EFFICIENCY TECHNIQUES
FOR IMPROVING I/O AND PROCESSING TIME IN THE DATA STEP

Neil Howard, ORI, Inc.

1. ABSTRACT

Saving money is uppermost in the minds of DP managers, and the SAS System is certainly a "programmer efficient" tool: minimal SAS code is required to perform traditionally code intensive tasks. Gratification and and results are immediate.

A familiar quote suggests that the SAS System is easy to use, and just as easy to abuse. Because the SAS System is becoming increasingly popular as a development tool, it is essential to evaluate techniques for its effective use. This tutorial will define efficiency, suggest factors to be considered in measuring efficiency, and provide comparisons of techniques that improve efficiency.

A primary area where the SAS System can be used more effectively is I/O. It should be noted that over 90% of the processing time in any given program involves reading and/or writing data (reference 1). This tutorial will illustrate how to avoid unnecessary I/O, minimize passes of the data, and control read-write loops.

These efficiency techniques also will be discussed in the context of the various resources influencing software development: development personnel, maintenance personnel, CPU time, clock time, region, and space.

By demonstrating the cost savings in the DATA step alone, it is hoped that users will realize that alternative techniques are available and should be evaluated in their approach to accomplishing tasks with the SAS System.

2. DEFINING EFFICIENCY

The dictionary defines efficiency as the ratio of effective or useful output to the input in any system. In the context of our industry and interests, efficiency is the total input (or resources) consumed or utilized by any system or program as compared to some standard such as an existing program or system. Our major concern is accomplishing specific tasks with minimal resources.

This definition is relative, in light of the resources listed above and two other critical elements: frequency of use and maintainability. As an example, consider a cost-benefit analysis of two programs A and B. Program A costs $100 to run while program B costs $1,000 to run. Initially you might conclude that program A is "cheaper". Yet, when you consider that program A is run daily and program B yearly, it is clear that B, on an annual basis, is less expensive. Because program A is run so frequently, more time and money should be invested in its development; it is critical that it be as efficient and maintainable as possible.

Development and production must be considered together when comparing alternative techniques. Figure 1 illustrates that Approach Y results in a program that does not cost much to develop, but costs more to maintain and run on a monthly basis. Approach X results in a program which is more efficient, but costs more to develop and less to run and maintain on a monthly basis.

FACTOR	APPROACH X	APPROACH Y
Development	$30,000	$5,000
Maintenance	$60/month	$50/month
CPU cost per run	$15	$100

Figure 1

It should be noted that monthly costs for A are $75 versus $150 for B. Assuming that both programs are run monthly, the following formulas are used to evaluate the total costs:

Approach X = $30,000 + (($60 + $15) * months)

Approach Y = $5,000 + (($50 + $100) * months)

The cost of each approach at 1, 10, 100, and 1,000 months is illustrated in Figure 2.

MONTHS	TOTAL COST APPROACH X	TOTAL COST APPROACH Y
1	30,075	5,150
10	30,750	6,500
100	37,500	20,000
1000	$105,000	$155,000

Figure 2

A natural question arises: How long will it be before Approach Y becomes more cost effective? This can be answered with the following equation:

$30,000 + (($60 + $15) * months)

= $5,000 + (($50 + $100) * months)

It will be nearly 28 years before the break-even point is reached. This analysis suggests that efficiency is relative and trade-offs must be considered in deciding how much effort should be devoted to optimizing the code.

3. EFFICIENCY TECHNIQUES

This tutorial examines I/O operations, and the results should guide the programmer in making design decisions. A requirements analysis will determine the tasks to be accomplished. The key will be identifying which tasks have multiple possible solutions and which ones can be done only with a specific PROC, a complicated calculation, or application-specific hard-coded statements: Where do we have choices and what are they.

Far too often, programmers don't realize that permanently stored SAS data sets can be referenced directly with a procedure call. Their code will make an unnecessary pass of the data. Figure 3 illustrates the right and wrong solutions.

```
***** WRONG: ******;
  DATA TEMP; SET SASDATA.ANALYSIS; RUN;
  PROC MEANS DATA=TEMP; RUN;

***** RIGHT:  ******;
  PROC MEANS DATA=SASDATA.ANALYSIS; RUN;
```

Figure 3

Or, a programmer may routinely use LIST input for reading raw data. There are more efficient styles. Merely switching to COLUMN or FORMATTED input may not realize tremendous savings, but using a combination of efficiency choices will make a significant difference. The savings will become more striking when processing larger data sets.

Another area might the calculation of the percent change from one year to the next, which must be done in the DATA Step, since no PROC applies. But calculating the average rate of rainfall over several months can be done in the DATA Step, a PROC MEANS, a PROC SUMMARY, etc.

If region is plentiful and clock time is limited, the choice for data reduction would be PROC SUMMARY. If region is tight, CPU time is inexpensive, and clock time is not a factor, then the choice might be PROC SORT with PROC MEANS (reference 8).

Factors influencing a decision in these cases will depend on experience, knowledge of the data, knowledge of the site's computer environment, and knowledge of the SAS System. One of the best resources for finding alternative approaches to typical programming problems is the Proceedings of previous SUGI conferences. Sections 8 and 9 (Bibliography) of this tutorial site other papers that also provide comparisons of techniques for efficiency and effectiveness.

4. MEASURING EFFICIENCY

A prototype is defined by the dictionary as an original form that serves as a model on which later stages are based or judged. Prototyping can be extremely useful in software development to:

- develop a rough version of a system or program to see if its works smoothly,
- refine requirements,
- develop two or more programs to perform the same process using different techniques (for the purpose of comparison).

Prototypes are useful to the programmer in judging the effectiveness and efficiency of various techniques. They are also helpful to the end user for previewing the entire system and sample output. Changes noted during the prototype phase are also more easily addressed.

The basis of this tutorial is, in fact, a prototype which measures efficiency. The prototype system written to compare techniques consists of:

- one DATA step to generate a non-SAS test file and a SAS data base with 250,00 observations and 6 variables, using nested DO loops with embedded assignment and OUTPUT statements,
- a series of DATA and PROC steps that read and write this data using different techniques,
- an external mechanism (i.e., a command procedure) that passes a different number of observations to each benchmark run.

The raw data and the SAS data set were generated using the DATA step shown in Figure 4, directing records to an external flat file and observations to an externally stored SAS data set.

The program required minimal changes to execute on an IBM 3090 under MVS/XA and on a VAX 8700 under VMS, both running Version 5.16 of the SAS System. The program was run with OPTIONS OBS=0 to determine the compile time for each step; then the program was run with 100, 500, 1,000, 5,000, 10,000, 50,000, and finally 100,000 observations, to evaluate the effect of data set size. A limited number of variables was included in the data sets, so it should be noticed that the efficiency savings in the examples to follow will be dramatically increased if more variables are involved.

```
DATA SASDB.SASDATA (DROP = I J);
  FILE RAWDATA;
  DO TEST = 1 TO 4;
    DO STATUS = 1 TO 5;
      DO FLAG = 1 TO 10;
        DO CODE= 1 TO 5;
          DO I = 1 TO 10;
            VALUE = UNIFORM(0);
            DO J = 1 TO 25;
              RANDOM = UNIFORM(0);
              OUTPUT;
              PUT @1 TEST @10 STATUS
                  @20 FLAG @30 CODE
                  @40 VALUE @60 RANDOM;
            END;
          END;
        END;
      END;
    END;
  END;
  STOP;
RUN;
```

Figure 4.

The following I/O operations were compared:

4.1 reading raw data

LIST vs. COLUMN vs. FORMATTED input

4.2 subsetting raw data

IF...THEN vs. trailing @ with IF...THEN

4.3 reading SAS data

DATA step loop vs. DO UNTIL loop

4.4 sampling SAS data

sequential vs. direct access

4.5 DATA step activities

recoding with IF...THEN vs. PROC FORMAT

the effect of the LENGTH of variables

use of DROP, KEEP statements vs. data set options

4.6 conversions

numeric to character, character to numeric

default activity vs. FUNCTIONS

4.7 SAS data set maintenance

DATA step vs. PROC DATASETS

concatenation in the DATA step vs. PROC APPEND

4.1 READING RAW DATA

The three DATA stepsin Figure 5 each read the external raw data file of 250,000 records, the first with LIST input, the second with COLUMN, and the third with FORMATTED input.

In the IBM environment, the most notable results of these runs were: LIST input required slightly more compile time; at 5,000 records, COLUMN and FORMATTED input were 5% more efficient and at 100,000 records 7% more efficient. As suggested earlier, this difference alone may not result in dramatic differences in execution time, but combined with other techniques that should become part of your standards, the savings will be substantial.

Method 1

```
DATA READ1;
  INFILE RAWDATA;
  INPUT TEST STATUS FLAG CODE VALUE RANDOM;
RUN;
```

Method 2

```
DATA READ2;
   INFILE RAWDATA;
   INPUT TEST 1 STATUS 10-11 FLAG 20-21 CODE 30
         VALUE 40-55 RANDOM 60-75;
RUN;
```

Method 3

```
DATA READ3;
  INFILE RAWDATA;
  INPUT @1 TEST 1. @10 STATUS 2. @20 FLAG 2.
        @30 CODE 1. @40 VALUE 16. @60 RANDOM 16.;
RUN;
```

Figure 5

On the VAX side, COLUMN and FORMATTED input were 3% more efficient at 5,000 records and 8% and 9% more efficient, respectively, at 100,000 records.

4.2 SUBSETTING RAW DATA

Figure 6 shows two techniques were compared for creating a subset of the raw data. To create SUB1, COLUMN input was used to read each record. The input statement was followed by an IF test, selecting records where the value of the variable RANDOM was < .1. To create SUB2, only the variable RANDOM was read initially, and a trailing @ was used to hold the record while the IF test was made. Only if the test was true was the rest of the record read in; otherwise control was returned to the Supervisor without processing (reading or writing) that observation.

Method 1

```
DATA SUB1;
  INFILE RAWDATA;
  INPUT TEST 1 STATUS 10-11 FLAG 20-21 CODE 30
        VALUE 40-55 RANDOM 60-75;
  IF RANDOM < .1 THEN OUTPUT;
RUN;
```

Method 2

```
DATA SUB2;
  INFILE RAWDATA;
  INPUT RANDOM 60-74@;
  IF RANDOM < .1 THEN DO;
    INPUT TEST 1 STATUS 10-11 FLAG 20-21 CODE 30
          VALUE 40-55;
    OUTPUT;
  END;
RUN;
```

Figure 6

The results of these runs were dramatic for both systems. On the IBM, at 500 records, Method 2 was already 34% faster and, at 1,000 records and beyond, proved to be at least 43% more efficient. On the VAX, at 500 records, Method 2 was 29% faster and, at 1,000 records, was 33% more efficient.

4.3 READING SAS DATA SETS

Two simple techniques were used to read in the observations from the SAS data set: the first used the DATA step itself as the read-write loop, each execution of the DATA step reading, then writing, one SAS observation sequentially. The second method put the SET

Method 1

```
DATA READSAS1;
  SET SASDB.SASDATA;
RUN;
```

Method 2

```
DATA READSAS2;
  DO UNTIL (LASTREC);
    SET SASDB.SASDATA END=LASTREC;
    OUTPUT;
  END;
  STOP;
RUN;
```

Figure 7

statement inside a DO UNTIL loop, each pass through the loop processing one observation. Using Method 2, all executions of the read-write loop are contained in one execution of the DATA step. Control is never returned to the SAS Supervisor. Whenusing this technique, the programmer is responsible for controlling the read, write, and housekeeping operations pertaining to the PDV. It is advisable to know what SAS Supervisor activities must be accounted for (reference 10).

On the IBM side, a noticeable savings was not realized until the 5,000 observation mark, when Method 2 proved to be 14% more efficient. At 100,000 observations, the difference was 15%. On the VAX, at 5,000 observations, Method 2 was 19% faster, and, at 10,000 observations, was 15% more efficient.

Several of the following DATA steps will use both of the above methods for reading SAS data, and will demonstrate that, when used with other efficiency techniques, the gain is greater.

Method 1

```
DATA SAMP1;
  SET SASDB.SASDATA;
  IF UNIFORM(0) < .2 THEN OUTPUT;
RUN;
```

Method 2

```
DATA SAMP2;
  DO UNTIL (LASTREC);
    SET SASDB.SASDATA END=LASTREC;
    IF UNIFORM(0) < .2 THEN OUTPUT;
  END;
  STOP;
RUN;
```

Method 3

```
DATA SAMP3 (DROP=I);
  DO I = 1 TO (.20*&TOTOBS);
    PTR=(INT(UNIFORM(0)*&TOTOBS) + 1);
    SET SASDB.SASDATA POINT=PTR;
    OUTPUT;
  END;
  STOP;
RUN;
```

Figure 8

4.4 SAMPLING SAS DATA SETS

To create subsets of the SAS data, three sampling methods were used. Simple IF tests in the DATA step read-write loop, IF tests within a DO UNTIL loop, and direct access using the POINT= option on the SET statement (see Figure 8).

At 10,000 observations, Method 2 required 15% less computer resources and at 50,000 and 100,000 proved to be 20% more efficient. With respect to Method 3, it has been shown that direct access is an efficient method when reading up to 20-25% of the data set, particularly when the input data set is extremely large and when "pointer files" into the data base are used (reference 4). It should also be noted that as the size of the sample decreases, direct access will become increasing more efficient.

4.5 DATA STEP ACTIVITIES

The programs in this section demonstrate that "little things" count. When packaged together, the "little things" add up to significant savings.

If your application includes many binary "flags" or integer code variables, consider representing these variables as character instead of numeric. As shown in Figure 9, taking four (4) variables in the test data base and converting them from numeric length 8 to character length 1 or 2 realized a 35% increase in the number of observations that could be stored per track on the IBM system. Under VMS, the original form of the data base used 9,384 out of 9,384 blocks of 512 bytes each. The data base with the conversions used 5,864 out of 5,865 blocks, representing a savings of nearly 38%.

```
DATA LENGTH (DROP=TEST STATUS FLAG CODE);
  DO UNTIL (LASTREC);
    SET SASDB.SASDATA END=LASTREC;
    CTEST   = PUT(TEST,1.);
    CSTATUS = PUT(STATUS,2.);
    CFLAG   = PUT(FLAG,2.);
    CCODE   = PUT(CODE,1.);
    OUTPUT;
  END;
  STOP;
RUN;
```

Figure 9

Recoding and classifications are common requirements within the DATA step. The methods compared in Figure 10 are: IF...THEN constructions, IF...THEN...ELSE statements, PROC FORMAT with the PUT function, IF...THEN...ELSE statements in a DO UNTIL loop, and the table look up (PROC FORMAT with the PUT function) in a DO UNTIL loop. It should be noted that for the table look up methods, a PROC FORMAT was run, but the CPU cost was not figured in the calculations. Formats that are frequently used should be stored permanently in libraries, and the cost of making them available to the program is not a factor. The PROC FORMAT that was run is included in Figure 10.

Because the look up table in each case is small, the relative efficient is difficult to detect in these examples, although the differences between doing any of the methods in a DO UNTIL loop was at least 15% more efficient than its DATA step loop counterpart.

The efficiency is apparent when the trade-offs are examined. Hard-coding IF...THEN or IF...THEN...ELSE is tedious and results in a tired programmer, a less readable program, and a program that is not easy to maintain. Using PROC FORMAT with the PUT function provides a readable and maintainable table that is separate from the DATA step and easier to generate and maintain; and the DATA step code to perform the look up is one line. For large look up tables, the PUT function look up is accomplished by a balanced binary search which is more efficient than the boolean tests required for IF....THEN...ELSE statements (reference 2).

The number of variables read into a DATA step or omitted from the output data set has an impact on the buffer sizes and makes a "little" contribution to the efficiency of the DATA step. DROP and KEEP

Assumed Activity

```
PROC FORMAT;
  VALUE NAME  1   = 'P'  2   = 'S'
              3   = 'R'  4   = 'B'
              5   = 'Z'  6   = 'O'
              7   = 'N'  8   = 'C'
              9   = 'M'  10  = 'E';
RUN;
```

Method 1

```
DATA RECODE1;
  SET SASDB.SASDATA;
  IF FLAG =  1 THEN CASE = 'P';
  IF FLAG =  2 THEN CASE = 'S';
  IF FLAG =  3 THEN CASE = 'R';
  IF FLAG =  4 THEN CASE = 'B';
  IF FLAG =  5 THEN CASE = 'Z';
  IF FLAG =  6 THEN CASE = 'O';
  IF FLAG =  7 THEN CASE = 'N';
  IF FLAG =  8 THEN CASE = 'C';
  IF FLAG =  9 THEN CASE = 'M';
  IF FLAG = 10 THEN CASE = 'E';
RUN;
```

Method 2

```
DATA RECODE2;
  SET SASDB.SASDATA;
  IF FLAG = 1 THEN CASE = 'P';
  ELSE IF FLAG =  2 THEN CASE = 'S';
  ELSE IF FLAG =  3 THEN CASE = 'R';
  ELSE IF FLAG =  4 THEN CASE = 'B';
  ELSE IF FLAG =  5 THEN CASE = 'Z';
  ELSE IF FLAG =  6 THEN CASE = 'O';
  ELSE IF FLAG =  7 THEN CASE = 'N';
  ELSE IF FLAG =  8 THEN CASE = 'C';
  ELSE IF FLAG =  9 THEN CASE = 'M';
  ELSE IF FLAG = 10 THEN CASE = 'E';
RUN;
```

Method 3

```
DATA RECODE3;
  SET SASDB.SASDATA;
  CASE = PUT(CODE,NAME.);
RUN;
```

Figure 10

Method 1

```
DATA SUBSAS1;
  SET SASDB.SASDATA;
  DROP RANDOM VALUE TEST CODE;
RUN;
```

Method 2

```
DATA SUBSAS2;
  SET SASDB.SASDATA(KEEP=STATUS FLAG);
  OUTPUT;
RUN;
```

Method 3

```
DATA SUBSAS3;
  DO UNTIL (LASTREC);
    SET SASDB.SASDATA(KEEP=FLAG STATUS) END=LASTREC;
    OUTPUT;
  END;
  STOP;
RUN;
```

Figure 11

statements can be used in the DATA step, as shown in Figure 11, Method 1, which does not change the number of variables read in, only those included in the output data set. But DROP and KEEP can also be effectively used as input data set options, Method 2, which limits the number of variables initially read into the Program Data Vector.

On the IBM, a 13% savings can be realized by combining the use of the input data set options with a SET inside a DO UNTIL loop. On the VAX, a 37% savings was realized. If you are dealing with data sets with a substantial number of variables, the savings will be more dramatic.

Method 1

```
DATA CONVERT1;
  LENGTH X $ 2 Y $ 1;
  SET SASDB.SASDATA;
  X=FLAG;
  Y=CODE;
RUN;
```

Method 2

```
DATA CONVERT2;
  LENGTH X $ 2 Y $ 1;
  SET SASDB.SASDATA;
  X=PUT(FLAG,2.);
  Y=PUT(CODE,1.);
RUN;
```

Method 3

```
DATA CONVERT3;
  LENGTH X $ 2 Y $ 1;
  DO UNTIL (LASTREC);
    SET SASDB.SASDATA END=LASTREC;
    X=PUT(FLAG,2.);
    Y=PUT(CODE,1.);
    OUTPUT;
  END;
  STOP;
RUN;
```

Figure 12

Method 1

```
DATA ALL1;
  SET SAMP1 SAMP2;
RUN;
```

Method 2

```
DATA ALL2;
  DO UNTIL (LASTREC);
    SET SAMP1 SAMP2 END=LASTREC;
    OUTPUT;
  END;
  STOP;
RUN;
```

Method 3

```
PROC APPEND BASE=SAMP1 DATA=SAMP2 FORCE;
RUN;
```

Figure 13

4.6 CONVERSIONS

Character to numeric, and numeric to character, conversions occur when:

- incorrect argument types are passed to a function
- comparisons of unlike type variables occur
- performing type-specific operations (arithmetic) or concatenation (character only)

The rule should be to perform any necessary conversions yourself and not allow the default conversions, which may yield unexpected or undesired results. As shown in Figure 12, Method 1 allows SAS to do the conversions, default techniques prevailing. Method 2 illustrates numeric to character conversion using the PUT function. Character to numeric would be accomplished using the INPUT function. Method 3 places the programmer controlled conversion techniques inside a DO UNTIL loop.

On the IBM side, at 10,000 observations there is an 8% difference between Method 1 and Method 3. On the VAX, at 10,000 observations there is a 34% difference. Not only are savings realized, the programmer is confident of the conversion results.

4.7 DATA SET MAINTENANCE

Two maintenance activities are evaluated in this section: concatenation of SAS data sets and modifying elements of the data set: changing a data set name, changing variables names, and adding LENGTH and FORMAT attributes to existing data sets.

Concatenation can be performed in the DATA step by naming two or more data sets on one SET statement; by default they will be concatenated. Figure 13 shows concatenation of two data sets previously created, SAMP1 and SAMP2, in a DATA step loop(Method 1) and a DO UNTIL loop (Method 2). Method 3 illustrates the use of PROC APPEND to identify a "base=" data set and append a "data=" data set.

On the IBM, with SAMP1 and SAMP2 of equal size, Method 2 was 12% more efficient than Method 1, and Method 3 was 65% more efficient than Method 1. On the VAX side, Method 2 was 26% more efficient than Method 1 and Method 3 was 80% more efficient than Method 1. The size of the data sets will be tremendous factor when using the SET statement code examples, whereas the size of the data sets is inconsequential to PROC APPEND.

Method 1

```
DATA SASDB.NEWNAME;
  SET SASDB.SASDATA;
  RENAME FLAG=TESTFLAG;
  LABEL STATUS = 'PATIENT STATUS AS OF 1/1/88'
        VALUE  = 'TEST RESULTS FOR WEEK OF 2/15/88'
        TEST   = 'TEST TYPE';
  FORMAT STATUS RANDOM 8.4;
RUN;
```

Method 2

```
PROC DATASETS LIBRARY=SASDB;
  MODIFY SASDATA;
    RENAME FLAG=TESTFLAG;
    LABEL STATUS = 'PATIENT STATUS AS OF 1/1/88'
          VALUE  = 'TEST RESULTS FOR WEEK OF 2/15/88'
          TEST   = 'TEST TYPE';
    FORMAT STATUS RANDOM 8.4;
    CHANGE SASDATA=NEWNAME;
RUN;
```

Figure 14

The need to modify existing datasets can best be met with PROC DATASETS, as illustrated by the two examples in Figure 14. Method 1 reads a SAS data set, changes its name, renames a variable, and adds LABELs and FORMATs to subsets of variables; but the activity takes place in a DATA step, requiring a pass of the DATA to accomplish the maintenance. PROC DATASETS, however, deals with the descriptor portion of the data set only, eliminating the need to pass the data even once.

On the IBM, at 5,000 observations, PROC DATASETS is 36% more efficient And on the VAX, PROC DATASETS is 98% more efficient. The number of observations will affect DATA step processing, but will not have any implications with PROC DATASETS.

5. THE BEST NEWS

The most dramatic results of the prototype were seen by simply evaluating the difference between reading raw data and SAS-stored data. A comparison of the simple data steps shown in Figure 16 was remarkable.

Method 1

```
DATA READ1;
  INPUT TEST 1 STATUS 10-11 FLAG 20-21 CODE 30
        VALUE 40-55 RANDOM 60-75;
RUN;
```

Method 2

```
DATA READSAS1;
  SET SASDB.SASDATA;
RUN;
```

Figure 15

Reading 100,000 observations using Method 2 proved 73% more efficient under IBM and 81% more efficient under VAX. These results strengthen the argument that data should be stored in SAS format—it is self-documenting, easy to maintain, and cheaper to process.

6. CONCLUSIONS

The relative efficiency of operations or PROCs may vary from Version to Version as the SAS System is upgraded, but the fundamentals of good design, an open mind, and consistent use of a combination of basic efficiency techniques will always apply.

7. OTHER PROTOTYPES

The focus of this tutorial was to measure the relative efficiency of different DATA step techniques and provide guidelines for prototyping other programs or systems. Similar "prototypes" exist that provide comparisons of methodologies:

- performing table look up IF...THEN...ELSE vs. PROC FORMAT with the PUT function vs. binary search vs. hashing algorithms (references 2 and 3),
- PROC TABULATE applications (reference 11),
- data reduction and summarization DATA step vs. PROC MEANS vs. PROC SUMMARY vs. PROC UNIVARIATE (reference 8),
- processing large data sets (references 6 and 7),
- testing SAS programs (reference 5),
- subset selection (reference 12),
- toolboxes (reference 9).

8. ACKNOWLEDGEMENTS

The author would like to thank Taube Wilson and Bob Pulgino for their contributions to this paper. This paper is published in the Proceedings of the Thirteenth SAS Users Group Conference, Cary, NC: SAS Institute, Inc. SAS is a registered trademark of the SAS Institute, Cary, NC. The author may be contacted at:

Neil Howard
Program Director
ORI, Inc.
601 Indiana Avenue, NW, Suite 1000
Washington, DC 20004

9. REFERENCES

1. Yourdon, E., Classics in Software Engineering, Edited by E. Yourdon, (New York, Yourdon Press, 1979)

2. Ray, Craig (1987), "A Comparison of Table Look-Up Techniques", Proceedings of the Twelfth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

3. Ray, Craig (1986), "Efficient Use of Table Look-Up Procedures", Proceedings of the Eleventh SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

4. Howard, Neil and Linda W. Pickle (1984), "Efficient Data Retrieval—Direct Access Using the POINT Option", Proceedings of the Ninth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

5. Howard, Neil, Linda W. Pickle and James B. Pearson, Jr. (1987), "Effective Methods of Testing Using the SAS System", Proceedings of the Twelfth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

6. Mopsik, Judie (1983), "Efficient Techniques for Large Data Files", Proceedings of the Eighth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

7. Dilorio, Frank C.(1985), "Efficient SAS Coding Techniques: Measuring the Effectiveness of Intuitive Approaches", Proceedings of the Tenth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

8. Sharlin, Joshua (1983), "Data Reduction and Summarization", Proceedings of the Eighth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

9. Henderson, Donald J. (1988), "A Toolbox Approach to Networking SAS Experts of Re-Using Instead of Reinventing the Wheel", Proceedings of the Thirteenth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

10. Henderson, Donald J. and Merry Rabb (1988), "The SAS Supervisor", Proceedings of the Thirteenth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

11. Norton, Andrew (1987), "Getting the Most from PROC TABULATE", Proceedings of the Twelfth SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.

12. Henderson, Donald J.(1982), "Selecting Subsets of Data", Proceedings of the Seventh SAS Users Group International Conference, Cary, NC: SAS Institute, Inc.