Paper 258-29

Guidelines for Coding of SAS® Programs

Thomas J. Winn Jr., Texas State Auditor's Office, Austin, TX

ABSTRACT

This paper presents a set of guidelines that could be used for writing SAS code that is clear, efficient, and easy to maintain. The guidelines described herein are widely regarded as good practices for SAS programmers of all ability levels, and for all types of computing environments.

INTRODUCTION

Besides being very powerful, the SAS programming language also is very flexible. It allows programmers to write valid code in several different ways in order to accomplish the same result. However, just because you *can* write SAS code in a particular way does not mean that you *should* do so. Issues involving programming style involve more than just aesthetic interest. SAS code which is written with careful attention to certain principles is much easier to read, easier to correct if it contains mistakes, and easier to change afterward. In my own coding, I strive to find the right balance between computer efficiency and maintainability.

The idea behind conforming to a good set of standards is not to make our lives unnecessarily difficult, but actually to make the accomplishment of our work easier. Like all good habits, following good practices may keep us out of trouble later. On rare occasions, in which expediency is extremely important, I may decide not to follow all of my own standards but, even then, I recognize that I would be better off if I did follow them. There always are risks associated with "cutting corners".

GENERAL RECOMMENDATIONS

First, I recommend that SAS programmers use a Software Development Life Cycle Methodology, whether developing complex systems or individual programs, whether manual or automated, and whether new or a modification of some existing project.

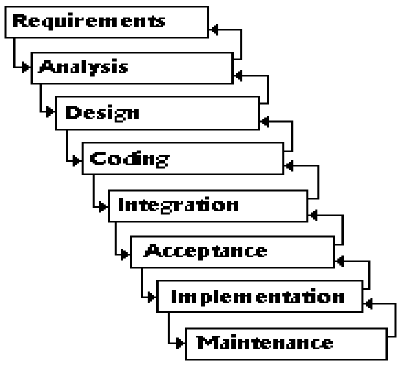

Figure – Typical Software Development Life Cycle Methodology

The steps are:

(1) Problem Definition – Identify the users' requirements – this is a statement of needs, from the users' perspective,
(2) Preliminary Analysis and Planning – Refine the common understanding (between users and programmers) concerning *what* the program or system is supposed to do – these are statements containing the functional specifications, the feasibility of the project, the proposed timeline, the resources needed, and a consideration of alternative solutions,
(3) Design – Determine *how* the program or system will get it done -- create detailed specifications for each process, the development work plan, and a protocol for validation,
(4) Coding and Testing of the components – programming activities,
(5) Integration with other systems, and more Testing,
(6) Obtain User Acceptance – this involves more Testing before user signoff for deliverables,
(7) Implementation, documentation, and user training,
(8) Post-Implementation Analysis, Maintenance, and Change Management.

These steps may be compressed to include only those activities that would be appropriate for a particular project. However many steps there are, each of them would include testing, to ensure that the desired outcomes were achieved. And each step also would seek an agreement of all of the parties involved regarding the outcomes. The essential walkthroughs will help to clarify understandings.

Moreover, I make it a practice to create a physical file folder for each project I work on, containing written documentation regarding each step performed. The folder would include copies of the initial (and subsequent) requests, a narrative description of the work performed, information regarding program inputs and outputs, program listings, SAS logs, pertinent correspondence, and requestor responses. I do this because I tend to forget some details over time. I firmly believe that there is no such thing as disposable code, and I have noticed that very few of the projects that I work on ever completely go away – similar requests seem to just have a way of reappearing. I try to save enough information about the work that I do, so that I could re-construct my results with ease. Chances are, these observations about me and my work also apply to you. So develop the habit of writing essential things down, and of systematically storing them for later reference. And don't just stuff things into the folder, add plenty of explanatory remarks which will give context to the included materials.

NAMES

Use meaningful, unambiguous names for libraries, programs, files, data sets, variables, formats, macros, and macro variables. Well-chosen names furnish important information, which may help the reader to grasp the meaning and intent of the code in which they are used. Generic names are okay, if circumstances make them necessary (for example, when you have a lot of very similar things to be identified), but I prefer to use short, descriptive names, wherever possible. Names like `A, B, C, X1, X2, X3, SPARKY, MUFFY, FLOPSY,` or `TOODLES` are meaningless, and provide the reader with no information. I used to work with a SAS programmer who routinely named SAS data sets after his children. In maintaining his programs, he wasted a lot of time and mental energy reminding himself how to distinguish each data set from the others.

In Version 6 and earlier releases of the SAS System, the maximum length of SAS names was 8 characters. With Versions 8 and 9, most SAS names may be up to 32 characters in length. It's nice to know that we now have so much flexibility in naming. Nevertheless, I still find that short names are easier for me to manage. Try to find the right balance between brevity and meaningfulness. And be consistent. If you have several programs which use the same data items, then use the same names in each program.

Here are some standards pertaining to naming:

- In naming, avoid cuteness, single-letter names, and names that too closely resemble one another.
- Names should be unique, short, and descriptive – in that order of importance.
- If longer names are needed, underscores may be used to separate words, in order to enhance readability. Examples: `DEBT_TO_ASSETS, EMPL_GROWTH_RATE, OP_INCOME_TO_ASSETS, INVENTORY_TO_SALES, LOG_OF_SALES, INVENTORY_TURNOVER, RETURN_ON_EQUITY.`
- If a user-defined format applies to only one variable, then name the format with a readily-recognizable form of the variable-name plus the suffix `FMT` . For example,

```
                PROC FORMAT;
                     VALUE $SEXFMT 'F' = 'FEMALE'
                                   'M' = 'MALE';
```

- Programs which are sequential should share the same name-prefix, but also have a numerical suffix which indicates their place in the sequence.
- Names of data files, which are shared by a group of individuals, should adhere to a naming standard which suits the needs of the group and the context of the data. For example, name components might include distinct identifiers for application, module, modifier, source, date-time, user-ID, sequence-number, and type-extension. Develop a standard that works, and use it consistently.

READABILITY & APPEARANCE

Write code that is readable. It will be easier to understand and to modify.

- ♦ Programs should look neat and orderly,
- ♦ The visual appearance of a program listing should mirror its logical structure,
- ♦ Use white space liberally.

Separate blocks of code, using indents and white space.

- ♦ Insert a blank line between SAS program steps; that is, before each `DATA` or `PROC` step.
- ♦ Be consistent with your indentation increments.
- ♦ Indent all statements in a logical grouping by the same amount.
- ♦ Left-justify all `OPTIONS, DATA, PROC,` and `RUN` statements. Indent all of the statements within a `DATA` or `PROC` step.
- ♦ Indent conditional blocks and `DO` groups, and do it consistently, The logic will be easier to follow.
- ♦ Align each `END` statement with its corresponding `DO` statement. This will make it easier to verify that they match.
- ♦ Remember to preface major blocks of code with explanatory comments.
- ♦ Consider inserting `PAGE` statements to force the SAS Log to begin tracing the execution of new modules on a new page.

Furthermore,

- Don't place more than one programming statement on a single line.
- Make sure that titles and footnotes used in printed reports accurately reflect the information being presented.
- Make sure to use date formats when reporting SAS dates.
- Do not extend any statement beyond column 72.

Here are a few lines of code from a SAS job:

```
PROC SUMMARY DATA=VNDRPMTS NWAY MISSING; WHERE AGENCY=324 AND
('01SEP2002'D<=EFF_DATE<='31AUG2003'D); CLASS OBJECT VEN_NO; ID
VEN_NAME; VAR AMOUNT; OUTPUT OUT=VNDRSMRY.AGY324(DROP=_TYPE_ _FREQ_)
SUM=; PROC PRINT DATA=VNDRSMRY.AGY324 N UNIFORM; VAR OBJECT VEN_NO
VEN_NAME AMOUNT; SUM AMOUNT;  FORMAT AMOUNT DOLLAR15.2; RUN;
```

The following lines would be much easier to follow:

```
*** CREATE A TAPE FILE CONTAINING SUMMARIZED PAYMENTS FOR AGENCY ******;
PROC SUMMARY DATA=VNDRPMTS NWAY MISSING;
     WHERE AGENCY=324 AND ('01SEP2002'D <= EFF_DATE <= '31AUG2003'D);
     CLASS OBJECT VEN_NO;
     ID VEN_NAME;
     VAR AMOUNT;
     OUTPUT OUT=VNDRSMRY.AGY324(DROP=_TYPE_ _FREQ_) SUM=;

*** PRODUCE A REPORT OF THE VENDOR PAYMENTS SUMMARY ********************;
PROC PRINT DATA=VNDRSMRY.AGY324 N UNIFORM;
     VAR OBJECT VEN_NO VEN_NAME AMOUNT;
     SUM AMOUNT;
     FORMAT AMOUNT DOLLAR15.2;
     TITLE1 'SUMMARY OF SELECTED VENDOR PAYMENTS FOR AGENCY #324';
     TITLE2 'DURING FY2003, ACCORDING TO OBJECT-CODE';
RUN;
```

For clarity, simplify complicated code and complex expressions,

- ♦ Insert parentheses in meaningful places in order to clarify the sequence in which mathematical or logical operations are performed,
- ♦ Break really complicated statements into a number of simpler statements,
- ♦ Don't be too clever,
- ♦ Even if you think that it may be more elegant to do so, refrain from using unconventional, obscure, and convoluted logic, except when you can't think of a simpler approach and, if you must, then insert lots of explanatory comments,
- ♦ Avoid implicit coding -- know your defaults, but don't overdo invoking them implicitly (for example, provide names on DATA steps and PROC steps that create data sets, and specify data set names as inputs to SAS procedures),

The SAS System certainly makes computing easy for the programmer. It provides data set names, as well as default types, widths, and formats for variables, even though the programmer may have neglected to code these items explicitly. Unless the programmer specifies otherwise, the SAS System assumes that all of the observations for all of the variables should be included in the output data set. And if the programmer fails to identify a particular SAS data set when invoking a SAS procedure, then the SAS System assumes that the programmer intended that the PROC should be executed using the most-recently-created data set within the job. Each SAS procedure has default options. This is all very convenient, so long as the actual outcome is what was intended. The advice is, *know your defaults, but don't overdo invoking them implicitly.* There needs to be a balance between programmer convenience and program completeness and clarity. The fundamental idea is to be intentional about telling the SAS System what you want, and not just to rely passively on SAS defaults. A reader of your code should be able to fully and correctly comprehend what you wrote. *Don't be too clever.* The programmer should strive to make the program easy to understand, and care also needs to be taken to ensure that the wrong data set isn't used, by accident.

INTERNAL PROGRAM DOCUMENTATION

Use numerous explanatory comments within the program,

- ♦ Use comments to provide a program header, a prologue which describes what the program was intended to accomplish, its author, and the circumstances of its having been written (what, why, for whom, by whom, and when); the prologue also should include a change control log, which describes subsequent modifications made to the program (what, why, for whom, by whom, and when),

- Use comments to present module descriptions,
- Use comments to explain subsetting or other conditional logic,
- Use comments throughout the code to document the program,
- For situations in which meaningful names would be too long, use inline comments (**/*** . . . ***/**) to more fully identify names whose required brevity obscures their meaning.

Here is a skeletal example of a program header. It would be placed very near to the beginning of the program:

```
***********************************************************************;
***  THIS SAS PROGRAM CREATES A REPORT (or DATA FILE) OF ALL . . .  ***;
***  FROM (database & tables) SATISFYING (conditions), AS OF (date).***;
***  IT WAS WRITTEN BY (programmer's name) IN RESPONSE TO A REQUEST ***;
***  THAT WAS RECEIVED ON (date) FROM (requestor's name) OF         ***;
***  (requestor's department), . . .                                ***;
***  THE PROGRAM-ID IS (fully-qualified-name and location).         ***;
***                                                                 ***;
***  ANY QUESTIONS ABOUT THIS PROGRAM, OR THE REPORT(or DATA FILE)  ***;
***  IT PRODUCES, SHOULD BE REFERRED TO (person responsible),       ***;
***  PHONE (xxxxxxx), EMAIL (zzzzzzzzzzzzzz).                       ***;
***********************************************************************;
***  PROGRAM MODIFIED SO AS TO . . .(actions), IN RESPONSE TO . . . ***;
***                                        (date) (initials/UserID) ***;
***********************************************************************;
```

Here is an example which illustrates the two different types of comments which may be used in a SAS job *(* comment; and /* comment */):*

```
***  IF ALL ELSE FAILS, COUNT HOW MANY NAME-COMPONENTS MAY SOUND    ***;
***  SIMILARLY. - - - - - - - - - - - - - - - - - - - - - - - - - - ***;
***  NOTE – THIS MACRO USES THE SOUNDEX FUNCTION, WHICH IMPLEMENTS  ***;
***         AN ALGORITHM THAT IS DESCRIBED IN D.E. KNUTH, "THE ART  ***;
***         OF COMPUTER PROGRAMMING, VOLUME 3, SORTING AND          ***;
***         SEARCHING", PUBLISHED BY ADDISON-WESLEY IN 1973. - - -  ***;
%MACRO SOUNDZ;
  DATA NONMATCS;
    SET NONMATCS;
    MATCSND = 0;  /* COUNTER FOR ITEMS THAT MAY SOUND ALIKE */
    %DO I=1 %TO 9;
     IF SOUNDEX(NAME&I._&CH1) NE '       '  AND
     (   INDEX(TRIM(SOUNDEX(NAME1_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME2_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME3_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME4_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME5_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME6_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME7_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME8_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0
      OR INDEX(TRIM(SOUNDEX(NAME9_&CH2)),TRIM(SOUNDEX(NAME&I._&CH1)))>0)
     THEN MATCSND = MATCSND + 1;
   %END;
   RUN;
%MEND SOUNDZ;
%SOUNDZ
```

And here is yet another example which illustrates the two comment-types*:*

```
*** WRITE SELECTED PAYROLL INFORMATION TO A FLATFILE ON DASD **********;
DATA _NULL_;
   SET COMBO2;
   FILE PAYDATA;
      PUT @1    CMPNUM          3.0  /* AGENCY NUMBER */
          @4    RPTNAME       $67.   /* EMPLOYEE NAME */
          @71   EMPLOYEE     SSN11.  /* EMPLOYEE S.S.N. */
          @82   CHKDATE  YYMMDD10.   /* PAYMENT DATE */
          @92   CKSEQ           3.0  /* CHECK SEQUENCE NUMBER */
          @95   LABRSEQ         3.0  /* COMPONENT SEQUENCE NUMBER */
```

```
@98   CHECKNO         11.0  /* PAYMENT NUMBER */
@109  PEREND    YYMMDD10.   /* PAYROLL PERIOD ENDING DATE */
@119  PCA             $5.   /* PROGRAM COST ACCOUNT */
@124  OBJCD           $4.   /* COMPTROLLER OBJECT CODE */
@128  DOCNUM          $8.   /* DOCUMENT NUMBER */
@136  FUND            $4.   /* FUND NUMBER */
@140  FY              $2.   /* FISCAL YEAR */
@142  AY              $2.   /* APPROPRIATION YEAR */
@144  GROSS           10.2  /* GROSS AMOUNT OF PAYMENT */
@154  ST_RET          10.2  /* STATE-PAID RETIREMENT AMOUNT */
@164  OASDITAX        10.2  /* STATE-PAID OASDI TAX */
@174  HICOTAX         10.2  /* STATE-PAID HI MEDICARE TAX */
@184  ST_INS          10.2  /* STATE-PAID HEALTH INSURANCE */
@194  FLAGB     $RETFMT.    /* RETIREMENT PLAN */ ;
```

REUSABILITY

Write the code that you use repeatedly as a macro, and then, instead of repeating your code, just invoke the macro. But beware! Since most of the operations of the SAS macro facility are carried out in the background, sometimes debugging them can be fairly mysterious. It is especially important for macros to be well-documented.

- Write code that can be re-used, with different parameters. Keyword parameters are preferable to positional parameters, because they are less likely to be specified incorrectly.
- Write the code you use repeatedly as a macro, and then, instead of repeating your code, invoke the macro.
- Avoid using global macro variables.
- If a macro is used by more than one program, put it into an AUTOCALL macro library.

EFFICIENCY

One method of improving the human efficiency of the programs that we write is to use a linearized, modular style of programming, Programs written in this manner generally require less programmer time for debugging, updating, or modifying.

- ♦ Avoid jumping to statement labels by `GO TO`, or `LINK` statements and `RETURN` statements,
- ♦ If possible, replace logic which jumps between subroutines with `DO ...END` and `IF ... THEN ... ELSE ...`-logic,
- ♦ End every `DATA` and `PROC` (except `PROC SQL`) step with a `RUN` statement,
- ♦ End every `PROC SQL` step with a `QUIT` statement.

Here is an example of a portion of a `DATA`-step which uses `GO TO` statements and subroutine labels in such a way as to cause the SAS System to jump to a new location, and to continue execution from that point onward:

```
LENGTH BEGINP 4  STOPP 4  SORTKEY $19;
IF BEGPER < 1000 THEN GO TO L350;
   ELSE GO TO L355;
RETURN;
L350:  STARTQTR = BEGPER;
       STARTMO = 1;
       BYY = INT(BEGPER/10);
       BQ = BEGPER – (10*BYY);
       IF BQ > 0 THEN BMM = (3*BQ) – 2;
          ELSE IF BQ = 0 THEN BMM = 1;
       GO TO L360;
L355:  BYY = INT(BEGPER/100);
       BMM = BEGPER – 100*BYY;
       STARTQTR = INT(((BMM + 2) / 3) + (BYY*10));
       IF         BMM IN (1,4,7,10) THEN STARTMO = 1;
          ELSE IF BMM IN (2,5,8,11) THEN STARTMO = 2;
          ELSE IF BMM IN (3,6,9,12) THEN STARTMO = 3;
       GO TO L360;
L360:  BEGINP = COMPRESS(STARTQTR || STARTMO);
       IF ENDPER < 1000 THEN GO TO L370;
          ELSE GO TO L375;
       RETURN;
L370:  ENDQTR = ENDPER;
       ENDMO = 4;
       EYY = INT(ENDPER / 10);
       EQ = ENDPER – (10*EYY);
       IF EQ > 0 THEN EMM = 3*EQ;
          ELSE IF EQ = 0 THEN EMM = 12;
```

```
             GO TO L380;
      L375:  EYY = INT(ENDPER / 100);
             EMM = ENDPER – (100*EYY);
             ENDQTR = INT(((EMM + 2)/3) + (EYY*10));
              IF        EMM IN (1,4,7,10) THEN ENDMO = 1;
               ELSE IF EMM IN (2,5,8,11) THEN ENDMO = 2;
               ELSE IF EMM IN (3,6,9,12) THEN ENDMO = 3;
             GO TO L380:
      L380:  STOPP = COMPRESS(ENDQTR || ENDMO);
             SORTKEY = COMPRESS(TPNUM || BEGINP || STOPP);
             KEEP TPNUM    BEGPER   ENDPER    BEGINP   STOPP
                  SORTKEY  TOTHRS   TRESULTS;
```

The preceding example uses a legitimate programming style, but the approach can lead to convoluted statements which may be difficult for a reader to follow. Actually, the preceding example isn't too arcane – it is modular, and fairly linear. However, it could have been made a lot more confusing by making additional separately-labeled subroutines out of the conclusions of each conditional statement, and then by shuffling all of the subroutines in the `DATA-` step. But you get the idea.

Now, here is the same program component, except that the execution is controlled by `DO-END` and `IF-THEN-ELSE` programming statements, which most SAS programmers find are easier to read and to understand:

```
      LENGTH BEGINP 4  STOPP 4  SORTKEY $19;
      IF BEGPER < 1000 THEN DO;
             STARTQTR = BEGPER;
             STARTMO = 1;
             BYY = INT(BEGPER/10);
             BQ = BEGPER – (10*BYY);
             IF BQ > 0 THEN BMM = (3*BQ) – 2;
                ELSE IF BQ = 0 THEN BMM = 1;
      END;
         ELSE DO;
             BYY = INT(BEGPER/100);
              BMM = BEGPER – 100*BYY;
              STARTQTR = INT(((BMM + 2) / 3) + (BYY*10));
              IF         BMM IN (1,4,7,10) THEN STARTMO = 1;
                 ELSE IF BMM IN (2,5,8,11) THEN STARTMO = 2;
                 ELSE IF BMM IN (3,6,9,12) THEN STARTMO = 3;
         END;
      IF ENDPER < 1000 THEN DO;
             ENDQTR = ENDPER;
             ENDMO = 4;
             EYY = INT(ENDPER / 10);
             EQ = ENDPER – (10*EYY);
             IF EQ > 0 THEN EMM = 3*EQ;
                ELSE IF EQ = 0 THEN EMM = 12;
      END;
         ELSE DO;
             EYY = INT(ENDPER / 100);
             EMM = ENDPER – (100*EYY);
             ENDQTR = INT(((EMM + 2)/3) + (EYY*10));
              IF        EMM IN (1,4,7,10) THEN ENDMO = 1;
               ELSE IF EMM IN (2,5,8,11) THEN ENDMO = 2;
               ELSE IF EMM IN (3,6,9,12) THEN ENDMO = 3;
         END;
      BEGINP = COMPRESS(STARTQTR || STARTMO);
      STOPP = COMPRESS(ENDQTR || ENDMO);
      SORTKEY = COMPRESS(TPNUM || BEGINP || STOPP);
      KEEP TPNUM    BEGPER   ENDPER    BEGINP   STOPP
           SORTKEY  TOTHRS   TRESULTS;
```

Group the executable statements. Place most of the non-executable statements in a DATA step before all of the executable statements. In particular, place variable attribute and other declarative statements near to the top of the DATA step, and ahead of the executable statements. There is a difference between declarative programming statements which deal with descriptor information (like `ATTRIB, LENGTH, INFORMAT, FORMAT`) or other declarative statements (like `ARRAY, RENAME, RETAIN`), and which take effect when the SAS System compiles the program, and programming statements which actually affect `DATA` step execution (like `INPUT, INFILE, SET,` MERGE, DO, IF, IF-THEN-ELSE, PUT, RETURN, and assignment statements). It is a good idea to group most of the declarative statements together at the beginning of the DATA step, to keep them from cluttering up the logic portion of the program. However, it also seems logical to place DROP or KEEP statements after the executable statements, instead of before them. One advantage of following this convention of putting like things together is that the reader could get a good idea which variables are created and their characteristics just by scanning the first few lines in a DATA step. An additional advantage of putting like things together is that it is easier to check for completeness and to find certain types of errors.

Here are some more standards that pertain to efficiency:

- In a DATA step, place most of the non-executable statements before the executable statements – exceptions include the DROP or KEEP statements, which may be placed after the executable statements.
- Define INPUT and PUT variables one per line, using @ pointer control.
- Screen data for unusual circumstances.

Errors often can be avoided by checking the data for invalid values. Unfortunately, erroneous or unforeseen data abound throughout the world, and your SAS program will find some of them. Experience can be your best teacher. Pay attention to the unexpected circumstances that aren't supposed to happen, but do, in the particular data files that you use. It is a good idea for you to do some preliminary data exploring with PROC FREQ, to become better acquainted with your data, *before* you put that new, special report into production. If the subject-matter experts tell you that only *n* possible conditions can be applicable in some given situation, you should be skeptical enough to expect more than that, and code accordingly.
.
The most important thing most SAS users can do to improve the computer efficiency of their programs is to decrease the number of input/output operations. Strive to minimize the number of passes through the data, as well as to minimize the number of DATA steps. If multiple data sets are to be created from the master data set, then create them all with one DATA step. The use of intermediate DATA steps to subset the data, to transform the data, or to perform calculations should be curtailed. Instead of using a subsetting-IF statement in a DATA step as a preliminary to running some SAS procedure, eliminate the need for a DATA step entirely by performing the subsetting in the PROC step, through the use of a WHERE statement. Wherever possible, take advantage of procedure output data sets as inputs to other PROCs, to avoid having to read the master data set again. Read and store only the fields that you need. And sort the data only when it is absolutely necessary.

In summary, to reduce the number of times the data are read:

- ♦ Minimize the number of passes through the data,
- ♦ Minimize the number of DATA steps,
- ♦ Read and store only the data that are needed,
- ♦ Sort the data only when it is absolutely necessary.

Here are a few more efficiency-related guidelines:

- When you read in an external file, use pointer controls, informats, or column specifications in the INPUT statement, to read only those fields you actually need.
- Store only the variables you need by using DROP or KEEP statements, DROP= or KEEP= options (eliminate variables from the output data set which are needed only during DATA step execution, and not afterward).
- When only one condition can be true for a given observation, use IF ... THEN ...ELSE ... statements (or a SELECT group), instead of a series of IF ... THEN ... statements without ELSE statements (In a sequence of IF-THEN statements without the ELSE, the SAS System will check each condition for every observation).
- When using a series of IF ... THEN ... ELSE ... statements, list the conditions in descending order of probability. This will save CPU time.,
- Use the LENGTH statement to reduce the storage space for variables in SAS data sets.
- Minimize workspace usage by using the DELETE statement in a PROC DATASETS step, to eliminate temporary data sets that are no longer needed by the program.
- Use the IN operator instead of a series of multiple logical OR operators.

CONCLUSION

By consistently using good coding standards for writing SAS programs, the code will be easier to follow, more efficient to run, and easier to maintain. As a result, programming errors will be reduced, and the quality of the results will be improved. Get yourself into the habit of following good coding standards, and always use them in writing your programs. You know what to do – just do it!

REFERENCES

SAS Institute Inc. (1990). *SAS Programming Tips: A Guide to Efficient SAS Processing,* Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999). *SAS OnlineDoc® Version Eight,* Cary, NC: SAS Institute Inc.

Winn, Thomas J., Jr. (1995). "How to Write SAS Code Which is Efficient and Easy to Maintain", *South-Central SAS Users' Group Proceedings of the Fifth Annual Conference*, pp. 263-269.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Tom Winn, Senior Systems Analyst
Information Systems Team
Texas State Auditor's Office
P.O. Box 12067
Austin, TX 78711-2067

Work Phone: (512) 936-9735
E-Mail: twinn@sao.state.tx.us