_chars {_i},
             _name);
11        chars {_i} =
             GETVARC (_dsid,
             VARNUM (_dsid,
             _name));
       END;

       /* Write numeric vars */
12     DO i=1 TO DIM (_nums) - 1;
          CALL VNAME (_nums {_i},
             _name);
          nums {_i} =
             GETVARN (_dsid,
             VARNUM (_dsid,
             _name));
       END;

       /* Write the obs */
13     OUTPUT;

14     _rc = FETCH (_dsid);
    END;

15  _rc = CLOSE (_dsid);
RUN;
```

1. Duplicate the PDV of the original dataset.

2. Ensure that there is at least one character and one numeric variable so that an attempt will not be made to define an array with zero elements.

3. Define _NUMS: an array with all of the character variables from WHATEVER.

4. Define _CHARS: an array with all of the numeric variables from WHATEVER.

5. Open the dataset WHATEVER in input mode. This creates a Data Set Data Vector (DSDV) for WHATEVER which contains all of the variables from WHATEVER.

6. Ensure the OPEN worked correctly.

7. Fetch the first observation. This will move one row of data from the DSDV to the SCL Data Vector (SDV).

8. Loop through WHATEVER until there are no more observations to read.

9. Loop through all of the character variables from the WHATEVER dataset. Notice that the variable _CHAR (the last variable in the array) is excluded.

10. Retrieve the name of the character variable.

11. Move the character variable from the SDV to the PDV.

12. Repeat the process of moving variables from the SDV to the PDV. This time for numeric variables.

13. Write the PDV to WHATEVER.

14. Fetch the next observation from WHATEVER.

15. Close WHATEVER.

CONCLUSION

The SET statement can frequently be used to solve problems in a data-driven manner and thus relieve programmers from having to modify code when data changes. Whenever possible, use programming idioms to avoid having to solve recurring problems each time they occur.

REFERENCES

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal (1996) *Pattern-Oriented Software Architecture: A System of Patterns*, Chichester, West Sussex, England: John Wiley & Sons Ltd.

Davis, James W., Steven Flint, Robert U. Anderson, "Techniques for Interrelating and Reducing Data Bases Using SAS," *Proceedings of the Seventh Annual SAS Users Group International Conference, pp.418-423.*

Henderson, Donald J., Merry G. Rabb, Jeffrey A. Polzin (1991), "The SAS System Supervisor - A Version 6 Update," *Proceedings of the Sixteenth Annual SAS Users Group International Conference, pp. 249-257*.

SAS Institute Inc (1995), *SAS Language: Reference*, *Version 6, First Edition*, Cary, NC: SAS Institute Inc.

ACKNOWLEDGEMENTS

SAS and SAS/AF are registered trademarks of SAS Institute, Inc. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The author may be contacted at:

Jack E. Fuller
Lead Software Engineer
Trilogy Consulting Corporation
5278 Lovers Lane, Kalamazoo, MI 49002
jefuller@sprynet.com